FPGA Pulse Monitor in a Laboratory Test Environment

A Thesis Presented to

The Faculty of the Computer Science Program

California State University Channel Islands

In (Partial) Fulfillment

of the Requirements for the Degree

Masters of Science in Computer Science

by

Susan Portugal

May 2014

APPROVED FOR THE COMPUTER SCIENCE PROGRAM

5/21/14

Advisor: Dr William Wolfe Date

5/21/14

Dr AndrzejBieszczad Date

5/21/14

Dr Peter Smith Date

APPROVED FOR THE UNIVERSITY

5-21-14

Dr Gary A. Berg Date

Non-Exclusive Distribution License

In order for California State University Channel Islands (CSUCI) to reproduce, translate and distribute your submission worldwide through the CSUCI Institutional Repository, your agreement to the following terms is necessary. The author(s) retain any copyright currently on the item as well as the ability to submit the item to publishers or other repositories.

By signing and submitting this license, you (the author(s) or copyright owner) grants to CSUCI the nonexclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video.

You agree that CSUCI may, without changing the content, translate the submission to any medium or format for the purpose of preservation.

You also agree that CSUCI may keep more than one copy of this submission for purposes of security, backup and preservation.

You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. You also represent and warrant that the submission contains no libelous or other unlawful matter and makes no improper invasion of the privacy of any other person.

If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant CSUCI the rights required by this license, and that such third party owned material is clearly identified and acknowledged within the text or content of the submission. You take full responsibility to obtain permission to use any material that is not your own. This permission must be granted to you before you sign this form.

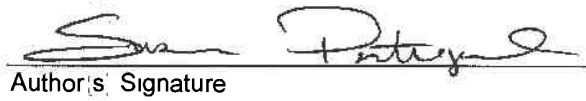
IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN CSUCI, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT.

The CSUCI Institutional Repository will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

FPGA Pulse Monitor In A Laboratory Test Environment
Title of Item

FPGA Pulse Monitor
3 to 5 keywords or phrases to describe the item

Susan Portugal
Author(s) Name (Print)



Author(s) Signature

May 16, 2014
Date

This is a permitted, modified version of the Non-exclusive Distribution License from MIT Libraries and the University of Kansas.

FPGA Pulse Monitor in a Laboratory Test Environment

by
Susan Portugal
Computer Science Program
California State University Channel Islands

Abstract

Manufacturers often resort to Application-Specific Integrated Circuits (ASIC) to meet their specific end-product requirements. Although ASICs are very effective in many applications, they have significant limitations. In particular, designers are often faced with a performance/cost trade-off. To get the performance they desire, designers often come up with costly ASIC designs. This clash between cost and performance has encouraged the evolution of new technology, pushing toward a more flexible, cost-effective, solution. This thesis will demonstrate how the integration of specialized hardware, Field-Programmable Gate Arrays (FPGA), programmed as a pulse monitor for capturing digital signals, and a graphical user interface (GUI) can provide low a cost solution with greater flexibility, increased data collection/storage, improved portability, higher efficiency, and enhanced performance.

Acknowledgements

The author would like to thank the faculty at California State Channel Islands University, employer (Engility Corp.), and family for their support.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: Introduction

Field-programmable gate array (FPGA) is a custom, user-designed, programmable logic chip. Prior to the introduction of the FPGA, the primary computer based application involved an application specific integrated circuit (ASIC). The production and application of these circuits were typically very expensive, very user specific and generally intended for high volume use, due primarily to the limited number of manufacturers. Once manufactured, they were un-changeable, and not subject to correction in the event errors were made during manufacture. However, with the vast expansion of computer based applications, it became increasingly necessary and obvious, that the need for a more user adaptable chip was necessary, in order to meet the demands of the end-user. Thus, the emergence of the FPGA; an integrated circuit that allows a user to customize by programming for their specific need. (1)

The importance of the emergence of the FPGA in today's rapidly evolving computer age cannot be overstated. Nonetheless, as with most systems, the system is only as good as the operator; in this case, the programmer. Integrating appropriate software applications, while utilizing generally common hardware devices, together with a FPGA, can result in a generally very cost effective unit, compatible to a multitude of tasks and environments, that is particularly user friendly, and typically, time efficient.

1.1 Purpose of the Project

In a laboratory environment that performs hardware and systems analysis, it is essential that test equipment, software simulations, and procedures be adaptable to the ever changing requirements that modern technology demands. Deadline driven projects and tight budget constraints fuel the necessity for efficient use of time, money, and physical assets. An FPGA can prove to be a valuable solution to many challenges that arise during the development and testing of new systems. In this case, there was a requirement to monitor multiple signals coming from a system component, perform calculations based on the signals, capture the calculated data in a file, and display the data to the end user. Capturing necessary digital signals, TTLs (Transistor-Transistor Logic signals) data was proving to be very inefficient, costly, and troublesome, in that not all data was being captured. There had to be a better means to capture, and retain necessary data without incorporating more high costs and sensitive equipment (oscilloscopes) to fit the demands of the user. The goal of this project is to use an FPGA as a timely, cost effective, and feasible solution to meet the requirements of the system under test, through the integration of a FPGA (Pulse Monitor).

The project initiated at my work site involves signal data collection. However, due to the proprietary and classified nature of the project, I am not able to discuss the true nature or purpose of this project, nor am I allowed to divulge specific data. Nonetheless, enough information will be discussed to allow for the understanding and project importance.

1.2 Available Solutions

There are three readily available solutions to monitor digital input signals. The first is an Agilent DSOX92004A Infiniium High-Performance Oscilloscope which cost $177,341 as seen in Figure 1. This oscilloscope is a very valid solution; however, there is a limit to the amount of data that is saved due to the scope saving all parametric data of the input signal, and with its stated cost, not a very cost effective solution. In addition to the state limitations, oscilloscopes usually only have two to four maximum inputs.

Figure 1. Agilent DSOX92004A Infiniium High-Performance Oscilloscope

The second solution is a lower cost oscilloscope, Agilent DSO1052B Oscilloscope, Figure 2, which cost $512.

Figure 2. Agilent DSO1052B Oscilloscope

This oscilloscope allows users to view the input pulses; however, there are no means to be able to continuously save all data.

The last available solution is a Spartan 3E Starter Kit which cost $199 as seen in Figure 3.

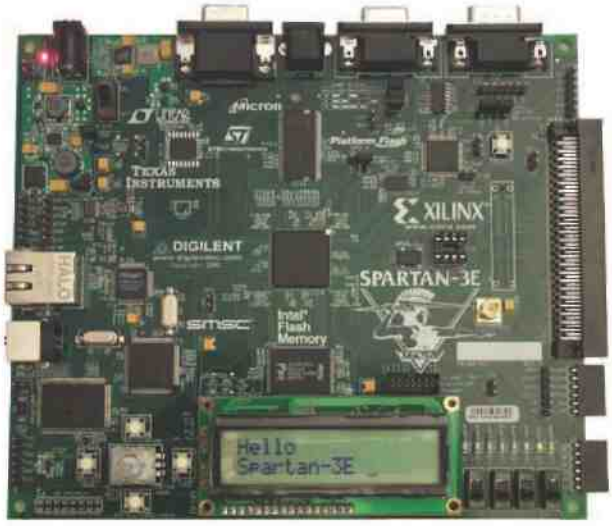


Figure 3. Spartan 3E Starter Kit

This solution was used but, the Spartan 3E Starter Kit was found to be very cumbersome. Input channels were selected by having to slide the switches on the board and required the user to view the data on the LCD panel located on the board. This was a very simple initial approach to solving the project requirements, however due to the size of the FPGA slices, not all requirements could be met.

1.3 History of FPGA

The first commercially available FPGA was invented in 1985. Up until 1985, the bulk of computer chip manufacturers were primarily interested in the mass production of chips. This was a very lucrative industry as the development of the chips was primarily a domestic venture, which dominated the global semi-conductor market. With little competition, cost remained high and out of the hands of many potential users. Low cost Asian manufacturers slowly began entering the market. This got the attention of US manufacturers as they began losing their market share. Simultaneously, with the market beginning to open up to more users, chip consumers began demanding more "specialized chips that could be utilized for specific applications". (2)

Most chip manufacturers were very reluctant to enter this application-specific market. Designing and manufacturing an array of different chips, meant that they could only be marketed primarily to smaller markets, and generally, at a smaller monetary return when compared to mass produced chips. The limited number of manufacturers, coupled with their reluctance to meet the demands of the consumer, resulted in the frustration of these application-specific circuit consumers. The few circuits that were made were often at a high cost and also at the mercy of the manufacturer. If there were defects in the chips, or if the needs of the consumer changed, the wait time to address the issues was often lengthy. This hold up could in turn equate to a loss of millions of dollars due to project stalls. (2)

During this time, Ross Freeman was working as a chip engineer at Zilog Corporation, a subsidiary of Exxon Corporation. Freeman realized that there might be a better way of meeting the need for application-specific circuits. His idea was to develop a sort of "blank computer chip that could be programmed by the customer." This technology later became known as "field programmable gate array" or FPGA.(2) Figure 4 is a Xilinx Spartan 6 FPGA Chip.

Figure 4. Spartan 6 FPGA Chip

Risks associated with faulty chips, could therefore be minimized and, in turn, allow for much greater flexibility for companies designing equipment that incorporated the chips. Freeman, who was a vice-president and general manager at Zilog at the time, approached his superiors and suggested that the development of FPGA devices could be a viable new avenue for Zilog. However, he was unable to convince executives at Exxon that his idea had merit

Freeman later left his position with Zilog. He teamed up with another former Zilog employee, Bernard Vonderschmitt. Together, in February 1984, they founded Xilinx and in 1985, the first FPGA was commercially marketed. The company's system basically consisted of an "off-the-shelf programmable chip and a software package that could be used to program and tailor the chip for specific needs." The technology was based on the arrangement of gates (the lowest level building block in a logic circuit) in complex formations called arrays; as the number of gates increased, the more complex were the functions that the semiconductor could perform.

Computer applications during this time (late 1980's and in the 1990s) continued to increase and with it, the market for application-specific circuits. Subsequently, the market for FPGA chips also increased substantially, resulting in both revenue and profit growth for Xilinx. "Sales rose to nearly $30.5 million in 1988 (fiscal year ended March 30, 1989) before rising to $50 million in 1989. Xilinx posted its first surplus—a net income of $2.92 million—in 1988 and went on to generate profits of $6 million in 1989." (2) Approximately 70% of Xilinx market, at this time, was primarily domestic, and in 1990 sold approximately $100 million worth of its products. Xilinx customers included "Apple Computer, IBM, Compaq, Hewlett-Packard, Fujitsu, Sun Microsystems, and Northern Telecom." Although there were other companies such as Actel and Altera Corporation selling technology that was similar to FGPA systems, Xilinx essentially controlled 100% of the FPGA market during the mid-1990. However, as most innovations breed competition, Xilinx dominance eroded, causing Xilinx to continue to

evolve/improve their technology in the form of introducing different "families" of FPGA chips and software, to "complement low-power applications/low-power devices such as portable and wireless communication gear, and digital cameras." By the end of 1996, Xilinx was employing more than 1,000 workers throughout North America, Asia, and Europe, selling more than 40 varieties of programmable logic products, with revenues of approximately $550 million. (2)

The 1990s were an explosive period of time for FPGAs, both in sophistication and the volume of production. In the early 1990s, FPGAs were primarily used in telecommunications and networking. By the end of the decade, FPGAs found their way into consumer, automotive, and industrial applications. (3) Figure 5 contains an FPGA timetable from *Field-Programmable Gate Array Explained.*

Year	**Market Size**
1985	First commercial FPGA technology invented by Xilinx
1987	$14 million
~1993	>$385 million
2005	$1.9 billion
2010 estimates	$2.75 billion

Year	Gates
1987	9,000 gates, Xilinx
1992	600,000, Naval Surface Warfare Department
Early 2000s	Millions

Year	FPGA Design Starts
1990's	10,000
2005	80,000
2008	90,000
2010	Estimated > 110,000

Figure 5. FPGA Timetable (4)

Chapter 2: Current Applications of FPGAs –Where FPGAs are Used Today

Due to their programmable nature, FPGAs are adaptable to various uses and markets. A small portion of uses or applications of FPGAs today are included in Figure 6 below.

Aerospace and Defense	Automotive	Switches and Routers
Avionics	Digital Signal Processing	Digital Cameras
Space	Image Processing	Printers
Audio	Broadcast	Servers
Security Systems	Medical (Ultrasound, MRI, X-ray, PET, CT Scanner)	Wireless Communications
Radar Systems	Industrial	Surgical Systems

Figure 6. Table of Common FPGA Applications (5)

2.1 FPGA Definitions

The most common FPGA architecture consists of an array of logic blocks (generally called Configurable Logic Block (CLB), or Logic Array Block (LAB), I/O pads, and routing channels. To better understand the architecture, it is important to first define the terminology, listed below found within Zeidman's *Architecture of FPGAs and CPLDs: A Tutorial*.

Term	Definition
Configurable Logic Block (CLB)	Location of function calculations.
I/OB	The block that connects an FPGA to other elements of the application.
Interconnect	Allows for the communication between IOB to CLB. It is the wiring resources in an FPD
Field-Programmable Device (FPD)	Refers to an integrated circuit that can be configured by the end user. Another name for FPDs is programmable logic devices (PLDs).
Programmable Logic Array (PLA)	A small FPD that contains two, programmable levels of logic; an AND-plane and an OR-plane.
SPLD	Refers to any type of *simple* PLD.
CPLD	Refers to a *complex* PLD that consists of an arrangement of

	multiple SPLD-like blocks on a single chip.
Programmable Switch	A programmable switch that can connect a logic element to an interconnect wire, or one interconnect wire to another.
Logic Block	A small circuit block that is replicated in an array in an FPD.
Logic Capacity	The amount of digital logic that can be mapped into a single FPD, typically measured in units of "equivalent number of gates in a traditional gate array."
Speed Performance	Measures the maximum operable speed of a circuit when implemented in an FPD. It is set by the longest delay through any path for combinational circuits and for sequential circuits; it is the maximum clock frequency for which the circuit functions properly.
Lookup Table (LUT):	A lookup table is utilized to change the input data into a more desirable output format. This is used in data analysis, particularly in image processing.
Full Adder	A logic element which operates on two binary digits and a carry digit from a preceding stage, producing as output a sum digit and a new carry digit. It is also known as a three-input adder.
Flip-flop	A circuit that has two stable states and can be used to store state information. It can be made to change state by signals applied to one or more control inputs and will have one or two outputs. It is the basic storage element in sequential logic
Mux (Multiplexer)	A device that selects one of several analog or digital input signals and forwards the selected input into a single line. They are primarily used to increase the amount of data that can be sent over the network, within a certain amount of time and bandwidth. It is also call a data selector. A mux also makes it possible for several signals to share one device or resource.

Figure 7. Key Terms (6)

2.2 FPGA Architecture

Again, to re-iterate, the most common FPGA architecture consists of an array of logic blocks (generally called Configurable Logic Block (CLB), or Logic Array Block (LAB), I/O pads, and routing channels as shown below in Figure 8.

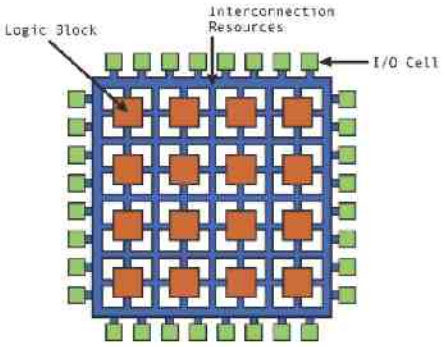


Figure 8. Typical FPGA Structure (7)

While the number of CLBs/LABs and I/Os required is readily determined from the design, the number of routing tracks required may vary considerably. Since unused routing tracks increase the cost (and typically decreases performance) without providing any benefit, FPGA manufacturers try to provide just enough tracks so that most designs will fit in terms of lookup tables (LUTs) and I/Os that can be routed.

In general, a logic block (CLB or LAB) consists of a few logic cells. A typical cell consists of a 4-input LUT, a full adder (FA) and a D-type flip-flop, as shown below in Figure 9. In this figure, the LUTs are split into two 3-input LUTs. In *normal mode* those are combined into a 4-input LUT through the left mux. The selection of mode is programmed into the middle multiplexer. The output can be either synchronous or asynchronous, depending on the programming of the mux.

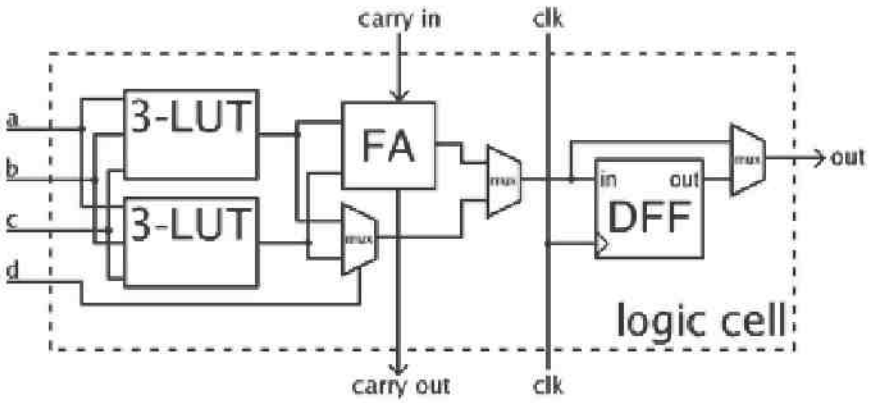


Figure 9. Basic Illustration of a Logic Cell (1)

A look-up table is simply a memory element, shown below in Figure 10.The output is determined by the state of the inputs.

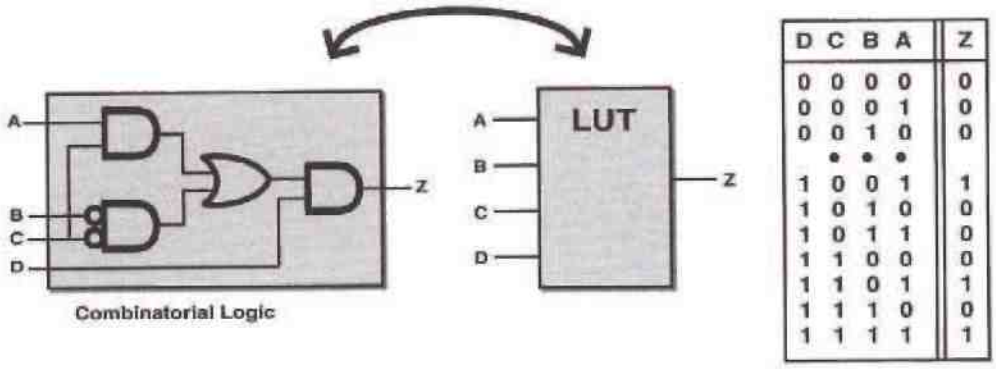


Figure 10. LUT (8)

In recent years, manufacturers have started moving to 6-input LUTs in their high performance parts. Since clock signals (and often other high-fan-out signals) are typically routed by special-purpose, dedicated routing networks, the signals are managed separately.

CLBs are a very basic building block of the FPGA. The CLB is a useful way to describe the size of an FPGA; however, not all CLBs are the same. Figure 11, list three different FPGAs produced by Xilinx with the associated FPGAs on the Xilinx evaluation kits.

- Each CLB contains two slices:
  - A CLB on a Virtex-5 FPGA contains the following:
    - Eight input lookup tables (LUT)
    - Eight flip-flops
  - A CLB on a Virtex-6 and Spartan 6 contains the following:
    - Eight lookup tables (LUT)
    - 16 flip-flops
- ❖ Virtex XC5VLX50T contains (Evaluation Kit $1,195)
  - 7,200 Slices
- ❖ Virtex XC6VLX240T contains (Evaluation Kit $1,795)
  - 37,680 Slices
- ❖ Spartan 6 XC6SLX16 contains (Evaluation Kit $295)

Figure 11. Xilinx CLB Comparison

The I/O block structure contains input/output registers, control signals, muxes and clock signals as illustrated below in Figure 12.

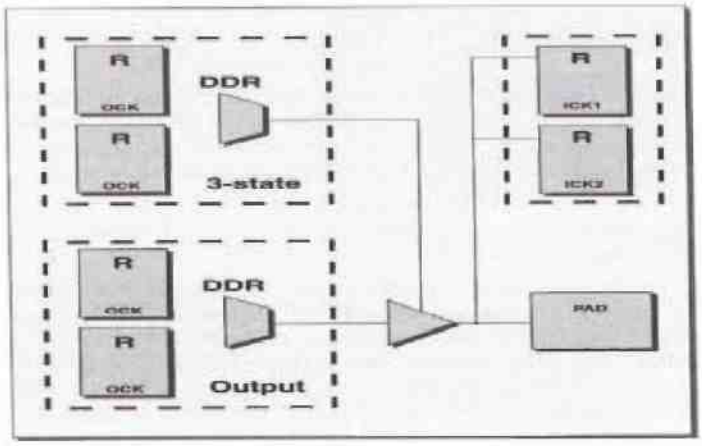


Figure 12. I/O block structure (8)

An I/O Block, as shown above in Figure 12, is used to bring signals into the chip and back off again. It consists of an input buffer and an output buffer with three-state, and open, collector-output controls. Typically there are pull up resistors on the outputs, and pull down resistors that can be used to terminate signals and buses.

The FPGA clock manipulation can be implemented with a phase-locked loop (PLL) or a delay-locked loop (DLL). PLLs generate the desired phase or frequency output by a voltage-controlled oscillator. PLL's are characteristically, analog circuits. DLLs access signals from a calibrated tapped delay line circuit internal to the FPGA to produce the desired clock phase or frequency. DLLs are digital circuits. (8) Figure 13 depicts an overview of a basic FPGA architecture.

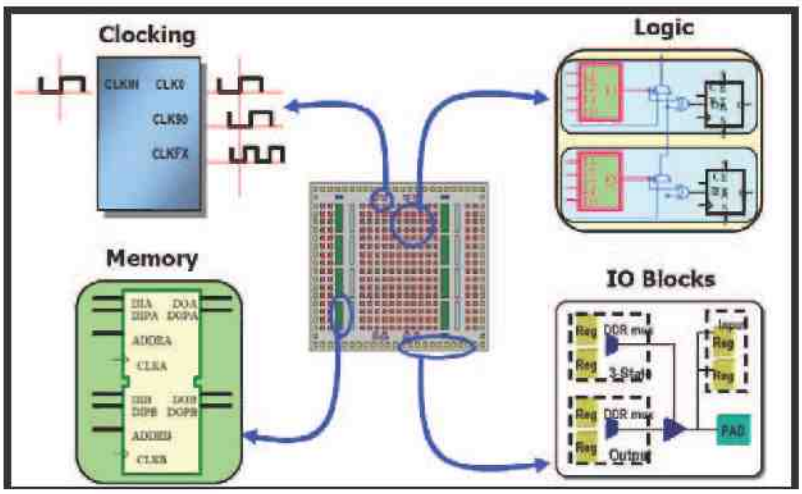
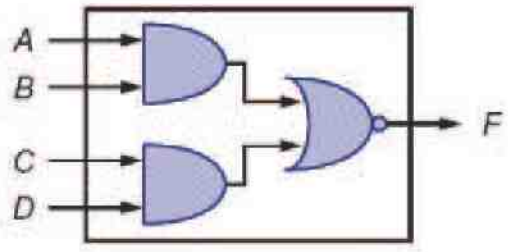


Figure 13. Basic FPGA Architecture (8)

Chapter 3: FPGA Management: Verilog and VHDL Software

The performance or action of an FPGA is typically controlled or managed by a hardware description language (HDL). Presently, there are two widely used standard HDLs—VHDL and Verilog. VHDL was developed under contract for the US Department of Defense as a part of the Very-High-Speed Integrated Circuits (VHSIC) program and subsequently became an IEEE standard language. Verilog was developed by Gateway Design Automation in 1985. Verilog HDL is a hardware description language used to design and document electronic systems. Verilog HDL allows designers to design at various levels of abstraction. It is the most widely used HDL with a user community of more than 50,000 active designers. (9) The difference between the two languages is slight. The choice of language is mainly personal preference, tool availability, and the industry. In the United States, the commercial industries tend to use more Verilog, while the aerospace and defense industries weigh more heavily in favor of VHDL.

Verilog was a proprietary language, but eventually became an IEEE standard language. Another programming language now in use, being utilized with FPGA's has been marketed by National Instruments under the name of LabVIEW. LabVIEW is a graphical programming language that includes an FPGA add-in module, allowing to target and program FPGA hardware. Verilog is the hardware program utilized in the subject project. Figure 14 below is an example of a Boolean expression coded in VHDL and Verilog.



Boolean Expression:	$F = \overline{A \cdot B + C \cdot D}$
VHDL Code 1:	F <= not ((A and B) or (C and D))
VHDL Code 2:	F <= (A and B) nor (C and D)
Verilog:	F <=~((A&B) | (C&D))

Figure 14. Diagram of Boolean Expression Represented in VHDL & Verilog (10)

Both the VHDL and Verilog code segments produce exactly the same hardware representation. (10)

Chapter 4: Pulse Monitor Project Details

4.1 Pulse Monitor Overview

The designed FPGA application project is a pulse monitor. The main goal of the project is to measure the pulse width of a digital (Transistor-Transistor Logic) TTL input signal. A pulse monitor measures pulse width by detecting the rising and falling edge of a digital signal as shown below in Figure 15, and counts the duration of the pulse using the system clock. Using the Ethernet port (10/100/1000 Tri-Speed Ethernet PHY), data is passed between the FPGA circuit board to a GUI software program. (11) For monitoring multiple digital signals, the software program can send commands to the FPGA board to switch between different digital input pins and/or command a reset.

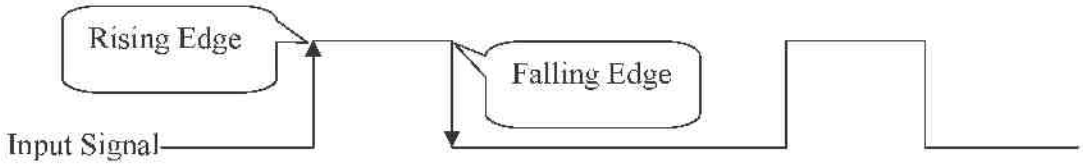


Figure 15. Digital Signal

Research went into the SP601, the FPGA circuit board involved in the project, to see whether or not it could prove to be a solution to cost effectiveness and results. It was found that this low cost solution would be beneficial where a high cost oscilloscope needed to be utilized, and yet would not be able to record data indefinitely, like the Pulse Monitor. Initially, there were several hurdles when using the provided SP601 Base Reference Design tutorial program. The SP601 Base Reference Design provides a User Guide (15) on how to operate the software, and how to download the bit file to the SP601 board, however there is no documentation for the actual Verilog or C♯ code flow. Therefore the user is left to reverse engineer both the Verilog and C♯ code.

4.2 Hardware

This project was developed using a Xilinx Spartan 6 FPGA SP601 Evaluation Kit using the XC6SLX16 CSG324-2C Spartan-6 FPGA logic chip which contains 14,579 logic cells as seen in Figure 16. (12)

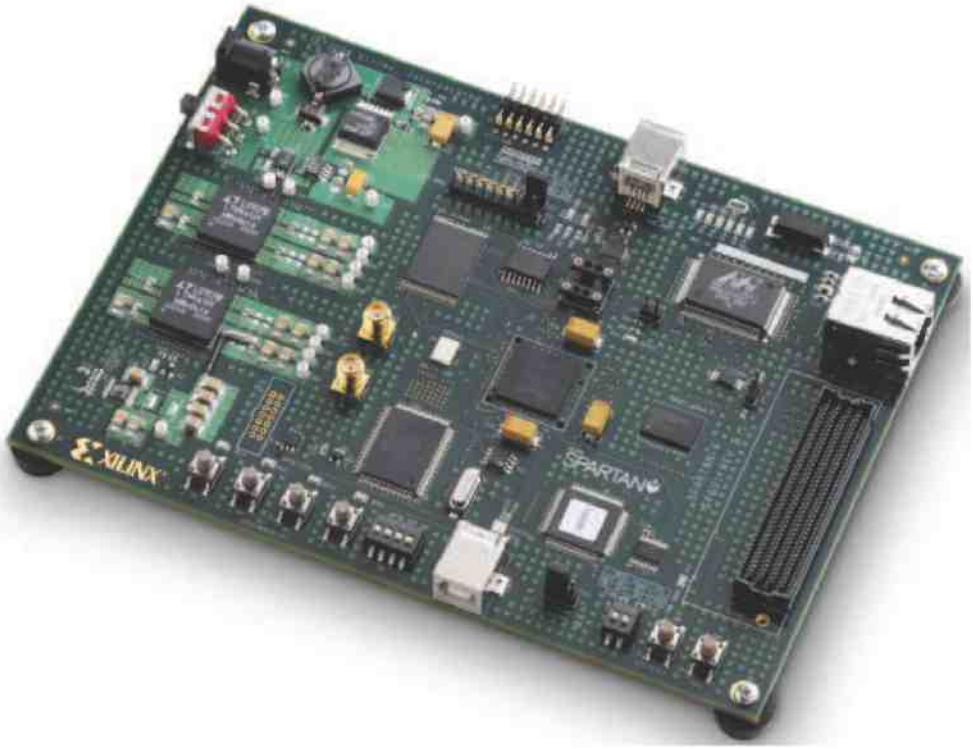


Figure 16. Spartan-6 FPGA SP601 Evaluation Kit (11)

The Spartan-6 FPGA Evaluation Kit was a replacement to its predecessor, the Spartan 3E Starter Kit that contained a XC3S500E FPGA, encompassing 10,476 logic cells. (13)

There were three main reasons why this evaluation board was selected for this project. The first reason was the large number of slices as in comparison to its predecessor, the Spartan 3E. The second reason the Spartan 6 Evaluation Board was selected was due to price of the board at $295, considered a very cost effective solution. And lastly, the Spartan 6 was chosen as it was the latest low cost FPGA produced by Xilinx at the time this project was initiated.

The specifications of the SP601 Evaluation Kit are shown below in Figure 17.

FPGA:
- XC6SLX16 CSG324-2C Spartan-6
- Configuration:
- Onboard configuration circuitry
- 8MB Quad SPI Flash
- 16MB Parallel (BPI) Flash
- JTAG

Memory:
- DDR2 Component Memory 128MB
- IIC 8Kb IIC EEPROM

Communication:
- 10/100/1000 Tri-Speed Ethernet PHY
- Serial (UART) to USB Bridge

Expansion Connectors:
- FMC-LPC connector (68 single-ended or 34 differential user defined signals)
- 8 User I-O (Digilent 2x6 Header)

Clocking:
- 200MHz Oscillator (Differential)
- Socket (Single-Ended) Populated with 27MHz Osc
- SMA Connectors (Differential)

Display:
- 4X LEDs
- Control:
- 4X Push Buttons
- 4X DIP Switches

Figure 17. SP601 Evaluation Kit Specifications (11)

4.2.1 Hardware Code

The pulse monitor hardware program is written in Verilog and the project was designed using Xilinx's ISE Project Navigator. Project Navigator organizes all files within the project (similar to Visual Studio) and processes the project design for implementation for the desired targeted Xilinx device, as in Figure 18 below. (14) Within the Project Navigator, users are able to produce a bit stream to load their design onto the designated FPGA. To generate a bit stream, the Project Navigator steps through a few Key processes. The first process is "Synthesize XST" which synthesizes the project code, checking for errors. The second process is "Implement Design" which verifies the project meets the specified timing requirements and maps and routing occurs. The last process is "Generate Programming File" which is where the bit stream file is created to be loaded onto an FPGA. This project utilized the original Xilinx project tutorial that was created to aid users to get started with the Spartan 601 Starter Kit, as the starting point. Using the basic communication between the FPGA and GUI, the pulse width monitoring was added. (15)(16)

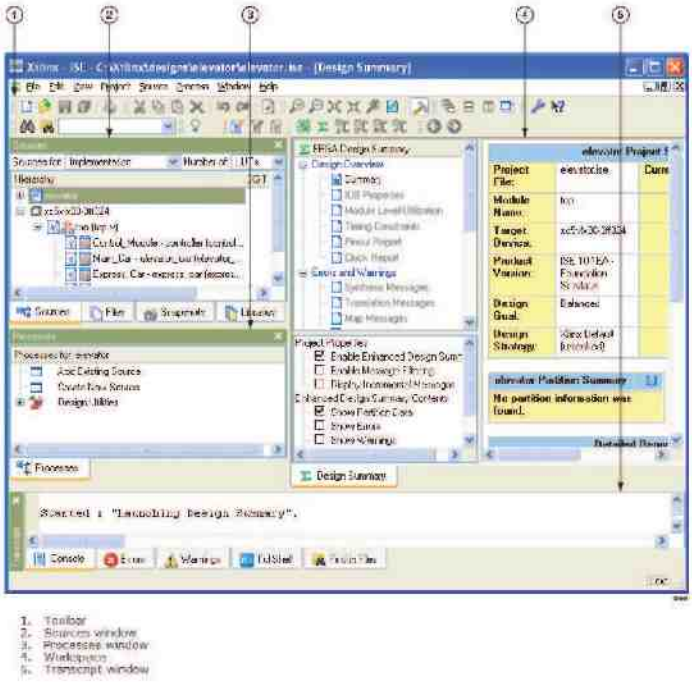


Figure 18. Xilinx ISE Project Navigator (14)

Within the ISE Design Suite, ISE iMPACT software (Figure 19) is used to program the PFGA with the bit stream that was created within the ISE Project Navigator software. ISE iMPACT software allows users to "Initialize Chain," which detects the type of FPGA connected to the computer and therefore load their desired design. This software also allows users to create a PROM file to be loaded into the EEPROM memory. Once the evaluation board is flashed with the created PROM file, the board will power-cycle with the same design.

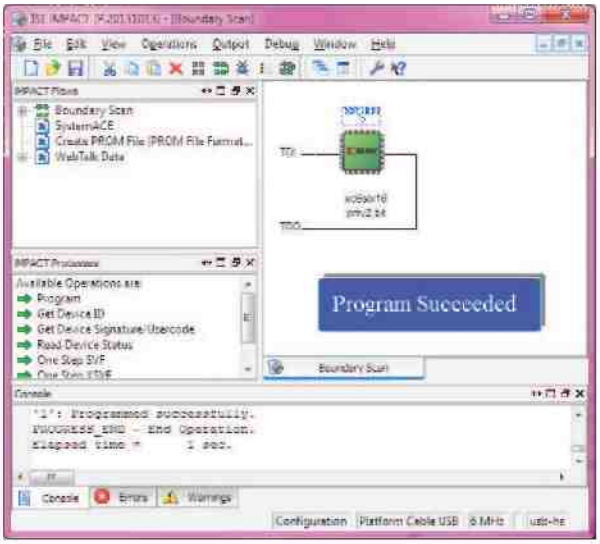


Figure 19. Xilinx ISE iMPACT

The sequential steps begin within the top level, main module of the source code which the pulse width is measured off the positive edge of the 125 MHz input clock. With the aid of two D flip-flops, the input pulse edge is detected. This is shown in Figure 20 below. The first D flip-flop synchronizes the input signal (pulse_in_q1) and the second D flip-flop stage is used for the edge detection (pulse_in_qq1). Next, values are compared for detecting a rising edge or a falling edge of the input signal. If either event occurs, the pulse width 32 bit integer counter value is saved and the counter is reset. Otherwise, if the signal neither rises nor falls, the counter increments. (17)

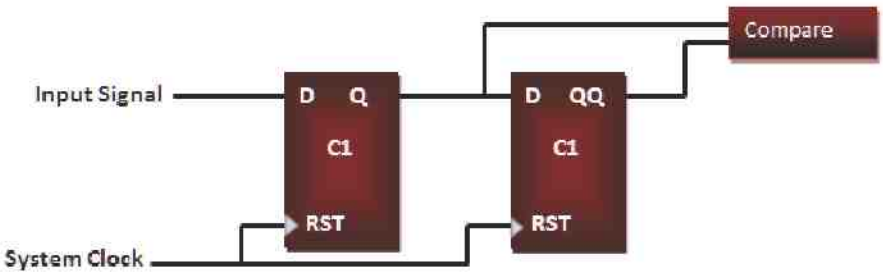


Figure 20. Edge Detection by D Flip-Flops (17)

Figure 21 is a timing diagram example to show how the D flip-flops are used for edge detection. The first black line is the system clock and the second green line is an example input signal. Using the system clock as a rest, the second D flip-flop (red line) holds the value of the input signal and the first D flip-flop (blue line) holds the current value of the input signal. Both d flip-flops are evaluated and set on the positive edge of the system clock. Therefore, as seen in Figure 21, for a rising edge, Q is equal to 1 and QQ is equal to 0 and Q is equal to 0 and QQ is equal to 1 for a falling edge detection.

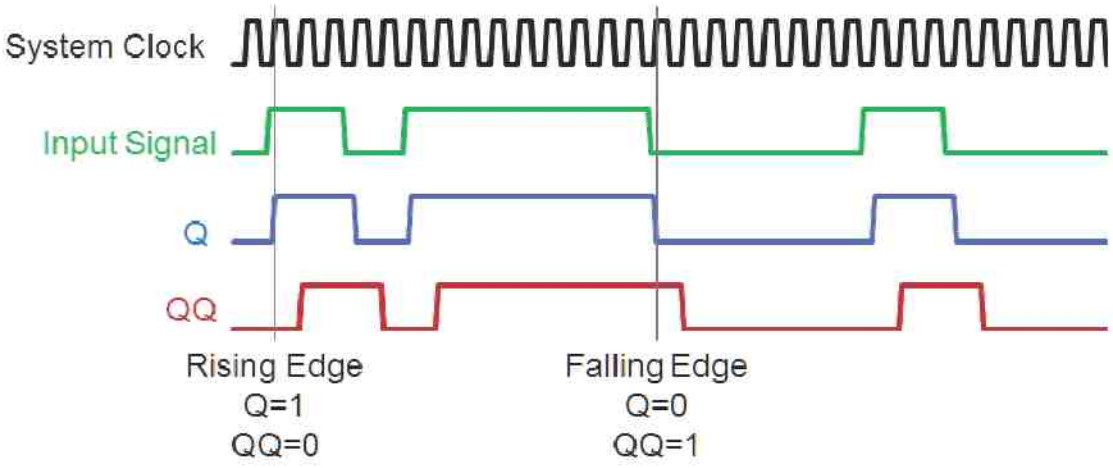


Figure 21. Pulse Monitor Example Input Timing Diagram

4.2.2 Verilog Code

The following Verilog code snippet in Figure 22 below shows the edge detection which is executed every positive edge of the 125 MHz system clock. The first D flip-flop stage is assigned the value of the input signal (INPUT_1). The next line takes in the second stage of the D flip-flop for comparison. After the two stages of the flip-flops acquire their values, the two values are compared to detect either a rising edge, falling edge, or continue counting. If the comparison results in a rising or falling edge, the counter value is saved in one of the three saved counter 32 bit integer. Each reported pulse represented by the counter saves the high value, low value, and system clock counter. The system clock counter is used by the GUI to insure the order of the input signal. To allow high frequency input signals, and remove the possibility of losing data, the Pulse Monitor saves the last three pulses of data for one input channel. Three pulses per input signal were chosen due to the size of the message sent between the FPGA and C# GUI.

```
//Count 1 Pulse Width
always@(posedge clk125)begin
pulse_in_q1 <= INPUT_1;     //1st D flip-flop stage
        pulse_in_qq1<=pulse_in_q1; // 2nd D flip-flop stage
        if (pulse_in_qq1 == 0 && pulse_in_q1 == 1) begin//Rising edge detected
                case(messageoutcount_1[1:0])
                        3:begin
                                count_1_pw_low_reported2 <= count_1_pw;
                                timecounter2_count_1<=timecounter end
                        2:begin
                                count_1_pw_low_reported1 <= count_1_pw;
                                timecounter1_count_1<=timecounter end
                        1:begin
                                count_1_pw_low_reported3 <= count_1_pw;
                                timecounter3_count_1<=timecounter end
                endcase
                count_1_pw <= 0; // Restart counter
        end
        else if (pulse_in_qq1 == 1 && pulse_in_q1 == 0) begin//Falling edge detected
                case(messageoutcount_1[1:0])
                        3:begin count_1_pw_reported3 <= count_1_pw;
                                messageoutcount_1<=1; end
                        2:begin count_1_pw_reported2 <= count_1_pw;
                                messageoutcount_1<=3 end
                        1:begin count_1_pw_reported1 <= count_1_pw;
                                messageoutcount_1<=2 end
                endcase
                count_1_pw_low <= 0; // Restart counter
        end
        else
                        count_1_pw <= count_1_pw + 1; end//End posedge clk125
```

Figure 22. Verilog Code for Edge Detection

The following Verilog code snippet in Figure 23 below shows that depending upon the user's desired channel selection, the user status message array is filled with three pulses of data and a time count for organization on the software side. If the FPGA receives a 0 command, each channel's counter and the overall time counter is reset to zero.

```
case(user_ctrl_en[7:0])//channel selected by user
8:user_status <={count_8_pw_reported1   count_8_pw_low_reported1  timecounter1_count_8,
                 count_8_pw_reported2   count_8_pw_low_reported2  timecounter2_count_8,
                  count_8_pw_reported3   count_8_pw_low_reported3   timecounter3_count_8,
                timecounter};
                                                .
                                                .
                                                .
1 user_status<={count_1_pw_reported1   count_1_pw_low_reported1  timecounter1_count_1,
                count_1_pw_reported2   count_1_pw_low_reported2  timecounter2_count_1,
                count_1_pw_reported3   count_1_pw_low_reported3,  timecounter3_count_1,
timecounter};
0  begin
                count_1_pw<=0;
                count_2_pw<=0;
                count_3_pw<=0;
                count_4_pw<=0;
                count_5_pw<=0;
                count_6_pw<=0;
                count_7_pw<=0;
                count_8_pw<=0;
                timecounter<=0;
                user_status<= {1111,80 d0};
        end
        default user_status<= {1010  80 d0};
endcase
```

Figure 23. Verilog Code for Channel Selection for Output Message

Although Figure 22 and Figure 23 show the main Verilog code that was inserted into the provided tutorial project it took substantial trial and error effort to get to this point.

In addition to detecting the rise and fall of the input digital signal the tutorial program also needed to be enhanced by sending a larger packet of data between the FPGA and the GUI than the original 16 byte per message format. Originally the FPGA sent 16 bytes of data based upon receiving a 1 or 0 from the GUI. The GUI would receive the message and decode the result to be either hello or goodbye. To accommodate the additional data, the message was expanded to 40 bytes.

```
Original:          reg [127:0] user_status=0     // 16 bytes
Pulse Monitor:     reg [319:0] user_status=0     // 40 bytes
```

The main module monitors the input channels and then the data is pushed to the Ethernet packet engine module that fills the FIFO to be sent to the GUI. Just as the main module was modified to accommodate a larger data register of 40 bytes the Ethernet packet engine module was therefore modified to accommodate the new message size. The

receiving buffer on the GUI side also needed to be expanded to accept the new message and decode the message received. If there is a mismatch of message size, either no data will be transferred or the GUI will not be able to handle the message received.

```
reg [567:0]tx_shift=0 // holds up to 71 bytes of packet data
Original: tx_shift<= {SERVER_MAC, MY_MAC 16'd20,
OP_USER_CONTROL+8'h80, {USER_CTRL_EN,USER_CTRL,USER_STATUS},
296'd0};
Pulse Monitor: tx_shift<= {SERVER_MAC, MY_MAC 16'd20,
OP_USER_CONTROL+8'h80 {USER_CTRL_EN,USER_CTRL,USER_STATUS},
80'd0};
```

4.3 Software Code

The Pulse Monitor GUI is programmed in C# with Visual Studio 2010. With the aid of SharpPcap.Packet.winPcap library, the Pulse Monitor sends and filters received User Datagram Protocol (UDP) messages from the FPGA. The Ethernet packet is structured so that it allows for one-to-one communication between the GUI and the FPGA; however, multiple programs can be setup to listen to the messages from the FPGA although only one program sending commands to the FPGA is desired. This also allows for multiple FPGAs to be setup on the same network for multiple nested combinations of communications.

4.3.1 Visual Studio C# Code

The Ethernet packet is structured with an array of data with header and desired commands. The header contacts a MAC address of the FPGA it is communicating to. Therefore, the FPGA only filters in messages sent with its MAC address in the header of the Ethernet message. After the MAC address is filtered, the FPGA responds to the type of message received, such as status or user defined channel. This low level raw Ethernet package, is sent between the GUI and FPGA due to the FPGA lack of a Network Interface Card (NIC). Lastly, the control word designates which channel the user is requesting data from. Figure 24 below defines the Ethernet packet sent from the PC to the FPGA.

```
wPcapDevice.SendPacket(CreateEthernetPacket(xilinxDeviceMac, newbyte[4] {
(byte)OPCODE.USER_DEFINED, control, controlWord[0], controlWord[1] }));
```

Figure 24. C# Send Packet (16)

When a message is received from the FPGA, the GUI insures that the message was received from the FPGA and processes the message depending upon what type of data is received; status or user defined pulse data. Figure 25 below shows how the PC filters the Ethernet packet and sorts by message type.

```
byte[] addressBytes = newbyte[7];
Array.Copy(packet.Bytes, 6, addressBytes, 0, 6);
MACaddr = newMAC(addressBytes);
if (addr Value  = xilinxDeviceMac Value)
return;
if (packet Data[0] == (byte)OPCODE.STATUS_RESPONSE)
ProcessStatusResponse(packet Data);
elseif (packet Data[0  == (byte)OPCODE USER_DEFINED_RESPONSE)
ProcessUserDefinedResponse(packet Data);
```

Figure 25. C♯ Ethernet Filter and Message Type Filter (16)

Once the desired response is obtained, the data from the FPGA is converted to double and multiplied by 8 to convert from 125 MHz count values to time in nanoseconds. The FPGA system clock is 125MHz which means that each clock cycle is: 1/125MHz=0.00000008s=8ns.

The GUI of the Pulse Monitor, in Figure 26 below, allows the user to select a desired input channel. The status of the FPGA will be populated within the Pulse Width Status text box. A connection status is located at the bottom of the GUI to notify users of connection status between the FPGA and the PC. The figure below demonstrates two scenarios; the first is a 10Hz input signal with a 50.0% duty cycle and the second is a 1 kHz input signal with a 15% duty cycle.

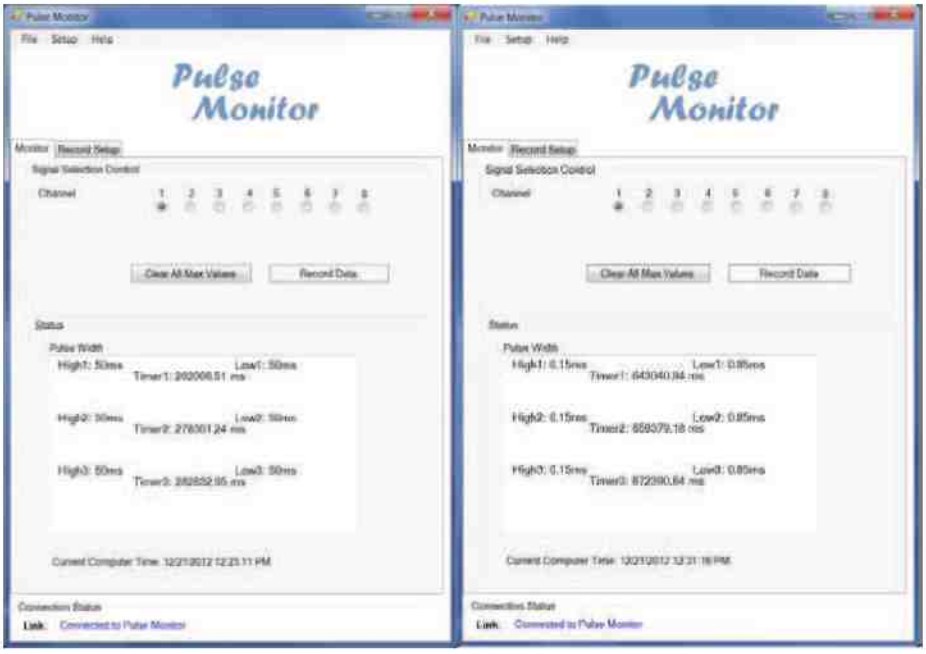


Figure 26. Pulse Monitor GUI

4.4 Pulse Monitor Enhancements

Using the basic building block of the Pulse Monitor, additional versions were created as new requirements were developed based upon the initial release of the Pulse Monitor, resulting in enhanced capabilities and performance. The initial Pulse Monitor monitors eight channels individually and saves only one channel of data at a time. After the Release of the first version of the Pulse Monitor, the following new requirements were developed to meet the needs of the users:

- Monitor eight channels simultaneously
- Allow users to view all eight channels simultaneously
- Save data for up to eight channels simultaneously
- Allow for monitoring differential input signals
- Monitor higher frequency input signal
- Post processing encoded signals

There were three versions of the Pulse Monitors to cover all of the new requirements. The first version improves upon the initial design by allowing simultaneous channel monitoring. The second version also allows simultaneous channel monitoring, and with the addition of the FMCX105 debugs card, allows for differential signal inputs. The third version only allows one channel input, but can sample the input signal at a higher frequency.

4.4.1 Multi-Channel Single Ended Input Signal Pulse Monitor

The initial design of the Pulse Monitor was first enhanced due to the need of monitoring more than one channel simultaneously. A maximum of eight channels was selected to be monitored due to hardware limitations of the FPGA J13 header pin out; 12 I/O (Input/Output) header with the following configuration: 8 I/O header pins, 2 ground pins, and 2 VCC pins. Figure 27 highlights the J13 header pin circled in red.

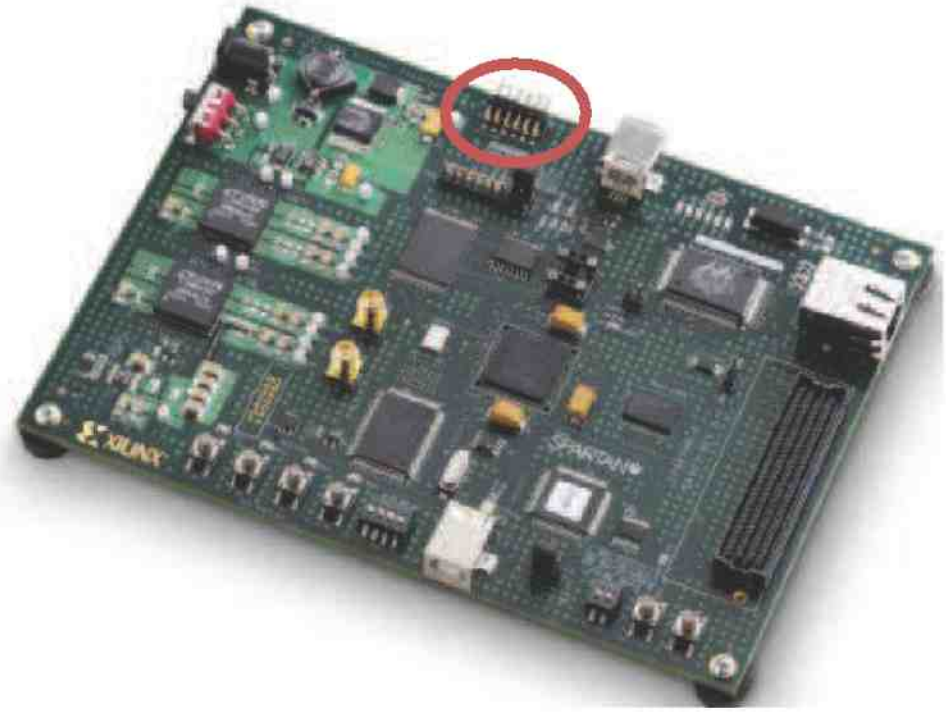


Figure 27. SP601 Evaluation Board with J13 Circled in Red

This design of the Pulse Monitor uses 10,638 slice registers of 18,224 available, which is 58% of the usable amount of slices on the SP601 FPGA. The Verilog code was enhanced in two areas. The user status messages, an array in the top module, was expanded to allow more data to be sent from the FPGA to the GUI, and the FIFO array (register tx_shift), in the Ethernet packet engine module was expanded to fit eight channels of data as seen in Figure 28.

User Status Message Change:

Original Pulse Monitor:

```
reg [319:0]user_status=0;// 40 bytes
```

Multi-Channel Pulse Monitor:

```
reg [319:0]user_status_output1=0;    // 40 bytes
reg [319:0]user_status_output2=0;    // 40 bytes
reg [319:0]user_status_output3=0     // 40 bytes
reg [319:0]user_status_output4=0     // 40 bytes
reg [319:0]user_status_output5=0     // 40 bytes
reg [319:0]user_status_output6=0     // 40 bytes
reg [319:0]user_status_output7=0;    // 40 bytes
reg [319:0]user_status_output8=0     // 40 bytes
```

Total of 320 bytes of data

FIFO Array (tx_shift) Change:

Original Pulse Monitor:

```
reg [567:0]tx_shift=0; // 71 bytes of packet data
tx_shift <= {SERVER_MAC, MY_MAC,16'd20,
OP_USER_CONTROL+8'h80,
{USER_CTRL_EN,USER_CTRL,
USER_STATUS}, 72'd0};
```

Multi-Channel Pulse Monitor:

```
reg [4991:0]tx_shift=0; // 624 bytes of packet data
tx_shift <= {SERVER_MAC, MY_MAC 16'd20,
OP_USER_CONTROL+8'h80,
{USER_CTRL_EN,USER_CTRL,
USER_STATUS1,USER_STATUS2,
USER_STATUS3,USER_STATUS4,USER_STATUS5,
USER_STATUS6,
USER_STATUS7,USER_STATUS8}, 2032'd0};
```

Figure 28. Verilog Changes for Multi-Channel Pulse Monitor

The GUI message handler was also changed within the C# project, to receive a larger packet of data and parse accordingly as seen in Figure 29. The original pulse monitor only allowed for a packet length of 40 bytes of data which was changed to a dynamic length determined upon the received messaged; a minimum of 20 bytes is required to process the message. The dynamic length is also implemented to allow for future growth.

```
privatevoid ProcessUserDefinedResponse(byte[] packet)
    {
if (packet.Length < 20)
return;
Queue<byte> response = newQueue<byte>();
for (int i = 1; i < packet.Length; i++)//originally:for (int i = 1; i <40; i++)
            response.Enqueue(packet[i]);
        UpdateUserStatus(response.ToArray());
    }
```

Figure 29. Visual Studio C# Changes for Multi-Channel Pulse Monitor

The GUI was also changed visually to allow users to view all eight channels of data simultaneously utilizing eight control lists as seen below in Figure 30.

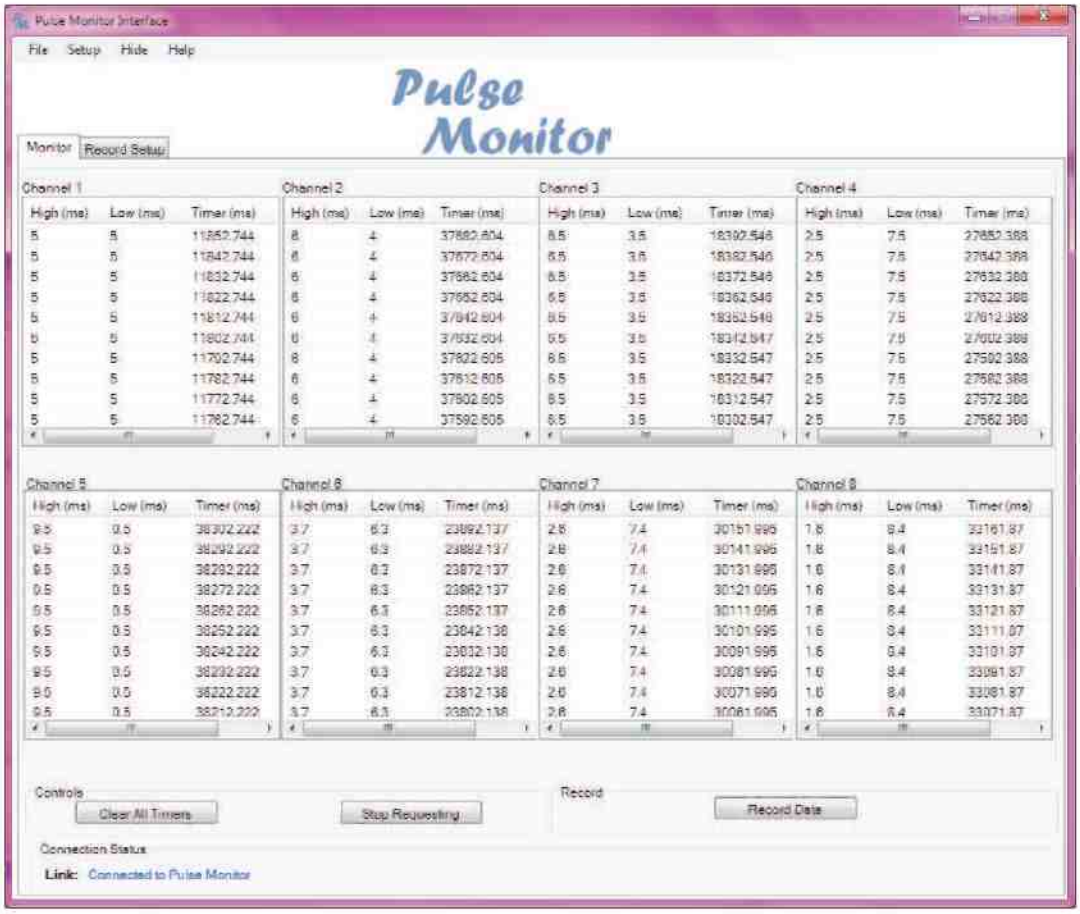


Figure 30. Multi-Channel Pulse Monitor GUI

4.4.2 Multi-Channel Differential Input Signal Pulse Monitor

Another expansion upon the original one channel and multi-channel pulse monitor was the need for monitoring differential signals simultaneously. Differential input signals are very effective in canceling out any outside noise that may be present in the surrounding environment. (18)In implementing eight differential input signals, modifications to both Verilog code and hardware were required. The input signals were remapped to accommodate a positive and negative input per channel for the VITA 57.1 FMC-LPC input connector. In order to connect to the input signals on the VITA 57.1 FMC-LPC input connector a Xilinx FMCXM105 Debug Card seen in Figure 31 was necessary in aiding to the addition of I/O pins that are not easily assessable on the SP601 Evaluation Board. There is only one GPIO HDR (General Purpose Input Output Header) pin-out located on the SP601 Evaluation Board which allows for eight I/O pin connections. This design of the Pulse Monitor uses 10,858 slice registers of 18,224 available, which is 59% of the usable amount of slices on the SP601 FPGA.

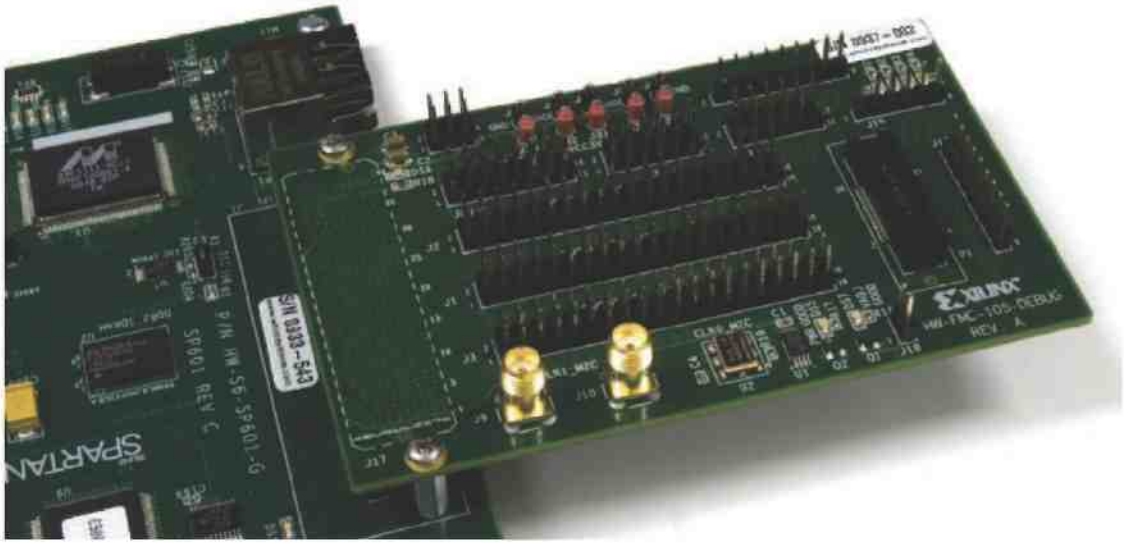


Figure 31. Xilinx FMC XM105 Debug Card (Cost: $159) (19)

Using Spartan-6 Libraries Guide for HDL Designs provided by Xilinx, the following change in Figure 32 was implemented to convert two signal differential inputs to an internal output buffer. Within the IDBUDS, a design level interface signal is represented as two distinct ports (I and IB), one is deemed the "master" and the other is the "slave." For the Pulse Monitor, INPUT_1_P(P for positive) is I, INPUT_1_N(N for negative) is IB, and INPUT_1 is the output buffer. (20)

```
// IBUFDS: Differential Input Buffer
// Spartan-6
// Xilinx HDL Libraries Guide, version 12.4
IBUFDS #(
.DIFF_TERM("FALSE"), // Differential Termination
.IOSTANDARD("DEFAULT") // Specify the input I/O standard
) IBUFDS_inst (
.O(INPUT_1), // Buffer output
.I(INPUT_1_p) // Diff_p buffer input (connect directly to top-level port)
.IB(INPUT_1_n) // Diff_n buffer input (connect directly to top-level port)
);
// End of IBUFDS_inst instantiation
```

Figure 32. Differential Input Verilog Code (20)

In addition to the changes made to implement differential inputs, changes to the User Constraints File (UCF) were required to reroute the input signal channels seen in Figure 33. The ucf file is an ASCII file that defines the project's constraints of the logical design. (21)

Original Pulse Monitor:	Multi-Channel Pulse Monitor:
NET 'INPUT_1' LOC = N17";	NET 'INPUT_1_n' LOC = 'C8";
NET 'INPUT_2' LOC = M18";	NET 'INPUT_1_p' LOC = D8";
NET 'INPUT_3' LOC = 'A3";	NET 'INPUT_2_n' LOC = 'A12;
NET 'INPUT_4' LOC = 'L15";	NET 'INPUT_2_p' LOC = 'B12";
NET 'INPUT_5' LOC = F15";	NET 'INPUT_3_n' LOC = "C6";
NET 'INPUT_6' LOC = 'B4";	NET 'INPUT_3_p' LOC = 'D6";
NET 'INPUT_7' LOC = F13";	NET 'INPUT_4_n" LOC = 'A11";
NET 'INPUT_8' LOC = P12";	NET 'INPUT_4_p' LOC = 'B11";
	NET 'INPUT_5_n' LOC = 'A2";
	NET 'INPUT_5_p' LOC = 'B2";
	NET 'INPUT_6_n' LOC = "F9";
	NET 'INPUT_6_p' LOC = "G9";
	NET 'INPUT_7_n' LOC = 'A7";
	NET 'INPUT_7_p' LOC = 'C7";
	NET 'INPUT_8_n' LOC = P7";
	NET 'INPUT_8_p' LOC = 'N6";

Figure 33. UCF Changes for Multi-Channel Pulse Monitor

The following Figure 34 references the FPGA Pin to the Schematic Net Name which is used in reference Figure 35 to correlate the appropriate pin on FMC XM105 J1Connector.

J1 FMC LPC Pin	Schematic Net Name	U1 FPGA Pin	J1 FMC LPC Pin	Schematic Net Name	U1 FPGA Pin
C10	FMC_LA06_P	D12	D1	FMC_PWR_GOOD_FLASH_TST_E	B3
C11	FMC_LA06_N	C12	D8	FMC_LA01_CC_P	D11
C14	FMC_LA10_P	D8	D9	FMC_LA01_CC_N	C11
C15	FMC_LA10_N	C8	D11	FMC_LA05_P	B14
C18	FMC_LA14_P	B2	D12	FMC_LA05_N	A14
C19	FMC_LA14_N	A2	D14	FMC_LA09_P	G11
C22	FMC_LA18_CC_P	R10	D15	FMC_LA09_N	F10
C23	FMC_LA18_CC_N	T10	D17	FMC_LA13_P	B11
C26	FMC_LA27_P	R11	D18	FMC_LA13_N	A11
C27	FMC_LA27_N	T11	D20	FMC_LA17_CC_P	R8
C30	IIC_SCL_MAIN	P11	D21	FMC_LA17_CC_N	T8
C31	IIC_SDA_MAIN	N10	D23	FMC_LA23_P	N5
			D24	FMC_LA23_N	P6
			D26	FMC_LA26_P	U7
			D27	FMC_LA26_N	V7
G2	FMC_CLK1_M2C_P	T9	H2	FMC_PRSNT_M2C_L	U13
G3	FMC_CLK1_M2C_N	V9	H4	FMC_CLK0_M2C_P	C10
G6	FMC_LA00_CC_P	D9	H5	FMC_CLK0_M2C_N	A10
G7	FMC_LA00_CC_N	C9	H7	FMC_LA02_P	C15
G9	FMC_LA03_P	C13	H8	FMC_LA02_N	A15
G10	FMC_LA03_N	A13	H10	FMC_LA04_P	B16
G12	FMC_LA08_P	F11	H11	FMC_LA04_N	A16
G13	FMC_LA08_N	E11	H13	FMC_LA07_P	E7
G15	FMC_LA12_P	D6	H14	FMC_LA07_N	E8
G16	FMC_LA12_N	C6	H16	FMC_LA11_P	B12
G18	FMC_LA16_P	B7	H17	FMC_LA11_N	A12
G19	FMC_LA16_N	A7	H19	FMC_LA15_P	G9
G21	FMC_LA20_P	N7	H20	FMC_LA15_N	F9
G22	FMC_LA20_N	P8	H22	FMC_LA19_P	N6
G24	FMC_LA22_P	R7	H23	FMC_LA19_N	T7

Figure 34. VITA 57.1 FMC LPC Connections on SP601 Spartan Evaluation Board (Desired pins selected in blue) (21)

FMC HPC Connector J17 Pin	Signal Name	J1 Connector (Odd Pins)	FMC HPC Connector J17 Pin	Signal Name	J1 Connector (Even Pins)
G6	FMC_LA00_CC_P[1]	1	C14	FMC_LA10_P	2
G7	FMC_LA00_CC_N[1]	3	C15	FMC_LA10_N	4
D8	FMC_LA01_CC_P[1]	5	H16	FMC_LA11_P	6
D9	FMC_LA01_CC_N[1]	7	H17	FMC_LA11_N	8
H7	FMC_LA02_P	9	G15	FMC_LA12_P	10
H8	FMC_LA02_N	11	G16	FMC_LA12_N	12
G9	FMC_LA03_P	13	D17	FMC_LA13_P	14
G10	FMC_LA03_N	15	D18	FMC_LA13_N	16
H10	FMC_LA04_P	17	C18	FMC_LA14_P	18
H11	FMC_LA04_N	19	C19	FMC_LA14_N	20
D11	FMC_LA05_P	21	H19	FMC_LA15_P	22
D12	FMC_LA05_N	23	H20	FMC_LA15_N	24
C10	FMC_LA06_P	25	G18	FMC_LA16_P	26
C11	FMC_LA06_N	27	G19	FMC_LA16_N	28
H13	FMC_LA07_P	29	D20	FMC_LA17_CC_P[1]	30
H14	FMC_LA07_N	31	D21	FMC_LA17_CC_N[1]	32
C12	FMC_LA08_P	33	C22	FMC_LA18_CC_P[1]	34
G13	FMC_LA08_N	35	C23	FMC_LA18_CC_N[1]	36
D14	FMC_LA09_P	37	H22	FMC_LA19_P	38
D15	FMC_LA09_N	39	H23	FMC_LA19_N	40

Figure 35. FMC HPC J17 Located on SP601 Board Connection to Connector J1 on FMC XM105 Debug Card Pin Assignments (Desired pins selected in blue) (22)

4.4.3 Expanded One Channel Pulse Monitor with Decoding

Expansion upon the original one channel pulse monitor, was further required for monitoring a higher frequency input. With the expansion of the packet of data sent from the FPGA to the computer as in the simultaneous multi-channel monitoring, instead of receiving the last three pulses of data for eight channels, the FPGA is capable of sending the last 24 pulses for one channel. In addition, it was also found that due to monitoring only one channel, it was more beneficial to send a counter, based upon the input clock, to send the transitions (high-to-low and low-to-high), whereby enabling to capture 54 pulses of data. This version of the pulse monitor is used to capture short, high frequency sequences of pulses where the data is saved and later ran through an algorithm, in order to decode the sequences looking for the desired pulse train sequences. This design of the Pulse Monitor uses 9,260 slice registers of 18,224 available, which is 50% of the usable amount of slices on the SP601 FPGA.

The following code snippet is the logic behind expanding the original one channel pulse monitor to output 54 pulses instead of the original three pulses. This version of the pulse monitor uses the system clock to mark where a transition occurred therefore allowing the GUI to subtract the transition times to reveal the high and low durations of the input pulse. A case statement is used to step through the output registered, in which once a rising edge is detected, the last saved high transition and the following low transition is saved to be sent to the GUI. The last 32 bits of the output registered is used for the GUI to insure that the messages received from the FPGA are in order; seen in Figure 36.

```
//Count 1 Pulse Width
always@(posedge clk125)begin
        pulse_in_q1    <= INPUT_1; //1st D flip-flop stage
        pulse_in_qq1 <= pulse_in_q_1; //2nd D flip-flop stage
if (pulse_in_qq1 == 1 && pulse_in_q1 == 0)begin//Rising edge detected
        case (messageoutcount_1[10:0])
        1:begin
                user_status_output[0:31]<=last_saved_high_trans;// high/low transition
                user_status_output[32:63]<=current_time_count;//low/high transition
                messageoutcount_1<=2; end
        2:begin
                user_status_output[64:95] <=last_saved_high_trans;
                user_status_output[96:127]<=current_time_count;
                messageoutcount_1<=3; end
        3:begin
                user_status_output[128:159]<=last_saved_high_trans;
                user_status_output[160:191] <=current_time_count;
                messageoutcount_1<=4; end
                                                .
                                                .
                                                .
        54:begin
                user_status_output[3392:3423]<=last_saved_high_trans;
                user_status_output[3424:3455]<=current_time_count;
                messageoutcount_1<=1;end
        default:
                messageoutcount_1<=1;
        endcase
        user_status_output[3456:3487] <=current_time_count;
        end
        else if (pulse_in_qq_1 == 0 && pulse_in_q_1 == 1) begin//Falling edge detected
                last_saved_high_trans<= current_time_count;end
else
        current_time_count <=current_time_count +1;end//End posedge clk125
```

Figure 36. Verilog Code for 54 Pulse Edge Detection

The GUI also received changes to accept 54 transition time stamps and calculate to appropriate high and low pulse values. The GUI also handles when the system clock rolls over by, subtracting the system clock max from the transition time and adding the following transition time value.

4.4.4 Pulse Stream Decoding

The GUI allows the user to define pulse streams, as shown in the figure below.

Symbol	Pulse Train(μs)
0	600,-1000,600,-1000
1	750,-400,750,-400
2	400,-1000,1000,-400
END	22000
SEQ	0112

Figure 37. Example Pulse Stream Sequence

The GUI then uses the user-defined pulse stream definition to identify these sequences of interest. The first symbol, 0, has a pulse train of 600 μs high, 1000 μs low, 600 μs high, and 1000 μslow. END indicates that there is a 22000 μs high pulse indicating the end of a sequence. SEQ indicates the desired sequence the output device is inputting to the pulse monitor. For example, sequence 0112 would result in the following pulse train: 600,-1000,600,-1000,750,-400,750,-400,750,-400,750,-400,400,-100,1000,-400 22000.

The algorithm is designed to mark all the locations of END pulses and then looks through the pulses between them for the desired sequence. Using the numbers from Figure 37 as an example, the program compares the first four pulse values to the 0 symbol pulse train and then compares the next four pulse values to the 1 symbol pulse train and so on for this example of sequence of 0112. If the sequence is found, the raw pulse data is outputted with the symbol and SEQ is outputted to indicate that the desired SEQ was found. If the sequence was not found, the algorithm then decodes the pulses looking for matches to the input symbol pulse trains. If no END pulses are located throughout the input file, the algorithm still decodes the symbols. Figure 38 below is a flow diagram of the pulse train decoder.

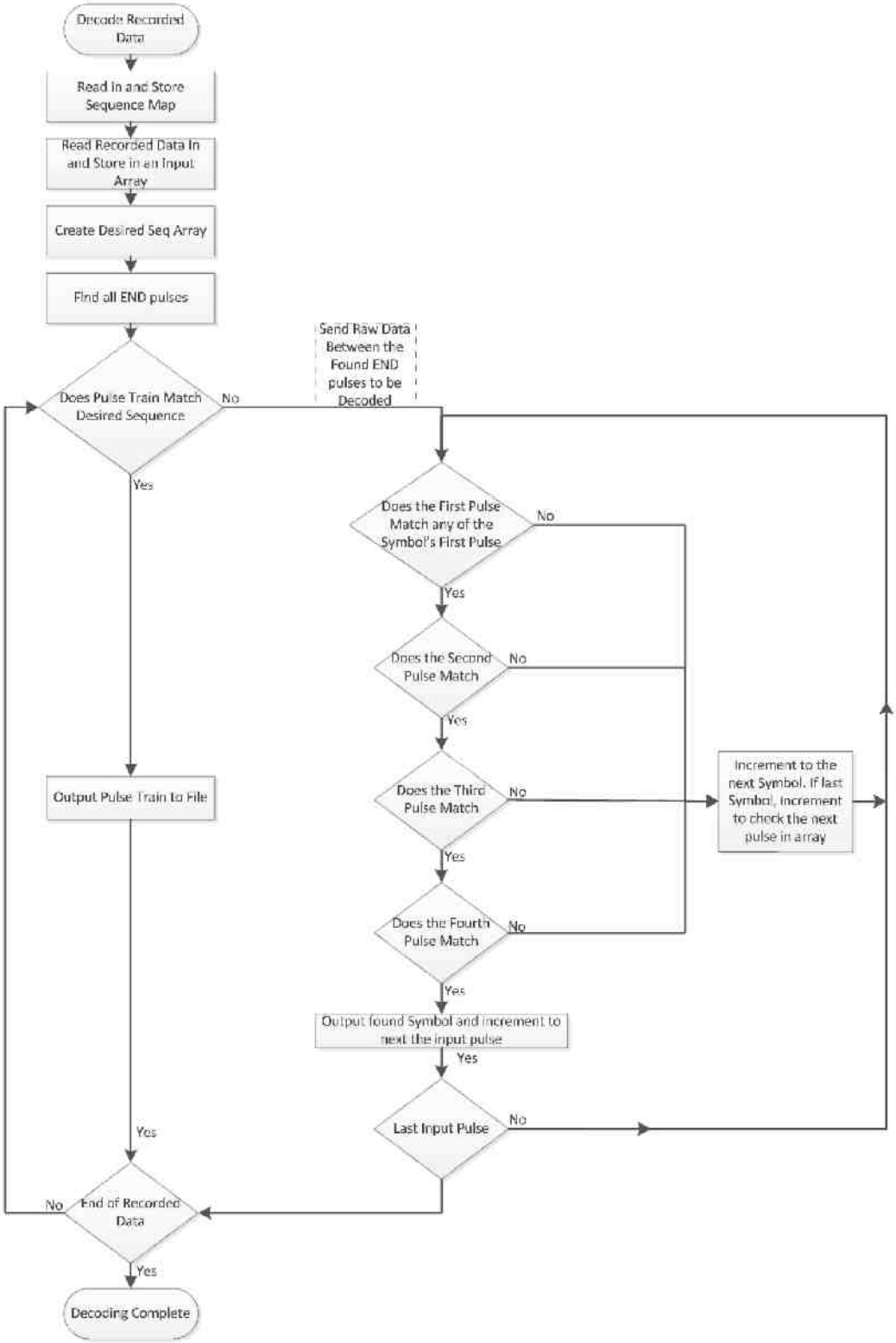


Figure 38. Pulse Train Decoding Flow Diagram

Chapter 5: Experiments

Testing of the pulse monitor was performed using a Tektronix Arbitrary Function Generator model AFG3022B (25MHz) as an input signal to the FPGA. Initially, tests were run on each channel individually to verify correct operation at each input. Next, both output channels from the function generator were used to test simultaneous signals into multiple input channels of the FPGA to test the enhanced Pulse Monitor.

Figure 39 below is an example of an output data file captured by the Pulse Monitor which was capturing a continuous 100 Hz input signal, with a 50% duty cycle with duplicate pulses removed. Due to the FGPA sending the last three pulses saved, the GUI may receive duplicate information; therefore it compares the input messages to the Time and doesn't save data that was already received. The rows highlighted in green indicate that two pulses were received within the same message transferred from the FPGA to the GUI.

Index	Channel	Pulse High(ms)	Pulse Low(ms)	Time(ms)	Message Received Time (ms)
1	1	5	5	8941	19.19
2	1	5	5	8951	23.27
3	1	5	5	8961	31.78
4	1	5	5	8971	45.5
5	1	5	5	8981	51.6
6	1	5	5	8991	72.17
7	1	5	5	9001	72.17
8	1	5	5	9011	81.39
9	1	5	5	9021	99.24
10	1	5	5	9031	103.25
11	1	5	5	9041	111.45
12	1	5	5	9051	127.19
13	1	5	5	9061	131.91
14	1	5	5	9071	141.67
15	1	5	5	9081	154.72
16	1	5	5	9091	161.85
17	1	5	5	9101	181.57
18	1	5	5	9111	181.57

Figure 39. Pulse Data from Output File (100 Hz with 50% Duty Cycle)

Figure 40 is a graphical representation of the data recorded from the Pulse Monitor found in Figure 39.

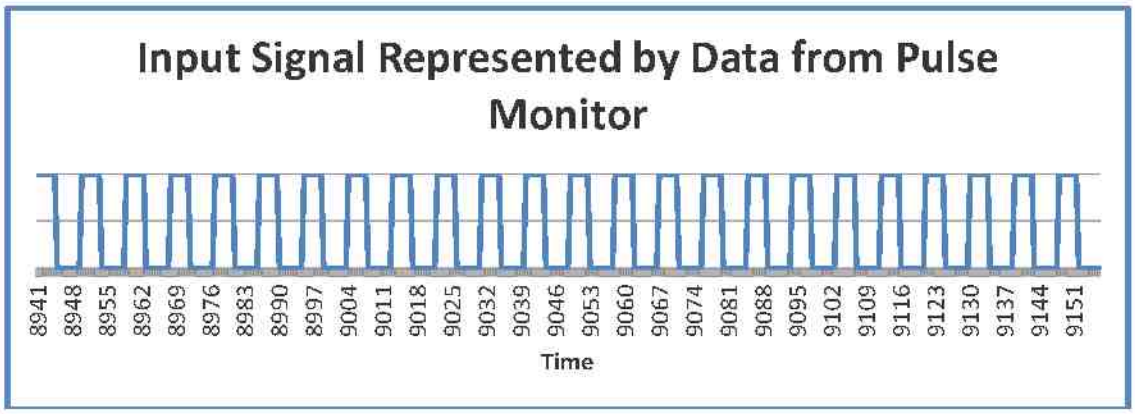


Figure 40. Input Signal Graph (100 Hz with 50% Duty Cycle)

Figure 41 below is another example of an output data file captured by the Pulse Monitor which was capturing a continuous 1 kHz input signal with a 50% duty cycle with duplicate pulses removed. The first three data rows in blue were received within the same messaged at time stamp 4.08ms and the fourth line in gray only shows one data point. There was only one pulse data reported for message received time 4.6 due to duplication in data within the same package of data. The following Figure 41 below shows how the true raw data from the FPGA is sent to the GUI which reveals the limitation of the Pulse Monitor by missing a pulse that should have been recorded at 437987 (highlighted in orange). The pulse was missed due to the high frequency of the input pulse and the approximate 4 ms delay between pulses received from the FPGA. The GUI receives the last three pulses saved by the FPGA which equates to 3 ms for this example. Therefore, if the time between messages received is greater than 3ms data will be missed.

Index	Channel	Pulse High(ms)	Pulse Low(ms)	Time(ms)	Message Received Time (ms)
1	1	0.5	0.5	437966	4.08
2	1	0.5	0.5	437967	4.08
3	1	0.5	0.5	437968	4.08
4	1	0.5	0.5	437969	4.6
5	1	0.5	0.5	437970	5.64
6	1	0.5	0.5	437971	6.73
7	1	0.5	0.5	437972	7.86
8	1	0.5	0.5	437973	8.96
9	1	0.5	0.5	437974	11.1
10	1	0.5	0.5	437975	11.1
11	1	0.5	0.5	437976	11.68
12	1	0.5	0.5	437977	12.75
13	1	0.5	0.5	437978	13.85
14	1	0.5	0.5	437979	14.5
15	1	0.5	0.5	437980	15.73
16	1	0.5	0.5	437983	20.62
17	1	0.5	0.5	437984	20.62
18	1	0.5	0.5	437985	20.62
19	1	0.5	0.5	437986	22.09
20	1	0.5	0.5	437988	25.6
21	1	0.5	0.5	437989	25.6
22	1	0.5	0.5	437990	25.6
23	1	0.5	0.5	437991	26.9
24	1	0.5	0.5	437992	27.51
25	1	0.5	0.5	437993	28.76

Figure 41. Pulse Data from Output File (1 kHz with 50% Duty Cycle)

Figure 42 is a graphical representation of the data recorded from the Pulse Monitor found in Figure 41. The graph is formatted to be similar to Figure 40 to show that 1 kHz is 10 times fast than 100 Hz. The orange arrow also points out the missing data that was highlighted in Figure 41.

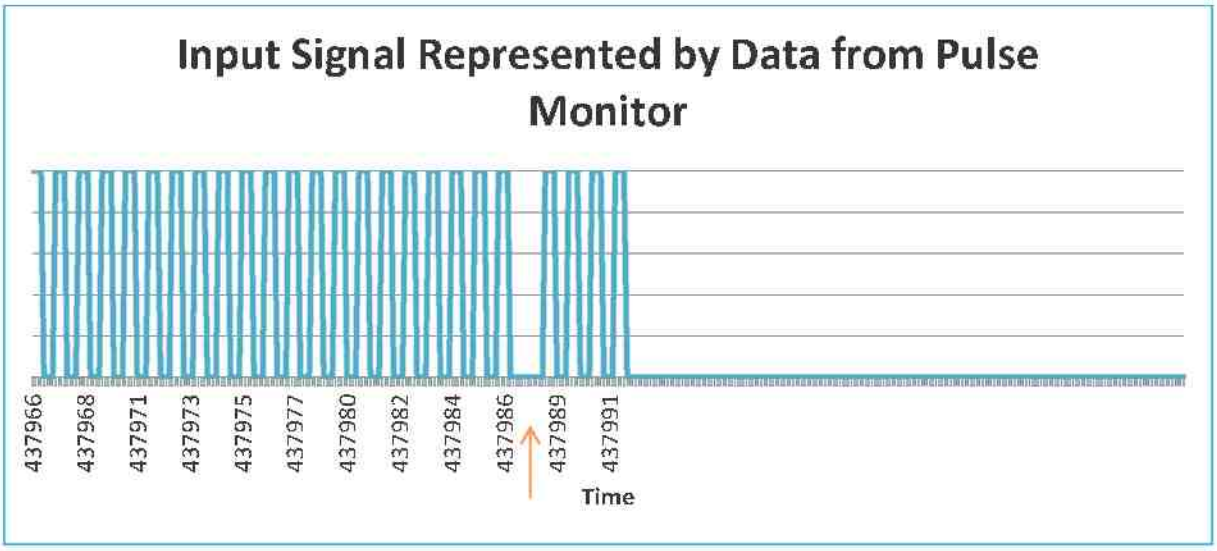


Figure 42. Input Signal Graph (1kHz with 50% Duty Cycle)

Chapter 6: Analysis of Results

Three pulses of data are captured and sent to the GUI upon request from the SP601. Based upon experimental data, it was found that the average time between messages received from the SP601 to the Pulse Monitor GUI, is 30ms over recorded time durations greater than 60 seconds. However, as the data presented in Chapter 5 demonstrates, an average message time received is 10ms for the continuous 100Hz input signal and 1.4 ms for the continuous 1kHz. Due to the process time of the Visual Studio C# code to translate the received 125 MHz clock counter data to milliseconds, record the process data, and also display to the control list, the data packages are buffered and lost once the buffer is full. The GUI has an average 30ms process time, therefore, the Pulse Monitor can correctly detect a continuous 100 Hz input signal; 3 pulses of data per 30ms. The GUI processing time is due to converting each of the pulse width count to milliseconds and displaying each channel to output to a list box. As in Figure 41 (continuous 1 kHz input signal with 50% duty cycle) it was found that with an input signal faster than 100 Hz, some data is captured; however, data is missed due to the processing of the GUI. If the application is only looking for triggering events, then the 125 MHz system clock on the SP601 is the limiting factor. Using the Nyquist Sampling Theorem, the sampling frequency must be twice the input frequency; therefore the Pulse Monitor can detect pulses as fast as 62.5 MHz. (18) However, it was found that for a testing environment with test leads, the nominal Nyquist Sampling Theorem value may not provide enough accuracy. Therefore, the Pulse Monitor was tested and found to be able to detect a 12.5 MHz (80ns) trigger event, which also happens to be the limit of the function generator the Pulse Monitor was tested with, as an input.

With the enhancements to the original Pulse Monitor, the last version utilized a larger message sent from the SP601 to the GUI, to receive 54 pulses of data per message. Based upon experimental data, it was found that it models the original design of processing an average of one message per 30ms and therefore was determined to correctly detect a continuous 1.8 kHz input signal at54 pulses of data per 30ms.

After extensive experiments, it was determined that if the processing of the messages is omitted from the Pulse Monitor GUI, the response time between packages is significantly lowered. With the aid of Wireshark to monitor the Ethernet traffic between the FPGA and the PC, when a Background Worker thread is added to send a request to the FGPA, the average time difference between response messages from the FPGA is 1.5ms. If real-time processing is not required, and the data can be post-processed, with the aid of Wireshark, a pcap file can be saved-- then the Pulse Monitor can monitor a continuous 2 kHz input signal. Therefore, the enhanced Pulse monitor can monitor a continuous 36 kHz input signal.

Figure 43 is a summary of the hardware usage comparisons:

Original Initial Pulse Monitor Design:

This design of the Pulse Monitor uses 6,604 slice registers of 18,224 available, which is 36% of the usable amount of slices on the SP601 FPGA

Multi-Channel Single Ended Input Design:

This design of the Pulse Monitor uses 10,638 slice registers of 18,224 available, which is 58% of the usable amount of slices on the SP601 FPGA

Multi-Channel Differential Input Design:

This design of the Pulse Monitor uses 10,858 slice registers of 18,224 available, which is 59% of the usable amount of slices on the SP601 FPGA.

Expanded Version:

This design of the Pulse Monitor uses 9,260 slice registers of 18,224 available, which is 50% of the usable amount of slices on the SP601 FPGA.

Figure 43. Pulse Monitor Slice Usage

With only using only half of the total number of slices, the Pulse Monitor still has room for future enhancements.

Chapter 7: Conclusion

The goal of this project is to use an FPGA as a timely, cost effective, and feasible solution to meeting the requirements of the system under test, a pulse monitor. Prior to the implementation of an FPGA, the attempt for better data compilation, speed, efficiency and cost efficiency was, at best, unsuccessful and costly. Being dependent upon manufacturers' designed chips, often resulted in errors, time lags, and most of all, frustration of not being able to adjust chips, on-site, to fit the needs of the user. With the introduction of the FPGA in the work site, work specific knowledge and requirements, and a working knowledge of HDL's, in this case Verilog, research and development time has been reduced, has been very cost effective, and most importantly, has not only met the demands of the work site, but has resulted in a better working system. Previous attempts to replicate this capability using an oscilloscope proved to be too time consuming, costly, and failed to meet the requirements of the system. Not only has the pulse monitor system proved to be more than anticipated, it can be readily shared with other users, requiring similar data. In additional to the original design of the Pulse Monitor, changes were implemented enhancing its capabilities and performance.

The original base reference program provided by Xilinx, was a great starting point to gain understanding and reverse engineering to manipulate in developing the Pulse Monitor. However, after analyzing the experimental data and overcoming timing issues within the processing and displaying received data from the SP601, further improvements could be performed to enhance the capabilities of the Pulse Monitor. Though there were several enhancements performed to the original Pulse Monitor, they all contain the same processing time of translating the received data from the SP601 from a 125 MHz count to a millisecond count value and displaying the data to the user on a GUI. Further enhancements to the Pulse Monitor would need to be incorporated such as by removing GUI updates and data translation to run significantly faster. Processing time is only an issue based upon the user's input signal attributes such as speed, continuous wave, burst, or trigger events. In addition to the timing issues of the Pulse Monitor, further investigation and possible recreating the Pulse Monitor may need to be performed to fully understand the discrepancies in the time difference between packages received by the GUI . After gaining such knowledge and familiarity in enhancing the base reference programs provided by Xilinx, new and improved programs could be created to better provide exactly what is required for the desired project. Other avenues for enhancements could be explored to improve the handling of the packet filter instead of the SharpPcap as well as removing the GUI and creating a win32 console application. Overall, the Pulse Monitor has its limitations. Nonetheless, there are several specific applications in which the Pulse Monitor is more than adequate in monitoring digital signals. The Pulse Monitor

is a great project as a stepping stone to obtaining further knowledge in FPGA signal processing, device communication, and data translation and manipulation.

7.1 Pulse Monitor for Other Applications

Typically, capturing and/or displaying signal data, whether it is in the medical, communication, broadcast, etc. environments, is often dependent upon the quality and type of the oscilloscope. However, with the integration of an FPGA, customizing to one's specific use, coupled with a pulse monitor, the cost of an oscilloscope is no longer a determining factor for capturing digital logic signals. The pulse monitor now becomes the efficient, cost saving avenue to accurately capture signals. There are several different applications for the Pulse Monitor in various forms, such as monitoring Global Positioning System (GPS) signals. GPS is a satellite based position and time system. GPS operates at 50 bits per second rate with each complete message transmitted at 12.5 minutes (750 seconds). (24) The Pulse Monitor would be able to detect the pulses and send raw data to the GUI. The GUI would then record the data received from the SP601 and decode the raw data into GPS messages.

Another application for the Pulse Monitor is to detect and decode Pulse-code modulation (PCM) signals that represent sampled analog signals. PCM data is used in many Telemetry systems to monitor on board aircraft sensors.

Chapter 8: Future Work

The Pulse Monitor program is one stepping stone towards future expansion. There are several avenues for expansion beyond the original and enhanced Pulse Monitor such as simultaneously monitoring more than eight channels, enlarging the data message to handle higher frequencies, creating an additional program to process pcap files, adding a playback feature, and/or adding the capability of real time graphing.

Currently the Pulse Monitor can have up to eight input channels for monitoring; this is limited by one of the header pins on the SP601 Evaluation Starter Kit board (J13). If additional hardware such as Xilinx FMC XM105 Debug Card is added, which is used in the enhanced Multi-Channel Differential Input Signal Pulse Monitor, the VITA 57.1 FMC HPC connector has 40 single ended inputs and 20 differential input signals adding an additional 12 or 32 input signals. (19) Differential inputs allows for a more stable input signal if electromagnetic interference (EMI) or radio frequency interference (RFI) is present, canceling out noise in the signal. (20)

Due to the real time processing time of the Pulse Monitor, it may be beneficial to add on to the current Visual Studio C# code to read in recorded pcap files created from Wireshark to post process data. This would allow for higher frequency input signals.

Another future avenue is to allow a playback functionality which outputs a saved pulse train. Currently the pulse monitor only monitors an input signal like an oscilloscope. However, with the flexibility of the SP601 Evaluation Starter Kit board utilizing connector J13 I/O pins, the pulse monitor can act like a function generator or mimic the recorded system.

Another possible alternative addition to the pulse monitor is to add the capability to allow users to visualize the input signal in a graphical form real time, as supposed to the just tabular data or graphing at a later time. Several graphs can be implemented to represent the data such as graphing all the high pulse dwell vs time, or low pulse dwell vs time or even graphing the current duty cycle vs time of any given high/low duration period.

Above are just a few minor expansions for the Pulse Monitor's future, however due to the nature of the FPGA there are many avenues the Pulse Monitor can be expanded.

References

1. Field-Programmable Gate Array. *Wikipedia.* [Online] [Cited: December 5, 2012.] http://en.wikipedia.org/wiki/Field-programmable_gate_array.
2. Xilinx, Inc. History. *Funding Universe.* [Online] [Cited: December 15, 2012.] http://www fundinguniverse.com/company-histories/xilinx-inc-history/.
3. **Mote, Dave and Ingram, Frederick.** Xilinx, Inc. International Directory of Company Histories. 2007. *Encyclopedia.com.* [Online] [Cited: December 21, 2012.] http://www.encyclopedia.com/doc/1G2-3479800102.html.
4. Field-Programmable Gate Array Explained. *Everything Explained At.* [Online] [Cited: December 23, 2012.] http://everything.explained.at/Field-programmable%20gate%20array/#Ref.4.
5. FPGA. What is a FPGA? *Xilinx.* [Online] [Cited: December 20, 2012.] http://www.xilinx.com/fpga/index.htm.
6. **Brown, S and Rose, J.** Architecture of FPGAs and CPLDs: A Tutorial. *Electrical and Computer Engineering Department University of Toronto.* [Online] [Cited: February 13, 2013.] http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf.
7. **Zeidman, B.** The Death of the Structured ASIC. *Chip Design.* [Online] [Cited: April 20, 2013.] http://chipdesignmag.com/display.php?articleId=386.
8. Chapter 2 FPGA Fundaments. *National Taipei University of Technology.* [Online] [Cited: April 20, 2013.] http://www.cc.ntut.edu.tw/~tylee/Courses/System_prototyping_and_HS_design/02_Chapter2_FPGA%20Fundaments-9501.pdf.
9. Verilog Resources. *Verilog.* [Online] [Cited: January 25, 2013.] http://www.Verilog.com.
10. **Serrano, Javier.** Introduction to Field Programmable Gate Arrays. *CERN Accelerator School.* [Online] June 2007. [Cited: January 30, 2013.] http://cas.web.cern.ch/cas/Sweden-2007/Lectures/Web-versions/Serrano-1.pdf.
11. Spartan-6 FPGA SP601 Evaluation Kit. *Xilinx.* [Online] [Cited: December 1, 2012.] http://www.xilinx.com/products/boards-and-kits/EK-S6-SP601-G.htm.
12. Getting Started with the Xilinx Spartan-6 FPGA SP601 Evaluation Kit. *Xilinx.* [Online] March 17, 2011. [Cited: December 2012, 2012.] http://www.xilinx.com/support/documentation/boards_and_kits/ug523.pdf.
13. Spartan-6 Family Overview. *Xilinx.* [Online] [Cited: December 1, 2012.] http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf.
14. Spartan-3E FPGA Family Data Sheet. *Xilinx.* [Online] October 29, 2012. [Cited: December 1, 2012.] http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf.
15. Project Navigator Overview. *Xilinx.* [Online] [Cited: April 20, 2013.] http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_project_navigator_overview.htm.
16. SP601 13.2 and Earlier Documentation. *Xlinx.* [Online] [Cited: December 1, 2012.] http://www.xilinx.com/products/boards/sp601/reference_designs.htm.
17. Synchronization and Edge-detection. *Doulos.* [Online] [Cited: December 20, 2012.] http://www.doulos.com/knowhow/fpga/synchronisation.
18. Single-Ended vs Differential Inputs. *Omega Engineering, Inc.* [Online] [Cited: April 20, 2013.] http://www.omega.com/techref/das/se-differential.html.

19. FMC XM105 Debug Card. *Xilinx.* [Online] [Cited: April 20, 2013.] http://www.xilinx.com/products/boards-and-kits/HW-FMC-XM105-G.htm.
20. **Xilinx.** Spartan-6 Libraries Guide for HDL Designs. *www.Xilinx.com.* [Online] 12.4, December 14, 2010. [Cited: September 16, 2013.] http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/spartan6_hdl.pdf.
21. —. SP601 Hardware User Guide. *www.Xilinx.com.* [Online] 1.7, September 26, 2012. [Cited: Spetember 16, 2013.] http://www.xilinx.com/support/documentation/boards_and_kits/ug518.pdf.
22. —. FMC XM105 Debug Card User Guide. *Xilinx.* [Online] June 16, 2011. [Cited: September 16, 2013.] http://www.xilinx.com/support/documentation/boards_and_kits/ug537.pdf.
23. Evaluating Oscilloscope Sample Rates vs. Sampling Fidelity How to Make the Most Accurate Digital Measurements. *Agilent Technologies.* [Online] November 11, 2012. [Cited: May 4, 2013.] http://cp.literature.agilent.com/litweb/pdf/5989-5732EN.pdf.
24. The GPS System Composition of the Data Signal. *Kowoma.de.* [Online] Kowoma. [Cited: April 20, 2013.] http://www.kowoma.de/en/gps/data_composition.htm.