MANCHESTER 1824

The University of Manchester

An FPGA-DSP Digital Tomograph with USB Connectivity

A dissertation submitted to The University of Manchester for the degree of Master of Science in Electronic Instrumentation Systems in the Faculty of Engineering and Physical Sciences

2005

Mohammed Al-Loulah

Supervisor: Dr. Krikor Ozanyan

School of Electrical and Electronic Engineering

Content

CONTENT **2**

LIST OF FIGURES **4**

LIST OF TABLES **5**

ABSTRACT **6**

DECLARATION **7**

COPYRIGHT STATEMENT **8**

ACKNOWLEDGEMENTS **9**

CHAPTER 1 .... **10**

INTRODUCTION **10**

1.1 PHILOSOPHY 10
1.2 TOMOGRAPHY & COMPUTATION 11
1.3 PROJECT ENVIRONMENT 12
1.4 BRIEF OVERVIEW, AIMS & OBJECTIVES 13

CHAPTER 2 .... **15**

DESIGN MOTIVATION **15**

2.1 THE DIGITAL CONFIGURABLE CHOICE 15
2.2 THE HYBRID ARCHITECTURE 16
2.3 USB SUPPORT 17

CHAPTER 3 .... **19**

DETAILED SOFTWARE DESIGN **19**

3.1 INTRODUCTION 19
3.2 MATHEMATICAL & NUMERICAL REPRESENTATIONS 19
3.2.1 HexConversion class: *19*
3.2.2 Complex structure: *22*
3.2.3 FFT class: *23*
3.3 USB-RELATED IMPLEMENTATIONS 24
3.3.1 RandomAccessBurst class: *24*
3.3.2 Acquisition class: *25*
3.3.3 Low-level USB functions: *25*
3.4 VISUALIZATION 26
3.5 OPERATING SYSTEM CONCEPTS 27
3.5.1 Theoretical background - Multithreading: *27*
3.5.2 Mutual exclusion solution adopted: *29*
3.5.3 Plot refreshing mechanism: *31*

CHAPTER 4 .... **34**

DETAILED DIGITAL HARDWARE DESIGN **34**

4.1 INTRODUCTION 34
4.2 HARDWARE CONCURRENCY 34
4.2.1 Identifying the problem *34*
4.2.2 Proposed solution *35*
4.3 INTER-PROCESS SYNCHRONIZATION 38
4.4 VHDL CODE MODULAR ARCHITECTURE 41

4.4.1 *A closer look at `SX2_Utilities` package* *42*

CHAPTER 5 47

DSP-RELATED DIGITAL HARDWARE DESIGN 47

5.1 DSP INTERFACE 47
5.1.1 *Introduction* *47*
5.1.2 *Parallel port description* *49*
5.1.3 *DSP_Driver module* *50*
5.2 SYNCHRONIZATION MECHANISM 53
5.3 DSP BUFFERING SCHEME 55

CHAPTER 6 57

EXPERIMENTAL RESULTS & ANALYSIS 57

6.1 MISCELLANEOUS HARDWARE DESIGN CONSIDERATIONS 57
6.2 FPGA INTERNAL LOGIC ROUTING 58
6.2.1 *Problem encountered & solution* *58*
6.2.2 *Analysis* *59*
6.3 USB2.0 ENDPOINT BANDWIDTH 65
6.3.1 *Streaming through isochronous endpoint* *65*
6.3.2 *Streaming through bulk endpoint* *68*
6.4 DSP BOOTING 72
6.5 FPGA METRIC ASSESSMENT 77
6.6 GUI SAMPLE OPERATION 80

CHAPTER 7 82

CONCLUSIONS & FUTURE WORK 82

7.1 CONCLUSIONS 82
7.2 PROPOSED FUTURE WORK 83

List of Figures

Figure 1.1: Overall System in logical block diagram representation............................... 14
Figure 3.1: 32-bit fixed-point signed integer [20] 20
Figure 3.2: 32-bit fixed-point signed fractional [20] 20
Figure 3.3: Density distribution plot.. 27
Figure 3.4: Threads & Processes [24]... 28
Figure 3.5: Host software source tree 33
Figure 4.1: Abstract solution for hardware mutual exclusion.. 35
Figure 4.2: Double buffering scheme 37
Figure 4.3: Inter-process synchronization illustration chart 40
Figure 4.4: `sx2WriteRegAsync` state machine... 44
Figure 4.5: Design module view.. 46
Figure 5.1: Parallel port block diagram [28]... 50
Figure 5.2: 16-bit write 52
Figure 5.3: 8-bit read cycle 53
Figure 5.4: DSP buffering scheme.. 56
Figure 6.1: SRAM-based pass transistors.. 60
Figure 6.2: Symmetrical FPGA Architecture 61
Figure 6.3: Limited Routability [35]... 61
Figure 6.4: A routing example [35] 62
Figure 6.5: DSP booting snapshot 74
Figure 6.6: Digital oscillator transmission snapshot... 75
Figure 6.7: Parallel port transfer operation 75
Figure 6.7: Digital Sinusoidal Oscillator 76
Figure 6.8: A snapshot of the `SoftScope`. 80
Figure 6.9: A snapshot of the `SpectrAnalyzer`. 81

List of Tables

Table 4.1: Select signal truth table 37
Table 5.1: dsp_sel operations 56
Table 6.1: Summary of Main synthesis report with ROM present. 77
Table 6.2: Summary of Main synthesis report excluding ROM. 77
Table 6.3: Summary of “Map” and “Place & Route” reports 79
Table 6.4: Summary of “Generating Clock” section of the “Place & Route” reports 79

Abstract

Tomography allows the imaging of an inaccessible cross-section for a given object by means of non-intrusive measurements taken at the periphery. Due to its ill-posed inverse nature, tomography is a highly demanding processing task.

Field Programmable Gate Arrays FPGAs are emerging not only as effective glue-logic solutions but also as attractive reconfigurable computing devices for hardware-dedicated embedded applications. However, despite their versatility, generic FPGAs tend to have limitations with respect to logic demanding implementations and routing. Coupled with powerful floating-point DSP in a co-processor fashion, an FPGA-DSP hybrid platform with USB interface is a suitable candidate for tomographic applications.

The Xilinx Spartan-3 90nm technology FPGA interfaces with the on-board SX2 USB device and the ADSP-21262 SHARC floating-point DSP as a true master controller. A logical peer-to-peer communication occurs between the FPGA and a multi-functionality GUI running on the host PC. The concurrent aspect of the design allows for maximum parallelism.

Throughout the 4-month project span, a software-hardware co-design was conducted resulting in the hybrid platform being made ready for a further relevant algorithmic exploitation.

Declaration

I declare that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

(1) Copyright in text of this dissertation rests with the author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the author. Details may be obtained from the appropriate Graduate Office. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.

(2) The ownership of any intellectual property rights which may be described in this dissertation is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

(3) Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the School of *Electrical and Electronic Engineering*.

Acknowledgements

Above all, I would like to thank my mother, may peace be upon her soul, for always believing in me. Her unwavering belief and devotion have always been an inspiration for the family.

I would like to express my gratitude to Dr. Krikor Ozanyan for his constant guidance throughout the project. Also I would like to thank Dr. Yang for the support that he provided throughout this year.

I would like to acknowledge the cooperation and help of Mr. Sergio Garcia-Castillo. Sergio provided me with the top-most board of his hierarchical multi-channel system.

Finally, I would like to thank all the staff and students at the school of EEE.

Chapter 1

Introduction

1.1 PHILOSOPHY

Ever since the concept of intelligent computing was introduced even before the advent of the transistor towards the middle of the twentieth century, people have always been able to come up with applications to defy what is realistic at the moment conceptualizing and defining a yet-to-come technology. It is commonly believed that technology steers evolution. This may or may not be true, but what is beyond doubt is that sometimes evolution demands that technology be made available. Back to our context, one of the driving forces in electronics industry is the industrial imaging problem commonly referred to as *Tomography.* Tomography essentially allows to view the internal composition and structure of an object by means of invasive non-intrusive measurements taken at the periphery of the object [1].

In its abstract mathematical sense, tomography was envisioned early in the nineteenth century [2] even long before von Neumann* conceptualized his Instruction Set Architecture (ISA) for modern microprocessors which was named after him later in tribute. Then it was not until 1917 that tomography has been officially declared a possibility when Radon† generalized the concept to include objects of arbitrary shape.

In this context, if we let the revolutionary digital processing means to be our technology, and we let tomography to be our evolution. Have we not waited for the availability of technology to evolve? Or has evolution spawned technology as a natural consequence to demand. One might argue quite strongly in support to either side. However, what is inarguably evident is that at some point in time, the world's collective reason or *world spirit* (according to Hegelian philosophy) is gradually escalating throughout history towards *becoming* more aware of its reason (the reason

---

* **Neumann**, John von (1903–1957), Hungarian-born U.S. mathematician. He developed game theory and quantum mechanics, and was a pioneer in computer theory and design.*

† **Radon**, Johann (1887-1956), Austrian mathematician. He demonstrated that any N-dimensional object can be "reconstructed" from an infinite number of (N-1)-dimensional "projections".

within) manifested in our context as the world's physical laws which govern the universe we live in. *Becoming* itself is the act of synthesizing *being* and *coming*. With *being* refers to the capacity to develop and *coming* refers to the realization and fulfilment of the potential [3].

1.2 TOMOGRAPHY & COMPUTATION

In tomography, the *reconstruction* of an object whose parameters are being interrogated with a certain sensing modality involves solving an inverse problem [4]. Fundamentally, the inverse problem is the act of regaining the true characteristics of an object from a measured set of data obtained after the interrogation of those characteristics through the use of a particular modality (attenuation in x-ray, permittivity in capacitance) [5]. The inverse problem can be traced back to physics whereby experimental results are fitted to a theoretical ideal model [4]. Due to its ill-posed nature, the complexity of the system of equations through which fitting is being performed is a function of the number of measurements and the desired precision. Thus intuitively as it may seem, the imaging of an inaccessible cross-section for a given composite of substances (typically found in process tomography) requires a considerable number of quantifying measurements for it to be of significance in the industrial environment (yielding acceptable resolution). The reason is that all tomographic modalities involve coupling and ill-conditioning [6]. Firstly, coupling refers to the fact that the contributions of many voxels on which a modality is being applied affect each single measurement. Therefore, for each frame a set of equations modelling the system's behaviour need to be evaluated with probably dynamically changing characteristics both in time (time variance vs. time invariance) and in space (shift variance vs. shift invariance). Secondly, ill-conditioning stems from the fact that images are very susceptible to noise unless *a priori* information exist. Therefore, numerical computation lies at the essence of tomography as both coupling and ill-conditioning are ever present in all tomographic modalities. As a result, an extra processing overhead is ultimately superimposed on top of a rather naïve "transcendental" reconstruction technique to accommodate for the above factors due to the cruelty of the physical world.

Often the coupling effect, or conversely the *non-local* effect of measurements is far more severe in lower frequency modalities. Meaning x-ray tomography measurements are very much local compared to other low-frequency modalities such as optical tomography. However, the high cost of nuclear modalities makes them unjustifiable in some cases of process tomography not to mention other contributing factors such as the difficulty and high cost of the associated maintenance.

1.3 PROJECT ENVIRONMENT

An ongoing project in the group of Sensors, Imaging and Signal Processing (SISP) at the school of Electrical and Electronic Engineering (EEE) is the design of a low-cost general architecture for digital tomography systems. The idea is to develop a hierarchical architecture of cascadable nodes responsible for the acquisition and processing of highly demanding tomographic data corresponding to measurement channels. Ultimately the system should be able to perform fully or partially onboard intensive computations, e.g. tomographic image reconstruction and then to pass the resulting stream of data to a computer for finalization, display, and storage.

The system design is centred around a very simple yet so powerful concept; reconfigurability. Tomographic image reconstruction systems are characterized by being highly specialized. Targeting a specific imaging technique, systems developed so far have proven to be difficult to be migrated from in order to facilitate other tomography systems utilizing the same platform. Resulting in little hardware reusability once the design is fully implemented.

Addressing this problem alongside the need to shorten development time, the system currently being designed aims at implementing a generalized architecture that can be tailored in terms of logic resources and expanded in terms of hierarchy as to suit a wider spectrum of tomographic modalities. Down at the bottom of the pyramid, nodes with intimate dependency on the underlying tomographic technique can be replaced altogether in worst case scenario without having a propagating effect up towards the top of the pyramid where all channels converge to a single node. This node performs the last processing tasks and/or packing before communicating the final bit stream to a host PC.

1.4 BRIEF OVERVIEW, AIMS & OBJECTIVES

The aim of this project is to utilize a hybrid architecture consisting of a Field Programmable Gate Array (FPGA) and a Digital Signal Processor (DSP) as its primary digital processing elements to implement a Universal Serial Bus (USB) interface to a host PC. The accomplishment of the proposed task dictates a rather extensive deployment of different programming languages through their underlying environments. The meticulous nature of project is a byproduct of the number of tasks associated with its successfulness. A detailed study of the general theory of USB is crucial for choosing a suitable USB2.0 device that can be integrated in the board architecture in a seamless manner. This was carried out from the very beginning and before commencing the actual project. Then VHDL as a means for synthesizing a synchronous core on the FPGA was investigated thoroughly. The VHDL code is the essence of the overall system as it is the true remote master which controls and synchronizes operations driven and initiated by an end user on the host PC. The VHDL core is designed to meet logical requirements in a Finite State Machine (FSM) manner as well as complex timing specifications as handshake and glue logic. Key issues tackled in VHDL range from logic design to inter-process synchronization. Further, C# programming language is used to implement a software interface which comprises in addition to its Graphical User Interface (GUI) classes to handle tasks including acquisition, file processing, multithreading, waveform plotting, mutual exclusion, complex numbers representation, decimal to two's complement conversion and vice versa, and FFT computation for assessing the spectral components of the signal being acquired. C# was adopted bearing in mind that manufacturers supply a free C++ Dynamic Link Library (DLL) which can be integrated into the C# code exploiting the interoperability feature of the .NET environment. Finally, an interface to a powerful onboard floating-point DSP equips the system with the capability to reload the DSP's firmware from the FPGA without using its expensive dedicated PCI-based JTAG debugger and emulator. This is achieved by deploying the bootstrap feature of the DSP through its parallel port.

The overall system is depicted below:

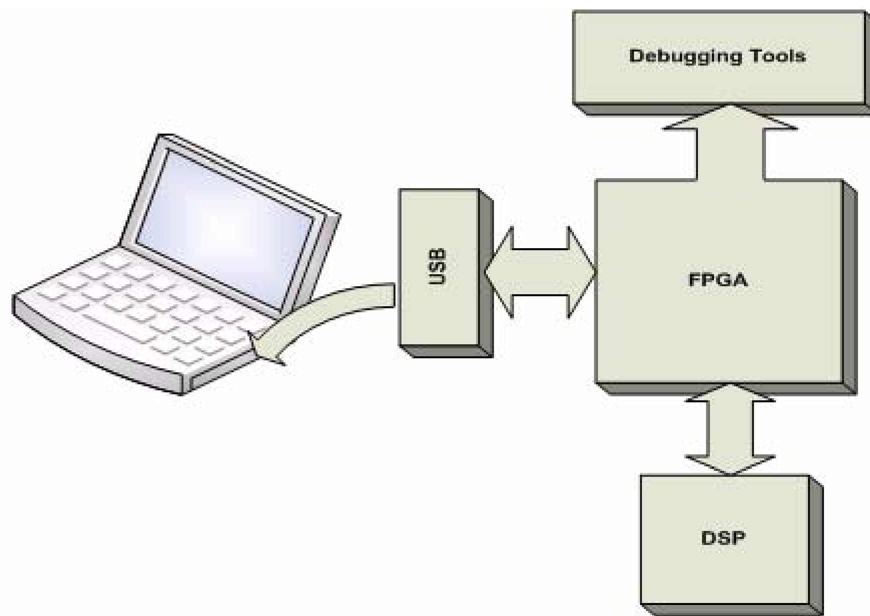

Figure 1.1: Overall System in logical block diagram representation

The roadmap to the successful completion of the project involves:

- Phase 0: USB specifications review, components choice, and identifying various system design requirements.
- Phase 1: extensive VHDL literature review, and migration from JAVA to C#.
- Phase 2: interfacing the FPGA to the USB device by implementing all necessary procedures whether for enumeration, configuration, or bidirectional data transfer.
- Phase 3: designing and implementing the PC-side GUI with all necessary functionalities as to control, format, display, and store the acquired data.
- Phase 4: reviewing the DSP specific architecture, implementing communication between the FPGA and DSP, and demonstrating the overall working scheme.

Phase zero was carried out during the design study module. Starting in April, the actual project realization began with phase one.

Chapter 2

Design Motivation

2.1 THE DIGITAL CONFIGURABLE CHOICE

It was discussed earlier that tomography is a demanding processing task. Yet the question that needs addressing is to decide on how and where to discharge the required computational load. Following on the recent breakthroughs in the contemporary state-of-the-art technologies, an ideal approach would be to employ a top-of-a-line nano system towards fabricating a dedicated system-on-chip which encapsulates in its dense customized resources all necessary processing yielding optimal performance in terms of mass, power consumption, and speed. However, in reality, important factors such as time-to-market, design complexity, and cost tend to influence people's adoption for a particular methodology.

Starting at the top of the hierarchy in its broadest sense, signal processing can be performed either in the analogue domain or in the digital domain. Although all tomographic modalities have their signals originating in the analogue domain, restricting analogue operation to preconditioning and other preliminary tasks can result in both prior and posterior advantages. Firstly, utilizing digital circuitry is more cost effective; digital signals are more immune to noise fluctuation alleviating the need for strong Electro-Magnetic Compatibility (EMC) inwards radiational shielding compared against full custom analogue circuitry. Secondly, digital solutions are highly reconfigurable due to their reprogrammability; in addition, they offer an unmatched versatility due to their in-circuit functional selectivity support. Furthermore, stand-alone digital design can be easily integrated and expanded into other systems.

Descending down the hierarchy, Application Specific Integrated Circuits (ASICs) are emerging as strong solution that is being adopted widely in industry. They are characterized by unmatched performance especially with respect to power

consumption and EMC performance. However, unless mass-produced, ASICs' cost can be quite unsustainable due to their extremely high prototyping cost.

2.2 THE HYBRID ARCHITECTURE

Reaching the purely digital layer in hardware design hierarchy, Digital Signal Processors (DSPs) are fully customized chips that are designed to perform DSP-related operations in the fastest and most efficient manner and most notably concurrently whenever possible. From barrel shifters to bit-reverse addressing modes, DSPs have all possible resources to implement a wide variety of processing operations. Nevertheless, since the architecture is preconfigured, the designer can not do much apart from customizing his / her firmware. In other words, DSPs have inherent limitations due to their instruction set architecture (ISA) giving rise to memory and instruction bottlenecks [7]. DSPs deliver their optimal speed performance when their pipelines are fully utilized resulting in the highest possible instruction throughput. Thus in fast streaming systems, such as the processing of tomographic data, responding to outside events compromises the effective processing instructions being executed and in turn limits the complexity for which the system can accommodate. Furthermore, DSPs suffer from relatively high power dissipation rates and often require extensive glue logic to interface with the rest of the system [8]. In general, DSPs have less versatility with respect to their peripheral capabilities when compared to regular microcontrollers. This is quite logical because DSPs targets data processing rather than controlling events. In ISA architecture the device utilizes the instruction path to execute one task at a time which results in tasks being scheduled for execution in a sequential fashion. Recently a new class of embedded solutions has emerged in an attempt to accommodate for broader applications. Named Digital Signal Controller (DSC), manufacturers (such as Microchip®) aims at offering low performance DSPs (typically 30 MIPS) with enhanced peripherals support such as Pulse Width Modulation (PWM) core for motor control applications [9].

On the other hand, FPGAs are much efficient in terms of power dissipation and are highly generic and reconfigurable in their architecture as they can be tailored to suit a particular application independently of the underlying silicon platform as opposed to ASICs [10]. Being able to support the industry's highest speed serial bit stream LVDS

[11], typically > 500 MHz, FPGAs are amongst few comprehensive configurable solutions available for developers to do so. In fact, FPGAs have been long adopted as the prominent solution for implementing glue logic, but have been just recently looked at as an effective DSP solution due to their ever increasing speed and logic density [7]. However, the caveat is that complex designs can be very demanding in terms of available resources. Moreover, designs could easily grow unmanageable up to the point when third-party Intellectual Property IP cores become inevitably necessary introducing additional cost. Otherwise, designers can find themselves reinventing the wheel in every single project, and possibly in every single task within a project. Whereas when buying an established DSP from a certain manufacturer, one will be charged neither for the effort nor for the research put into designing and implementing the product simply because of the mass-production aspect associated with it.

Real-life experience has shown that a hybrid scheme utilizing both solutions in a complementary fashion is highly desirable to strike a balance between the merits and disadvantages of both technologies. The result is rapid hardware reconfigurability for a wide range of applications [12-16]

2.3 USB SUPPORT

USB is emerging as the industry's unified standard since its introduction in 1998. Increasing number of peripherals across different platforms is adopting USB to support a host-device connectivity. This has led to a quick embrace of USB by many manufactures [17] in the quest to satisfy the huge demand due to its powerful features which include: auto detection and configuration, easy expansion using hubs, reliability, and low cost. Recently USB standard was enhanced to include 'On-The-Go' (OTG) functionality [18] enabling point-to-point data exchange between remote products which added even more popularity. Now the wireless USB standard is being finalized [19] with predictions that it will soon replace Bluetooth. USB2.0 High-speed delivers a theoretical bitrate of 480 Megabits/second which makes it a suitable candidate for the project's PC interfacing aspect as it is roughly estimated that the aggregate data stream generated by the overall system hierarchy would be around 25

Mega-Byte per second. In addition, its endpoint-based modular protocol facilitates a straightforward communication scheme defined on top of USB.

Chapter 3

Detailed Software Design

3.1 INTRODUCTION

In this chapter, a detailed treatment of the system's software architecture is presented. Rather than addressing all the theory of the software design which is beyond the scope of this report, the following discussion will attempt to explain important system-specific issues associated with the software. These issues are grouped according to their logical nature as follows: mathematical & numerical representations, USB-related considerations, plotting and visualization, all the way up to Operating System (OS) concepts. Selections of C# code will be listed whenever necessary to facilitate the discussion. However, for full code listing please refer to appendix C at the end of the report.

3.2 MATHEMATICAL & NUMERICAL REPRESENTATIONS

3.2.1 *HexConversion* class:

Starting in a strictly mathematical fashion, it was imperative that the system be able to translate the numeric format supplied by both the FPGA and the DSP. Whether integer or fractional, the FPGA and the DSP use a signed two's complement notation. In response, a class named `HexConversion` was developed to handle the task of converting data from hexadecimal to double and vice versa. The double type is the one used by the Math class in C# so it is logical to convert everything into double as it is most likely that the acquired samples will be subjected to further mathematical processing whether for visualization or for packing in matrices. The following paragraph will explain the fixed-point signed fractional and signed integer numeric notations before proceeding with the actual class implementation.

Consider a 32-bit fixed-point format. This format can denote a signed integer or signed fractional number as shown in figures 3.1 and 3.2.

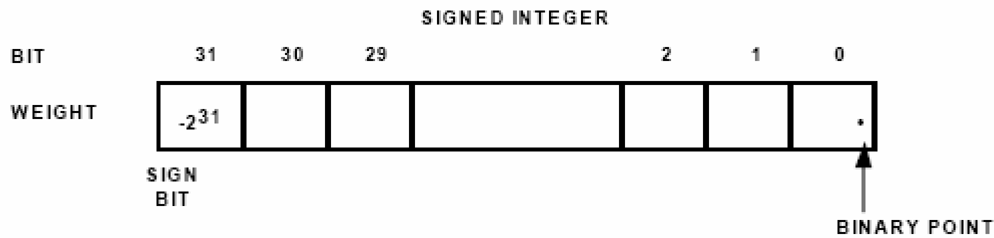

Figure 3.1: 32-bit fixed-point signed integer [20]

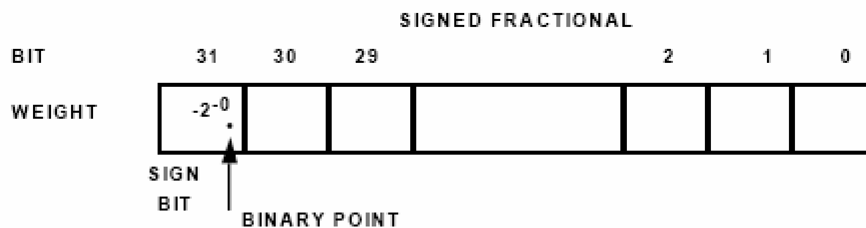

Figure 3.2: 32-bit fixed-point signed fractional [20]

These representations can be generalized to any number of bits n.

As illustrated in the figures, a signed integer format of n-bits can accommodate for a span of values $[-2^{n-1}, +2^{n-1} - 1]$ inclusive. One the other hand, a signed fractional number of n-bits can represents faithfully numbers ranging from -1 to almost +1.

As widely perceived, in high level programming languages such as C#, it is not possible to perform low-level logical operations such as shift and concatenate. This is partially true as it is possible to perform shifting on the `byte` type in C#. However, the result is automatically parsed into an `int` type which renders it unusable for further binary-level operations. In order to go around this problem, the string type is used as an intermediate type to which all numeric conversions will be parsed. This makes the conversion an easy and straightforward matter as all types in C# have a default method `ToString()` associated with them that facilitates this operation with zero overhead.

The `HexConversion` class consists of six static methods. Static methods are used to guarantee that no objects of the class are allowed to be instantiated. This ensures that modifications introduced to an object will affect the original object and not another replica in the memory. The six static methods are:

- `hexDigits` : returns a hex character in correspondence to a hex nybble passed in an integer argument.
- `ToHexString` : returns the hex string equivalent of an array of bytes. This method makes use of `hexDigits` and C# built-in binary manipulation operators; >> and `&`.
- `decimal2hex` : converts a decimal fixed-point fractional number `val` to its hex representation over n-bits by implementing the following equation: $Floor\left(2^n + val \times 2^{n-1}\right) \& \left(2^n - 1\right)$. If `val` is positive, the first left hand side of the equation will overflow yielding zero in the sign bit. "Anding" the result with ones discards bits beyond the nth digit. If `val` is negative, a weighted full resolution range will be subtracted from 2^n (effectively two's complementing `val`) yielding a one in the sign bit to indicate a negative number. Flooring the first stage of the equation gets rid of unwanted fractions in case the resultant value is not a whole number.
- `int2hex` : converts a signed integer to its signed hex representation over n-bits.
- `lookupHex` : returns the equivalent integer value of a passed hex character wrapped in double format in order to minimize further explicit parsing.
- `hex2integer` : returns an integer equivalent of the passed two's complement hexadecimal number. It is easier to comment on the actual code for this function rather than explaining it in words.

Listing 3.1

```
public static int hex2integer (string input, int n)
{
        int i, val;
        double j=0;
        for (i=0; i<input.Length; i++)
                j += (double)Math.Pow(16, input.Length-1-i)*lookupHex(input[i]);

        if ( j >= Math.Pow(2,n-1) )
                val = -(int)(Math.Pow(2,n) - j);
```

```
        else
              val = (int)j;

        return val;
}
```

The “for” loop simply decodes the hex number packed in the string input to its numerical equivalent. Then if the result is found to be greater than the maximum positive representable number, the result is complemented with respect to n-bits and negated.

- `hex2decimal` : returns the fractional decimal representation of the passed hex parameter. It is implemented exactly in the same way as `hex2integer` except that the resultant value needs to be normalized first as follows:

Listing 3.2

```
if ( j >= Math.Pow(2,n-1) )
      val = -(Math.Pow(2,n) - j)/Math.Pow(2,n-1);
else
      val = j/Math.Pow(2,n-1);

return val;
```

A typical utilization of this class within the programme is presented bellow:

Listing 3.3

```
double[] dbuffer = new double[buffer.Length/2];
for (int j=0; j < buffer.Length; j+=2)
{
      // Reference Operation :
      //dbuffer[j/2] = buffer[j] + buffer[j+1]*Math.Pow(2,8);
      byte[] intermediate = new byte[2];
      intermediate[0] = buffer[j+1];
      intermediate[1] = buffer[j];

      dbuffer[j/2] = HexConversion.hex2integer(
                        HexConversion.ToHexString(intermediate),
                        16
                        );
}
```

The code shows how a signed two’s complement sample present over two consecutive bytes can be converted to its integer number representation and stored in a double buffer location for further visualization at a later stage.

3.2.2 *Complex* structure:

One of the functionalities that the software offers is a real-time FFT computation for the incoming stream of samples. In order to perform FFT, first a complex structure was defined to realize complex numbers representation. It is worth pointing out that `Complex` is defined as a structure rather than class in order to avoid the overhead associated with calling a constructor for every single class allocation in an N point complex array when computing FFT recursively which might result in a slight degradation in the overall software performance. The implementation of the complex structure is straightforward. It consists of three public methods; `Complex`, `conjugate`, `ToString`. The latter overrides the default .NET `ToString` method. In addition, the structure overloads basic arithmetic operators such as +, -, and * for scalars and complex numbers.

3.2.3 *FFT* class:

A real-time FFT computation is performed on the host PC. It provides a means to assess the frequency content of the incoming signal. It runs on a different thread than the mainstream thread in a multithreaded application. The multithreaded nature of the software will be discussed in details later alongside some other OS concepts deployed in the software's architecture.

The class *`FFT_Trans`*, consists of three static methods `fft, ifft, and convolve`.

Firstly, the `fft` method computes the FFT of the complex array `X[]`, assuming its length is a power of 2 based on a recursive implementation of the radix 2 Cooley-Tukey FFT algorithm [21]. The recursive approach of the Cooley-Tukey FFT is very simple to code indeed as one neither has to keep track of the separation of constituent points of the butterfly in a given stage b_w nor the separation of points having the same weighting factor within a given stage b_f. However, the caveat is that an FFT recursive algorithm is more demanding in terms of memory resources. For example, a 1024 point FFT will require 10 recursions, one per butterfly stage, to yield the final result. (Had this been the case fifteen years earlier, one would have worried about this being inefficient way to implement FFT. Nevertheless, modern computers as so powerful that we no longer care much about efficiency and even tend to trade it for simplicity.) What is crucial is that while regression is in progress, a termination case has to ensure that the final recursive call converges to a value upon which all remaining calls

depend for them to evaluate and present a valid value to the parent call. In this case, the base condition is when the number of points is equal to one which corresponds to `y` being equal to `x`. Generally, the even and odd components of the DFT will be extracted whereby an FFT transform for each is called so that at a later stage they will be used with the proper weighting factor to reconstruct the final result.

Secondly, the `ifft` computes the inverse FFT of the complex vector `X[]`. Implicitly it calls the `fft` method. The only additional caution is that the complex array is conjugated both before and after the direct FFT call to account for the positive sign in the IDFT formula. Also the returned array `Y[]` is divided by the total number of points.

Finally, method `convolve` computes the convolution of two complex time-domain vectors `x` and `y` based on Fast Linear Convolution [22]. The two vectors are taken to the frequency domain where they are multiplied and the result is inverse-transformed to the time domain again.

3.3 USB-RELATED IMPLEMENTATIONS

In this category, various classes and aspects associated with USB are presented. The following discussion comprises two classes namely `RandomAccessBurst` and `Acquisition` in addition to low-level USB driver functions.

3.3.1 *RandomAccessBurst* class:

To start with, a class named `RandomAccessBurst` was implemented which maintains operations related to a USB-packet. In the Object Oriented Programming (OOP) philosophy, programmers are encouraged to wrap objects of the same level of abstraction or category into a separate class. Hence `RandomAccessBurst` class encapsulates all packet-related entities in one object that is easily and securely maintained through its properties and methods. In addition to some private data members, the class has two public data members as far as the functionality of the class is addressed, `burst` which is a double array of fixed size (here 512) to hold a packet, and `index_of_burst` which is an `int` type to keep track of the received packet's index. Moreover, the class utilizes a property `BurstIndex` to set and get the private

`index_of_burst` variable, and two public methods `SetBurst` and `GetBurst` to set and get the private `burst` array.

3.3.2 *Acquisition* class:

This class builds on top of the `RandomAccessBurst` class to provide file-processing services necessary for storing and retrieving `RandomAccessBurst` objects to/from files by means of internal private binary reader and writer data members. Public methods include: `OpenFile` which create/open file containing empty records, `GetBurst` which retrieve a `RandomAccessBurst` depending on its index, and `AddBurst` which add a burst to file at position determined by a parameter signifying burst number.

Typical working scenario that illustrates the usage of the above two classes is presented in listing 3.4:

Listing 3.4

```
acquisitionProxy = new Acquisition();
acquisitionProxy.OpenFile( fileName );
.
.
.
int newBurstIndex = Var.USB_Packet.BurstIndex + 1;
RandomAccessBurst bPacket = new RandomAccessBurst(newBurstIndex, dbuffer);
acquisitionProxy.AddBurst(bPacket, newBurstIndex);
```

3.3.3 Low-level USB functions:

The .NET framework does not provide implementations to support low-level I/O operations. Moreover, literature on this subject is extremely hard to find as Windows Operating System in its essence is not an open source OS as opposed to other operating systems such as Linux and UNIX. As a result, it is almost always the case that one can never find an open source implementations in support to Windows OS. This makes the development of windows applications with intimate relation to the hardware layer on which the OS operates be restricted and monopolized by big corporations rather than individuals.

Initially the plan with regards to the USB windows driver was to *marshal* the unmanaged C++ static library, which is freely available from Cypress, into the managed world to allow .NET applications to use the C++ DLL. Relying on my

computer engineering background, the risk of digging deep in such specific programming task seemed to be feasible at the time. After investing a great deal of time in an attempt to produce a wrapper code for marshalling the unmanaged DLL, it turned out that the task requires more than just a computer engineer. In fact a solid software engineering background coupled with dedication and time is inarguably needed especially that the task dictates a specialized expertise in .NET framework rather than general programming skills.

Choosing not to compromise other tasks in the project which are more relevant to the project's environment, a third party software was used to generate the Windows USB driver plus the low-level DLL file to interface with the USB device. After some experimentation, the DLL was easily incorporated in the software resulting in a dynamically adjustable USB driver which was crucial for investigating various USB transfer types as to suit the nature of application.

3.4 VISUALIZATION

The signal being acquired through USB is required to be plotted both in time domain and frequency domain after the application of FFT. To achieve this task, two classes were implemented; `SoftScope` and `SpectrumAnalyzer`. They provide the user with the ability to perform various tasks such as controlling the x-y axes scale, normalization factor, offset etc emulating the real operations of an oscilloscope or spectrum analyzer. Further to these operations, `SpectrumAnalyzer` also allow for choosing the display and operation settings including linear and dB scale, windowing type, and number of frequency bins. Both classes run in parallel to the parent application on separate threads allowing it to be free of the burden associated with constant monitoring and control. The real-time refreshing procedure is maintained via special mechanism that will be discussed later in the multithreading section of the OS concepts.

As for plotting waveforms, a freely available plotting library for .NET by Matt Howlett and Paolo Pierini is deployed [23]. The library handles all plotting-related operations from scaling to background colour. This is in line with the whole purpose behind OOP programming which is to provide classes to facilitate code reusability enabling programmers to be more productive concentrating on system-level

integration and development rather than starting from scratch in every single project. One additional consideration while choosing a suitable plotting library is the ability to support density pixel plots commonly needed in tomographic distribution images. The next figure depicts such a plot.

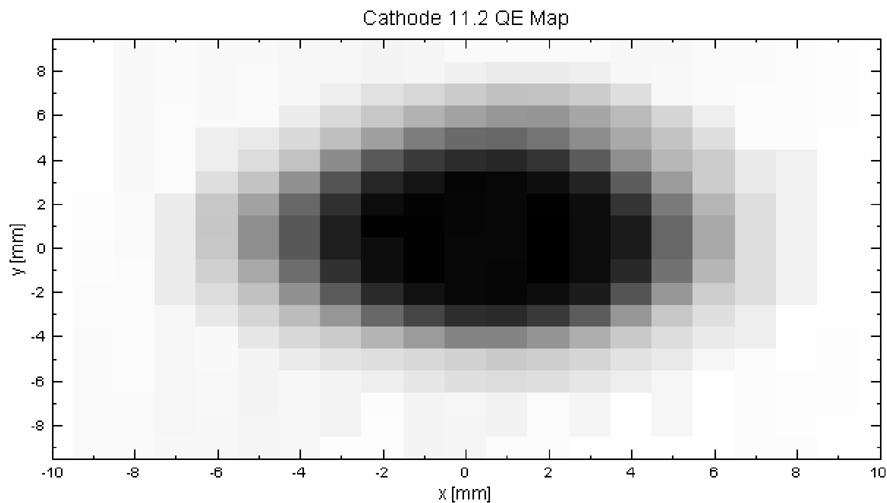

Figure 3.3: Density distribution plot

3.5 OPERATING SYSTEM CONCEPTS

At this point, in order to proceed with presenting the remaining features of the software design, an extremely important concept needs to be imported first from the Operating Systems world. Multithreading is one powerful feature around which many of the aspects of real-time systems revolve. Yet at the same time, multithreading is Pandora's Box, which once opened, a huge number of inconveniences can stem from thus gradually degrading the system and even eventually rendering it totally erroneous.

The following discussion will explain in great details this concept as it will be revisited again in the Detailed Hardware Design chapter. Although the discussion will be conducted in a software-orientated manner, the same concepts remain totally valid in an embedded hardware environment such as an FPGA or a DSP.

3.5.1 Theoretical background - Multithreading:

A thread is essentially a portion of a programme that can execute. Multithreading refers to the ability of an OS to support multiple threads of execution within a single process [24]. A process is a collection of one or more threads that can run simultaneously. Of course single processor PCs have only one processor which is capable of executing only one task at a time. However, the OS schedules and dispatches among processes and threads resulting in a concurrent effect when observed over a small interval of time in which the switching occurs. Historically the privilege of using multithreading was granted to the OS only. Now .NET framework grants this flexibility to users using any .NET language. Multithreading gives the programmer a greater control over the timing of application-related events. When the nature of the program does not require serializing tasks, multithreading becomes extremely useful. The following figure illustrates various scenarios for multithreading.

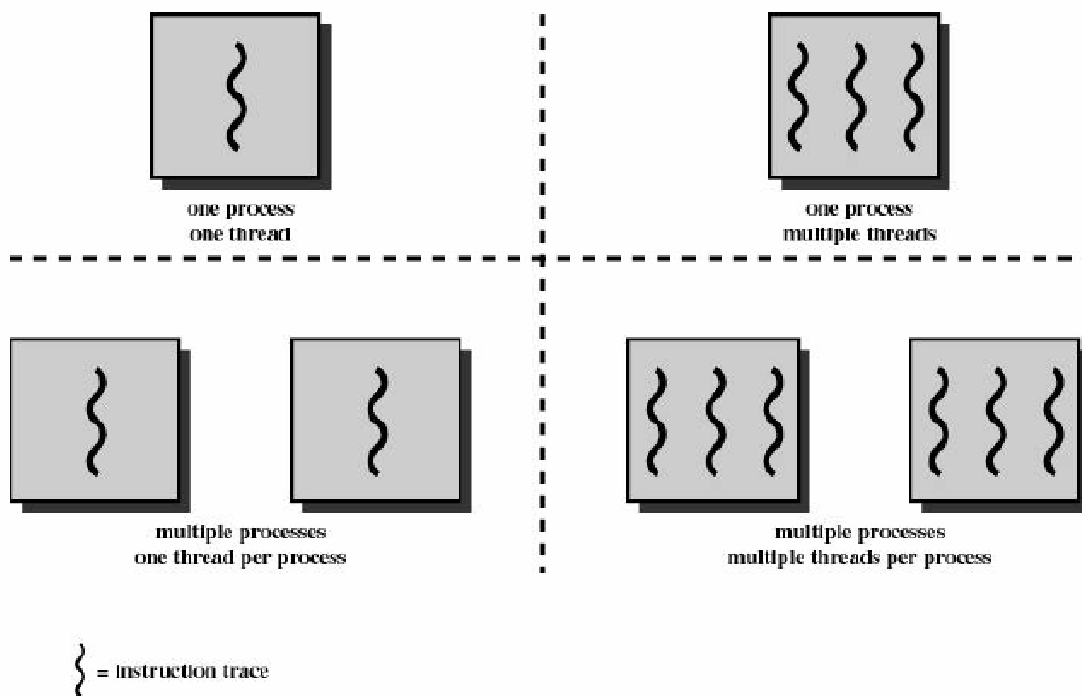

Figure 3.4: Threads & Processes [24]

The major benefit of multithreading is that a programmer can make use of a concurrent execution scheme without performance degradation because the use of threads does not require an explicit intervention from the OS's kernel in order to perform dispatching and monitoring unlike processes.

It was established that concurrency is a powerful concept that can improve the software architecture of a program leading to faster execution time and better processor utilization. However, the risks associated with concurrency are enormous. In order not to drift too far in explaining all sorts of difficulties arising from concurrency, the discussion will be restricted to the problem relevant to the project's nature. The main problem that needs attention is the scenario when two threads try to access a shared variable. This problem is referred to as the mutual exclusion problem. The solution has to ensure that only one process (or thread) is allowed to access the shared memory location at a time such that data integrity is maintained.

The proposed software architecture consists of a parent process which is the `DefaultInterface` class. Utilizing a user-friendly menu, an end-user creates a new session from the options menu. The session in turn creates/opens a file to which subsequent packet reception will be registered as `RandomAccessBurst` objects. Then a session configuration form enables the user to configure a read/write continuous/single operation to/from a chosen pipe. The operation will commence immediately. The user may choose to view the incoming signal by pressing the oscilloscope button. The oscilloscope thread operates transparently and independently of the parent process and may spawn the spectrum analyzer thread upon user's request.

3.5.2 Mutual exclusion solution adopted:

When the software is fully operational, one parent process `DefaultInterface` runs in parallel alongside two major child threads; `SoftScope` and `SpectrumAnalyzer`. As stated above, this parallelism must not result in multiple accesses to a shared memory resource. In order to accomplish mutual exclusion, all global variables are encapsulated within a class called `GlobalVariables`. Within `GlobalVariables`, private data members are protected by means of the `Monitor` class. `Monitor` class is part of C# `Threading` namespace which provides thread synchronization. Listing 3.5 describes how class `Monitor` can be used to protect shared variables.

Listing 3.5

```
public RandomAccessBurst USB_Packet
{
      get
```

```
        {
                // obtain lock on this object
                Monitor.Enter( this );

                // tell waiting thread (if there is one) to
                // become ready to execute (Started state)
                Monitor.Pulse( this );

                RandomAccessBurst RABCopy = this.usbPacket;

                // release lock on this object
                Monitor.Exit( this );

                return RABCopy;

        } // end get

        set
        {
                // acquire lock for this object
                Monitor.Enter( this );

                // set new value
                this.usbPacket = value;

                // tell waiting thread (if there is one) to
                // become ready to execute (Started state)
                Monitor.Pulse( this );

                // release lock on this object
                Monitor.Exit( this );

        } // end set
}
```

Method `Enter` is used to obtain lock on an object. Before releasing lock on the object using method `Exit`, method `Pulse` tells the thread that has been blocked (if any due to its attempt to access this object) to become ready to resume executing as this object is about to be released [25]. Attention needs to be drawn to an important consideration in the "get" property due to one special case. While a thread in the "get" property and about to release lock on object, another thread could be assigned the processor immediately after the monitor is released and before the return executes. In this case the first thread would receive the new value modified by the second thread. Therefore, copying the critical object first ensures that the first thread receives the original value and not the one which has just been updated by the second thread.

All critical variables within class `GlobalVariables` use this simple yet powerful class. .NET framework provides other alternatives for ensuring mutual exclusion such as class `Mutex`. In general, unless the complexity of the design dictates the

deployment of the more sophisticated approaches in remedy to a certain situation, it is always advisable to keep the design as simple as possible.

One more issue needs to be mentioned at last for the sake of completeness. When instantiating the new thread, its constructor receives the `GlobalVariables` object as a parameter which will be stored to a similar local instance originally set to reference a null. In OOP languages such as C# and JAVA, no explicit use of pointers is performed by programmers. However, one must bear in mind that setting an uninstantiated instance of an object to a previously instantiated one (using the keyword `new`) is equivalent to copying a pointer (reference) to an object into the un-instantiated copy. Therefore, effectively the new thread's constructor stores a reference to the original object in a suitable local uninitialized object of the same type without allocating new memory space for it.

3.5.3 Plot refreshing mechanism:

In the following paragraph, the problem of refreshing a windows form will be addressed. The `oscilloThread` thread will be utilized as the object of discussion. However, the same result applies to the `spectrThread` in exactly the same manner.

Windows forms suffer from a legacy inherent limitation requiring that methods called from outside the control's creation thread be marshalled to (executed on) the control's creation thread [26]. The nature of refreshing a plot within a windows form suggests that refreshing be scheduled on a regular basis to deliver a steady rate. This can be handled by a background thread that allows the interface to remain responsive while refreshing is being performed in the background without having to poll on the event. Still due to the above mentioned limitation, the outside call needs to be marshalled on the `oscilloThread` thread.

To accomplish this refreshing mechanism, first the method `scopeRefresh` was defined. This method is called from the background thread. It is called through a `BeginInvoke` call so that it is always "marshalled" to the thread that owns the `plotSurface` control. In turn `BeginInvoke` requires a delegate as an argument. In .NET framework, a delegate is equivalent to a function pointer. The following piece of code shows various entities defined within `SoftScope` class.

Listing 3.6

```
//     Background Thread
private delegate void ScopeRefreshDelegate();

private Thread refreshThread;
.
.
private ScopeRefreshDelegate scopeRefreshDelegate;
.
.
scopeRefreshDelegate = new ScopeRefreshDelegate(scopeRefresh);
```

Upon selecting the “Start Refreshing” option in the View menu list, the following code will be executed:

Listing 3.7

```
refreshThread = new Thread(new ThreadStart(ThreadProcedure));
refreshing = true;
refreshThread.Start();
```

Now the `refreshThread` is instantiated and started. Listing 3.8 shows the actual thread procedure. This method runs in a background thread to refresh the `plotSurface`.

Listing 3.8

```
private void ThreadProcedure()
{
      while (true)
      {
            try
            {
                  // Perform a BeginInvoke call to the list box
                  // in order to marshal to the correct thread.
                  // Begin the cross-thread call.
                  IAsyncResult r = BeginInvoke(scopeRefreshDelegate);
            }
            finally
            {
                  // You are done with the refresh

                  // Raise an event that notifies the user that
                  // the refresh has terminated.
                  // You do not have to do this through a
                  // marshaled call, but
                  // marshaling is recommended for the
                  // following reason:
                  // Users of this control do not know that it is
                  // multithreaded, so they expect its events to
                  // come back on the same thread as the control.
                  BeginInvoke(onRefreshComplete, new object[] {this,
                  EventArgs.Empty});
            }
            Thread.Sleep(100);
      }
}
```

The comments in listing 3.8 describe the overall operation of the thread. One last note needs to be pointed out with regards to the refresh rate. The refresh rate of the plot is determined by the number of milliseconds passed as argument in the method `Sleep`. In this case, 100 ms results in 10 Hz refreshing rate. `Sleep` method instructs the thread to give up its time slice and stop execution for a certain number of milliseconds.

In conclusion, the source tree of the host application is shown in figure 3.5.

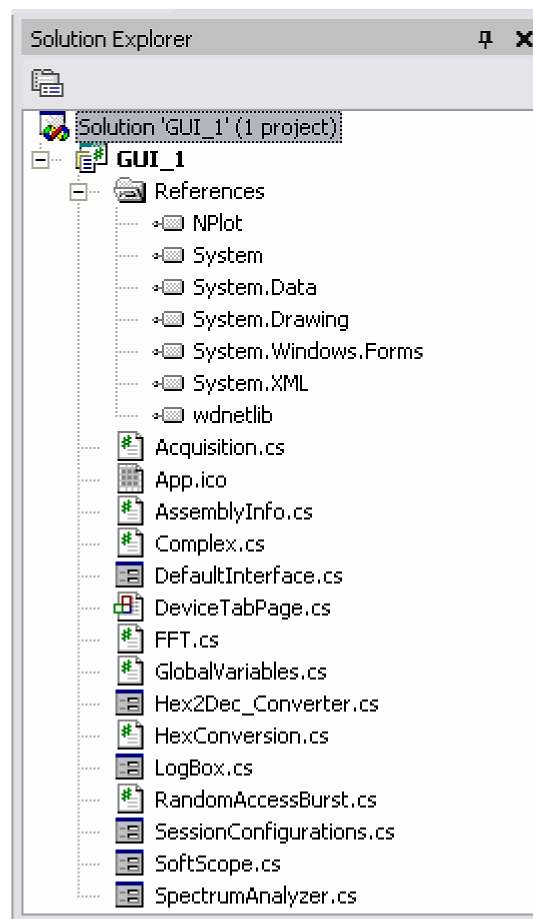

Figure 3.5: Host software source tree

Chapter 4

Detailed Digital Hardware Design

4.1 INTRODUCTION

In this chapter, important digital hardware design considerations will be tackled. Rather than explaining the code, an emphasis on the functional behaviour of various design entities, whether procedures or modules, will be placed. The reader is to be directed to the VHDL appendix for the commented detailed code.

4.2 HARDWARE CONCURRENCY

4.2.1 Identifying the problem

Following on the multithreading discussion presented in chapter 3, hardware threads or processes should also ensure mutual exclusion. In VHDL, a behavioural architecture contains one or more processes running in parallel. However, no multi-source signals are allowed. A multi-source signal is one that can be written to (modified) in two or more distinct processes. This synthesis constraint guarantees internal signals integrity. One the other hand, depending on the nature of the system, restricting modify-accesses to one hardware process may result in a sequential execution scheme. Therefore, arises the question whether concurrency should be lost in the favour of meeting synthesis constraints, as in such a case, the adoption of FPGAs would become questionable, with the end result resembling that of an ISA computing solution.

In order to meet synthesis constraints and still have as many parallel processes as possible accessing a shared resource, the following scheme depicted in figure 4.1 is applied. The shared resource receives only one modifying signal. A multiplexing process selects which of many connected signals to route to the shared resource in accordance to a selecting input signal. The inferred multiplexer process is an

asynchronous one with a sensitivity list consisting of all input signals including the select signal. Again the select signal can not be modified but in one process. This process is an arbiter process that listens to a combination of requests from all relevant processes to supply the select signal accordingly. The arbiter process has to account for all possible scenarios including the prohibited ones in order not to result in any unexpected behaviour.

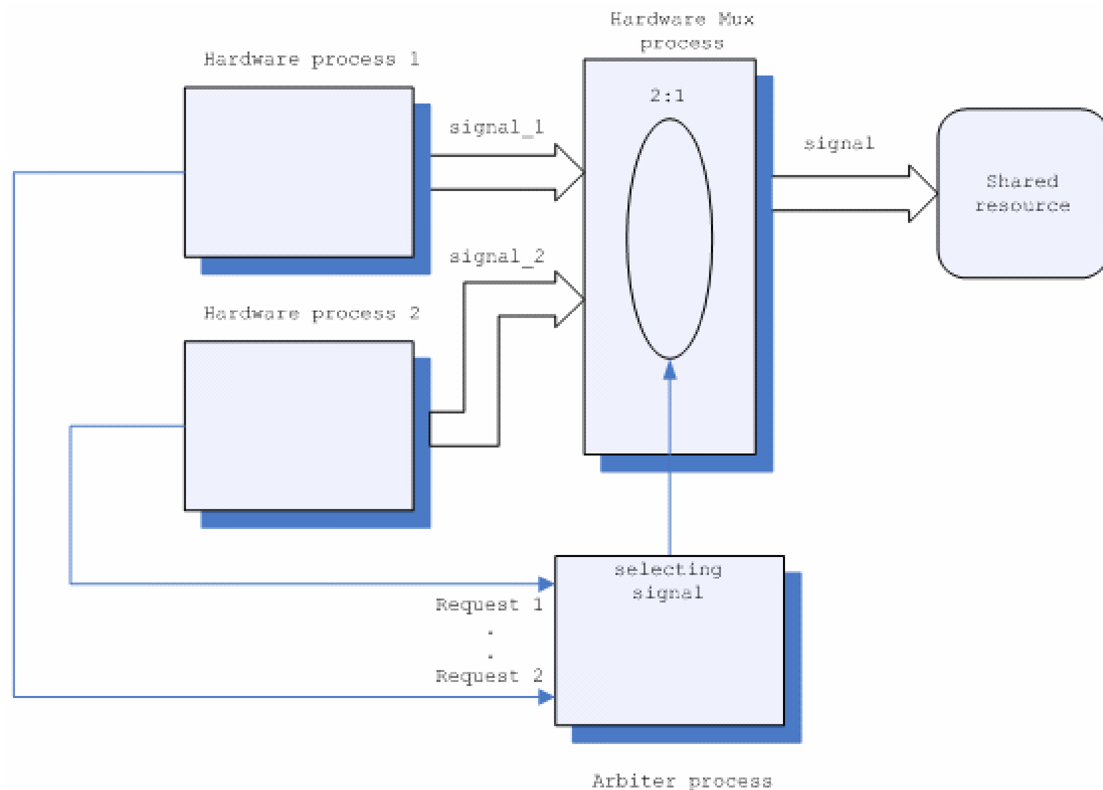

Figure 4.1: Abstract solution for hardware mutual exclusion

Another issue to be kept under consideration is that in VHDL, a shared resource can be as tiny as a mere signal, or as large as a fully functional module such as an external bus driver whose use is restricted to one process at a time.

4.2.2 Proposed solution

For assessing the PC USB connectivity, Xilinx's core generator was used to produce a digitally-synthesized sine waveform which is sampled at relatively high frequency. The sampling clock is the system's clock divided by eight $25MHz/8 = 3.125MHz$ leaving a good temporal margin (8 clock cycles) in which to manipulate the sample. Each sample is represented over 16-bit two's complement integer notation. Thus in

order to maintain a continuous stream of uninterruptible packets, sample packing and transmission have to occur simultaneously. This is achieved by implementing a double buffering scheme in which simultaneous read and write operate seamlessly.

The Spartan-3 component library has a built-in dual-port RAM that enables reads/writes operations from/to the same block RAM. Furthermore, one can choose among various configurations for partitioning the total available bits. Thus in principle while filling the upper half of the buffer, the lower half can be read out as to accomplish a smooth read/write working scheme. Nevertheless, the use of two distinct RAM blocks was necessary simply because each block can accommodate for up to 1K 18-bit words and the software design requires that a constant 512 samples be sent at a time.

In VHDL code, two processes were implemented that operate interchangeably in a loop to supply the SX2's FIFO with packets at a constant rate. This constant rate along with the endpoint bandwidth will be assessed later in chapter 5. Process `sine_packing` fills one of the RAM buffers with samples whenever instructed to by the process which will consume this buffer. In turn, `sine_packing` uses services from `new_sample_rdy` process which operates at the sampling clock to flag the readiness of new sample. This was necessary as in VHDL a process is only allowed one clock to operate on which conforms to what is expected from a transparent hardware modelling language. The transmitting process, in this case the mainstream process, detects the availability of a full buffer by means of synchronization flags and commences the transfer immediately. The inter-process synchronization mechanism will be discussed in the next section.

Figure 4.2 is the metaphoric hardware representation of the double buffering scheme.

Two 1:2 demultiplexers alongside a 2:1 multiplexer are used for the read enable, write enable, and read data respectively. This hides which buffer is being currently accessed from the producing and consuming processes. Because write is always performed by `sine_packing` process and read is always carried out by the mainstream process no further multiplexing is needed although it is possible. Moreover, no arbiter process is implemented as the selecting signal is always modified by the mainstream process which decides when to start/stop `sine_packing` process by means of synchronization flags.

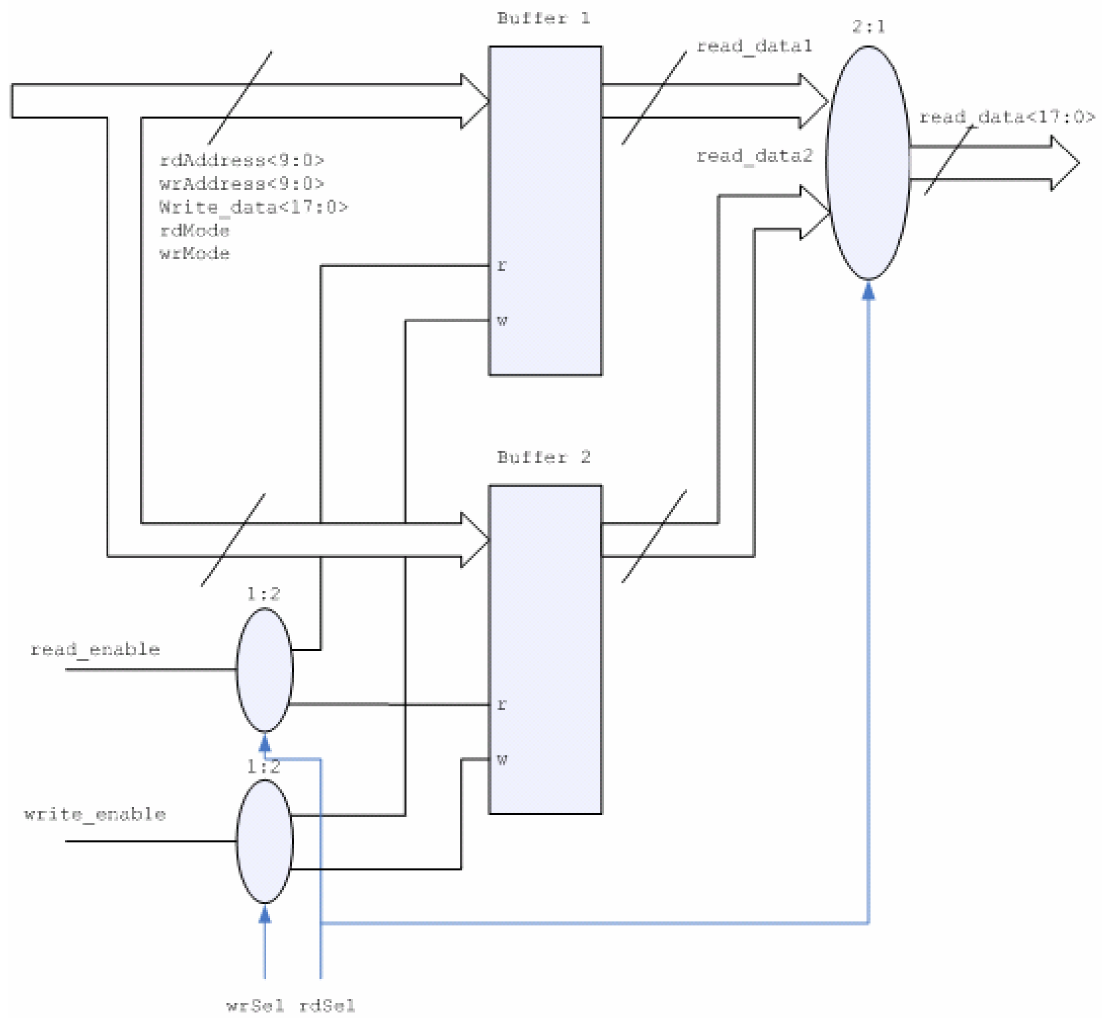

Figure 4.2: Double buffering scheme

Table 4.1 shows the truth table of the logic operations set by the select signal.

Table 4.1: Select signal truth table

buffSel<1:0>		OPERATION
wrSel	rdSel	
0	0	**Prohibited**
0	1	Write buffer 1 Read buffer 2
1	0	Read buffer 1 Write buffer 2
1	1	**Prohibited**

It is worth pointing out that the multiplexing process is inferred rather than instantiated, i.e. the logical operation is achieved in an implicit manner using a case statement which account for both the multiplexing and demultiplexing actions.

Listing 4.1 shows the VHDL implementation of the above schematic.

Listing 4.1

```
myDemux : process (write_enable, read_enable, rwSel, read_data1, read_data2)
begin
      case rwSel is
            when "00" =>      -- NOT ALLOWED, however in case
                  write_enable1 <= '0';
                  write_enable2 <= '0';
                  read_enable1 <= '0';
                  read_enable2 <= '0';

                  read_data   <= (others => '0');
            when "01" =>
                  write_enable1 <= write_enable;
                  write_enable2 <= '0';
                  read_enable1 <= '0';
                  read_enable2 <= read_enable;

                  read_data   <= read_data2;
            when "10" =>
                  write_enable1 <= '0';
                  write_enable2 <= write_enable;
                  read_enable1 <= read_enable;
                  read_enable2 <= '0';

                  read_data   <= read_data1;
            when "11" =>      -- NOT ALLOWED, however in case
                  write_enable1 <= '0';
                  write_enable2 <= '0';
                  read_enable1 <= '0';
                  read_enable2 <= '0';

                  read_data   <= (others => '0');
            when others => NULL;
      end case;
end process;
```

4.3 INTER-PROCESS SYNCHRONIZATION

Inter-process synchronization refers to the mechanism by which a timely order is maintained among a group of processes. The presentation of this section was intentionally delayed after the introduction of hardware concurrency section so that the above discussion justifies the need for such arrangement.

Inter-process synchronization is realized via declaring a custom record type `InterprocessSync`. In turn, `InterprocessSync` deploys another custom record type `status_flag`. Listing 4.2 shows the declaration of both types.

Listing 4.2

```
      type status_flag is (INPROGRESS, FINISHED);

  --  This record type maintain synchronization between parent & child
  --  processes
      type InterprocessSync is           -- InterprocessSynchronization
                                         -- Record type
      record
            syncFlg     : std_logic;     -- Synchronization Flag
                                         -- Modify Authority: parent
                                         -- process
            status      : status_flag;   -- Status Flag, Modify Authority:
                                         -- child process
      end record;
```

The first type `syncFlg` can be thought of as the asynchronous resent of the child process. It is solely modified by the parent process. The second type is meant to be modified by the child process upon the completion of the required operation. The parent process can poll on this flag occasionally or constantly to test whether a certain task has finished. For a successful synthesis of state machines, the Xilinx XST synthesis tools demand that state machines be written in a manner that conforms to the predefined VHDL language templates; otherwise, the code becomes unsynthesizable despite its logical synthetical correctness. For this reason, status flags have to be polled on rather than included in the sensitivity list of a process.

Listing 4.3 shows a typical scenario for using this synchronization mechanism.

Listing 4.3

```
      when PACK_IN_BUFFER =>

            case indexer is
                  when 0 =>
                        if (buffAddress = "1000000000") then
                              indexer := 4;
                        else  -- a packet has been stored
                              smplRdy.syncFlg <= '1';
                              increment(indexer);
                        end if;

                  when 1 =>
                        if (smplRdy.status = FINISHED) then
                              smplRdy.syncFlg <= '0';
                              increment(indexer);
                        end if;
.
.
```

```
.
new_sample_rdy : process (smplRdy.syncFlg, clk_sampling)
begin

       if (smplRdy.syncFlg = '0') then
              smplRdy.status <= INPROGRESS;
       elsif (clk_sampling = '1' and clk_sampling'EVENT) then
              smplRdy.status <= FINISHED;
       end if;

end process;
```

Process `new_sample_rdy` simply detects an active sampling clock edge. In more complex processes, a state for halting the sequential execution has to be inserted. This is important in order to ensure that while the parent process has not yet acknowledged the FINISHED flag by deasserting its sync flag, the child process is kept trapped in an idle state.

Figure 4.3 presents a graphical illustration of the inter-process synchronization scheme explained above applied on the `dsp_packing` process.

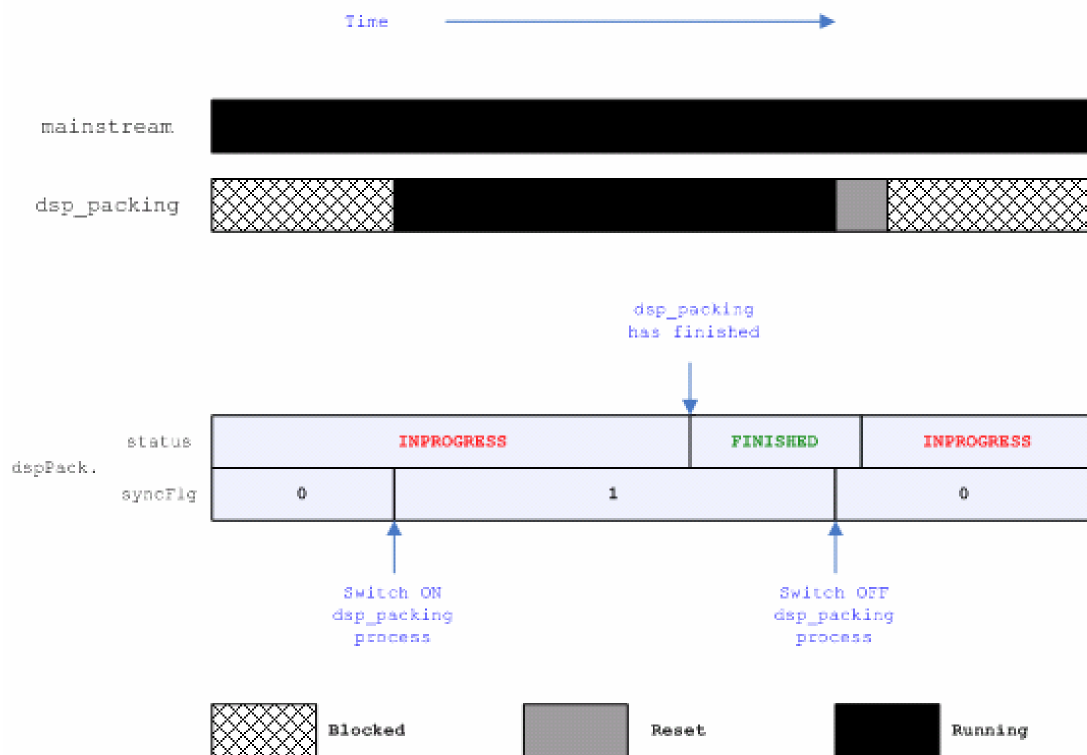

Figure 4.3: Inter-process synchronization illustration chart

4.4 VHDL CODE MODULAR ARCHITECTURE

In order to converge to the final solution in a systematic manner, it was necessary to adopt a top-down modular approach that allows for formality, segregation, generality, and incrementality [27]. Moreover, the system should exhibit high cohesion and low coupling as set by the general guidelines for embedded systems.

As a result, the digital design architecture consists of the following modules:

- `Main` Module: At the very top of the design hierarchy, it is in this module that all other modules are instantiated and linked (though processes) to deliver the system's final desired functionality.

- `bus_driver` Module: Although this module does not include much logic operations, it is supplemented by a set of functions defined in package `SX2_Utilities` as to implement SX2 USB device timing diagrams. In essence, it communicates external signals to/from internal peers after a possible application of an intermediate conditioning logic. The partitioning of the bus into external and internal falls under the general design practices that facilitate the functional modularity of the system. The internal bus always has an active high polarity whereas the external one can be dynamically tuned with respect to the input `active_s` signal signifying the desired active state. A simple xnor gate implements this truth table. Also this module controls the state of the I/O buffer according to the current operation being performed.

  Originally, during the work on this project, this module generated the synchronous state machine for read/write operations utilizing internal signals that instruct the module when to do so. Later, it turned out that the device starts up in the asynchronous mode and thus implementing asynchronous accesses was necessary for the initial configuration of the chip. Then realizing that configuration is only done once and that a 16-bit FIFO access alleviates the need for a synchronous time-efficient one, the synchronous interface was substituted with the asynchronous one altogether. Furthermore, the asynchronous interface has the advantage of being more EMC friendly due to the absence of the interface clock which is always desirable.

- `RAM_Buffer` Module: Equips a block of dual-port RAM with necessary logic to implement an architecture which is capable of operating in either FIFO mode or addressable RAM mode.

- `clkdivider` Module: is a modification of the Xilinx's Digital Clock Manager (DCM) library component that provides the system's synchronizing clock, other divided versions, and a doubled version. The divided versions are used to sample the diagnostic sine waveform at lower rates than that of the system's clock which are $\frac{clk}{2}$, $\frac{clk}{4}$, $\frac{clk}{8}$, $\frac{clk}{10}$, and $\frac{clk}{16}$. The twice as much version is used in the `dspPack` process to provide more clock resolution for manipulating the incoming/outgoing asynchronous data.

- `sinewaveform` Module: supplied by Xilinx's core generator as a means of assessing the maximum achievable USB bandwidth and demonstrating the co-designed GUI software functionalities.

- `DSP_Driver` Module: This module is an asynchronous state machine that is controlled solely by the DSP's strobing signals.

- `synchronizer` Module: This module synchronizes the signals outputted by `DSP_Driver` to the internal clock in `Main`. The synchronization mechanism will be discussed later in the DSP-related Hardware Design chapter.

4.4.1 A closer look at `SX2_Utilities` package

`SX2_Utilities` contains type definitions and constants used throughout the code. For instance, both USB descriptor and internal register definitions are implemented as arrays of constant values. While `mySX2Descr` is a 1D array of 148 values detailing various USB enumeration fields (such as VID, PID, and Endpoints configurations), `sx2RegsDef` is a 2D array whose first column corresponds to the indices of internal registers and second column corresponds to their desired bit patterns. VHDL automatically allocates ROM blocks for these arrays without explicit instantiation and initialization from the designer. Furthermore, `SX2_Utilities` contains two classes of functions and procedures; simple utility functions and SX2 timing procedures which are meant to be called from within a process on active clock edges until completion. In addition, by passing current and next state signals as arguments to a procedure, this

procedure can alter the current state of the sub FSM whenever desired once completed.

Utility functions and procedures include:

- `strobe`
- `de_strobe`
- `toggle`
- `next_step`
- `reset_step`
- `increment`
- `reset_index`

SX2 procedures are:

- `single_async_read`
- `single_async_write`
- `sx2ReadRegAsync`
- `sx2WriteRegAsync`
- `interrupt_status_read`
- `latch_addr_asyn_WR`
- `signle_fifo_async_read`
- `signle_fifo_async_write`
- `end_packet_asyn`
- `waiting_loop`

It is beyond the scope of this chapter to describe the implementation of every single function. Nevertheless, the state machine for `sx2WriteRegAsync` will be detailed as a sample to illustrate the general concepts deployed in the implementation of all procedures.

Figure 4.4 shows the state machine for `sx2WriteRegAsync` procedure. The syntax of the flowchart is meant to convey the algorithmic behaviour of the procedure independently of the actual VHDL code. For instance, `wait(SETUP_TIME)` signifies

that addr is allowed a window of SETUP_TIME in which to settle before asserting signal wr. In the VHDL code, succession of states containing the statement next_step(dispatcher) are inserted to generate the necessary timing. That is, the clock resolution determines how flexible the resultant timing can be. The manufacturer specifies minimum timing parameters for each asynchronous operation. Ultimately those parameters are functions of many physical properties such as temperature, fanout, and drive current. Thus it is advisable that the designer leaves a relaxed margin accounting for possible variations due to the mentioned factors.

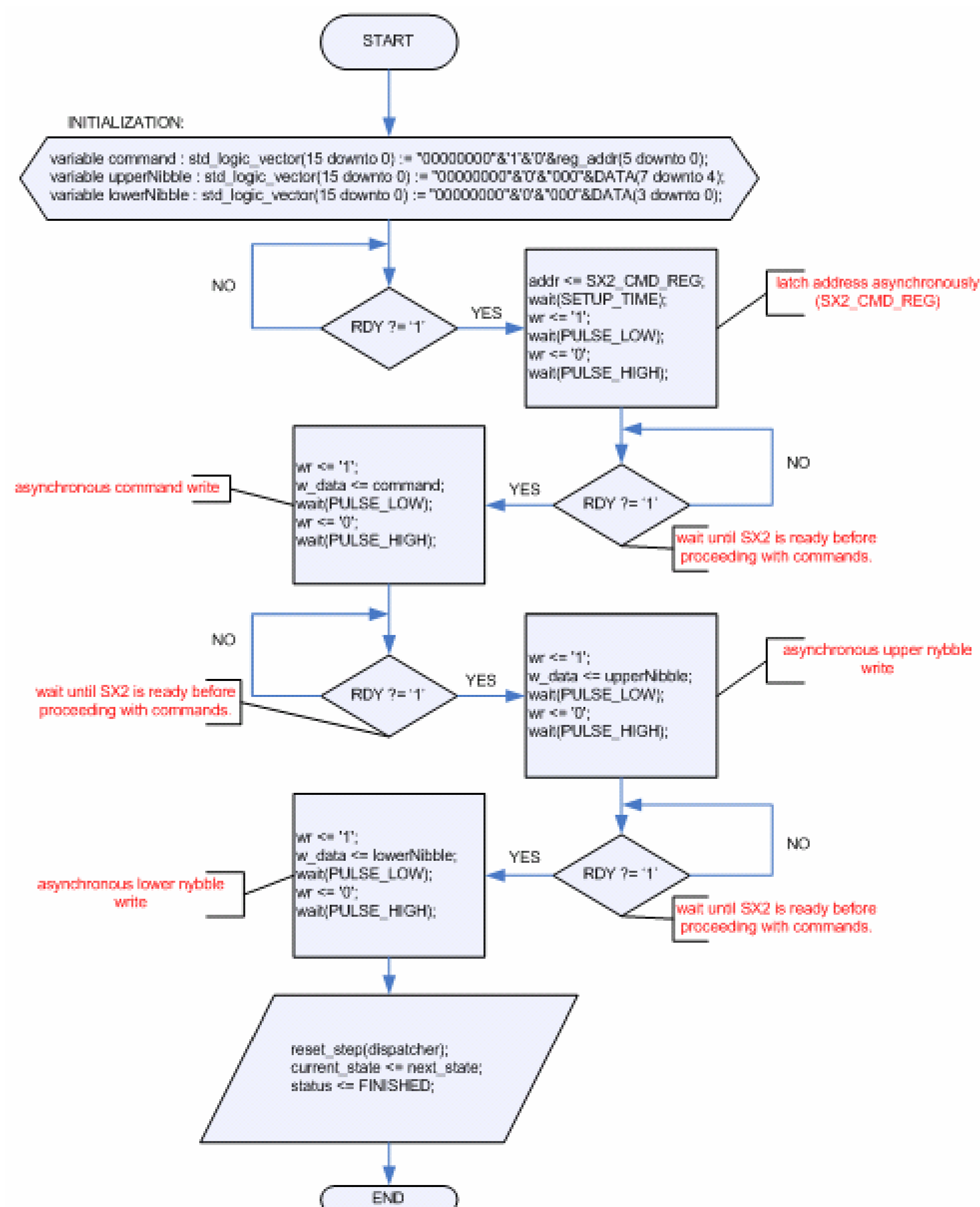

Figure 4.4: sx2WriteRegAsync state machine

Listing 4.4 shows roughly how timing parameters are being generated.

Listing 4.4

```
case dispatcher is
      .
      .
      when 10 =>
            next_step(dispatcher);  --    \
                                    --     \
      when 11 =>                    --     / 2*40 ns = 80 ns > 50 ns
            next_step(dispatcher);  --    /
```

As shown in the flowchart, the procedure checks and polls if necessary on the READY flag after each operation before proceeding to the next. Towards the end, the dispatcher is reset, the current state is assigned the next state, and the status flag is assigned the finished state. The finished flag is the terminating condition upon which the synchronous calling program will stop evaluating the procedure on active clock edges. Listing 4.5 demonstrates a typical situation for calling a procedure from within the mainstream process.

Listing 4.5

```
when CONFIG =>
      case index is
            when 0 =>         -- Write register
                  val := sx2RegsDef(i,1);
                  if (val = X"FE") then
                        status <= INPROGRESS;
                        current_state <= SETDESCR;
                        next_state <= SETDESCR;
                        i := 0;
                        index := 0;
                        LED1 <= '1';
                  elsif (status = FINISHED) then
                        status <= INPROGRESS;
                        i := i + 1;
                        index := 1;
                  else
                        sx2WriteRegAsync(dispatcher, next_state, current_state,
                                                status,
                                                wr, RDY, addr, w_data,
                                                sx2RegsDef(i,0), val
                                                );
                  end if;
```

The procedure is called successively until one of the conditions of the if-statement is met. The programmer may or may not wish to alter the current execution state according to the combination chosen for `current_state` and `next_state`. It is extremely important that the nesting of the if-statements ensure that worst case output propagation can occur within the system's clock period. For this reason, a relatively slow clock (25 MHz) was chosen for the mainstream process responsible for

initialization and packets sending which are the operations that make use of the `bus_driver` module. While it was necessary to use faster clock of more time resolution when dealing with external asynchronous signals coming out of the `dsp_driver`.

Finally, the module view of the developed design is presented in figure 4.5. The figure shows that the use VHDL test benches was vital in the theoretical behavioural model simulation and in the post-place & route model simulation. As a simulation means, VHDL acquaints the designer with additional powerful concepts such as inertial and transport delays which emulate the realistic behaviour of hardware.

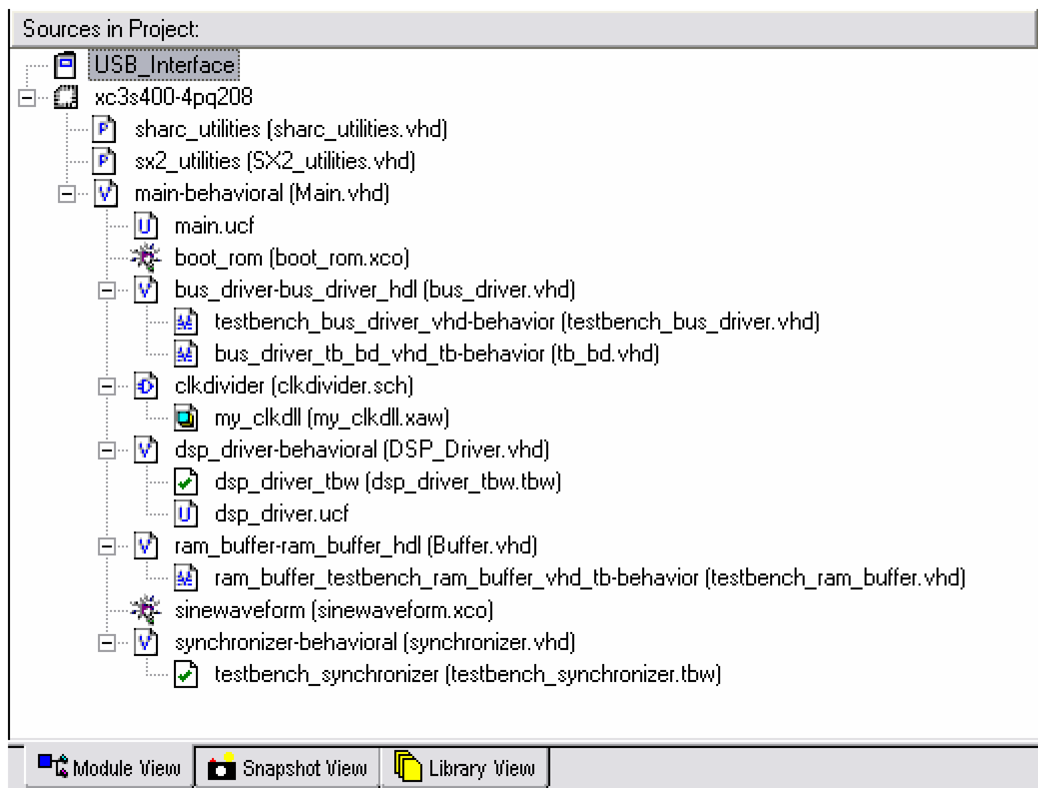

Figure 4.5: Design module view.

Chapter 5

DSP-Related Digital Hardware Design

Out of the various implemented VHDL modules, two are directly involved with the outside asynchronous DSP signals. These are `dsp_driver` and `synchronizer` modules. `dsp_driver` interfaces to the DSP's parallel port strobe signals, while `synchronizer` makes the `dsp_driver` signals align with the internal clock of the system. In the coming two sections, both modules will be investigated thoroughly revealing all design aspects that had to be met.

5.1 DSP INTERFACE

5.1.1 Introduction

The ADSP-21262 is a powerful 32-bit DSP that is ideal for many applications including the imaging problem [20]. Its architecture combines a powerful 400 MMACS or 800 MFLOPS processing core with high performance dedicated buses for Program Memory (PM), Data Memory (DM), and I/O. This enables multiple accesses to be issued in a single instruction. To be more precise, the DSP can access two data operands from memory blocks, fetch an instruction from cache, and perform a DMA transfer in every instruction. Furthermore, the DSP has 22 DMA channels that support a wide variety of peripherals including one parallel port, serial ports, SPI ports (Serial Peripheral Interface), and input data ports [28]. Thus, in order to make use of the Direct Memory Access feature of the DSP, one of the mentioned peripherals has to be exploited. In a DMA-driven transfer, the processing core is only involved at the beginning and end of the transfer and is free to perform other tasks during data transfer. In other words, a DMA-driven access requires no intervention from the processing core, resulting in transparent access to the DSP's on-chip memory with maximum processor utilization. In the project's context, processor utilization is defined as the effective instructions (whether computational or control) that the DSP

executes while performing the intended algorithm as opposed to those performed by the DSP to transfer data or to interact with events originating from the outside world. In mathematical sense, processor utilization can be described as follows:

$$utilization = \frac{effective}{overhead + effective} \times 100\%$$

As far as the throughput is concerned, the parallel port is the most promising peripheral to interface with. When interfacing to intelligent or memory-mapped peripherals, the parallel port can accommodate for up to 132 Mega bytes per second. Clearly, such a raw high bit stream of 1.056 Gbps is incommunicable to PC via USB2.0. Nevertheless, the availability of such high bandwidth link between the FPGA and DSP adds to the versatility of the system and allows for more sophisticated exploitation of the overall platform. For instance, if the DSP was thought of as a centralized floating-point processing unit, the presence of such a high throughput port could facilitate a back-and-forth operation upon requests from a master FPGA. The system being developed at the moment presumes that operations are being solely initiated and controlled by the DSP as a true master. This needs not always be the case, as it can waste precious processing utilization. At the far end of the system, specifically, in the acquisition nodes, FPGAs have been so far restricted to preliminary conditioning DSP operations in a complementary fashion, whereas the master DSP performed the rest of the tasks including algorithm-control ones. However, emanating from the nature of the problem being tackled, assigning control and interface tasks to the FPGA and highly demanding processing tasks on the floating-point DSP may improve the design complexity limit for which the underlying platform can accommodate e.g. hardware implementation of neural network [16].

From a first glimpse, it may seem that there is a caveat associated with the use of the parallel port. Due to its very nature, it is dominantly being controlled by the DSP. Contemplating this for a second, the solution may seem quite trivial. One can associate the read/write operations with interrupts upon which the DSP commences reading from or writing to the FPGA. This brings back balance to the proposed master-slave relation between the FPGA and DSP respectively. In a single task system, the relation between the FPGA and DSP is of little importance as the DSP is always expected to perform a predefined set of tasks in the fastest possible manner.

On the other hand, when looked at as a floating-point localized processing unit in the platform, the DSP's task can be very much dynamic. An example can be a non-linear adaptive algorithm that is being controlled by the FPGA with zero utilization overhead. The FPGA can request different operations from the DSP depending on the current situation while convergence is in progress. Of course, there is a tiny overhead with regards to implementing a protocol that facilitates various scenarios. However, with such good degree of coupling available between the reconfigurable fabric and the CPU [29], such issue can not pause serious bottleneck problems in this working scheme. Moreover, from the DSP's perspective, the algorithmic overhead accompanying such a scheme is minimal as decoding a readily available command is faster than going through the chain of deduction which has led to the issuing of this command in the first place. In very simple words, if the FPGA is allowed to do what it is good at (customized parallelism) and the DSP is guaranteed a fully operational pipeline (effective instructions), the overall platform can be tuned to approach even the highest computationally demanding class of applications.

5.1.2 Parallel port description

Before presenting the asynchronous VHDL design developed to interface with the DSP's parallel port, the latter needs to be described first. Figure 5.1 shows a block diagram of the parallel port.

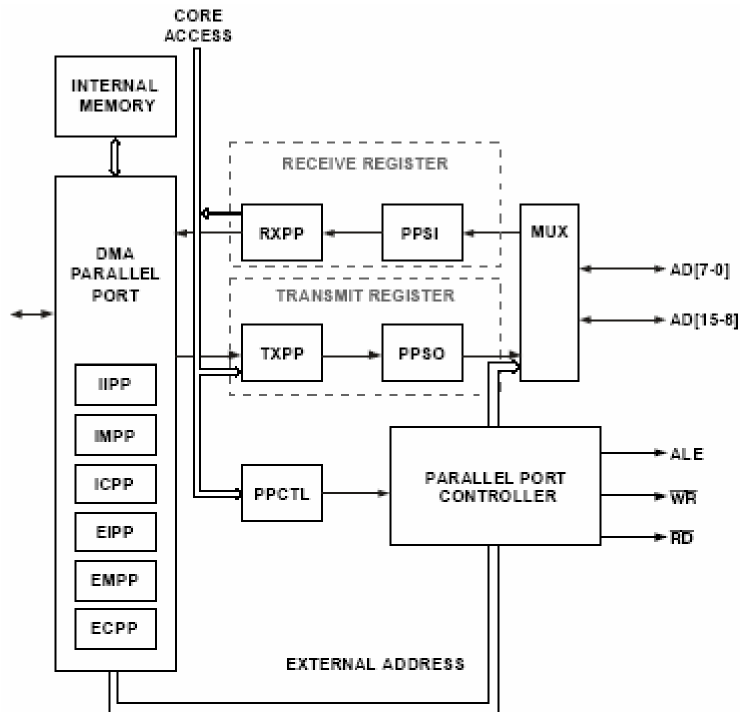

Figure 5.1: Parallel port block diagram [28]

The parallel port utilizes three signals and a 16-bit bus.

- Address/Data bus (`AD<15:0>`)
- Read strobe ($\overline{RD}$)
- Write strobe($\overline{WR}$)
- Address latch enable (ALE)

The use of these pins will become apparent while explaining the VHDL code.

5.1.3 *DSP_Driver* module

`DSP_Driver` is the module responsible for interfacing with the DSP's parallel port pins. This module consists of seven concurrent processes five of which are responsible for the asynchronous detection of ALE, $\overline{RD}$, and $\overline{WR}$. Processes `WAIT_ALE_proc`, `WAIT_WR_Ris_proc`, `WAIT_WR_Fal_proc`, and `WAIT_RD_Ris_proc` are edge-sensitive while `WAIT_RD_Fal_proc` is level-sensitive. This arrangement is important because of the nature imposed by the 8-bit mode interface. The 8-bit read interface was necessary to be implemented in order to successfully boot the DSP from the FPGA. While the 8-bit write interface was not implemented because it simply

offers no advantage over the 16-bit mode, although modifying the current design to include the 8-bit write mode is straightforward. Fundamentally, the parallel port boot mode uses the 8-bit interface in order to be able to address up to $2^{24} = 16\,\text{MB}$ of external memory space. The DSP driver port is segregated logically into external and internal. The external portion interfaces to the parallel port's signals plus a DSP enable pin by which to activate the parallel port operations. The DSPEN pin is used to instruct the DSP to start transmission/reception. The internal portion or the port includes:

- `ioe_AD`: controls the input-output tri-state buffer for bidirectional AD bus instantiated in `Main`. Because of synthesis purposes, Xilinx design tools requires that buffers be instantiated only in the top module of the design hierarchy. `Ioe(0)` controls the lower tri-state buffer byte and `ioe_AD(1)` controls the upper one. Again this was done in order to meet the 8-bit mode timing specifications. Since two separate processes (`WAIT_RD_Ris_proc` and `WAIT_RD_Fal_proc`) are required to modify `ioe_AD` in accordance to the mode of operation (8-bit/16-bit), it is necessary to have process `ioe_AD_mux_proc` multiplexing between `ioe_AD_Ris` and `ioe_AD_Fal` subject to whether `RD_bar` is high or low.

- `dsp_commence_syncFlg`: can be thought of as the asynchronous reset of the five processes to be manipulated by the process which will make use of this module.

- `dsp_status`: is implemented using a concurrent VHDL signal assignment. For the 16-bit read from DSP operation (wait on $\overline{WR}$ with ALE cycle present), it is analogous to the FINISHED flag whose functionality was explained in the previous chapter. When ALE cycle is only sent at the beginning of the transfer, it is the result of the constant comparison of the ever toggling flags upon both $\overline{WR}$ edges. For 8-bit write to the DSP operation (wait on $\overline{RD}$), it simply detects a $\overline{RD}$ falling edge on which to place data on the lower bus byte. That is, the functionality of this pin changes according to the desired mode of operation set by `dspAccessMode` and `dspRdWr`.

- `dspAccessMode`: determines whether the module is to expect an 8-bit or 16-bit interface mode.

- dspRdWr: of particular importance for the addr_mux_proc process that determines how to pack the internal 24-bit dsp_address_out. The reception of a valid address depends on the current operation being performed (read/write) and its mode (8-bit/16-bit) as well. Internally this is accomplished via local signals signifying the occurrence or relevant external events namely validDataInRis and validDataOutFal.
- dsp_address, dsp_data_in, dsp_data_out, and dsp_enable are self-explanatory.

Figure 5.2 shows a typical timing diagram for the a16-bit mode read operation (wait on $\overline{WR}$)

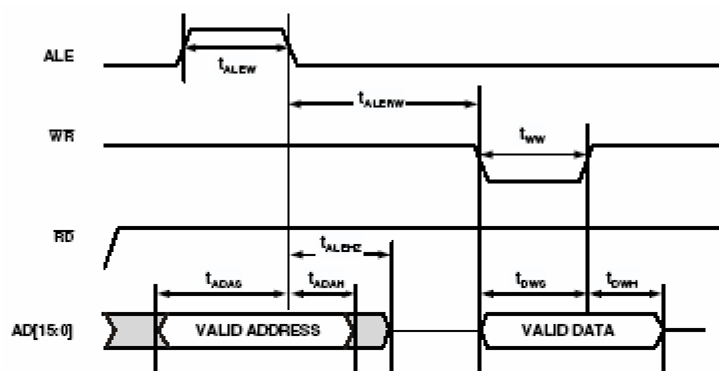

Figure 5.2: 16-bit write

This is a simple example in which the module receives two 16-bit words both on the falling edge of the ALE and the rising edge of the $\overline{WR}$. Two internal signals are toggled upon each reception. By continuously comparing these signals one can determine when both words have been received. At the end of the ALE cycle, a 16-bit address word has been received. After toggling the validAddr signal, dsp_status (defined as validAddr xnor validDataInRis) goes to zero because a valid data has not yet been received. When the validDataInRis signal goes high upon the rising edge of $\overline{WR}$, dsp_status becomes true again signifying that a valid address-data combination is now available to be sampled from the internal portion of the interface bus. Utilizing this simple yet powerful mechanism, an external process can tell when to sample the interface signals by probing the dsp_status signal. However, the issue of synchronizing these signals with respect to the internal clock is yet to be tackled.

For this reason, `synchronizer` is first applied to `dsp_status`, `dsp_address`, and `dsp_data_in` before a synchronous process can receive these signals.

As with regard to the 8-bit write mode (wait on $\overline{RD}$), every 256 $\overline{RD}$ cycles, the DSP updates the upper 16-bit address (`ADDR<23:8>`) in one ALE cycle. The least address byte (`ADDR<7:0>`) is supplied on the upper `AD` byte every $\overline{RD}$ cycle. Thus on the falling edge of $\overline{RD}$, `DSP_Driver` reads `ADDR<7:0>` and places a valid data byte on the lower `AD` bus as soon as it becomes ready. On the rising edge of $\overline{RD}$, the DSP latches the data byte with zero hold time requirements. That is why `WAIT_RD_Fal_proc` was implemented as level-sensitive rather than edge-sensitive. When including `dsp_data_out` in the sensitivity list, later changes made to `dsp_data_out` are allowed to propagate after the occurrence of the falling edge. Figure 5.3 illustrates this scenario.

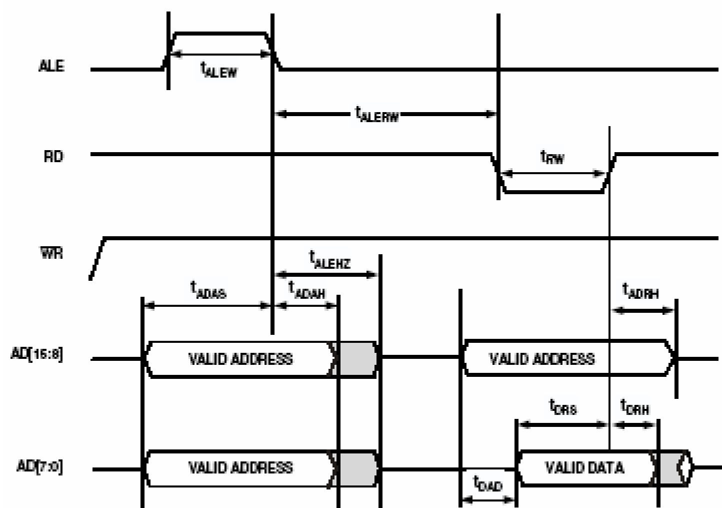

Figure 5.3: 8-bit read cycle

5.2 SYNCHRONIZATION MECHANISM

It was established earlier that the reception/transmission of words in the `dsp_driver` module occur asynchronous to the system's clock due to the fact that they are controlled solely by the parallel port signals. This generates the need to synchronize the incoming signals in order to guarantee their integrity.

synchronizer takes as inputs dsp_status, dsp_address, dsp_data_in, in addition to the clock to which these signals are to be synchronized. It outputs synchronized versions of the above signals namely; sync_out, sync_address_out, and sync_data_out. Moreover, sync_out is a trimmed version of dsp_status that extends over one clock cycle precisely. This ensures that even though dsp_status might still be on for sometime until the reception of new valid address in a new *ALE* cycle, it is only detected once in the synchronous process that polls on dsp_status. This arrangement relieves the internal clock resolution from any dependency whatsoever on the external asynchronous signals, provided that the internal clock resolution is fast enough to allow for the detection of the DSP's signals and to act upon them accordingly.

synchronizer accomplishes its functionality through the use of two signals namely event0, and flag, and three processes of which one is completely asynchronous. proc0 operates on async_signal and assert/deassert signal event0 whenever async_signal is high/low. In turn, proc1 and proc2 have both clk and event0 in their sensitivity lists. Listing 5.1 shows the VHDL code for both processes.

Listing 5.1

```
proc1 : process (clk, event0)
begin
if (clk = '0' and clk'EVENT) then
      if (flag = '0' and event0 = '1') then
            sync_out <= '1';
            sync_address_out <= async_address_in;
            sync_data_out <= async_data_in;
      else
            sync_out <= '0';
      end if;
end if;
end process;

proc2 : process (clk, event0)
begin
if (clk = '0' and clk'EVENT) then
      if (flag = '0' and event0 = '1') then
            flag <= '1';
      elsif (flag = '1' and event0 = '0') then
            flag <= '0';
      end if;
end if;
end process;
```

Always synchronized to the falling edge of the clock, proc1 transmits its input port to its output port if event0 is detected while flag is zero. Otherwise, sync_out is pulled

low. `proc2` ensures that during one high cycle of `event0`, `flag` is only allowed to stay low for one clock cycle. Unless `event0` is reset by `proc0`, `flag` stays high.

Synchronizing these processes to the falling edge is extremely important for the following reason. Since the system's internal logic is operating on the rising edge, the incoming signals should be allowed a finite time in which to settle and propagate before being sampled again at the rising edge. Otherwise, depending on logic routing, `sync_out`, `sync_address_out`, and `sync_data_out` may or may not happen to coincide with the rising edge resulting in probable error on random basis which might even affect certain bits non-uniformly. This conforms to the classical rule in digital logic design; always transmit at one clock edge and receive on the other.

5.3 DSP BUFFERING SCHEME

Similar to the packing scheme encountered earlier for the digitally synthesized sine waveform, the DSP utilizes a buffer to be read from or written to by two distinct processes interchangeably. However, this time only one buffer is needed. Because the two processes alternate between read and write operations, all buffer signals except for the data read out have to be multiplexed ensuring that no multi-source signals occur. In the terminology depicted in figure 5.3, `Prs` suffix denotes process signals while `Thrd` signifies thread ones. While there is no ultimate distinction between processes and threads hardware-wise, a hardware thread or a spatial thread [30] refers to a portion of synthesized sequential logic that complements and aids the main process. That is, process `dsp_packing` is a thread while process `mainstream` is a process. Figure 5.4 shows the metaphoric schematic equivalence of the developed VHDL code.

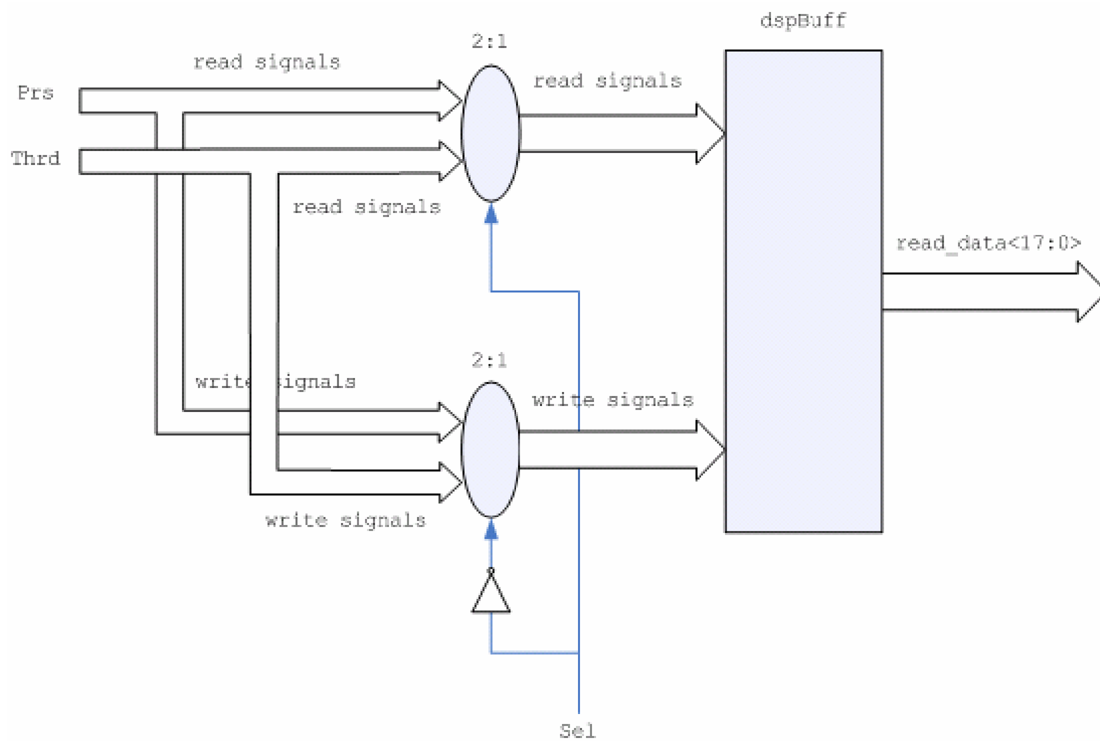

Figure 5.4: DSP buffering scheme

Table 5.1 illustrates the interchangeable operations set by signal `dsp_Sel`.

Table 5.1: dsp_sel operations

Dsp_Sel	OPERATION
0	read `Prs` write `Thrd`
1	write `Prs` read `Thrd`

Chapter 6

Experimental Results & Analysis

In this chapter, various experimental issues along with their analysis will be presented. The chronological order of the following subsections reflects the natural course through which these issues were tackled.

6.1 MISCELLANEOUS HARDWARE DESIGN CONSIDERATIONS

During the VHDL hardware design phase, a rather unexpected set of considerations needed to be addressed. Some of the attempted solutions were found after investigating some third-party reference designs while others were completely spontaneous. The following discussion will describe and detail all encountered cases along with their solutions.

Firstly, Right from the very beginning, when resetting the SX2 chip, a minimum delay of $200\mu s$ had to be inserted both while reset is low and after reset is pulled high. The second delay interval allows the internal logic of the SX2 to stabilize before proceeding with configurations. This is relatively straightforward and logical. However, experimentations showed that unless SX2 pins were tri-stated while reset is in progress, the SX2 chip may behave quite strangely in the subsequent operations. This may be due to some sort of leakage current that affects the state in which SX2's internal logic starts up after reset.

Secondly, while configuring the internal registers of the SX2, it turned out that a delay after each register write operation had to be inserted. Otherwise, the chip stops generating interrupts. This particular problem needs clarification from Cypress. The solution for it was found when investigating a third-party C driver code written originally for an embedded OS called VxWorks [31]. In the C code, the delay is specified as a call to a function whose parameter is an integer signifying the units of

delay needed. In VHDL, a very relaxed delay of $300\mu s$ was appended to each register write operation.

Thirdly, when writing the USB descriptor, the programmer has to ensure that complete 500 bytes are written to the internal memory, regardless of the real length of the customized descriptor. In VHDL, after committing 148 bytes to the descriptor's RAM, consecutive fictitious writes had to be performed until the index becomes 500. It seemed as if the descriptor RAM required those fictitious writes to increment some internal counter which is logically 'and'-ed with whatever triggers enumeration later.

Fourthly, for the enumeration to occur properly, the ENUMOK interrupt flag had to be polled on. In other words, experimentations showed that it was not sufficient to expect the occurrence of ENUMOK directly after the SX2 had been connected to the PC. Rather, the FPGA repeatedly waits on interrupts and reads them. Unless the current interrupt is ENUMOK, the FPGA keeps on waiting for yet another interrupt. This is unexpected as the manufacturer specifies that once the SX2 is connected to the PC, ENUMOK is the first interrupt to occur.

Finally, when switching to the 16-bit FIFO access mode, a dummy packet has to be committed first before starting regular packet sending process. Initially, all FIFO accesses were carried out using the default 8-bit mode. Then realizing that packing rate can be twice as much when utilizing the 16-bit FIFO access mode, it seemed obligatory to exploit this feature. Nevertheless, once the WORDWIDE bit in the EPxPKTLENH register is set, the very next packet to be committed though that endpoint has to account for the fact that its reception on the PC side would be erroneous. Therefore, a dummy packet had to be sent first in order to allow the internal FIFO logic to adapt to the new configurations.

6.2 FPGA INTERNAL LOGIC ROUTING

6.2.1 Problem encountered & solution

At some point during the project development, after a continuous connection was successfully established, the mere inclusion of any form of additional logic caused the system to randomly stop transmission or even collapse altogether. That is, once any extra logic was added to the VHDL design, the system either interrupted packet

transfer after a random number of packets or failed to enumerate with the peer PC in the first place. This was amongst the most serious unforeseen difficulties encountered in the project. Doubt was applied systematically to virtually every aspect that had been achieved thus far. The similar reference designs investigated utilized high-end DMA controllers [32] [33] with typical internal clock of 130 MHz and thus much more time resolution. Coupled with some inconsistencies in what turned out later to be PC-related problems (refer to section 6.3), the robustness of the 25 MHz FPGA asynchronous interface was even questioned. The effect of the FPGA internal logic routing on the stability of the system's performance was unprecedented. After spending a significant amount of time trying to troubleshoot this problem, it was not until the addition of a mere signal had caused a system's failure that the nature of the problem has started to emerge. This indicated that something wrong took place in the FPGA. Between a fully working design with no additional signal routed to the debugging port, and a totally failing scenario with the inclusion of that signal, there was a mere debugging signal. The sudden deterioration of the system was at last traced at the synthesis & implementation options.

In the *synthesize process* properties, the following modifications were introduced:

- Optimization goal & effort: speed & high respectively.
- FSM encoding: gray.
- Max fanout: 100.

In the *implement process* properties, the following changes were made:

- Optimization strategy: speed as opposed to area.
- Place & route effort: high.

After re-performing the *synthesize* and *implement* processes, the resultant design was downloaded again to the FPGA, only this time to work with the newly added logic.

6.2.2 Analysis

The difficulty of the problem is due to its twofold nature. Depending on the type of the logic added, the system either collapsed or degraded. The best theory fitted to these symptoms can explain to a high degree of confidence the failure of the system, but not the random pattern manifested in the number of packets received before

transmission is interrupted. The following discussion will attempt to explain the encountered behavior. In addition, the discussion will be restricted to symmetrical volatile-memory FPGAs such as SPARTAN-3.

In volatile-memory FPGAs, reprogrammable Static-RAM cells control pass transistors that steer signals to make routing paths [34] (Figure 6.1). These switches are characterized by being both resistive and capacitive resulting in large delays (RC), which dominate that of a CLB. It is the incremental nature of these delays that makes routing so critical (cumulative effect) and path dependent.

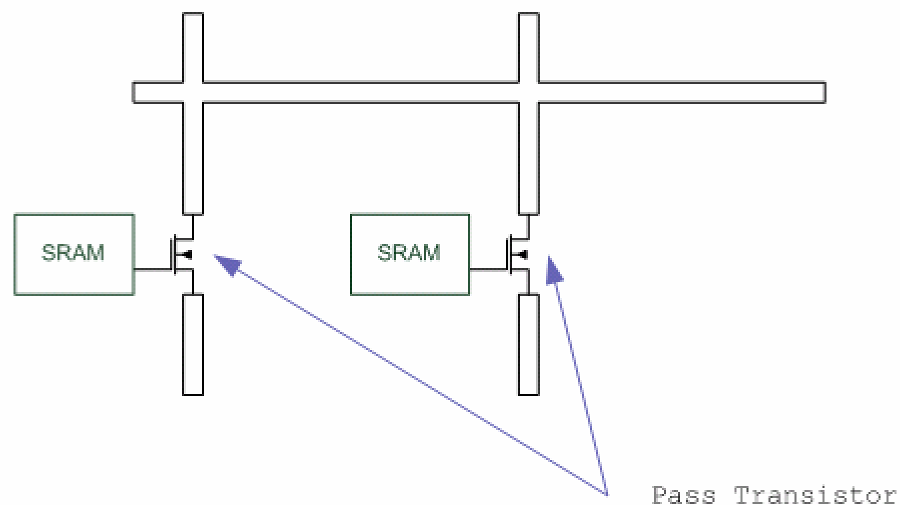

Figure 6.1: SRAM-based pass transistors

As shown in figure 6.2, CLBs and IOBs connect to interconnect segments in wiring channels, and wiring channels intersect at switchboxes (SBs). On one hand, interconnect segments are logical connects i.e. they can be made up of portions interleaving several mask layers. One the other hand, due to their large size and capacitive nature, pass transistors within a switchbox have limited connection possibilities of further variable lengths as depicted in figure 6.3. In other words, the uniformity of interconnect segments whether in wiring channels or in switchboxes is sacrificed in the favor of area (logic density)

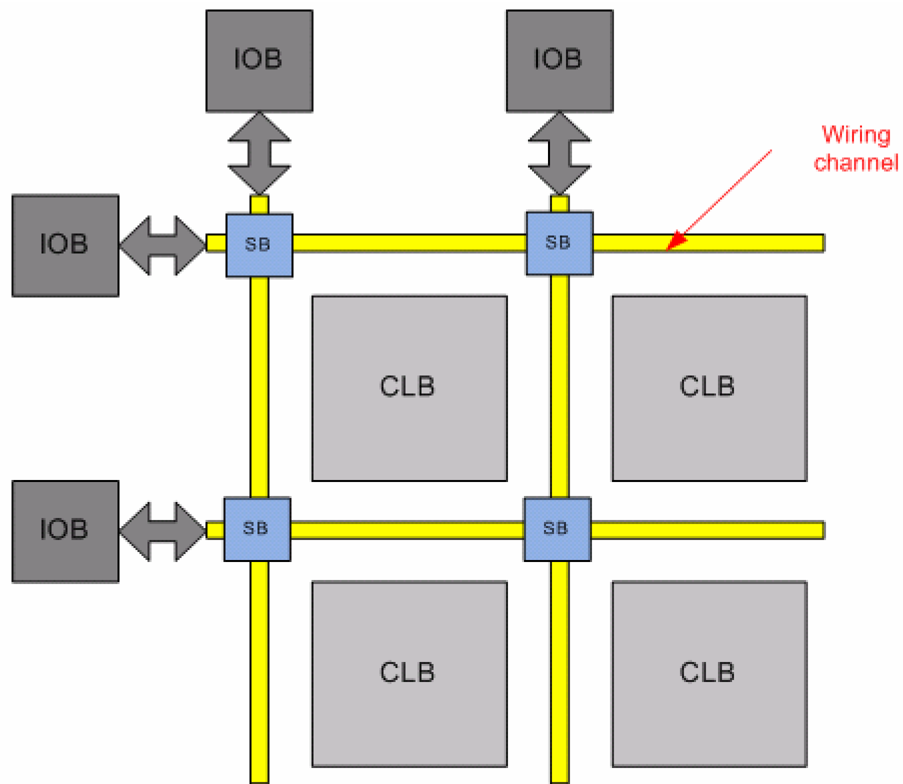

Figure 6.2: Symmetrical FPGA Architecture

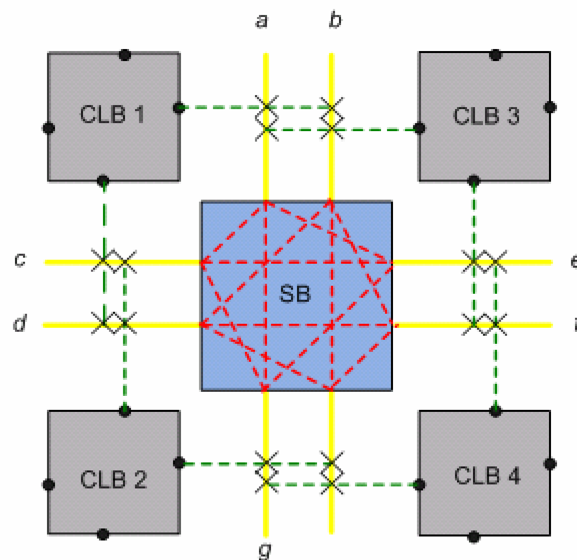

Figure 6.3: Limited Routability [35]

Figure 6.4 shows a typical routing scenario for the CLBs and SB depicted in figure 6.3.

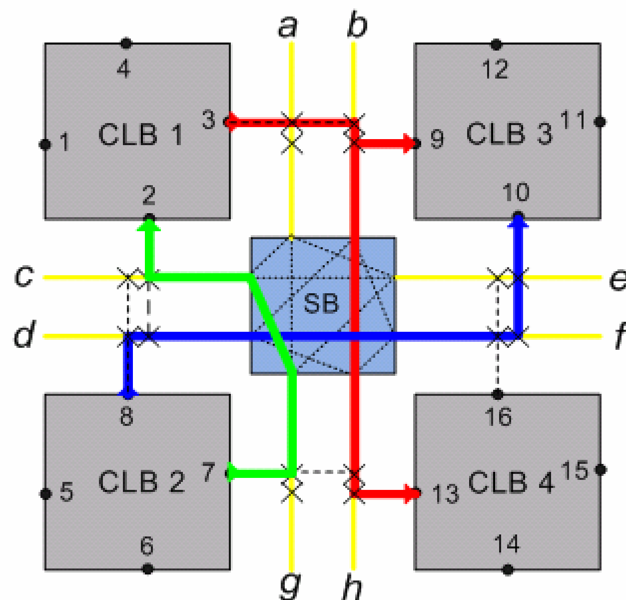

Figure 6.4: A routing example [35]

Also FPGAs architecture addresses some other needs such as dedicated clock routing for minimal skew problems.

Furthermore, routers typically use additional strategies that might help in optimizing designs of which the following are typical: Firstly, routers might employ unused logic resources as routing resources. Secondly, utilizing silicon-characteristic parameters, routers can model the effects of various resources to calculate delays which would be used to assess potential propagation delay of a route for a given option.

As a matter of fact, routing can be thought of as a search problem whose states (branches in a search tree) are partial routes. The above techniques specific to this search problem are used as domain-specific knowledge which help in converging to an optimal solution. In addition to testing whether the final goal is reached, estimator function assesses the remaining distance to the goal state, and successor function yields a combination of possible states to which the current state can go. The most important function in the search problem is the Heuristic function which assesses the goodness of a partial route against other options. In the routing problem, the Heuristic function will attempt to minimize critical path delay under timing and routing constraints [36]. The Heuristic function may use the following condition:

$a_u + D_{uv} \le a_v \quad \forall (u,v) \in E_t$ where

a_u is the arrival time at node u

a_v is the arrival time at node v

D_{uv} is the delay along $path(u,v)$

E_t is the matrix of routes relative to the current state in the search tree $T = \{T_1,...,T_m\}$

In order to evaluate this heuristic function, some general techniques for solving optimization problems with difficult constraints are deployed such as Lagrangian Relaxation [35].

On top of what has been mentioned so far, the fact that various constraints, whether user or architectural, are strongly coupled with global routing problem requires that routing be performed simultaneously through cooperative processes. Wiring channels offer limited flexibility to neighboring resources (IOBs or CLBs). This makes user pin assignments propagate to interconnect segments allocations. The latter is coupled through SBs constraints to interconnect segment assignments of other channels rendering routing a global problem. That is, routing can not be spatially decomposed into smaller independent manageable problems. Fundamentally, this adds more breadth to the search problem. For this reason, simultaneous routing can utilize cooperative concurrent processes with excessive-delays-based heuristic function as a way of casting preference.

It was established earlier that routing delays are more predominant than that of the abstract logic ones and that performance is highly influenced by cell placement [37]. The key factor behind the total collapse of the system is its self-timed aspect. As enough clock cycles were inserted to generate timing specifications set by the SX2, there was no need to specify timing constraints. At some stage, the inclusion of additional resources changed routing and in turn degraded performance due to less optimized paths that violated SX2 timings. Depending on the magnitude of the additive logic resources, the system either collapsed or degraded. Additive resources necessitate that routing be placed through difference paths in order to accommodate for the optimal utilization of area and pin assignments. Changing optimization strategy to timing-based rather than area-based improves critical paths delay. While altering the effort properties to high forces the synthesizer and implementer (place and

route) into considering more states in the search tree which will improve performance eventually. Decreasing the fanout property from 500 to 100 relieves the system from some unnecessary burdens. Gray encoding in principle is more noise immune as only one bit is likely to change while in a given state. The latter option was modified out of desperation and is of no significant impact inside the FPGA, except that it might consume a little bit of extra logic as its encoding is more logic demanding that a regular adder. However, what is left yet to be inferred is the following: in some cases, the inclusion of additional logic had caused the SX2 to interrupt its transmission after some arbitrary number of packets. A possible cause for this is accidental worst case propagation delay scenarios which had taken place on a near-random basis. Combinational logic circuitry may be forced to propagate through all possible logic levels depending on its inputs. As SX2 functions were developed in an almost behavioral manner, it is extremely difficult if not impossible to trace this back because the Xilinx synthesizer automatically produced their logic resources. More sophisticated IDE tools such as ModelSim allows the developer to examine his/her designs interleaving among different representations. A logic design developed in VHDL can be examined more closely by checking its RTL equivalence. Recently, a huge amount of effort is being put towards implementing full behavioral synthesizers [38-42] that will significantly improve both complexity and time-to-market issues associated with contemporary cutting-edge logic designs.

6.3 USB2.0 ENDPOINT BANDWIDTH

6.3.1 Streaming through isochronous endpoint

The USB2.0 specifications define the isochronous transfer as being periodic with continuous communication between host and device, which is typically used for time-critical information. This is achieved by encapsulating time-indicative information in the data. It does not support error detection and correction (CRC), thus unreliable. Typical application is video streaming at which a minor loss of data may not be even observable [43].

Unlike bulk transfer, when using isochronous transfer, a pre-negotiated portion of the available bandwidth in each microframe is reserved by specifying the *wMaxPacketSize* field of the endpoint descriptor. Moreover, the latency of the delivery is specified via the *bInterval* field of the endpoint descriptor. The latency is the rate at which the host retrieves isochronous packets from the USB device per microframe.

In principle, both bulk and isochronous endpoints can yield identical throughput when operating in an equivalent manner. The criterion upon which bulk and isochronous endpoint are considered equivalent is a function of size, availability of bandwidth in a microframe, and polling interval. Size intuitively refers to both endpoints having the same packet size. Since bulk type uses bandwidth leftovers in a microframe, the rate at which a bulk endpoint operates is governed by the production of its data plus the current status of the microframe. When operating alone, a bulk endpoint can use as much bandwidth as the microframe can accommodate bearing in mind the size of the bulk packet. On the other hand, an isochronous endpoint must be pre-allocated a certain amount of bandwidth to be used every polling interval. Effectively, this allows the isochronous endpoint to operate at a rate lower than that of a microframe. A technique called PID sequencing orders isochronous packets sent within one microframe [44].

Away from ideal conditions, an isochronous endpoint is more promising in terms of throughput. While a bulk endpoint is protected by CRC that has to be acknowledged by the recipient, an isochronous endpoint does not guarantee data integrity. The overhead of handshaking in bulk type forces the resending of a packet whose just one bit might have toggled during propagating through the physical medium. If used for very high-bandwidth streaming, the slightest error in a packet might be unrecoverable resulting in transmission stoppage. Whereas in an isochronous type, data in packets might get affected by minor errors and still be received properly. The key difference between the two types is the handshaking overhead that deploys a CRC error check before flagging the successful reception of a packet.

In order to optimize the USB bandwidth, an isochronous endpoint was first utilized through which data were streamed. The appropriate modifications were introduced to `sx2RegsDef` double dimensional array so that all relevant registers were configured accordingly. In addition, the high-speed section of `mySX2Descr` array describing endpoint 6 was setup for the intended task. Surprisingly, the obtained results were not up to the expectations. The following paragraph discusses the background of the problem that has led to some disappointing outcomes.

When first released as a legal document, USB did not support isochronous transfer type. This type was appended at a later stage out of necessity. As it is the case with any introduction of novel solution in the IT world, many organizations operating across the spectrum start working in parallel to welcome the new offspring. For this reason, Windows XP service pack 1 did not support isochronous transfer at first. Microsoft released a patch to solve this problem [45]. Nevertheless, many people reported inconsistencies associated with the isochronous type. Microsoft claims that this problem is solved altogether in the service pack 2. One the other hand, Intel® have identified that 82801 USB2.0 chipset may have intermittent communication or connection problems [46]. Intel is not planning any long-term fixes to these issues and spontaneous solutions may propagate as deep as a bios update.

This research was conducted after identifying the strange problem. Yet all what could be deduced about it does not exceed the realm of speculations. For each packet, a time interval of 6 to 7 seconds was observed when streaming through an isochronous endpoint of 1024 KB size, one transaction per microframe, and a polling interval of 1 ($2^{(1-1)}$). Clearly as it may seem, a 7 seconds interval is totally illogical. An

oscilloscope was utilized in order to examine a flag indicating whether the double-buffered endpoint FIFO is full or not. The examination showed that the flag is almost all the time asserted except for a tiny interval in which the FPGA is allowed to write more data to the buffer. This reveals that the problem is solely related to the rate at which the PC is fetching packets. Indeed, it turned out that manual consecutive single reads yielded a faster rate than that of the automated sequence. This was achieved in the program by bypassing the stage in which the user specifies various transaction parameters and substituting it with a press of a button. In an attempt to remedy this problem by means of programming, a polling thread was developed whose job is to wait for the transmission to complete before requesting yet another transmission. The approach was part of an attempt to trace the source of the problem narrowing down potential causes. Since manual consecutive reads yielded a better rate than the automated one supplied as part of the USB DLL, it seemed logical to try a customized automated approach that gets around the potential source of the problem. Listing 6.1 shows part of the code deployed.

Listing 6.1

```
if (!rdActivePipe.IsIsochronousPipe())
{
      rdActivePipe.SetContiguous(true);
      rdActivePipe.UsbPipeTransferAsync(true, TIME_OUT,
      new D_USER_TRANSFER_COMPLETION(ListenCompletion));
}
else
{
      pollOnPacketDelegate = new PollOnPacketDelegate(PollOnPacket);
      onPollingComplete = new EventHandler(OnPollingComplete);
      this.PollingComplete +=new EventHandler(DefaultInterface_PollingComplete);

      pollOnPackThread = new Thread(new ThreadStart(PollingThread));
      pollOnPackThread.Start();
}
.
.
public void PollOnPacket ()
{
      // Wait for transmission to finish
      while(rdActivePipe.IsInUse());

      int rdlBuffSize = rdActivePipe.GetBuffSize();
      byte[] lbuffer = new byte[rdlBuffSize];

      rdActivePipe.UsbPipeTransferAsync(true, lbuffer, rdlBuffSize, TIME_OUT,
            new D_USER_TRANSFER_COMPLETION(TransferCompletion));

      while(rdActivePipe.IsInUse());

      if(rdActivePipe.GetTransferStatus() != (int)wdu_err.WD_STATUS_SUCCESS)
      {
```

```
            LogMsg(string.Format("Transfer Failed! Error {0}: {1} ",
                  rdActivePipe.GetTransferStatus().ToString("X"),
                  wd_status_string_module.GetStat2Str(rdActivePipe.GetTransferS
            tatus())));
      }
else
{
.
.
```

The resultant isochronous listening approach involved a long time in the busy-waiting state rendering the overall GUI irresponsive (trapped in `while(rdActivePipe.IsInUse())` statement). In short, the polling thread literally ate too much processor execution time according to the OS vocabulary.

Digging deeper in the DLL, the original C++ implementation of the call-back function was examined. It turned out that the C++ call-back function uses multithreading which is identical to what had been attempted. After some research in the example codes provided by the third party software, a webcam application was spotted which utilizes isochronous transfer as a means to communicate the video stream. Examining the C++ specific implementation, the C++ function used for transferring the isochronous packets used a specific parameter which was not supplied in the C# wrapped DLL. The DLL was modified to include the same constant parameter and the test was conducted again. Nevertheless, the results remained the same. With this, the attempts to solve the problem associated with the isochronous type were brought to an end, because of restrictions by the project timescale and resources. Unfortunately, this leaves uncertainty and speculations about the identified OS-related problem.

To conclude, it is worth pointing out that some software packages such as Labview® do not support isochronous type at all. This demonstrates how the late inclusion of the isochronous type in the USB2.0 specifications affected the third-party support for this particular transfer type.

6.3.2 Streaming through bulk endpoint

After giving up on the isochronous option, the bulk type was used. The initial testing results showed a pattern of totally random number of packets that were received before the transmission was interrupted altogether. Gradually an increasing delay was

inserted until a continuous stream of packets was observed in the soft scope. The following calculations detail the timing aspects associated with that working scheme.

Digital sampling rate: $\frac{25MHz}{8} = 3.125MHz$ ➔ sampling period = 320 ns

With 2 Bytes per sample ➔ packing rate: 6.25MBytes/sec

Packet frequency = $\frac{6.25MBytes/\sec}{Bytes_per_packet} = \frac{6.25 \times 10^{+6}}{1024} = 6{,}103.515625Hz$

Packet period = $163.84\mu s$

Effective packet period = packet period + delay inserted = $163.84\mu s + 20\mu s = 183.84\mu s$

Effective packet frequency = 5439.51262 Hz

In order to slow down the system, the `mainstream` process was forced to poll on the status flag of the `sine_packing` process. Listing 6.2 shows how is this done in VHDL.

Listing 6.2

```
when SINE_PACK_SEND_LOOP =>

      case index is
            when 0 =>
                  rwSel <= "01";
                  increment(index);

            when 1 =>
                  sinPack.syncFlg <= '1';
                  increment(index);

            when 2 =>
                  if (sinPack.status = FINISHED) then
                        sinPack.syncFlg <= '0';
                        increment(index);
                  end if;

            when 3 =>
                  rwSel <= "10";     -- toggle
                  thread_current_state <= IRRELEVANT;
                  thread_next_state <= IRRELEVANT;
                  thread_status <= INPROGRESS;
                  increment(index);
```

The overhead associated with committing a packet to the USB device is primarily determined by its iterative loop. Eleven clock cycles constitute the time needed for

each sample to be written to the endpoint FIFO. Thus roughly the overhead of the sending process is:

$$sending_overhead = 11 \times 1024 \times t_{clk} = 11 \times 1024 \times 40ns = 450.56\mu s$$

The total time taken before a packet is fully committed to the USB device FIFO is:

$$overall_time = 450.56\mu s + 183.84\mu s = 634.4\mu s$$

Of course, this timing can be speeded up by concurrently filling one of the buffers while sending is in progress. This was the very reason behind adopting a double-buffering scheme which was designed with a high bandwidth interchangeable operation in mind. Since the performance was no near a bottleneck scenario whereby consumption is faster than production, neither concurrent operations nor time-efficient automatic FIFO mode were deployed. Had not this been the case, the packing process could have been assigned a faster clock and the RAM buffer could have been operated in the FIFO mode.

The final rate at which packets were being sent can be computed as follows:

$$hardware_bitrate = 1024\frac{Byte}{packet} \times (634.4\mu s)^{-1}\frac{packet}{\sec} = 1.614123MBps = 12.912988Mbps$$

The duration of a high-speed microframe has no effect on the hardware bitrate. After the first packet has been committed, the production of the next packet occurs while transmission is in progress. The production of a packet refers to all operations starting from buffering it in the FPGA and ending with writing it to the endpoint FIFO.

As there was no means by which to observe what is actually taking place in the USB protocol, the assessment of the resultant bitrate had to be done by programming on the PC side. In order to do so, the following was done. Realizing that the refresh thread has a frequency of 10Hz (a sleep interval of 100ms), the occurrences of two consecutive refresh operations enclose in between a certain number of received packets. In the `SoftScope` class, three private members were added; `previousBurst` to hold the index of the last packet received when refresh was last called, `average` to store the computed bitrate, and `flag` to decide whether bitrate has been already computed. The code shown in Listing 6.3 was inserted to the refresh method

Listing 6.3

```
if (flag == 2)
{
```

```
	average = (double)(Var.USB_Packet.BurstIndex - previousBurst)*1024/100e-3;
}

line = String.Format("Average endpoint throughput {0:F} KBps", average/1000.0F);
lines.SetValue(line, 2);

flag++;

previousBurst = Var.USB_Packet.BurstIndex;
```

The reason why `average` is computed after the second increment of the flag is in order to allow `previousBurst` to adapt to the previous `BurstIndex` first. `average` signifies the number of bytes received in a 100 ms interval (thread sleep interval). Surprisingly, the obtained average was only 30.72 KBytes per second i.e. thirty packets/sec (245.76 Kbps). Moreover, this figure degraded to nearly 20 KBps after some time. The manufacturer specifies that the maximum achievable throughput per endpoint is 24 MBytes per second. Nearly a factor of 800 separates the results measured on the PC side from what the manufacturer promises (24 MBps). A factor of 15 is the difference between the rate at which data is being sent from the FPGA and the feasible rate specified by the manufacturer.

Contemplating this matter for some time, various tests were conducted in an attempt to elucidate this problem. With exactly identical setup, removing the line in which the full flag is tested before proceeding with the program results in a random number of packets being received by the PC before the transmission is interrupted altogether. This means that at some point in time, the FPGA is required to slow down the rate of packet production. Relying on this, the best conceivable explanation for this strange phenomenon is the following. As more traffic is being placed on the differential USB pair, packets are becoming bursty due external noise sources affecting the PCB trace. At some stage, the occurrence of an error in a packet becomes inevitable whereby the PC does not acknowledge one reception forcing the USB device into a resend operation which conform to why polling on the FIFO flag is needed. The USB device handles automatically low-level USB protocol requests such as a resend operation. In plain English, although enumeration shows that transmission is taking place in the high-speed mode, the actual achieved bitrate is degraded due to noise when trying to exceed certain threshold bitrate. Unfortunately, the only way to defend this theory is to use a USB protocol analyzer which was not available. A USB protocol analyzer shows what is actually happening in the USB transmission.

On the PC side, it was observed that the CPU utilization increased dramatically once streaming had begun. The processor on which the GUI was run is Intel Mobile Centrino®. The Centrino processor is part of a new trend towards having a real mobile processor as opposed to a scaled-down version of a regular P4. The Centrino processor is characterized by having a dynamic pipeline whose certain stages are shut down when the processor is not fully functional. This allows for a maximum conservation of power in accordance to the status of the processing load. The applications being run concurrently determine the processing load and in turn processor utilization. It was observed that once streaming had started, the cooling fan of the Centrino started to work immediately indicating more processing load. In addition, the CPU's clock was noticed to be working at its maximum speed. This hints that a significant amount of computational load had led to such a reaction from the processor, which is in line with the reasoning so far.

At last, it is worth stressing that the results obtained are with respect to streaming applications which are beyond the scope of this project. This effort was carried out in an assist to the on going research which aims at streaming an aggregate of 25 MBytes/sec supplied by 32 acquisition channels. People involved in that project are urged to investigate this problem further. As with regard to the project's aims, interfacing a digital tomograph to a PC was fulfilled.

6.4 DSP BOOTING

The ability to boot the DSP from the FPGA had not been an objective set right from the beginning of the project. Later on, realizing that the SHARC DSP supports a parallel port boot mode, and stemming from a practical need, this feature has been added to the system through a software-hardware co-design. At some stages during the development of the project, access to the PCI-based JTAG debugger and emulator was restricted because of personnel unavailability. This resulted in an interruption in the 4-month timetable that compromised the successful completion of the DSP part. The idea of programming the DSP from the FPGA seemed to be the solution for having no access to the JTAG. In addition, Analog Devices VisualDSP++ IDE has a 90 days trial version period which is enough for the project's development time span.

VisualDSP++ can generate a boot loader file for either an external SPI memory or a parallel EEPROM connected to the parallel port. That is, if the FPGA can emulate an 8-bit parallel EEPROM, the DSP can be booted from the FPGA upon reset.

The first problem that was overcome is the following. The DSP rest pushbutton halted the generation of the board clock whenever pressed. This means that the FPGA received no clock while the duration of reset. After consulting the board's designer, it was feasible to disconnect the reset signal from the clock generation unit. The second issue tackled was to do with the boot kernel. At first, the boot loader was parsed as a constant 8-bit array of around 4K entries. It took the synthesizer a great deal of time every time the design had to be recompiled. Moreover, when the boot method was tried, the oscilloscope revealed that at some points entries lost lock with addresses. The addresses read from the DSP were substituted by an internal incremental index only to yield identical results. What was inferred from the oscilloscope is that after around 3K addresses had passed, an entry whose address was supposed to be a particular value appeared after four addresses. This meant that there were inconsistencies between the entries and the addresses when approaching the end of constant array. Realizing that constant arrays are implemented as distributed memories, the entry-address inconsistent behavior towards large indices (4K entries) is likely to be cased by long-distance cumulative propagation delays. After reasoning about the problem, the solution was found in the Xilinx core generator facility. A 4K distributed memory ROM was generated. In the software GUI, an option for parsing a VisualDSP++ loader file into a Xilinx coefficient file was added. The Xilinx coefficient file has a specific format that includes a memory radix and a memory content vector. After examining some Xilinx coefficient files generated by MATLAB's filter design HDL coder toolbox, the format was comprehended and integrated into the GUI. The content of the 4K distributed ROM was initialized using the COE file and the boot operation was tested again. The result was a successful booting that was verified using the oscilloscope. Figure 6.5 shows a snapshot of the boot process.

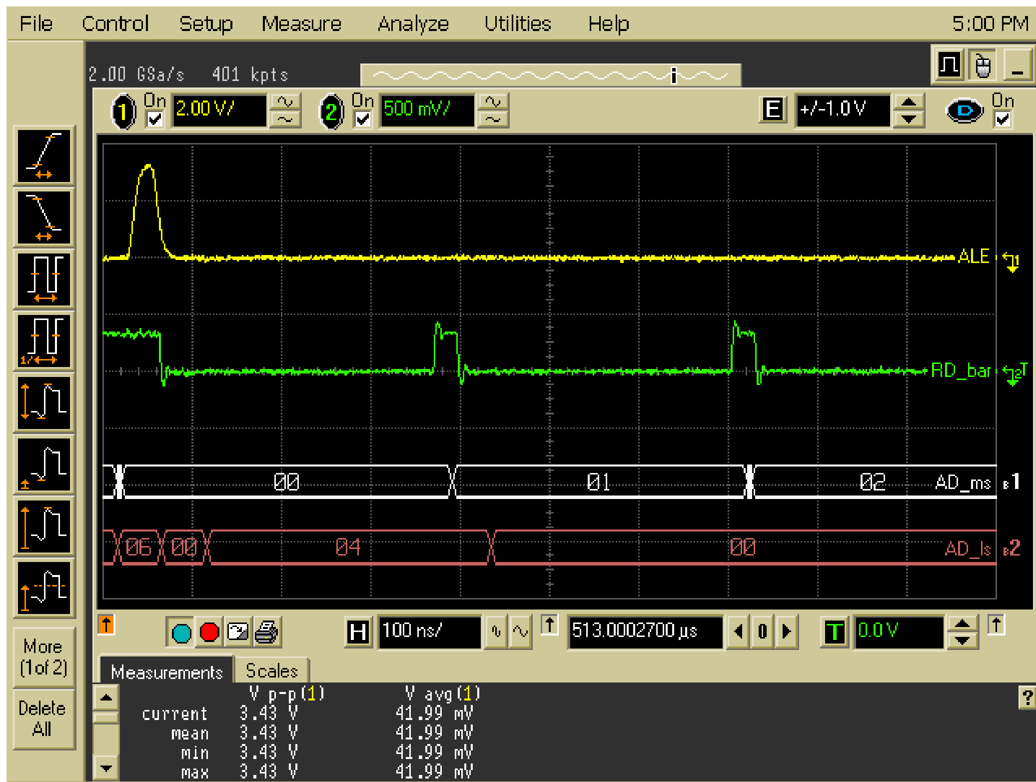

Figure 6.5: DSP booting snapshot

As a demonstrating example of the working scheme, a simple digital oscillator (critically stable biquad IIR filter) was implemented. For information about general IIR DSP filter implementations please refer to [47] [48]. Of course, this assembly language filter implementation is specific to the underlying ADSP-2126x SHARC architecture [20]. Details about the theory of self-sustained digital sinusoidal oscillators can be found at [49].

The DSP's timer0 was configured to deliver 8 KHz sampling frequency. Every timer0 interrupt, the digital oscillator algorithm is called and the parallel port is set up to transfer the content of the 32-bit word `Sinusoid`. For a complete assembly code listing please refer to the appropriate appendix. Figure 6.6 shows how transmission is initiated every 125 µs.

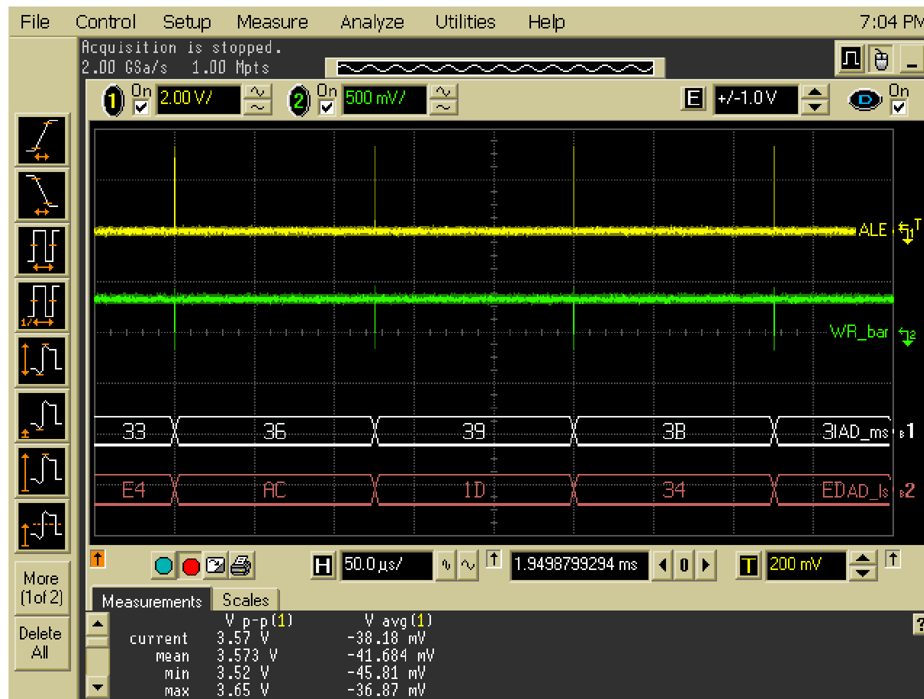

Figure 6.6: Digital oscillator transmission snapshot

The parallel port transfer operation consists of one ALE cycle per transaction. This is achieved by setting the parallel port external modifier to zero resulting in maximum data throughput. Figure 6.7 depicts this transaction whose $\overline{WR}$ cycle duration is 88 ns.

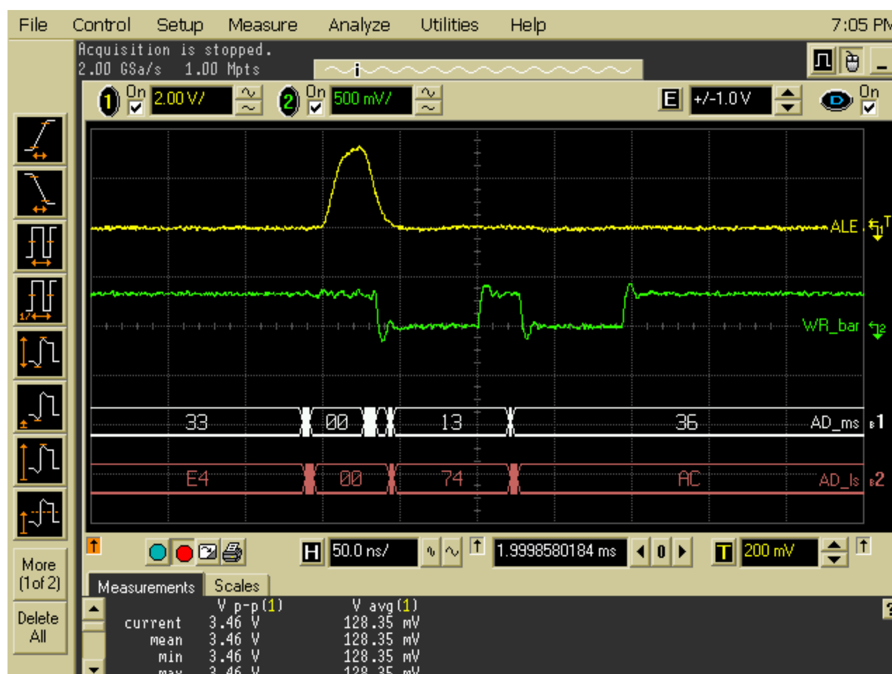

Figure 6.7: Parallel port transfer operation

Determined by the initial conditions of the digital sinusoidal oscillator with the sampling rate in mind (8 KHz), a 100 Hz sinusoid was generated. Samples were

buffered in the FPGA and were communicated to the PC in demonstration to the overall successful implementation. Figure 6.7 shows the waveform received at the PC side.

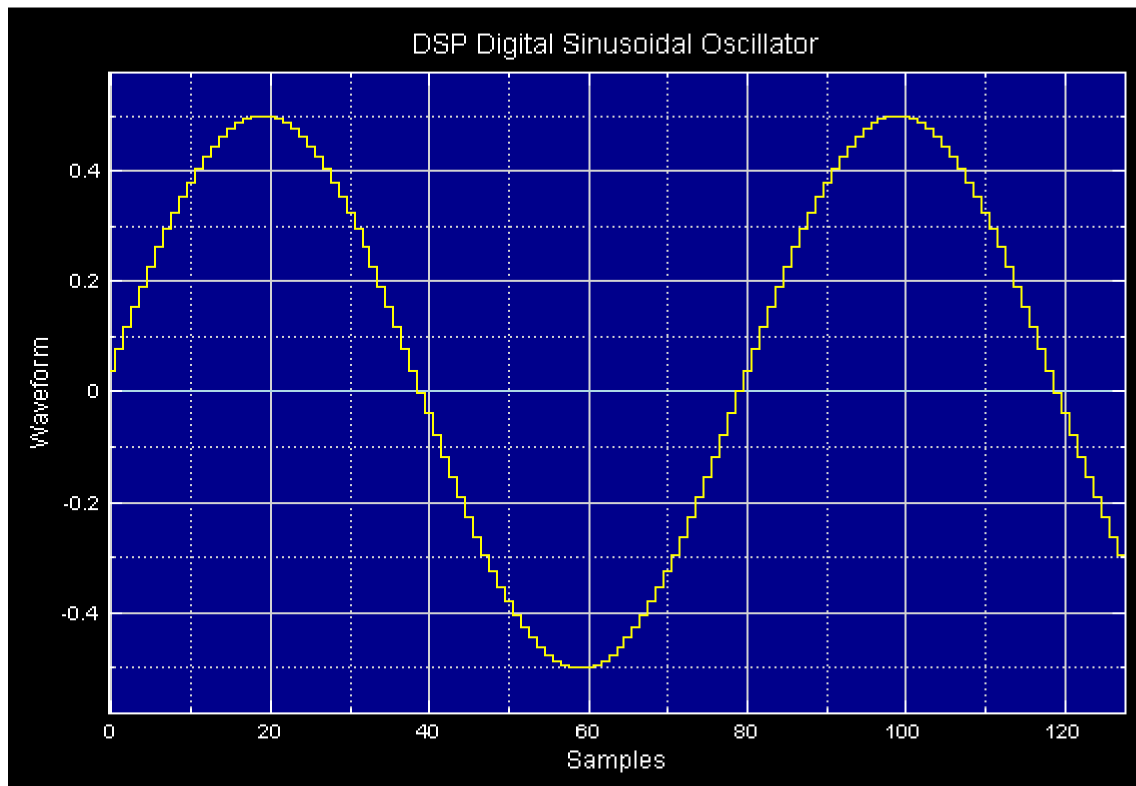

Figure 6.7: Digital Sinusoidal Oscillator

Figure 6.7 shows 128 32-bit samples sent over 1024 KB packet. The hex values had to be converted first to double type utilizing the `hex2decimal` method of class `HexConversion`. The quantization step was intentionally kept visible conveying a discrete sampled signal. Classically, a low-pass reconstruction filter is applied in the analogue domain to smooth out higher frequencies contained in the sharp-edge transitions. This can be also done digitally in the DSP or the FPGA or even on the PC side. However, since the waveform is merely meant to prove successful platform exploitation, there was no need for further processing.

6.5 FPGA METRIC ASSESSMENT

Synthesis was performed using Xilinx XST 6.1.03i. The summary of the synthesis report is shown in table 6.1.

Table 6.1: Summary of Main synthesis report with ROM present.

		Main (with distributed ROM)	
Cell Usage	BELS	6047	
	FFs/Latches	598	
	CLK Buffers	3	37%
	IO Buffers	71	50%
Device Utilization	SLICES	2005	55%
	SLICE Flip Flops	598	8%
	4 input LUTs	3580	49%

Then the distributed ROM was commented out and Main was re-synthesized. The summary is shown in table 6.2.

Table 6.2: Summary of Main synthesis report excluding ROM.

		Main (without distributed ROM)	
Cell Usage	BELS	1940	
	FFs/Latches	570	
	CLK Buffers	3	37%
	IO Buffers	71	50%
Device Utilization	SLICES	93	25%
	SLICE Flip Flops	570	7%
	4 input LUTs	1447	20%

As with regard to timing summary section of the synthesis report, the maximum frequency was not included int the summary presented above for the following reason.

Examination of this section showed that the theoretical maximum frequency is 45.595MHz with a maximum combinational path delay of 10.356ns. Firstly, the scenario at which 10.356ns delay takes place was traced back to the source net RD_bar_i with destination DB_d<7> through five levels of combinational logic. The irrelevant debugging pin was commented out and the design was re-synthesized. Again the report showed a maximum combinational path delay of 9.638ns occurring between source RD_bar_i and destination AD_io<7> through four levels of logic. Not only the destination pin is supposed to meet a certain setup time of 3.3ns (specified by the DSP manufacturer), but also RD_bar_i is totally irrelevant to the internal clock on which operations are being carried out. In addition, $\overline{RD}$ process was intentionally implemented in this way in order to allow for the propagation of data after the address is read. In other words, this case imposes a restriction on the active duration of $\overline{RD}$ rather than on the internal clock. $\overline{RD}$ being an external clock has confused the synthesizer in imposing restrictions on the internal clock, or at least the synthesizer has presented a fact which is up to the designer how to interpret it. Secondly, still these are never precise figures as the design deploys two clocks namely clk and clk2x. The low clock figure is likely to be with respect to clk which is already operating at 25 MHz due to its relatively high load and other USB-related factors such as addressing distributed constant arrays (descriptor and register configuration). This means that the DSP process may be able to deploy yet faster clock resulting in more time resolution. Thus more exact figures can be obtained from the "place & route" report in the form of clock skew and delay.

Since it is obvious that Main without a distributed ROM component presents a more faithful reflection of the implemented design, the following tables provide further analysis of "map" and "place & route" reports for Main without distributed ROM. Table 6.3 provides a summary of major metrics extracted from "Map" and "Place & Route" reports. Table 6.4 lists the fanout, skew, and delay associated with both clocks.

Table 6.3: Summary of "Map" and "Place & Route" reports

			Main	
Map Report		SLICES	757	21%
		SLICE Flip Flops/Latches	339	4%
	4 input LUTs	Total	1,374	19%
		used as logic	1,257	
		used as route-through	117	
	Gate count	Total	83,768	
		JTAG for IOBs	3,408	
Place & Route Report		Average Connection Delay	0.991 ns	
		Average Connection Delay for 10 worst nets	3.909 ns	
		Max pin delay	5.052 ns	

Table 6.4: Summary of "Generating Clock" section of the "Place & Route" reports

			Main	
Place & Route Report	`clk`	Fanout	150	
		Net skew	0.153 ns	
		Max delay	0.458 ns	
	`clk2x`	Fanout	49	
		Net skew	0.097 ns	
		Max delay	0.396 ns	

Table 6.4 shows that `clk` has in general larger metrics than that of `clk2x`.

The parametric analysis reveals that significant additional computations can be performed exploiting more than 75% left free resources. However, it was established earlier that free resources are far from being a faithful "linear" assessment of what can be further implemented in the FPGA. The overall resources utilization is a compromise between timing and density (due to limited routability). Yet the availability of dedicated units such as multipliers makes additional DSP computations feasible in their abstract sense without taking into account the associated control and

datapath which have to be implemented using CLBs. Pipleling the datapath can improve timing performance. The Xilinx Core Generator can be deployed in producing a pipelined multiply-accumulate DSP components. Even control FSMs can be improved in terms of timing. The nesting of conditions in a state must be kept vertical in one level whenever possible. Insertion of null conditions in the "else" case reduces accidental latches although ultimately the implementation of FSM depends on the manufacturer template to which the designer must adhere.

6.6 GUI SAMPLE OPERATION

Figures 6.8 and 6.9 shows snapshots of some of the features of the developed GUI.

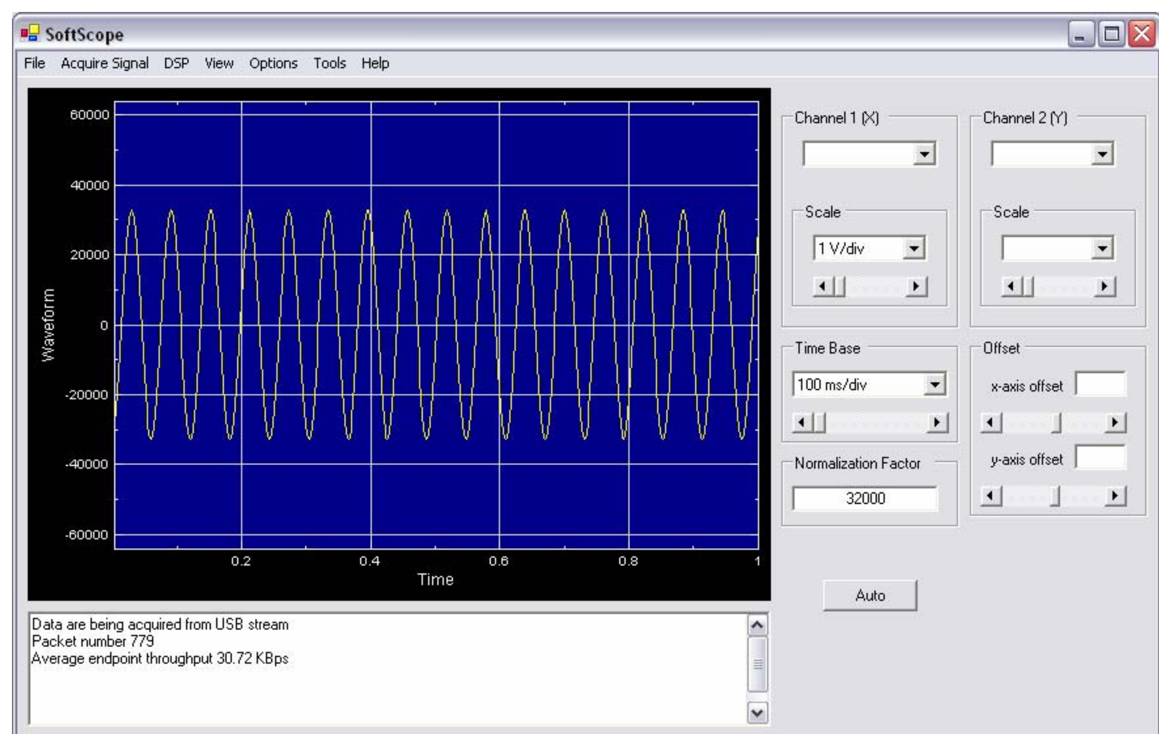

Figure 6.8: A snapshot of the `SoftScope`.

A digitally-synthesized sine wave of 100 KHz sampled at 3.125 MHz is displayed in figure 6.8. `SoftScope` comprises various fields emulating the operations of an actual oscilloscope. Although all presented fields are properly linked to the `GlobalVariables` corresponding variables, only y-scale and normalization factor are activated at the moment. Others are left to be easily included later in the refresh method. For instance, the timebase field (x-scale) was left unconnected because it

essentially depends on the sampling frequency which is application specific. Depending on the sampling frequency, a proper time-increment should be accounted for in the refresh operation. A possible generic situation is to send the sampling frequency as a header field in each packet. The application can decode this field to extract the inverse time-increment. Once the final application parameters are decided these fields can be activated in a very straightforward manner.

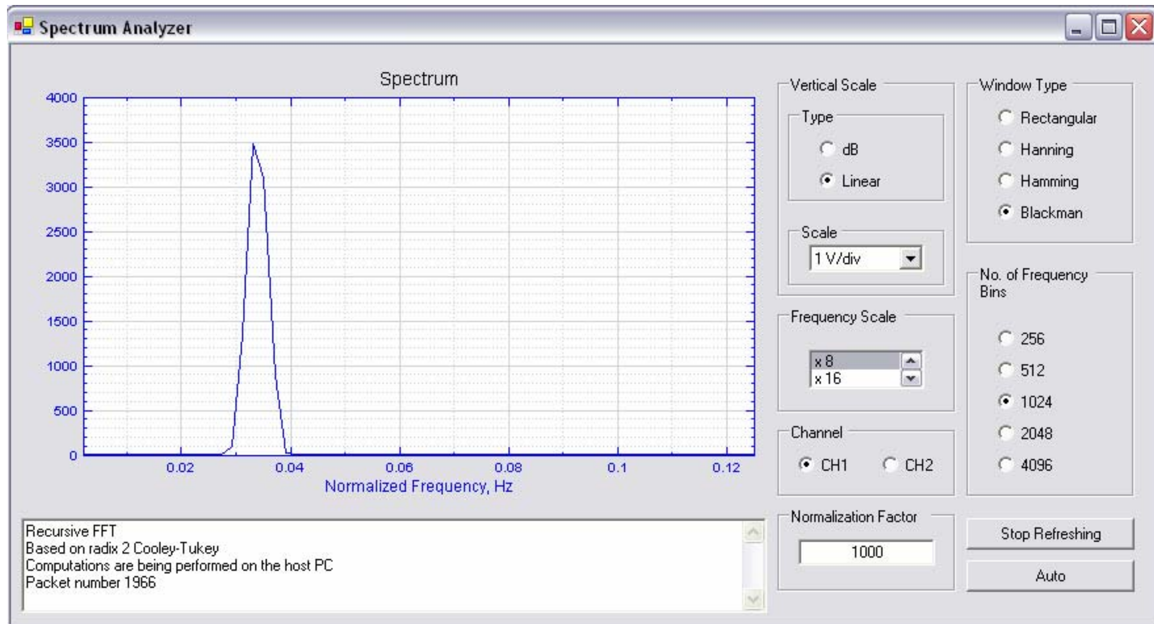

Figure 6.9: A snapshot of the `SpectrAnalyzer`.

In response, figure 6.9 shows a single harmonic residing at a normalized frequency of 0.032 Hz. This corresponds exactly to what is expected as $\frac{100KHz}{3.125MHz} = 0.032Hz$.

This result is obtained with a Blackman windowing function, linear scale, 8x frequency scale, and 1024 frequency bins. Only the channel radio buttons are inactive at the moment as a single channel is currently in operation.

Chapter 7

Conclusions & Future Work

7.1 CONCLUSIONS

Throughout the time span of the project, a software-hardware platform of a digital tomograph with USB PC connectivity was developed. The hybrid architecture is now ready for implementing some serious processing tasks. The combination of a well coupled fine-grained reconfigurable fabric with a floating-point DSP constitutes a skeleton architecture for targeting the high class of the embedded computing applications. The FPGA-controlled USB link not only can communicate results to a user friendly GUI of multi-functionalities, but can facilitate an in-circuit emulation and debugging when used in conjunction with the reconfigurable FPGA. The philosophy of having a master FPGA stems from the desire to control both the USB device and DSP through the use of synchronized concurrent processes and still at the same time offers enough reconfigurable fabric for further customized computations. The Spartan-3 has dedicated multiplier blocks enabling the realization of auxiliary DSP operations on the reconfigurable fabric necessary for algorithmic control tasks. Effectively yielding a system that can be configured and controlled to a large extent from within one design environment which was a goal set right from the beginning during design study phase.

The previous acquaintance of JAVA facilitated a rather rapid migration to C# effectively shortening software development period. VHDL was investigated thoroughly as a means for digital design. The transition between the formally adopted schematic-based approaches to advanced VHDL logic design is a major outcome of this project. VHDL was used not only as a hardware modelling language but also as a simulation tool via VHDL test benches. A methodological hardware-software co-design approach was demonstrated. Vast exposure to USB protocol is in itself an important consequence as USB is expected to be the next predominant solution in the embedded systems environment. Practical FPGA issues such as routing impact were

interesting to be realized away from their ever present literature treatment. The core architecture of the SHARC DSP was studied thoroughly allowing the coding of some classical simple DSP problems. Again previous experience with fixed-point DSPs accelerated this process.

However, due to limited time-scale no major novel hardware implementations were attempted. The relatively large number of tasks tackled in the project did not allow for further investigations and optimizations of some encountered issues such as USB endpoint bandwidth in streaming applications (30.72 KBps per endpoint). With 50 MHz clock, a DSP transfer rate of 22.7272 MBps could be handled successfully with almost 5 clock resolution ratio per $\overline{WR}$ cycle (88 ns per 16-bit word). By deploying faster clock this figure can be easily improved. The unpredictability of hardware behaviour consumed all the advancements made in the project timetable.

Finally, from software programming, hardware design, OS concepts, to artificial intelligence, many aspects from computer engineering were deployed into the successful realization/reasoning of/about this hardware-software platform with a significant element of research.

7.2 PROPOSED FUTURE WORK

Floating-point representation support in the HexConversion class can be added so that floating-point formatted words can be communicated from the DSP and decoded on the PC side for further packing, visualization, or processing.

Exception handling management may be investigated later and added to the software design. Since the software was developed rather quickly, exception handling was not investigated thoroughly. For instance, an out-of-sequence commands might trigger some exceptions (null objects) resulting in the collapse of the software session. Exception handling is a very straightforward issue that only requires time. Constant debugging of the potential causes will eventually yield exception free software.

Since the skeleton architecture is now ready for serious utilization attempt, it may be rational to think about the realization of a relevant embedded computing application. In the tomography context, this could be a full or partial reconstruction technique. Modelling can be done using VHDL test benches as an input to the FPGA master

controller simulating the effects of various system parameters including a DSP stimulus. Therefore, the customized control and computing of the reconfigurable fabric can be verified in abstraction. Furthermore, the DSP share of the algorithmic implementation can be simulated in the VisualDSP++ IDE. Later on, the overall performance of the system can be in-circuit debugged and logged (to PC) through the use of the FPGA in conjunction with the USB device. However, as convenient as this might be, the utilization of this system is associated with a significant learning curve. On one hand, VHDL requires a considerable effort and time before being fully digested as a digital synthetical and simulating tool. On the other hand, for efficient assembly coding, the DSP meticulous architecture has to be fairly researched. The SHARC DSP supports a rather advance comprehensive instruction set that reflects its architecture faithfully. As an example, for efficient coding, instructions such as delayed branch have to be exploited whenever possible in order to optimize the use of the pipeline. As DSP algorithms are centred around iterative loops, coding enhancements as little as a delayed branch instead of ordinary branch are not to be taken slightly in highly demanding processing applications such as the reconstruction problem. An improvement of 0.002% in a given function which is called 10K times/sec yields a 20% system improvement. Various issues such as stack current context save and retrieve are of intimate relation to the underlying architecture. Another example can be the challenging issue of inter-process synchronization whose solution is described in the "Dining Philosophers" problem in one of its forms. These are directly imported from the OS literature. A DSP is essentially an embedded processor which means that if one were to exploit its processing power, classical OS solutions remain perfectly applicable and just scaled down to an embedded environment (processes are relevant to the context of operation rather than generic). Thus it is of no surprise that tackling embedded OS problems is inevitable when programming in assembly or even higher level languages such as C which still requires awareness of such issues before deploying a readily-supplied solutions. Therefore, in order to produce significant results, future projects should account for the significant learning curve associated with the system not to mention the theoretical background of the intended application itself. With these considerations in mind, a minimum of fully dedicated MPhil students are likely to meet these requirements more than MSc students.

The USB PCB high-speed design considerations were not part of the project tasks. As it was concluded that noise might be a contributing factor to the poor USB endpoint bandwidth performance, future investigations of the PCB recommended considerations and shielding are worth the effort. The availability of USB protocol analyzer in USB2.0 projects gives more insight to the various protocol operations such as bit stuffing or handshaking packets. Thus shortening development cycle and reducing speculative blind effort. Further examination of the OS considerations with respect to USB streaming application will aid in explaining the unexpected behaviour of the isochronous transfer type.

Support for a set of requests through the bidirectional endpoint 0 can be added to contribute in the versatility of the overall system. Nevertheless, such tasks require considerable research in the USB protocol.

Replace the dip switch that controls the boot sequence of the DSP with a digitally-controlled one to be used by the FPGA as to in-circuit reprogram the DSP without the need for explicit intervention from the user. The DSP reset switch also is to be replaced by an FPGA pin. Because of the relatively long initialization duration of SRAM-based FPGAs, all on-board chip initializations should proceed that of the FPGA. Thus allowing the FPGA to reset all the on-board chips is a more coherent design practice.

Having such large distributed ROM present in the FPGA might compromise the further sustainable computational tasks. Physically, RAM blocks within the FPGA exist as 18Kbit configurable entities. This particular Spartan-3 has 16 RAM blocks present over two columns. If a sufficient number of blocks were linked together, a fairly large RAM space becomes addressable. Then pointers can be used to refer to a starting address in this RAM space. From the PC side, the DSP loader file can be sent over say four packets and buffered in the available RAM space alleviating the need for a ROM block. Later on, the same space can be used for other purposes and different processes can share a multiplexed RAM space.

Bibliography

[1] Williams, R.A. and M.S. Beck, *Process tomography: principles, techniques, and applications*. 1995, Oxford; Boston: Butterworth-Heinemann. xxv, 581p, 3 – 12.

[2] Holder, D. and o.P. Institute, *Electrical impedance tomography: methods, history and applications*. Series in medical physics and biomedical engineering. 2005, Bristol: Institute of Physics. xiii, 456, 295 – 340.

[3] Hegel, G.W.F. and L.S. Stepelevich, *Preface and Introduction to The phenomenology of mind*. The Library of liberal arts. 1990, New York; London: Macmillan: Collier Macmillan.

[4] Williams, R.A. and M.S. Beck, *Process tomography: principles, techniques, and applications*. 1995, Oxford; Boston: Butterworth-Heinemann. xxv, 581p, 13 – 36.

[5] A. Reader, “Tomography and the Inverse Problems”, Lecture notes DIAS MTP 2004 – 2005.

[6] Holder, D. and o.P. Institute, *Electrical impedance tomography: methods, history and applications*. Series in medical physics and biomedical engineering. 2005, Bristol: Institute of Physics. xiii, 456, 3 – 56.

[7] Dick, C. *Rediscovering signal processing: a configurable logic based approach*. 2003. Pacific Grove, CA, USA: IEEE.

[8] Bilsby, D.C.M., R.L. Walke, and R.W.M. Smith. *Comparison of a programmable DSP and a FPGA for real-time multiscale convolution*. 1998. London, UK: IEE.

[9] *dsPIC30F Data Sheet General Purpose and Sensor Families*, Microchip Technology Inc, 2003.

[10] Langen D., J. C. Niemann, M. Porrmann, and H. Kalte, *Implemetation of a RISC Processor Core SoC Designs – FPGA Prototype vs. ASIC Implementation*. 2002. IEEE proceedings.

[11] L. Fanucci, A. Renieri, C. Rosadini, C. Sicilia, and D. Sicilia, *Generic Sensor Interface for On-Board Satellite Applications*. IEIIT.

[12] Van den Keybus, J., et al. *DSP and FPGA based platform for rapid prototyping of power electronic converters and its application to a sampled-data three-phase dual-band hysteresis current controller*. 2002. Cairns, Qld., Australia: IEEE.

[13] Mann, S., et al. *A flexible test-bed for developing hybrid linear transmitter architectures*. 2001. Rhodes, Greece: IEEE.

[14] Abbiati, R., A. Geraci, and G. Ripamonti, *Self-configuring digital processor for on-line pulse analysis.* IEEE Transactions on Nuclear Science, 2004. **51**(3,

pt.3): p. 826.

[15] Yin-Tsung, H., C. Cheng-Ji, and C. Bor-Liang. *A rapid prototyping embedded system platform and its HW/SW communication interface generation and verification*. 2002. Bali, Indonesia: IEEE.

[16] Sung Su, K. and J. Seul. *Hardware implementation of a real time neural network controller with a DSP and an FPGA*. 2004. New Orleans, LA, USA: IEEE.

[17] http://www.usb.org/. [Retrieved 11 September 2005].

[18] *On-The-Go Supplement to the USB2.0 Specifications*, http://www.usb.org/developers/onthego/. [Retrieved 11 September 2005].

[19] *Wireless USB Specifications*, http://www.usb.org/developers/wusb/. [Retrieved 11 September 2005].

[20] *ADSP-2126x SHARC DSP Core Manual*, Analog Devices Inc, 2004. p2-9.

[21] Ifeachor, E.C. and B.W. Jervis, *Digital signal processing: A practical approach*. 2nd ed ed. 2002, Harlow: Prentice Hall. xxiii, 933p, 121 – 132.

[22] Ifeachor, E.C. and B.W. Jervis, *Digital signal processing: A practical approach*. 2nd ed ed. 2002, Harlow: Prentice Hall. xxiii, 933p, 273 – 301.

[23] NPlot – A charting library from .NET. Copyright © 2003 – 2005 Matt Howlett and others. http://www.nplot.com/. [Retrieved 11 September 2005].

[24] Stallings, W., *Operating systems: internals and design principles*. 4th ed. 2001, Upper Saddle River: Prentice Hall. xviii, 779.

[25] Deitel, H.M., *C#: a programmer's introduction*. Deitel developer series. 2003, Upper Saddle River, N.J.: Prentice Hall. xlix, 862.

[26] Microsoft .NET Framework SDK v1.1 Documentations.

[27] Laplante, P. and o.E.a.E.E. Institute, *Real-time systems design and analysis: an engineer's handbook*. 3rd ed ed. 2004, Piscataway, N.J.: Ieee. 512, 225 – 260.

[28] *ADSP-2126x SHARC DSP Peripheral Manual*, Analog Devices Inc, 2004.

[29] Todman, T.J., et al., *Reconfigurable computing: architectures and design methods.* IEE Proceedings-Computers and Digital Techniques, 2005. **152**(2): p. 193.

[30] Vuletic, M., L. Pozzi, and P. Ienne. *Programming transparency and portable hardware interfacing: towards general-purpose reconfigurable computing*. 2004. Galveston, TX, USA: IEEE Comput. Soc.

[31] *Cypress EZ-USB SX2 Driver for VxWorks User's Guide*, WindRiver Systems, 2001.

[32] *Bulk Transfers with the EZ-USB SX2 Connected to an Intel XScale DMA Interface*, Cypress application note AN052, 2003.

[33] *Bulk Transfers with the EZ-USB SX2 Connected to a Hitachi SH3 DMA Interface*, Cypress application note, 2002.

[34] Trimberger, S. *Effects of FPGA architecture on FPGA routing*. 1995. San Francisco, CA, USA: ACM.

[35] Seokjin, L. and M.D.F. Wong, *Timing-driven routing for FPGAs based on Lagrangian relaxation.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2003. **22**(4): p. 506.

[36] Luger, G.F., *Artificial intelligence: structures and strategies for complex problem solving*. 4th ed. 2002, Harlow: Addison-Wesley. xxiii, 856.

[37] Patel, H., *The 40% Performance Advantage of Virtex-II Pro FPGAs over Competitive PLDs*. Xilinx white paper WP206, 2004.

[38] Iyoda, J. and M. J. C. Gordon, *Higher-Level Hardware Synthesis in HOL.* University Cambridge Computer Laboratory.

[39] Pursley, D. J. and B. L. Cline, *A Practical Approach to Hardware and Software SoC Tradeoffs Using High-level Synthesis for Architectural Exploration*. Forte Design Systems, 2003.

[40] Sanguinetti, J. and D. Pursely, *High-Level Modelling and Hardware Implementation with General-Purpose Languages and High-level Synthesis*. Forte Design Systems, 2002.

[41] Johnson, David, et.al, *Design automation of a receiver: breaking the RTL cycle Time barrier using Behavioral Compiler*. DesignCon98, 1998.

[42] Pursley, D. J., *Using behavioal clustering to improve quality of results for DSP designs*. Forte Design Systems.

[43] *USB 2.0 Specification*, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Chapters 5-9-10.

[44] *Streaming Data Through Isochronous/Bulk Endpoints on EZ-USB FX2 and EZ-USB FX2LP*, Cypress application note AN4053, 2005.

[45] Microsoft USB Isochronous Data Transfer Issues Patch. http://support.microsoft.com/default.aspx?scid=kb;en-us;Q307271. [Retrieved 11 September 2005].

[46] *USB Device Resource*. http://www.guidenet.net/resources/usb.html. [Retrieved 11 September 2005].

[47] El-Sharkawy, M., *Real time digital signal processing applications with Motorola's DSP56000 family*. 1990, Englewood Cliffs, N.J.: Prentice Hall. 398p.

[48] Gaydecki, P. and o.E.E. Institution, *Foundations of digital signal processing: theory, algorithms and hardware design*. IEE circuits, devices and systems series; 15. 2004, London: Institution of Electrical Engineers. xxii, 462 p.

[49] Proakis, J.G. and D.G. Manolakis, *Digital signal processing: principles,*

algorithms, and applications. 3rd ed ed. 1996, Upper Saddle River, N.J.; London: Prentice-Hall International (UK). xv, 968, [48] p.

[50] www.xilinx.com

[51] www.analog.com

[52] www.cypress.com

[53] www.microsoft.com

Other Readings

[1] Navabi, Z., *VHDL: analysis and modeling of digital systems*. 2nd ed ed. 1998, Boston: McGraw-Hill. xxii, 632 p.

[2] Charles H. Roth, Jr, *Digital Systems Design Using VHDL*. 1997, Boston: ITP, 470 p.

[3] Ashok K. Sharma, *Programmable Logic Handbook: PLDs, CPLDs and FPGAs*. 1998, McGraw-Hill.

[4] *ADSP-21160 SHARC DSP Instruction Set Refrence*, Analog Devices Inc, 1999.

APPENDICES

A. VHDL Code

A.1 Main.vhd
A.2 bus_driver.vhd
A.3 DSP_Driver.vhd
A.4 synchronizer.vhd
A.5 RAM_Buffer.vhd
A.6 SX2_Utilities.vhd

B. SHARC Assembly Code

B.1 main.asm
B.2 _initAlgorithm.asm
B.3 _init_timer0.asm
B.4 _algorithm.asm
B.5 PP_CON.asm
B.6 tmr0_isr.asm

C. C# Code

C.1 Acquisition.cs
C.2 Complex.cs
C.3 DefaultInterface.cs
C.4 FFT.cs
C.5 GlobalVariables.cs
C.6 Hex2Dec_Converter.cs
C.7 HexConversion.cs
C.8 RandomAccessBurst.cs
C.9 SoftScope.cs
C.10 SpectrumAnalyzer.cs

A. VHDL Code

B. SHARC Assembly Code

C. C# Code