Capstone Senior Design 2 Final Report

Indiana-Purdue University Fort Wayne
Department of Engineering
May 10, 2013

Project Title

Implementation of a Real-Time Spectrum Analyzer

Team Members

James Darabi
Chinmayi Avasarala
Tracy Cline

Faculty Advisor

Dr. Todor Cooklev

Sponsored By

JDSU

Table of Contents

Section 1

1.1 Acknowledgements

First and foremost, the team would like to thank JDSU for sponsoring this project. Specifically, we want to extend our gratitude to Ben Maxson, Adam Gray, and Dan Chappell from JDSU for taking the time to work with us as their experience and expertise proved to be invaluable.

We also want to thank our faculty advisor, Dr. Todor Cooklev, for providing us with guidance and support throughout the entire design process. In addition, we want to thank Dr. Wang for providing technical documentation and assistance.

Section 2

2.1 Abstract

Spectrum analyzers support research and development, testing, and design in countless applications. JDSU has requested the design and implementation of a real-time spectrum analyzer (RTSA) to assist field technicians with troubleshooting cable television networks. The RTSA requires a processing bandwidth of 85 MHz. The analog input signal will be digitized with a 12-bit ADC at a sampling rate of 204.8 MSPS which is equivalent to a sampling period of approximately 4.9 ns. The digitized values will be accumulated in a 12-bit by 1024-sample buffer. Initially, this buffer will take 5 µs to accumulate 1024 samples at which point the first 1024-point FFT will be triggered. In order to avoid information loss in the spectrum being measured, a 1024-point FFT will be triggered every 2.5 µs to provide a 50% overlap in the resulting output. FFT output values will be quantized to the nearest magnitude value. Finally, the quantized values will be output to a host PC for display and evaluation. A serial output buffer will be designed such that 0.1 to 1 second of data can be accumulated. An FPGA-based solution will be realized which meets the high performance design specifications provided by JDSU while minimizing cost.

Section 3

3.1 Background

Spectrum analyzers support research and development applications ranging from radio frequency identification and radar to radio frequency component development (amplifiers, synthesizers, etc.) and transmitter measurements. Spectrum analyzers are also dominant tools in wireless testing, including cellular systems and device design and manufacturing. Put simply, spectrum analysis is the core tool for evaluating RF signal characteristics.

Signal interference negatively affects transmission coverage and mobile capacity, limiting the overall network performance. Unavoidable signal interference is becoming more prevalent in the wireless community with an increasing number of active transmitters in the RF spectrum. The presence of many signals is making a very complicated environment which must be routinely monitored in order to maximize service performance. Wireless service providers have traditionally used spectrum analyzers to monitor the performance of their transmission signals as well as the presence of adjacent frequencies in order to identify possible sources of interference. Unfortunately, the use of conventional spectrum analyzers has some limitations such as an inability to identify intermittent signals.

As new RF applications are emerging, the RF signal is evolving in its characteristic and complexity, becoming more intermittent, shorter in duration and more variable in frequency than the RF signals of the past. It has become a necessity to capture all possible information in a signal for better precision and accuracy in analyzing signals. As such, today's RF signals are more difficult to analyze with a traditional swept spectrum analyzer, which is limited in its digital modulation analysis and multi-domain capabilities.

When analyzing electrical signals it is often advantageous to view the signal in the frequency domain as opposed to the time domain. There are two basic approaches to obtaining and visualizing the frequency spectrum of a signal. The traditional swept approach and the Fast Fourier Transform FFT approach.

3.1.1 Swept-Tuned approach

The traditional approach was to sweep through the frequency band of consideration and measure the power of the input signal over that band. The results are displayed on a frequency verses power plot. This is an analog approach and has a couple of drawbacks. There is a slice of time when the internal frequency oscillator is returning from the high end of the frequency band to the low end that the input signal is not monitored. Also there is no memory from one sweep to the next. Either of these drawbacks may result in short duration signals being missed.

3.1.2 Pure FFT approach

Another approach is to use a Fast Fourier Transform FFT based approach. To understand how this is accomplished one needs to understand the Discrete Fourier transform DFT since the FFT is just a fast

algorithm thereof. The DFT transforms a sampled signal in the time domain into the frequency domain. The input signal is sampled at a fixed time interval (T) called the sampling period. The sampling frequency (f_s) is the reciprocal of the sampling period. A fixed number of samples (N) are collected into a sampling frame and the FFT is calculated. The result is a sequence of N complex numbers describing the input signal in the frequency domain.

There are constraints on the values of f_s and N. First, the sampling frequency must be high enough to cover the input signal's bandwidth (BW), or at least that portion of the signal's bandwidth that is of concern. The minimum sampling frequency is defined by Nyquist to be $f_s > 2BW$. Second, N is constrained by the desired frequency resolution and required sampling frequency. Frequency resolution is f_s/N. To increase the measured signal's frequency resolution, one may decrease sampling frequency. However, sampling frequency is already constrained by the input signal's bandwidth. The only other variable is N. One can do zero-padding (adding 0's to a sequence) to increase the value of N.

When sampling a signal in the time domain the resulting DFT resolves the signal frequency components to the span $0 – 1/2f_s$. If the input signal has frequency components that are greater than the Nyquist frequency, they will be aliased somewhere into this band. In order to eliminate aliasing, a filter must be employed to remove any frequencies outside the desired bandwidth. A near perfect window can be implemented using digital filtering techniques.

To derive the power of the spectrum, the absolute value of the FFT must be squared and multiplied by $2/N^2$. Spectrum analyzers typically display data in units of dBm. Units of dBm are calculated by $20 \log_{10} P$ where P is power in mW. Some limitations to the purely FFT based approach:

- Bandwidth capability is limited by the sample speed and calculation speed of the underlying hardware.
- There is still a blind spot that occurs between sampling and computation.
- Due to windowing techniques used to minimize the phenomenon known as leakage, a signal's level may appear reduced if it appears at the edge of an FFT's sampling frame.

3.1.4 Real-time Approach

This approach alleviates the problems associated with the FFT. The real-time approach uses FFT frame overlapping to minimize the effects of a signal occurring at the edge of one FFT frame. To eliminate blind spots, the real-time spectrum analyzer is continuously accumulating sample points into the FFT sample space. FFT calculations are computed at the calculation rate with overlapping sample frames. Since sampling, computation, and output display are performed continuously, the chance that a short duration signal will be missed is almost eliminated.

Successive FFT frames may be retained and compared to other frames over time allowing a changing signal to be visualized. Features such as persistence and spectrogram are common. Persistence is a

feature that fades an older signals effect over several screen updates instead of immediately discarding on the next screen update. The spectrogram is a 3D representation of power, frequency, and time.

3.1.4 MATLAB Implementation of the FFT

Computing the discrete Fourier transform according to the formal definition is not efficient as it requires □(□^{2}) arithmetic operations to be performed. The most commonly used fast Fourier transform algorithm, the Cooley-Tukey FFT algorithm, can reduce the required arithmetic operations to □(N log □) providing a significant performance advantage while producing the same results as the DFT.

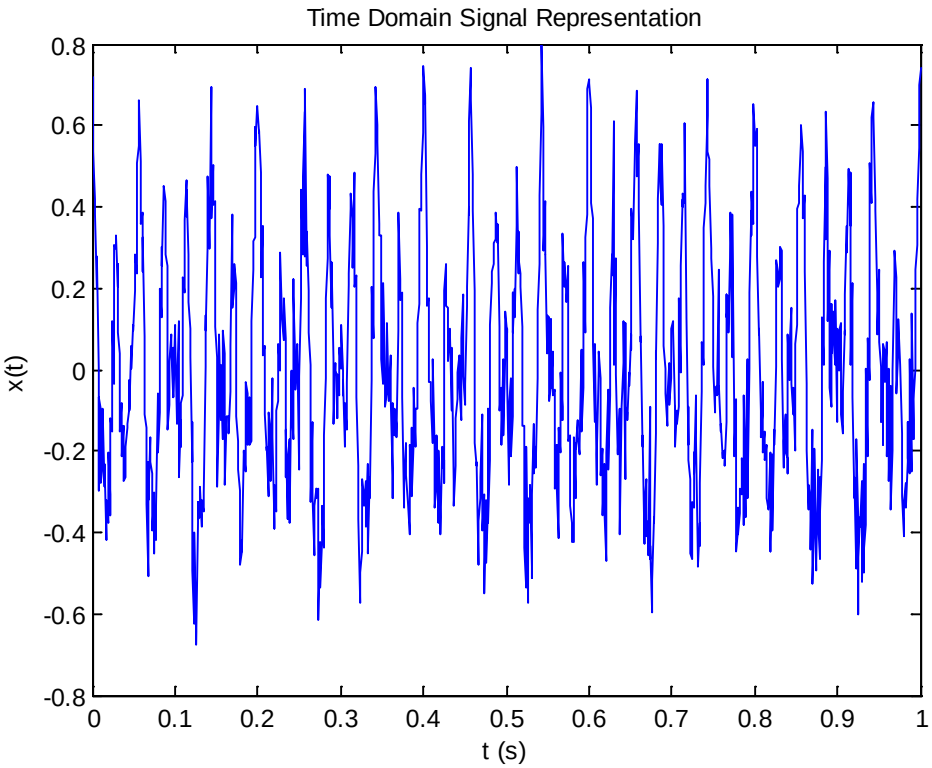


Figure 1: Time Domain Signal Representation of 3 sinusoids of frequencies 20, 35, and 70 Hz with amplitudes of 0.2, 0.15, 0.3 respectively in the presence of white Gaussian noise.

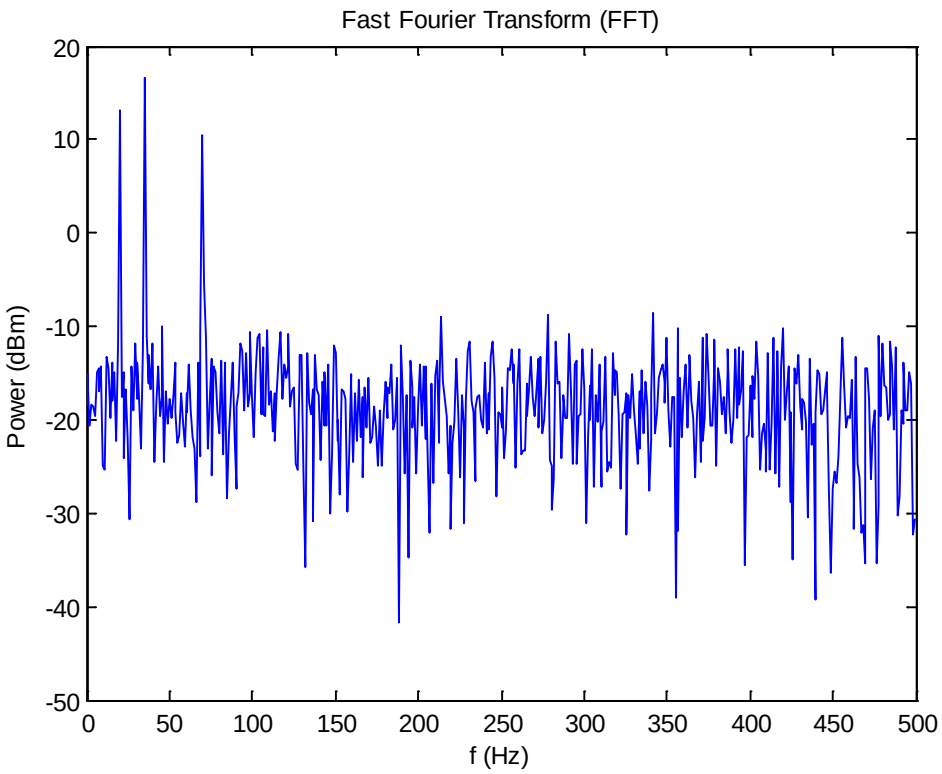


Figure 2: Fast Fourier transform (FFT) of 3 sinusoids of frequencies 20, 35, and 70 Hz with amplitudes of 0.2, 0.15, 0.3 respectively in the presence of white Gaussian noise.

Section 4

4.1 Problem Statement

4.1.1 Requirements and specifications

- The signal overlap should be at least 50% Overlap.
- The FFT output shall be at least 256 bins, but no greater than 1024 bins.
- The accumulation period shall range between 0.1 seconds to 1 second, after which, the frequency count resulting at the end of the accumulation period should be output to a host PC for display and interpretation.
- Processing Bandwidth: 85 MHz processing bandwidth must be used for implementing the RTSA.
- The project has a budget of $3000.

4.1.2 Given parameters

The given, or fixed, parameters are those that will be the guidelines for the design of the system. Some of the given parameters include:

- Sampling Rate: 204.8 MSPS
- The RF Front End is provided by JDSU, and it cannot be altered.

4.1.3 Design Variables

The design variables include components in hardware and software. The hardware includes the FPGA boards and its configurations. It may also include analog interface to the ADC. The software includes the GUI component that needs to be developed for user interface.

4.1.3.1 Hardware Alternatives

- Field Programmable Gate Array (FPGA) board with an ADC
    - Number of Logic Elements
    - Number and Size of Embedded Multipliers
    - Size of Embedded Memory
    - Number of I/O Pins
    - ADC Resolution and Number of Channels
    - Clock Speed
    - Incorporating an built-in FFT algorithm
- Digital Signal Processor (DSP)

- Arithmetic Format
- Data Width
- Speed
- Memory Organization
- Ease of Development

4.1.3.2 Software Alternatives

- GUI interface
  The 3 variables that need to be represented in the GUI interface are:
    - Power (dBm)
    - Time (sec)
    - Frequency (MHz)

The representation can be done in various ways. One possible alternative is the heat diagram. Heat diagrams plot power in terms of frequency (x-axis) and time (y-axis). The color represents the amount of power of the input signal. An example for the heat diagram is shown in Figure 2 below.

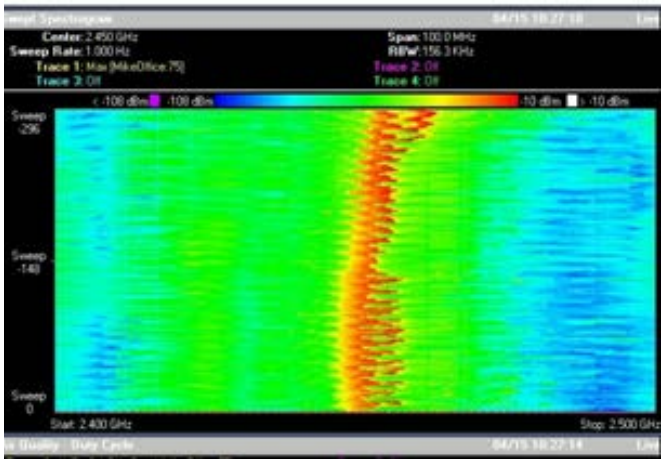


Figure 3: Sample heat diagram

Other representations include 2D plots of frequency versus power as shown in Figure 3 (left) or time versus power as shown in Figure 3 (right).

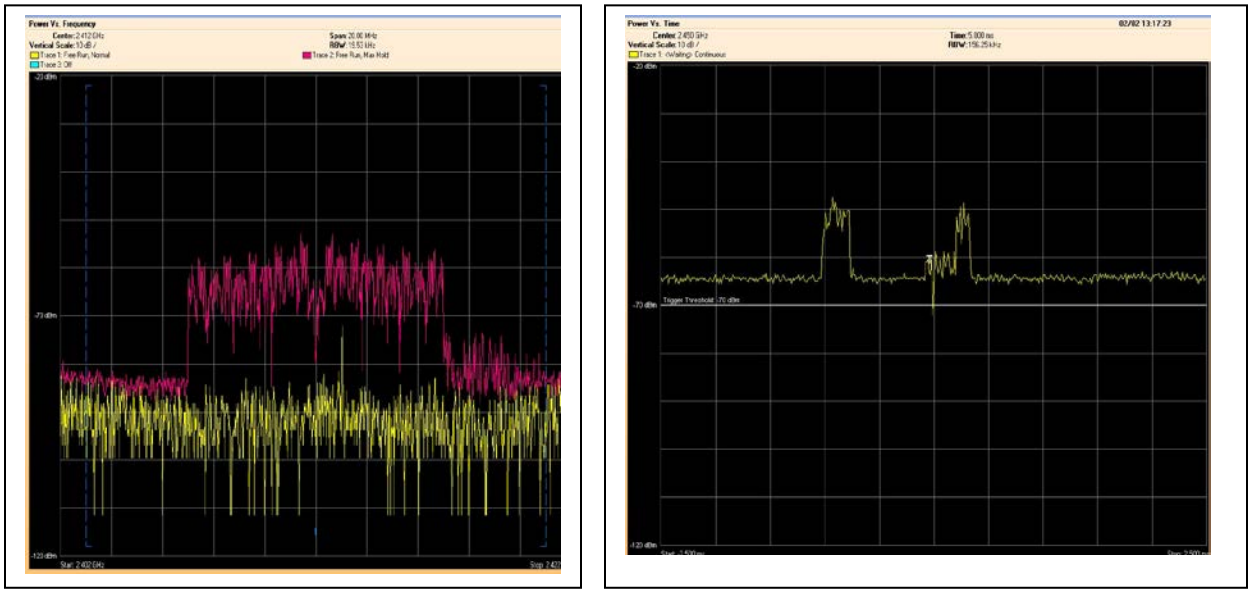


Figure 4: Frequency versus power, and time versus power plots

Other considerations for a GUI interface include:

- Ease of use on multiple OS
- Analysis Features

- Data Transfer: Calculating the FFTs can be done directly after each sample has been obtained, or it can be done after an overlap has been established.

4.1.4 Limitations and Constraints

The RTSA must follow limitations and constraints that are outside the scope of the predefined parameters. The main constraint includes the cost and using low power components.

- Low power components: It is also mentioned in the requirement that the RTSA technology is evolving using low power components. Thus, it would be advisable to use low power components for implementing this project.
- The bandwidth of the RF front-end will be limited to that of the hardware JDSU provides.
- The bandwidth of the down converter is limited in the same manner to that of the hardware provided by JDSU.
- Computation speed of the FFT's and other required computations are also limited by the commercially available hardware.

- Time: Due to a restriction of a school year to design and build the system, the design must be manageable within the time allotment.

4.1.5 Additional Considerations

There are other factors controlling the design of the RTSA. These factors include time management, accessible tools and other hindering factors.

- Tools: In the initial stages of planning and designing, the tools and equipment available in IPFW laboratories are limited. Thus, the tools from JDSU should be obtained far ahead of schedule for successful completion of the project.
- Testing: The system should be designed in a way that enables individual testing of components. The GUI component and the hardware components should be capable of independent testing for effective trouble shooting.

Section 5

5.1 Detailed Design

The detailed design consists of the following components as shown in the Figure 12:

- ADC converter
- Hanning window
- IP Core XC6SLX45T (FFT calculations)
- Magnitude block
- Average calculation block
- Log converting block
- Serial output

The purpose and functionality of each component is evaluated below.

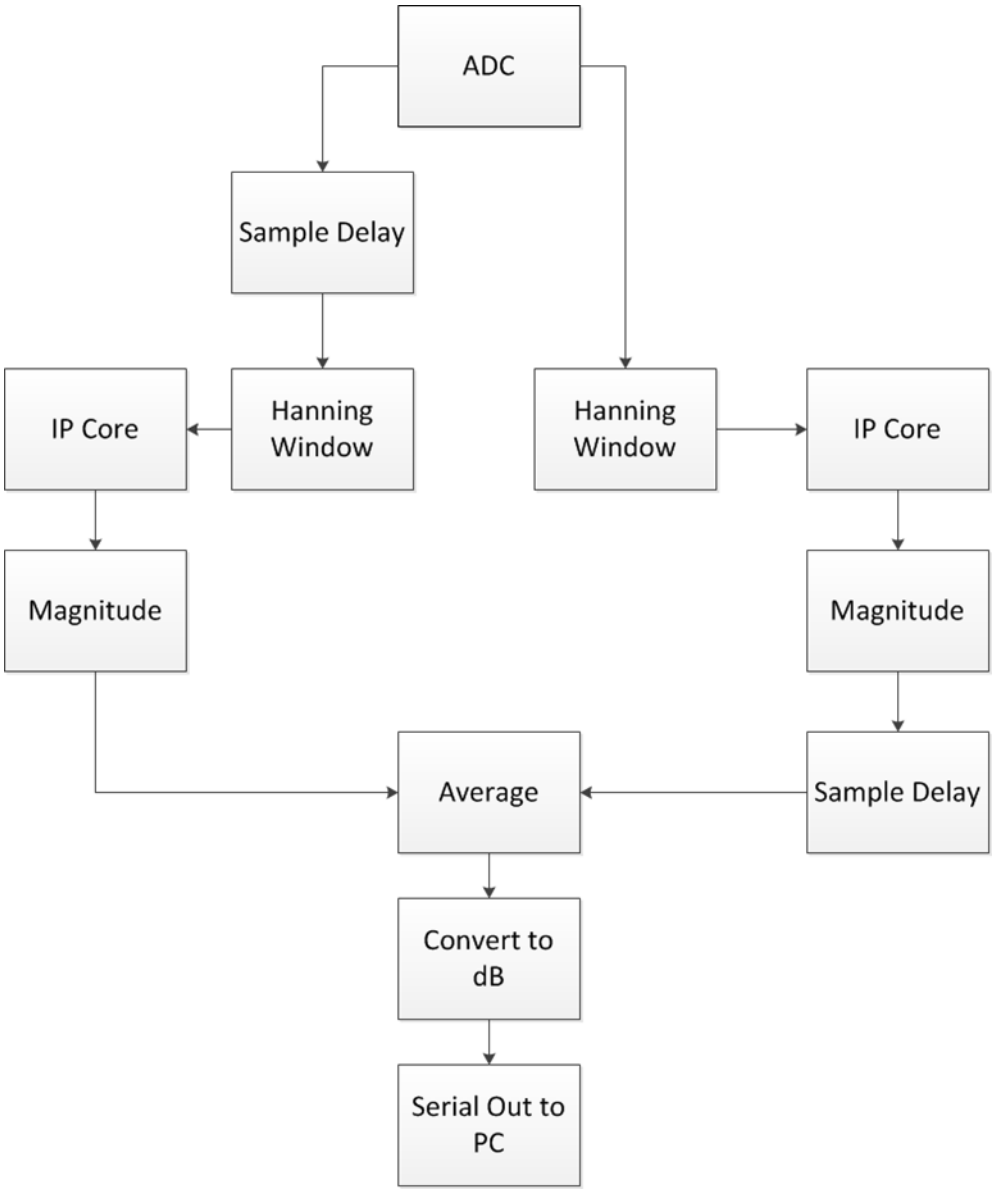


Figure 12

5.1.2 Analog-to-Digital Converter

The RTSA requires a 12-bit analog-to-digital converter (ADC) with a sampling rate of 204.8 MSPS. The team has selected the Texas Instruments ADS54RF63 Evaluation Module which is a 12-bit, 550 MSPS ADC that interfaces with the Spartan-6 XC68LX45T via a Texas Instruments high speed ADC to FMC header adapter card.

5.1.3 Hanning Window

Most real world signals measured by a spectrum analyzer are aperiodic. Because the FFT assumes a periodic signal, leakage will occur when an aperiodic signal is processed. In order to avoid this issue, a Hanning window is typically applied to the input signal, thus making it period by forcing the end points of signal to zero. The team will implement the Hanning window using look-up tables (LUTS).

5.1.4 IP CORE

The Spartan 6 FPGAs have the following configuration as shown in Table 2.

Table 2: Spartan 6 FPGA features

Device	Logic Cells[1]	Configurable Logic Blocks (CLBs)			DSP48A1 Slices[3]	Block RAM Blocks		CMTs[5]	Memory Controller Blocks (Max)[6]	Endpoint Blocks for PCI Express	Maximum GTP Transceivers	Total I/O Banks	Max User I/O
		Slices[2]	Flip-Flops	Max Distributed RAM (Kb)		18 Kb[4]	Max (Kb)						
XC6SLX4	3,840	600	4,800	75	8	12	216	2	0	0	0	4	132
XC6SLX9	9,152	1,430	11,440	90	16	32	576	2	2	0	0	4	200
XC6SLX16	14,579	2,278	18,224	136	32	32	576	2	2	0	0	4	232
XC6SLX25	24,051	3,758	30,064	229	38	52	936	2	2	0	0	4	266
XC6SLX45	43,661	6,822	54,576	401	58	116	2,088	4	2	0	0	4	358
XC6SLX75	74,637	11,662	93,296	692	132	172	3,096	6	4	0	0	6	408
XC6SLX100	101,261	15,822	126,576	976	180	268	4,824	6	4	0	0	6	480
XC6SLX150	147,443	23,038	184,304	1,355	180	268	4,824	6	4	0	0	6	576
XC6SLX25T	24,051	3,758	30,064	229	38	52	936	2	2	1	2	4	250
XC6SLX45T	43,661	6,822	54,576	401	58	116	2,088	4	2	1	4	4	296
XC6SLX75T	74,637	11,662	93,296	692	132	172	3,096	6	4	1	8	6	348
XC6SLX100T	101,261	15,822	126,576	976	180	268	4,824	6	4	1	8	6	498
XC6SLX150T	147,443	23,038	184,304	1,355	180	268	4,824	6	4	1	8	6	540

Table 3 lists the speed grade with respect to model number.

Table 3: The speed grade availability in IP Core for Spartan 6 devices

Device	Speed Grade Designations		
	Advance	Preliminary	Production
XC6SLX4[1]			-3, -2, -1L
XC6SLX9			-3, -3N, -2, -1L
XC6SLX16			-3, -3N, -2, -1L
XC6SLX25			-3, -3N, -2, -1L
XC6SLX25T			-3, -3N, -2
XC6SLX45			-3, -3N, -2, -1L
XC6SLX45T			-3, -3N, -2
XC6SLX75			-3, -3N, -2, -1L
XC6SLX75T			-3, -3N, -2
XC6SLX100			-3, -3N, -2, -1L
XC6SLX100T			-3, -3N, -2
XC6SLX150			-3, -3N, -2, -1L
XC6SLX150T			-3, -3N, -2

The speed grade of the chosen IP Core is 3, which is the highest speed grade provided. It also uses MCB's (Memory control blocks) which may be needed for memory allocation and management at a later stage.

The performance of an IP Core can be evaluated from the spec sheets provided by Xilinx.

Data is provided on the XC6SLX150T, the features and performance features of the IP Core are listed below:

- Implementation : Streaming using radix 2 algorithm
- Input data width handled : 16 bits
- The output is scaled
- The IP core allows the user to choose between fixed point and floating point arithmetic
- It allows scaling and rounding options
- The maximum clock frequency is 219 (which is greater than 204.8 requirement)
- The maximum number of DSP slices required by the IP Core is 16
- The maximum number of LUT/FF pairs needed for the computation is 2796

The XC6SLX45T has the same performance as the XC6SLX150T, except that it has 45,000 Logic units, instead of 150,000 logic units and 58 DSP slices verses 180. The maximum clock frequency of the IP Core is 219. Therefore, it can handle the required sampling rate. The number of DSP slices used by the IP Core is 16, and the XC6SLX45T has 58 DSP slices. Also, the number of LUTs used is 2712, while the Spartan board provides 43,661 LUTs. The number of DSP slices required for the FFT implementation is 16 in addition to 2712 LUTS. The XC6SLX45T will meet these requirements. JDSU already owns a development board that uses the XC6SLX45T.

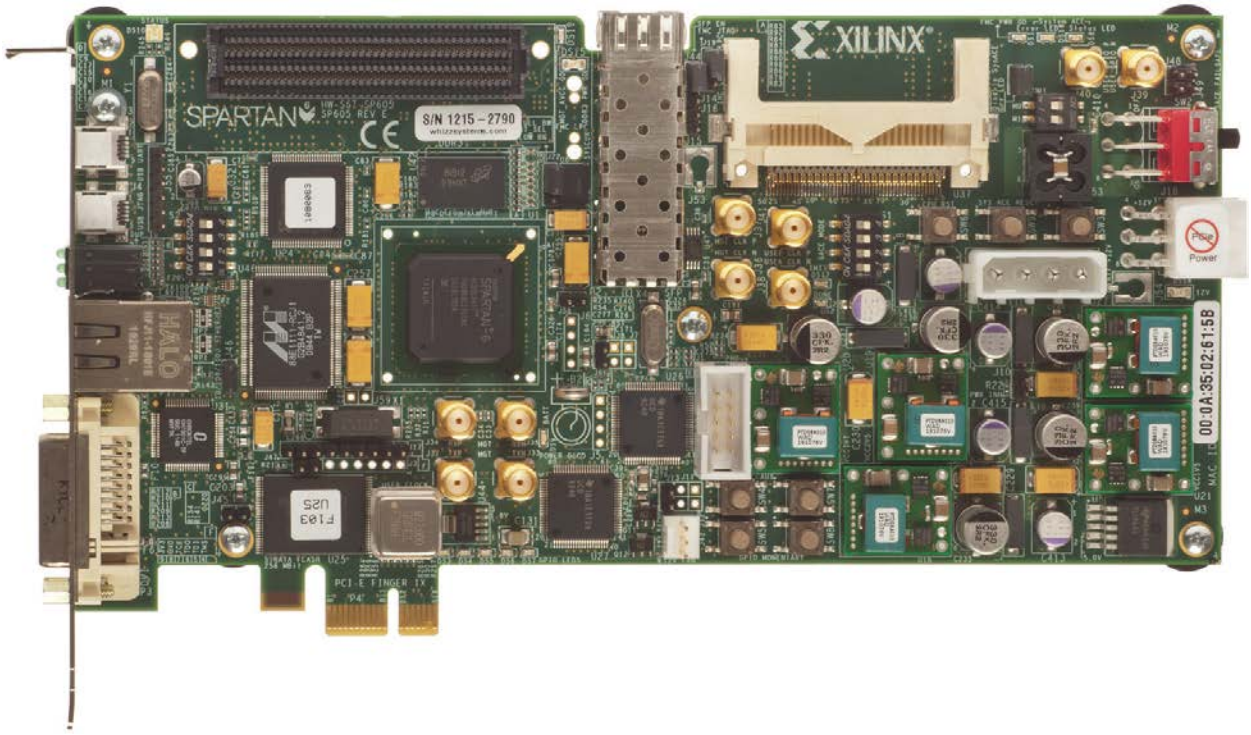


Figure 13: Spartan 6 SP605

Thus, using the Spartan 6, this IP core can be utilized with relative ease to develop the design and meet the requirements and specifications.

5.1.5 Sample Delay

The input samples for the second IP Core are delayed by 512 samples. This is done to get a 512 sample overlap (50% overlap) on the FFT output.

There is another delay of 512 samples inserted before the Log section. This is to make sure that the two FFT frames are in phase to be summed together for the actual value of the FFT. These delays will require 2 Kbytes of memory.

5.1.6 Magnitude

The complex values are presented to the magnitude block in 32-bit fixed-point format. A possible method of determining the magnitude is the LogiCORE IP CORDIC v5.0. The CORDIC core implements a generalized coordinate rotational digital computer (CORDIC) algorithm to iteratively solve trigonometric equations, and a broader range of equations, including the hyperbolic and square root equations.

The magnitude and phase of the input vector is returned using the translate function. The LogiCORE IP CORDIC v5.0 Product Specifications shown in Table 4 indicates that for a Spartan 6 FPGA operating in full pipelining mode, 1059 LUT6-FF pairs, 986 LUT's, 992 FF's will be required. This specification also shows a

maximum clock speed of 298 MHz. These resource requirements will need to be doubled since there will be two implementations.

Table 4: CORDIC v5.0 Specifications

Parameter/ Result	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10	Case 11
Function	Rotate	Trans	SinCos	Atanh	Square Root	Rotate	Trans	SinCos	Atanh	Rotate	Trans
Architecture	Word Serial	Word Serial	Word Serial	Word Serial	Word Serial	Parallel	Parallel	Parallel	Parallel	Word Serial	Word Serial
Input/Output Width	16	16	16	16	16	16	16	16	16	32	32
Round Mode	Trunc	Trunc	Trunc	Trunc	Trunc	Trans	Trans	Trans	Trans	Nearest Even	Nearest Even
LUT6-FF pairs	586	503	18	342	415	1196	1059	1050	1166	1245	928
LUTs	511	463	17	300	389	1116	986	1010	1128	1079	803
FFs	427	335	17	234	318	1104	992	976	1083	851	631
Block Rams	0	0	0	0	0	0	0	0	0	0	0
XtremeDSP slices	0	0	0	0	0	0	0	0	0	0	0
Max Clock Frequency	205	195	225	169	334	334	298	334	298	164	164

The output of the LogiCORE IP CORDIC is the 16 bit values of magnitude and phase rotation. Only the magnitude will be used. The 16 bit values indicating the magnitude of the two FFT operations will be averaged together by summing them together and then right shifting one bit. Right shifting rounds the result by truncating the LSB. Since the magnitude is always positive, rounding is toward zero.

5.1.7 Average

Corresponding values from the two FFT frames will be averaged together using an arithmetic mean.

5.1.8 Convert to dB

The input to the logarithm block is a 16 bit unsigned binary number representing the magnitude of the output signal. To convert this to the requested units of dB, the following equation must be applied.

$$DB = 20 \times \log_{10}(\text{Magnitude})$$
$$DB = 20 \times \log_{10}2 \times \log_{2}(\text{Magnitude})$$

The value $20\log_{10}2 \approx 6.0206$ is a constant. The base 2 log of the input signal must be computed and multiplied with this constant. Since VHDL has no log function available one has to be developed. Understanding that the log of a number represented in scientific notation is simply the exponent plus the log of the significand provides a possible algorithm.

The exponent can be determined from the location of the first 1 in the 16 bit number. The significand is the binary value that results if a decimal point is inserted immediately after the leading 1 in the 16 bit binary value.

To determine the base 2 exponent, a 16 to 4 decoder will be implemented that finds the first occurrence of a '1' bit. The 16 bit number will then be shifted (right or left) to achieve a 5 bit significand. A look-up table will be used to cross the 5-bit significand to a logarithm.

The look-up tables only need to provide enough precision to meet the design requirement of 0.5 dB output resolution. Since the constant multiplier is 6.0206 the resolution must be <= 0.08304. A table with 32 values will be implemented allowing for a logarithm resolution of 0.05 and an output resolution of 0.19 dB meeting the requirements.

5.1.9 Serial Output to Host PC

The data arrives at the input of the serial block at a rate of 1 data point every 4.9 ns or 204.8 M data points/s. As a result of the windowing techniques employed by the Hamming windows, the beginning and ending FFT output data-points will be reduced towards zero. If only the middle 850 output samples are used, the average output rate of the serial block is reduced to 170 M data points/s. Using 850 of the 1024 available data points reduces the bandwidth from 102.4 MHz to 85 MHz which is the specification. Since the input to the real-time spectrum analyzer is real valued, the FFT will be symmetrical about the vertical axis. Therefore only one half of the data points presented to the serial block will be used. Since one half of the samples will be discarded, the average output rate is reduced to 85 M data points/s.

The data output from the serial block is required to cover a 96 dB range with resolution of 0.5 dB. This requires 8 bit unsigned integer format at a minimum. The serial block must be able to deliver the output data at a rate of 680 Mbit/s or buffer overflow will occur. A Xilinx FPGA with a gigabit transceiver is capable of 3.2 Gbit/s on the Spartan 6 line of FPGA's. While USB ports subscribing to the USB 3.0 standard are capable of 5 Gbit/s and would meet the requirements, the chosen design board does not have USB 3.0 capacity. Instead we are currently limited to USB 2.0 standard ports. The USB 2.0 standard is capable of 480 Mbit/s.

The evaluation boards can communicate over the PCI bus or the SFP (Small form-factor pluggable transceiver) at a rate of 3.2 Gbit/s which would meet the requirements. Since the specifications require communication to the host PC over USB, a method of averaging 2 successive samples and transmitting as a single sample was chosen as a solution. Using this method, a transmission rate of 340 Mbit/s will be required falling within the USB 2.0 standard's specification. The solution maintains the real-time characteristics of the device (no samples lost) while achieving a manageable transmission rate.

A time span of 0.1 – 1.0 sec was given as a requirement for the max time interval for the host PC data upload. This will require a buffer space of 4.3 to 42.5 Mbytes for data point accumulation.

Section 6

6.1 Bill of Materials

Table 5 - BOM

Part	Cost	Quantity
TI ADS54RF63 Evaluation Module	$300	1
FMC-ADC-ADAPTER	$50	1
Spartan-6 XC68LX45T	$570	1

Total Cost: $920

Section 7

7.1 Build Process

After creating the design in System Generator, the design had to be transferred to the ISE as VHDL source code. The following procedure was used to accomplish this task.

Double click the System generator block on the Simulink drawing and configure as follows:

- In compilation tab
  (1) Set Compilation Options - "HDL Netlist"
  (2) Synthesis Tool - "XST" (Default)
  (3) Hardware description Language - "VHDL"
  (4) Target Directory - "./netlist" (Default)
  (5) Project Type - "Project Navigator"
- In Clocking Tab
  (1) FPGA Clock Period to "4.9 ns"
  (2) Clock Pin Location – leave blank
  (3) Multirate Implementation - "Expose Clock Pins"
  (4) Simulink System Perion – "1"
- In General Tab – defaults are ok

After configuration settings were made, The generate button was clicked, and the netlist files were created.

In the netlist subfolder of the project's directory a file named project_mcw.xise was created. This file is the ISE tools Project Navigator file for the System Generator project. The ADC portion of the project needed to be incorporated into this design and the corresponding signal and clock pins connected.

Files AdcInputMain.vhd and IDDR12.vhd (created for the ADC) were added to project_mcw.xise as source files. The top level project file was amended to incorporate the ADC. In the appendix, both the original and the modified versions of project_mcw.vhd are shown.

Some issues arose while implementing the design. The two main problems that presented themselves were related to timing constraints. The first issue came from generation of the system generator files. During generation timing constraints were added in two places, file "project_mcw.ucf" as well as an invisible file "project_mcw.xcf". To disable ISE from reading constraints from "project_mcw.xcf" the "Use Synthesis Constraints File" check box was unchecked in the Process Properties – Synthesis Options tab. This tab is available by right clicking Synthesize-XST folder in the lower design pane and selecting "Process Properties".

The other timing constraint error was due to the clock signal's name. In the project_mcw.ucf (constraints file) the net referenced was "clk_1". This had to be renamed to "clk_1_sg_x7" to match the clock signal in "project_mcw.vhd.

7.1.2 Analog-to-Digital Converter

ADC Build

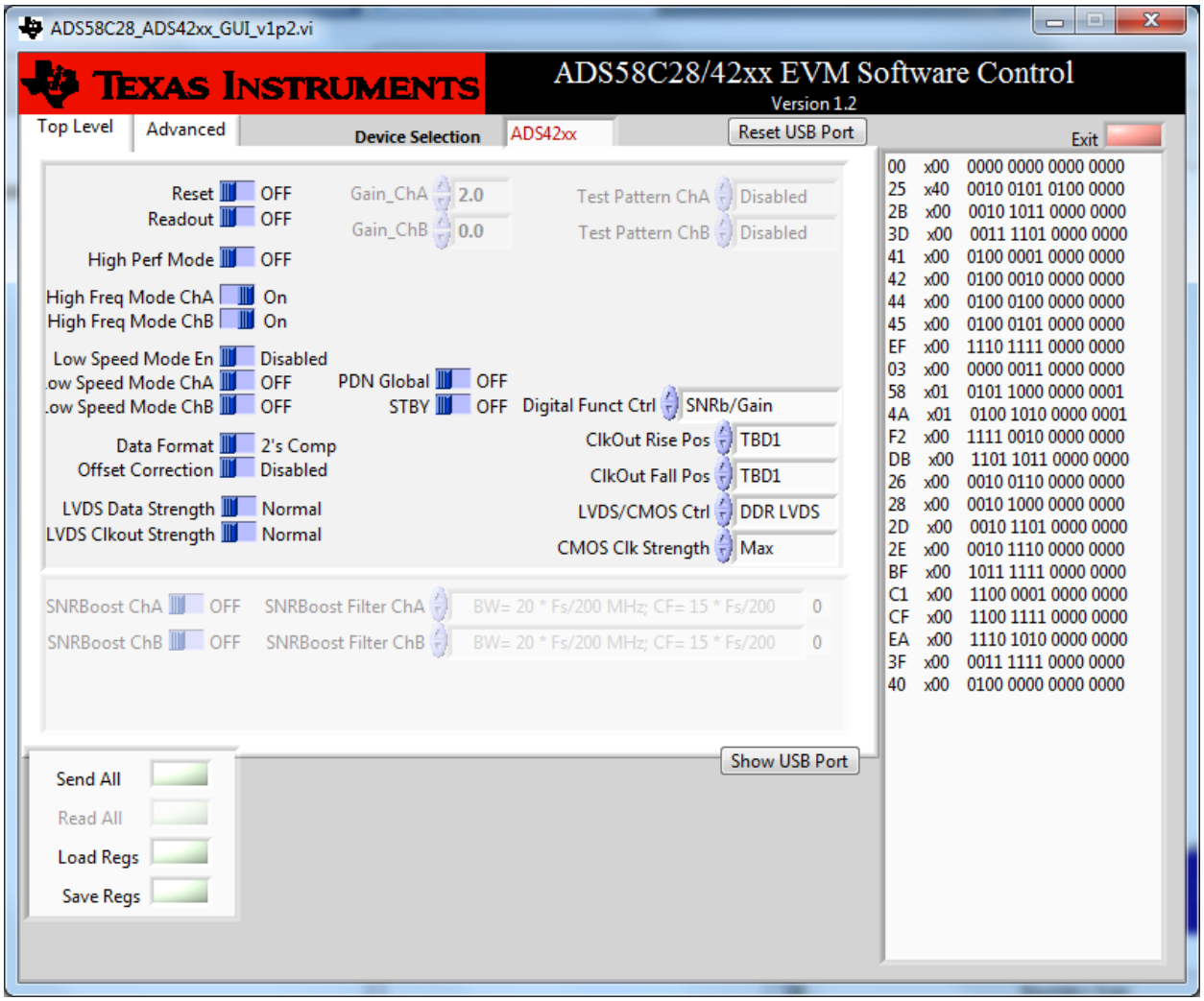
The design requires the TI ADC4229 EVM (ADC evaluation board) to connect to the SP605 evaluation board via the FMC (VITA 57 standard) header using the FMC_ADC_ADAPTOR. The pinout of the ADC4249 EVM board has to be mapped to the Xilinx board in the constraints file. The ADC4229 EVM board can output the 12-bit signal in one of two methods, CMOS (single ended), or LVDS (differential). Since LVDS is the preferred method for high speed signals and both boards are capable of this format, LVDS was chosen. Using the interface connector schematic for the FMC_ADC_ADAPTOR and the SP605 user's guide, the mapping was determined for the 12-bit signal input and the user constraints file was updated.

Since the FFT will be running in real-time pipe-lining mode, the clocking for the ADC and the FPGA's FFT block have to be synchronized. The easiest way to do this was to provide the required the 204.8 MHz clock signal to the ADC4229 EVM board's clock input and then configure the ADC board to provide a synchronized clock to the FPGA board via the FMC header. This clock signal is also in LVDS format and the corresponding mapping was added to the constraints file. The mapping is shown in box below.

```
NET "CLK_IN_N"                          LOC = "AB11";  ## D21 on J2
NET "CLK_IN_P"                          LOC = "Y11";   ## D20 on J2
NET "DATA_IN_FROM_PINS_N(0)"            LOC = "A17";   ## C19 on J2
NET "DATA_IN_FROM_PINS_P(0)"            LOC = "C17";   ## C18 on J2
NET "DATA_IN_FROM_PINS_N(1)"            LOC = "Y14";   ## G28 on J2
NET "DATA_IN_FROM_PINS_P(1)"            LOC = "W14";   ## G27 on J2
NET "DATA_IN_FROM_PINS_N(2)"            LOC = "AB14";  ## H29 on J2
NET "DATA_IN_FROM_PINS_P(2)"            LOC = "AA14";  ## H28 on J2
```

Configuration of the ADC board was performed using information provided in the ADS42xx EVM User's Guide and TI provided software ADS58C28_ADS42xx_GUI.

The LVDS inputs from the ADC board are presented to the SP605 board in DDR (double data rate) format. This means that the even bits of information are presented on DATA_IN_FROM_PINS(0 – 5) on the rising clock edge and the odd bits are presented on the falling clock edge. VHDL code was created to read the data from the ADC. The code is contained in two files, "AdcInputMain.vhd" the top level, and "DDR12.vhd". These files are included in this report as attachments.



7.1.3 Hanning Window

The Hanning window was implemented using block ram.

7.1.4 IP Core

The FFT IP Core was provided by Xilinx and implemented in Simulink.

7.1.5 Sample Delay

The primary sample delays were implemented using FIFO buffers. While the smaller sample delays were implemented using shift register.

7.1.6 Magnitude

The Magnitude block is used to find out the magnitude after the FFT block outputs the real and the imaginary part of the result. The magnitude can be found out by using the formula sqrt(real part^2 + imaginary part^2).

The Simulink provides predefined Cordic blocks in the Xilinx set to find out the magnitude of the FFT output. However, it uses too many DSP slices 40 of 56 DSP slices), and the resources on the Spartan 6 board could not accommodate the Cordic blocks.

The group implemented the magnitude block by using an approximation to the magnitude formula. The pseudo code for the approximation algorithm is as follows:

Pseudo Code:

```
if ( x > y )
   mag ≈ x + 0.5 * y
else
   mag ≈ y + 0.5 * x
```

This algorithm is implemented in VHDL and implemented in the Simulink design using a black box tool. The VHDL code for this block is provided at the end of the document.

Unit tests have been performed on the block to check if the approximation is close enough.

7.1.7 Average

The average block is used to average out the samples obtained from the 2 magnitude blocks attached to two different FFT units. This involves addition of the two results and shifting the output by a bit to the right, to obtained the divided value.

7.1.8 Convert dB

The db_converter block is essentially designed to convert the magnitude of the FFT output into a db scale. This task is accomplished using smaller components designed in VHDL into a huge module using port mapping techniques, and importing this VHDL code into simulink design using a “black box” from Simulink Library browser.

The general formula used to compute the db value is $20\log_{10}$(Magnitude). The Magnitude can be represented as ($X*2^N$) where X€ R and N € N.

Formula Derivation:

$dB = 20\log_{10}(X*2^N)$

$= 20\log_2(10)\ \log_2(X*2^N)$

$= 20\log_2(10)\ \log_2(N+\log_2 X)$

$= 20\log_2(10N) + 20\log_2(10\ \log_2 X)$

$= 6N + 6\log_2 X$

Functionality of the db_converter block:

This block gets a 16 bit input from the adder (which sums the FFT samples from the 2 cordics). The 16 bit input has to be checked and tested to locate the position of leading '1' in the number. The position of the leading '1' is assigned to the value 'N'. For the 0.5 db precision, the team has choosen to consider the 5 bits after the leading '1' into consideration. Thus, X is assigned to be the 5 most significant bits after the leading '1'. Later, the value of 6N and $6\log_2 X$ is found and added to obtain the final result.

The VHDL code to implement this db_convert block has the following components:

- Db
- Six_log_two
- Adder

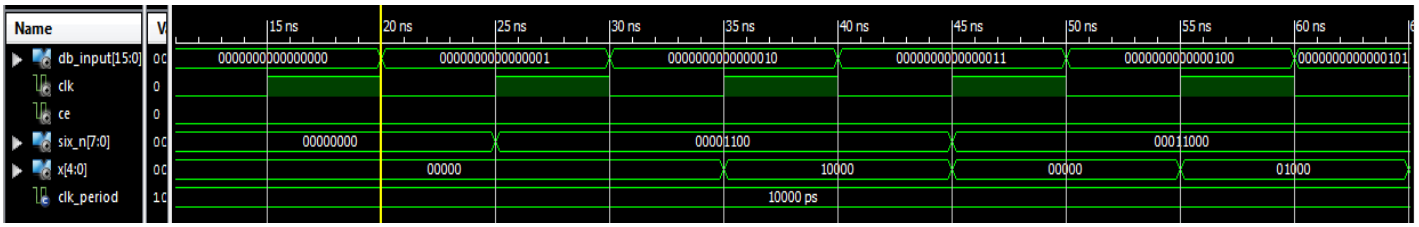
Db component:

This block acts like a 16*4 decoder. It finds the position of a leading '1' and outputs the value of 6N, along with a 5 bit X value. The code of this component is attached at the end of the document. The value of 6N is an 8 bit fixed point value, in which the binary point is placed at index 0. It has to be noted that the process uses a Clock pulse to compute the result. Thus, output changes on the rising edge of the clock.

A VHDL test bench is used to test the outputs for all the possible values of the 16 bit DB_INPUT. A snippet of the VHDL test bench code is as follows:

```
for I in 0 to 65535 loop

        DB_INPUT <= conv_std_logic_vector(I,16);

        wait for 10 ns;

end loop;
```

The final waveform obtained after the test bench was implemented is as follows:



Six_log_two component:

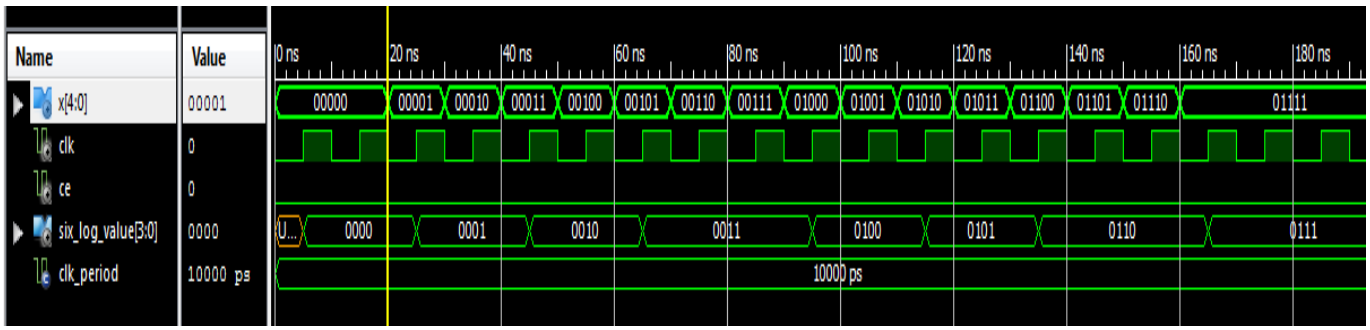
This block uses the input form the db block to find out the value of $6\log_2 X$. The input to this block is the value of X. The result obtained from this block is a 4 bit fixed point value, in which the binary point is placed at index 1. Thus, the 2 least significant bits represent the value after the decimal point.

The basic logic used to implement this block is similar to that of a standard decoder, which checks the condition of an input to result in a specific output. The value of $6\log_2 X$ is approximated tot hr nearest 0.5, and the output conditions are applied.

The VHDL implementation of this block is shown in the appendix. A snippet of the VHDL test bench to verify the output of this block is as follows:

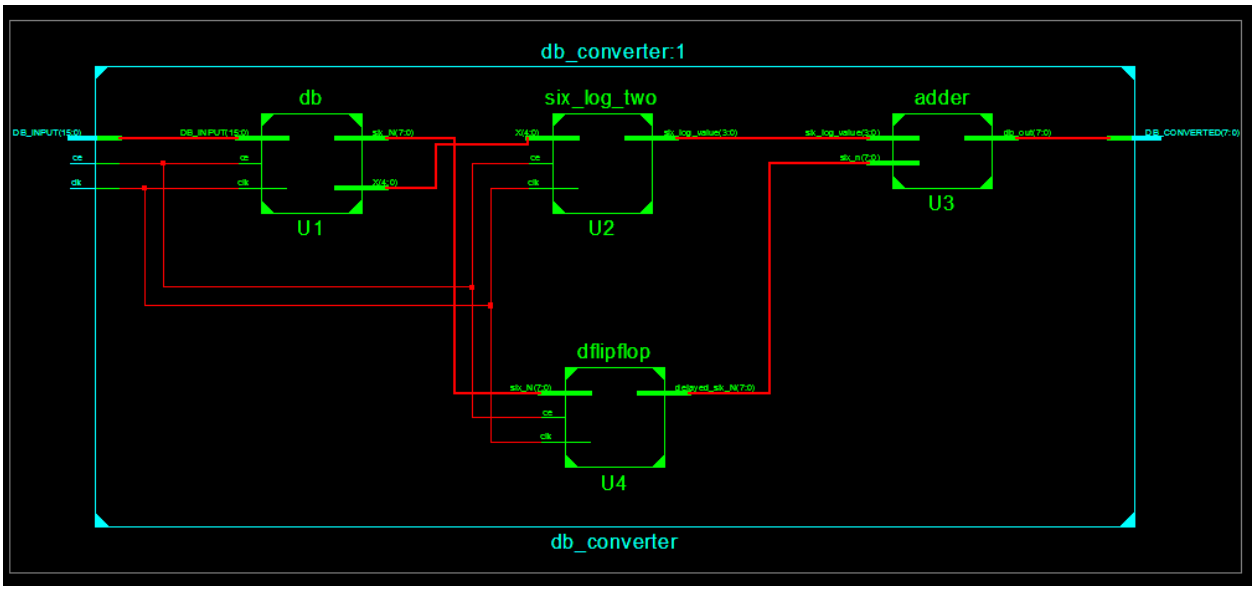
```
for I in 0 to 15 loop

        X <= conv_std_logic_vector(I,5);

        wait for 10 ns;

end loop;
```

The final wave-form obtained after the test bench was implemented is as follows:



Final Assembly:

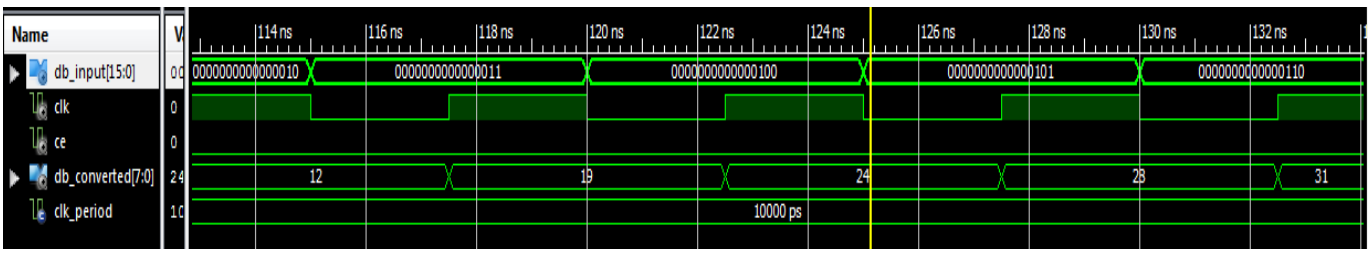
The final assembly involves an adder and a delay component along with the 2 VHDL blocks mentioned above. The adder sums the value of 6N and $6\log_2 X$ to obtain the final result. As there is a delay between obtaining the 6N value and $6\log_2 X$, a delay block is inserted after the db block, to make sure that 6N and $6\log_2 X$ values are sent as inputs to the adder at the same time. The component layout after incorporating the delay and the adder into the system is as shown in the figure below.



The Synthesis report of the above-mentioned design is as follows:

```
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                  : 1
 32x4-bit single-port distributed Read Only RAM        : 1
# Adders/Subtractors                                    : 2
 16-bit adder                                          : 1
 8-bit adder                                           : 1
# Registers                                             : 25
 Flip-Flops                                            : 25
# Multiplexers                                          : 27
 5-bit 2-to-1 multiplexer                              : 14
 8-bit 2-to-1 multiplexer                              : 13

=========================================================================
```

The above-mentioned design has also been tested using the VHDL test bench. The test bench resulted in the waveform shown in the figure below.

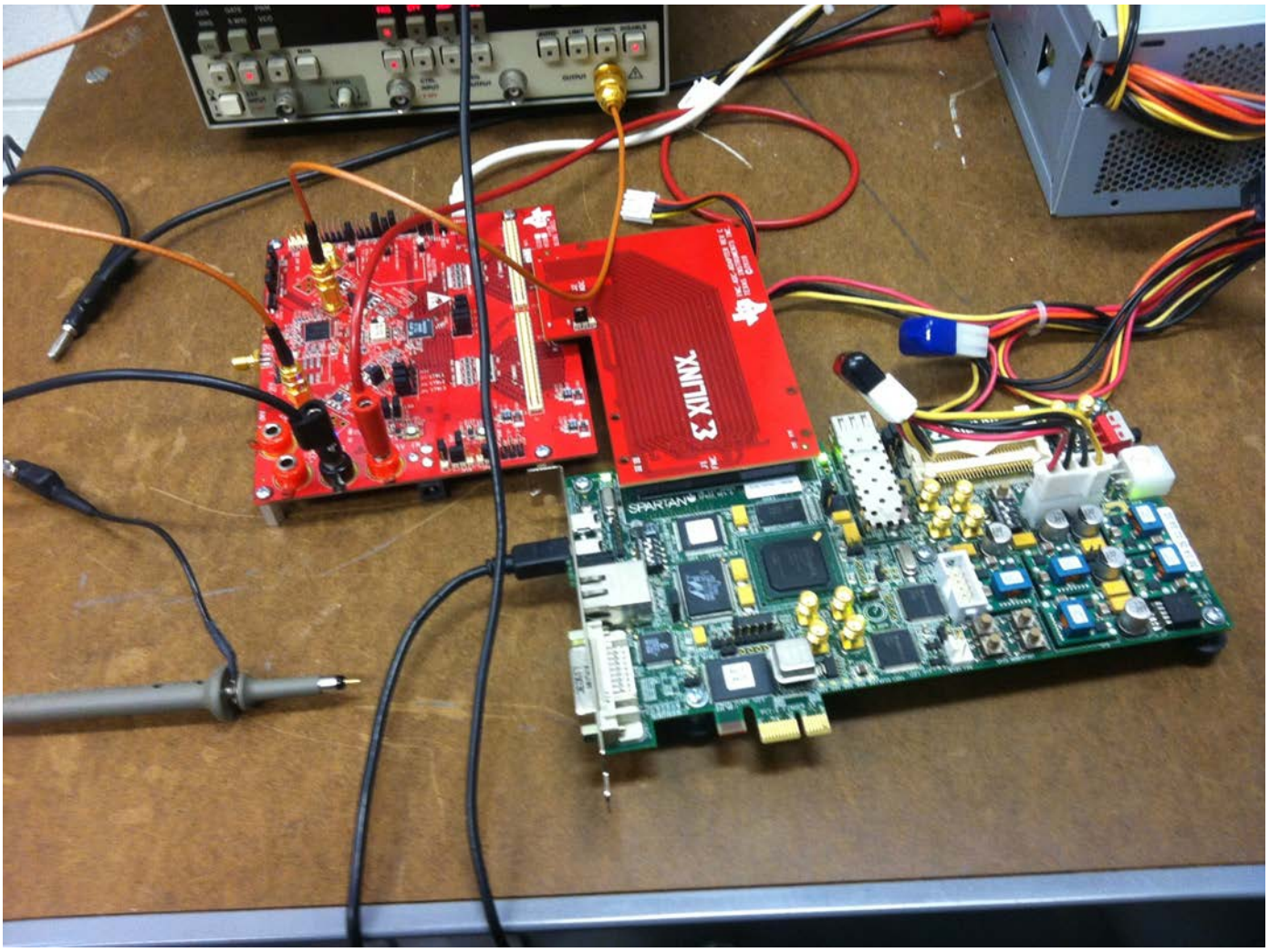


Section 8

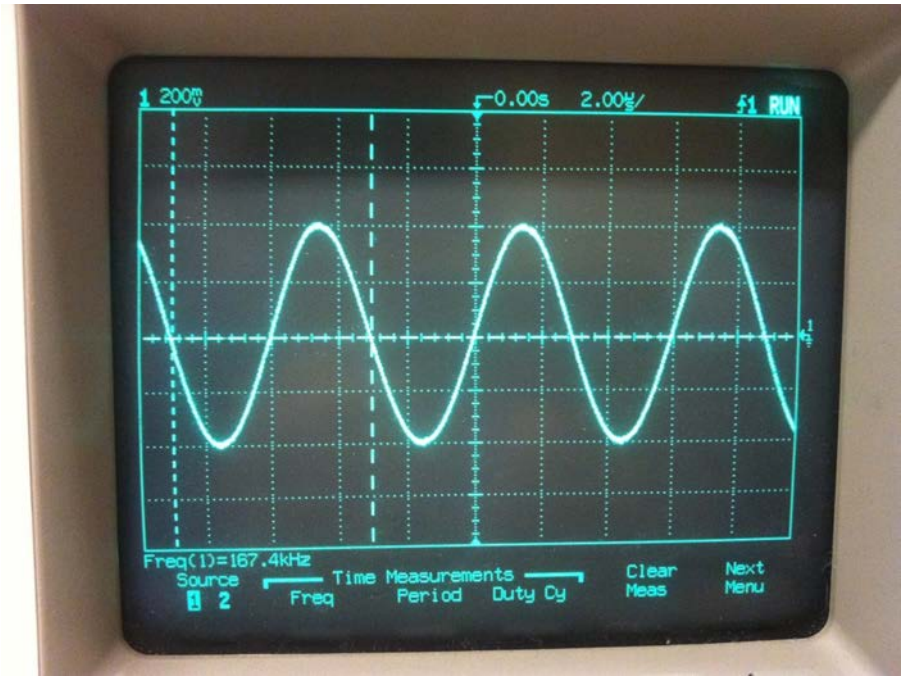
8.1 Unit Testing

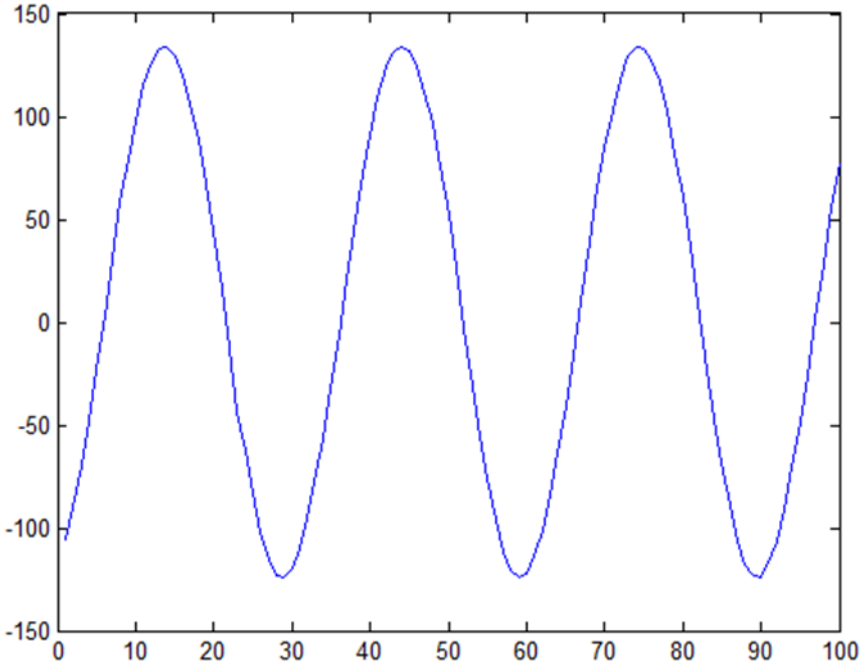
8.1.2 Analog-to-Digital Converter

The analog-to-digital converter was tested by sending an analog sine wave in at a known frequency and analyzing the corresponding digital signal.



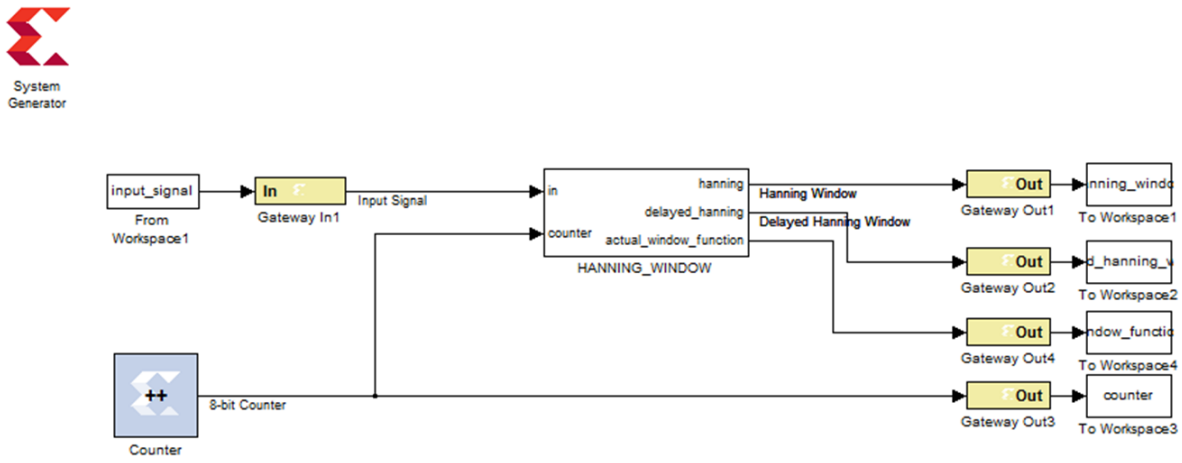
The two images below show the analog sine wave in and the digital sine wave out. Both the magnitude and frequency measurements are correct.

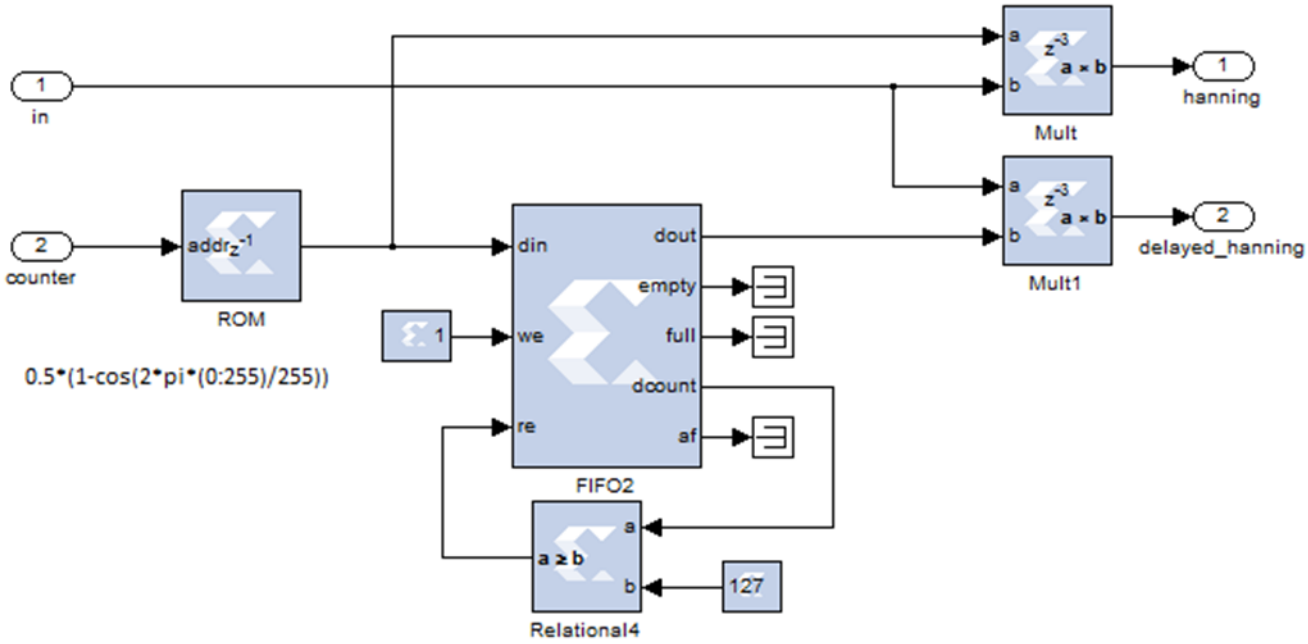




8.1.3 Hanning Window

The Hanning window was tested graphically by sending a sine wave in and visually inspecting the resulting Hanning window.



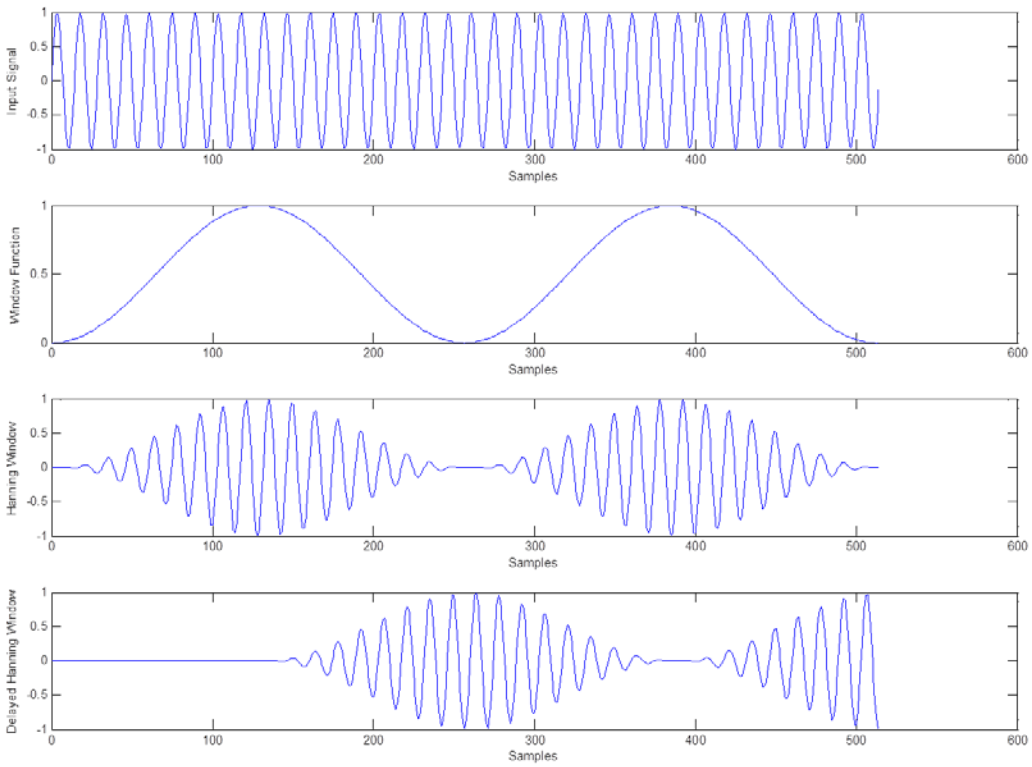


The following MATLAB scripts import and export data to and from the MATLAB workspace into the Simulink model (in this case, the Hanning window module).

```
clear
clc

f = 70;
fs = 1e3;
T = 1 / fs;
num_samples = 10000;
n = [0:num_samples];
t = n*T;
x =  sin(2*pi*f*t);

input_signal = [n; x]';
```

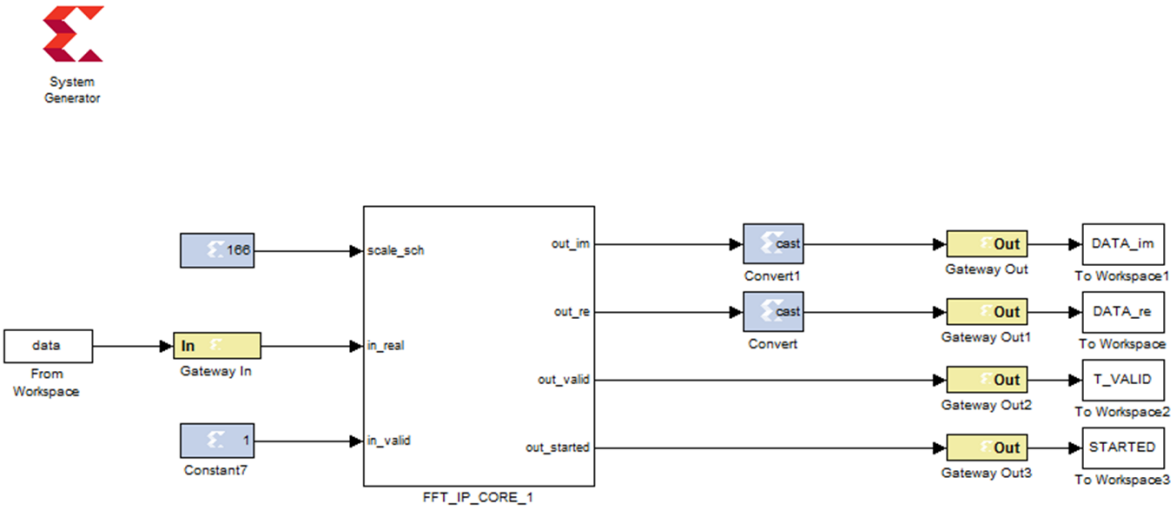
```
clc

hanning_window_latency = 3;
fft_frame_size = 256;

L = 2 * fft_frame_size + hanning_window_latency;

subplot(4,1,1)
plot(n(1:L), x(1:L))
xlabel('Samples')
ylabel('Input Signal')

subplot(4,1,2)
plot(n(1:L), window_function(1:L))
xlabel('Samples')
ylabel('Window Function')

subplot(4,1,3)
plot(n(1:L), hanning_window(1:L))
xlabel('Samples')
ylabel('Hanning Window')

subplot(4,1,4)
plot(n(1:L), delayed_hanning_window(1:L))
xlabel('Samples')
ylabel('Delayed Hanning Window')
```

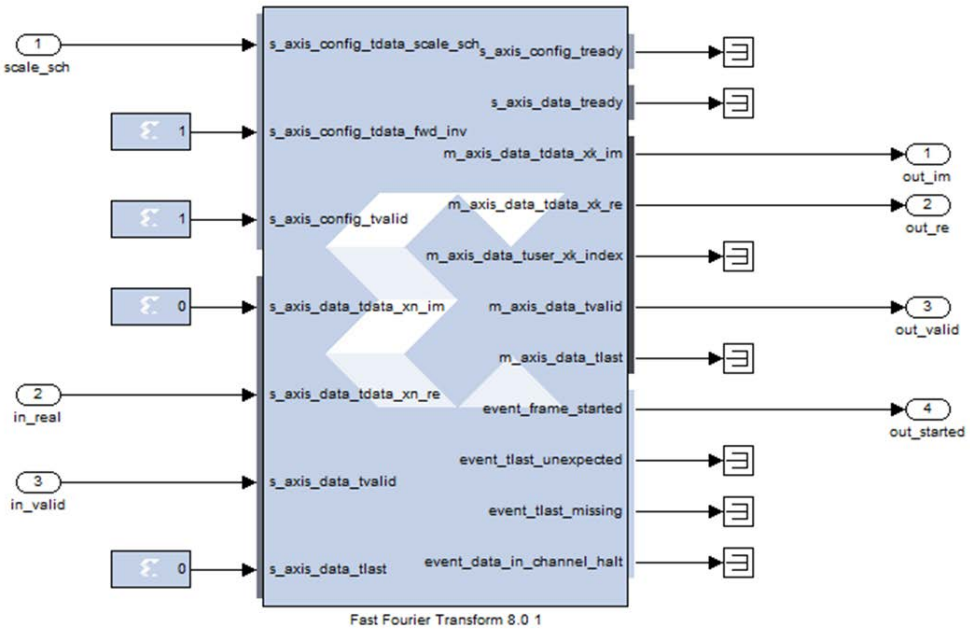
The image below shows the input sine wave, the window function, and the resulting Hanning windows. Using the MATLAB data markers, a visual inspection confirms the correct delay and magnitude values.



8.1.4 IP Core

The FFT IP Core was tested in the same way as the Hanning window. A sine wave was sent from the MATLAB workspace into the IP Core. The results were graphically inspected.



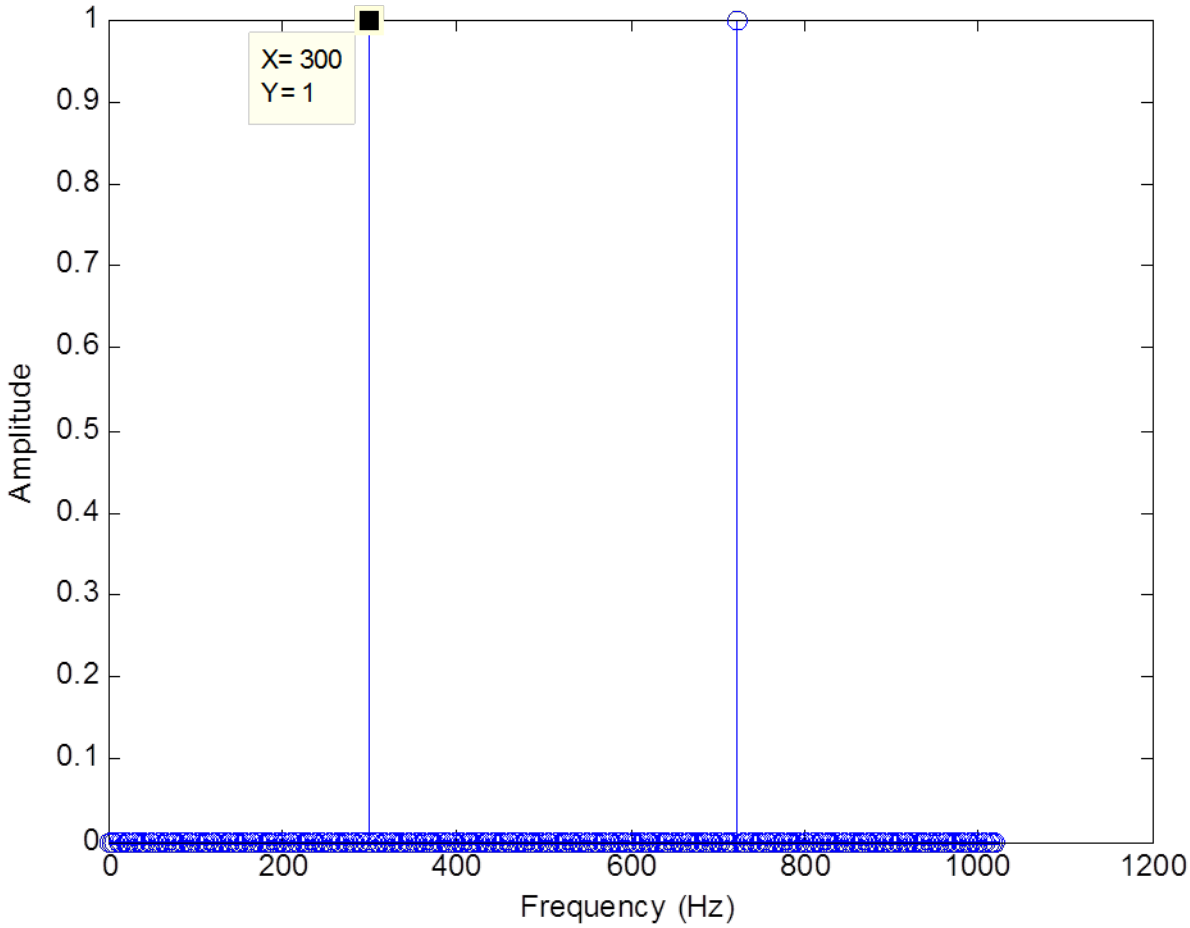


The following MATLAB scripts import and export data to and from the MATLAB workspace into the Simulink model (in this case, the FFT IP Core module).

```
clear
clc

f = 300;
fs = 1024;
T = 1 / fs;
n = 0:10000;
t = n*T;
x =  sin(2*pi*f*t);

data = [n; x]';
```

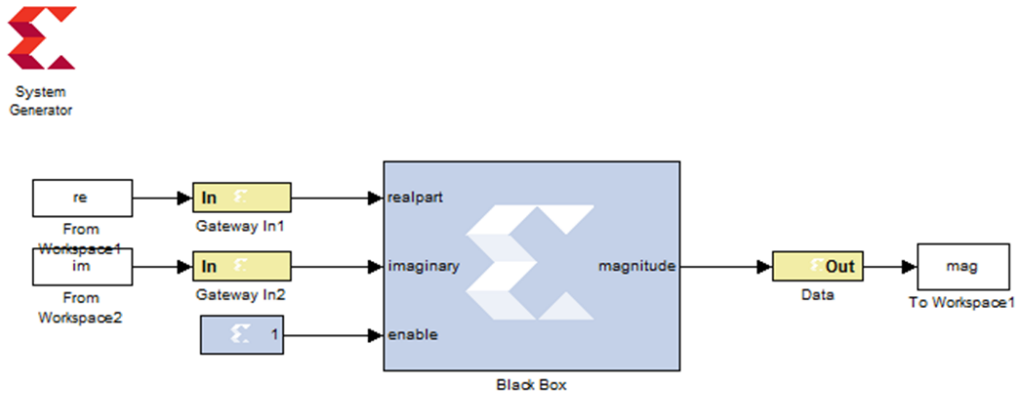
```
T_VALID = T_VALID';
start = min(find(T_VALID));

DATA_im = DATA_im';
DATA_re = DATA_re';
MAG = sqrt(DATA_im.^2 + DATA_re.^2);

N = 256;
delta_f = fs / N;
f = [0:N-1] * delta_f;
stem(f, MAG(start: start + N - 1))
xlabel('Frequency (Hz)')
ylabel('Amplitude')
```

The following image clearly shows a pure sine wave at frequency 300 Hz. As seen in the aforementioned MATLAB scripts, the frequency resolution was selected to avoid any spectral smearing.



8.1.5 Magnitude

The magnitude module was tested by using it to calculate the Pythagorean identity and plotting the results in comparison to a perfect unit circle.

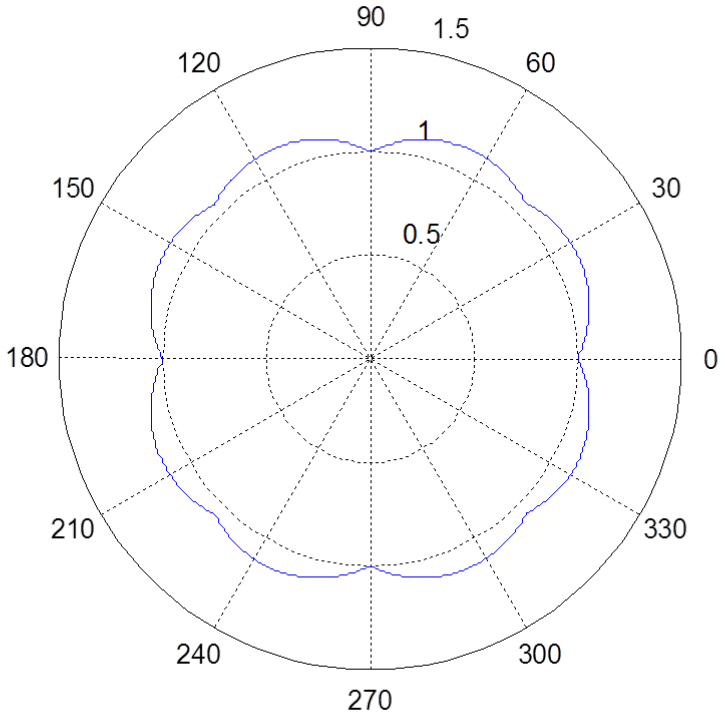


The following MATLAB scripts import and export data to and from the MATLAB workspace into the Simulink model (in this case, the Magnitude module).

```
clear
clc

theta = 0:0.001:2*pi;
n = 0:length(theta)-1;
x = 0.999*cos(theta);
y = 0.999*sin(theta);

re = [n; x]';
im = [n; y]';
```

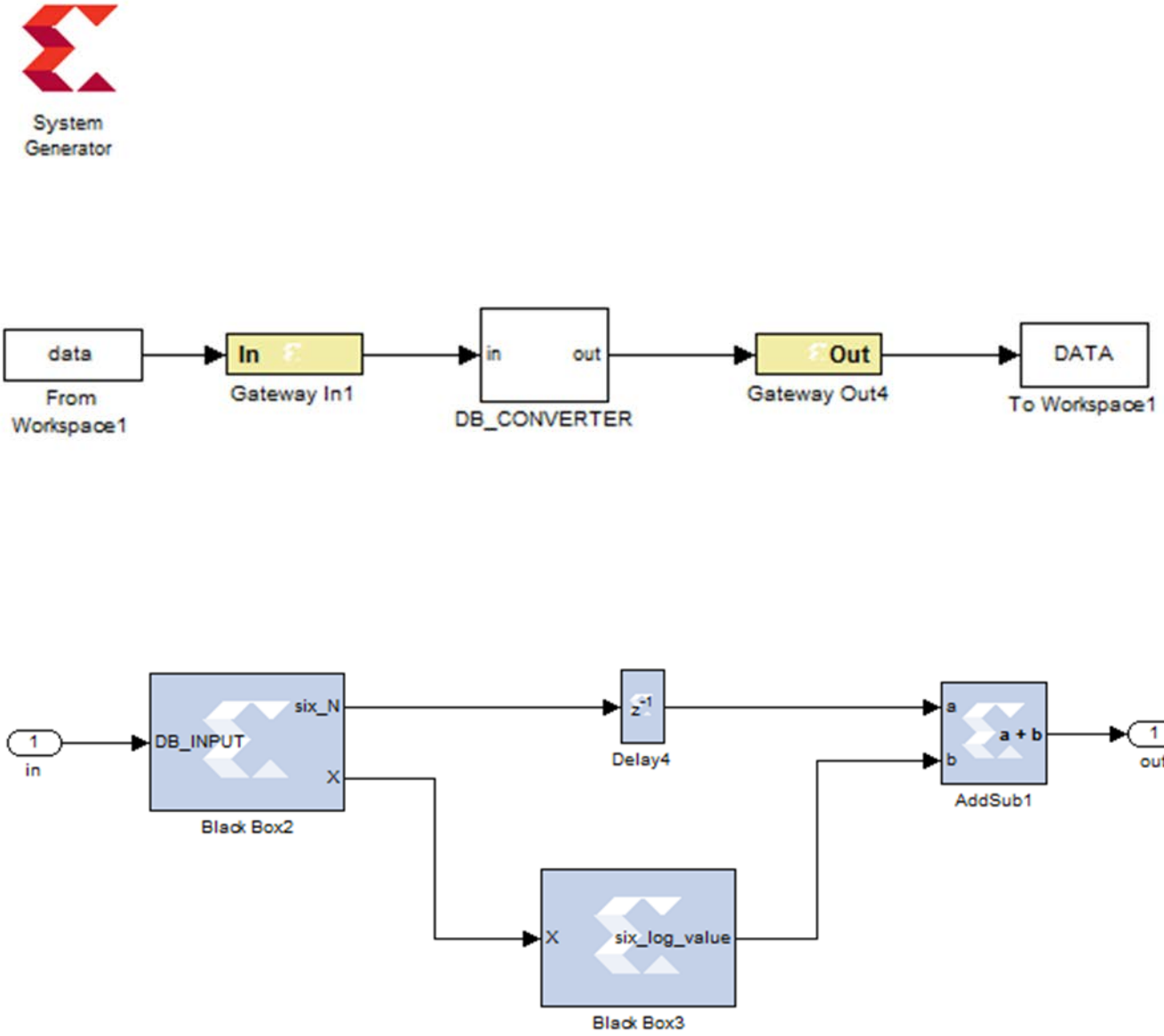
```
SIXTEEN_BIT_SIGNED = 32768;
trunc_mag = mag(1:length(theta))'/SIXTEEN_BIT_SIGNED;
polar(theta, trunc_mag)
```

The image below shows the results of the magnitude module. Clearly it is not a perfect unit circle which is expected because the algorithm used is not an exact magnitude calculation as discussed in Section 5.

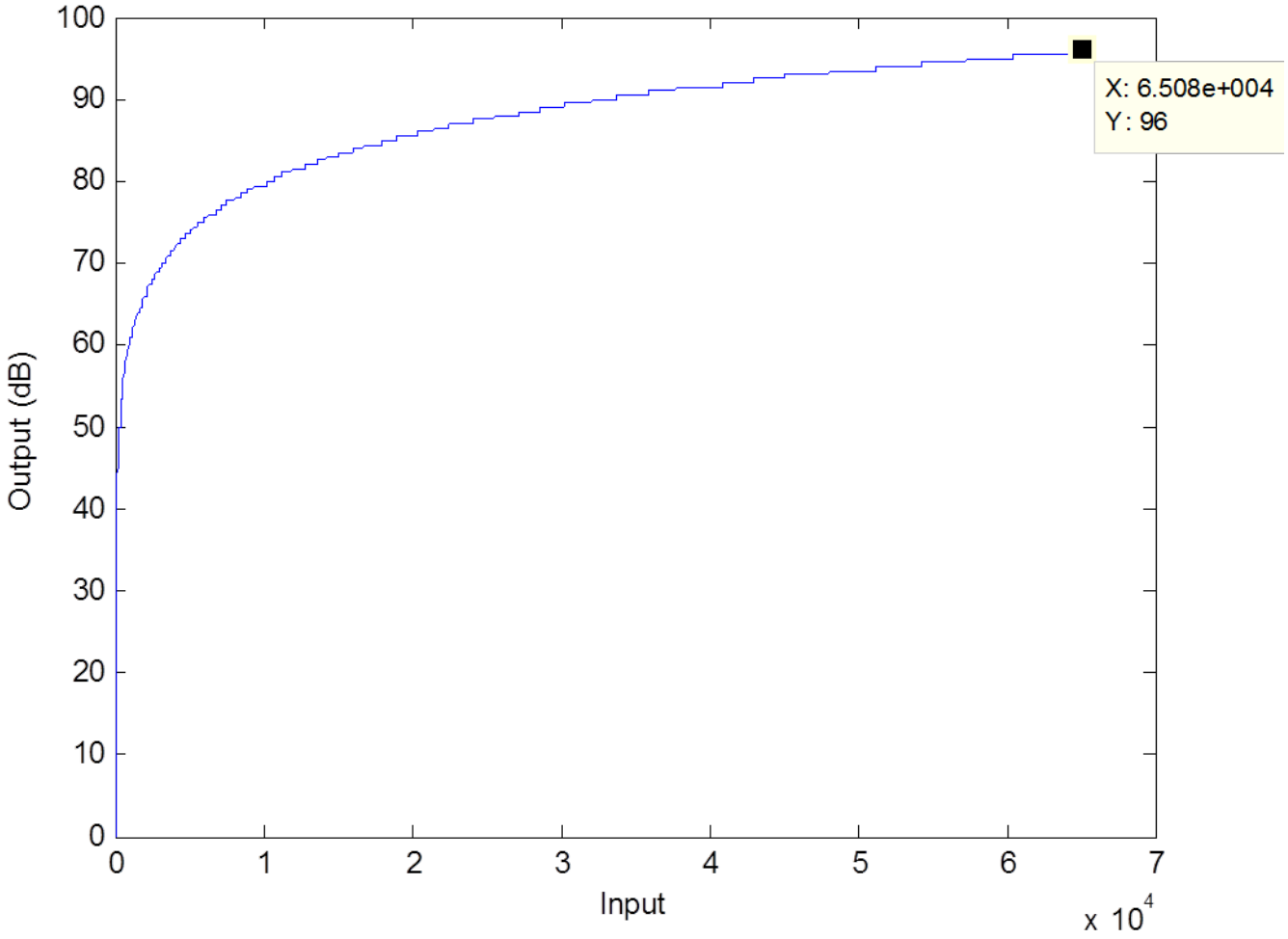


8.1.6 Convert dB

The dB converted was tested by sending integer numbers between 0 and 65635 from the MATLAB workspace into the dB Conversion module. This simulates all possible values of a 16-bit number.

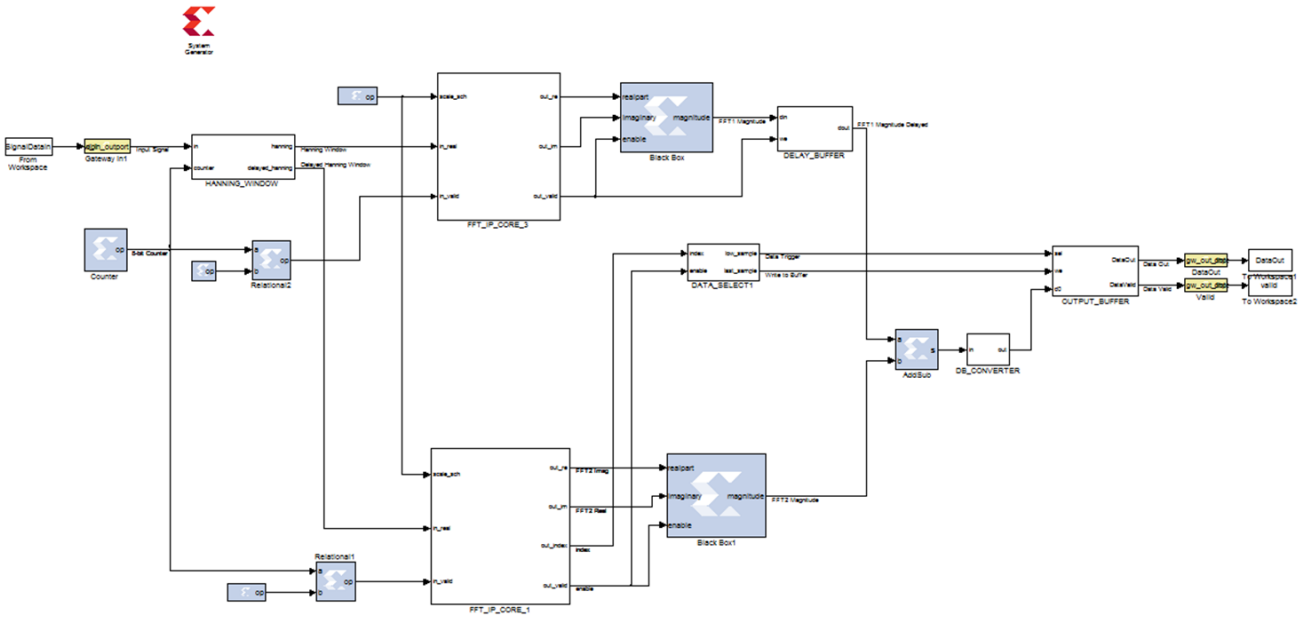


The plot below shows the characteristic graph of a logarithm. The MATLAB data marker clearly shows that the desired range of 96 dB is met.

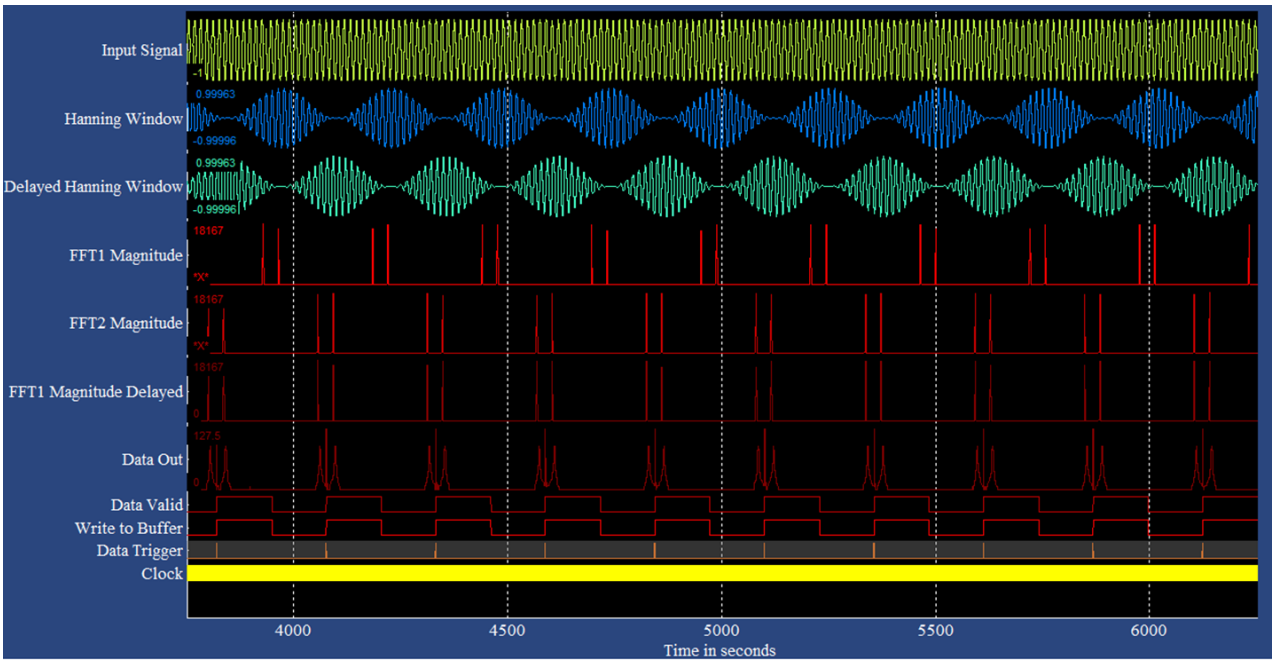


8.2 Integration Testing

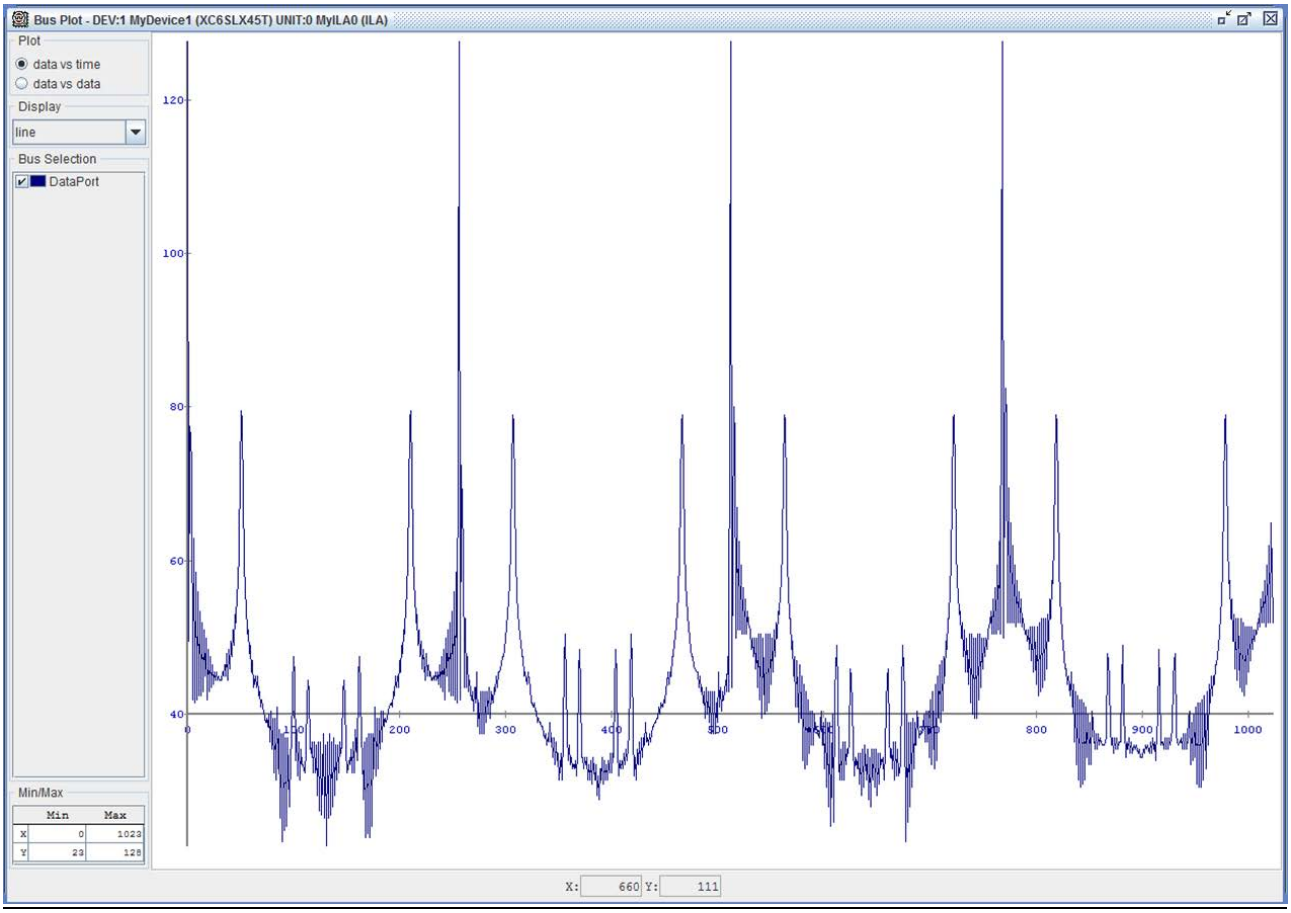
After unit testing, all of the individual modules were put together and tested as a single, integrated system. The image below shows this system.



The waveform below was used to visually inspect or probe various lines to validate the design.



Finally, Chipscope was used to test the entire system in real-time. The image below shows a screenshot of the functioning real-time spectrum analyzer.



Section 9

9.1 Evaluation and Recommendations

- Additional Time
    - More robust unit and integration testing
    - Further optimization
    - Provide an input buffer from ADC to meet the timing constraints
    - USB driver Development and interface
- Additional Resources
    - Higher system performance can be achieved with more advanced hardware
    - The team projects that the 150T version of the Spartan 6 can effectively run a 1024 point FFT.

9.2 Conclusion

The design failed to meet the speed specification of 204.8 M samples/sec. The report summary reveals the timing constraint of 4.9 ns failed due to the ADC interface. The ADC is designed to operate at 250 MHz so it should meet specifications. In one document for TI's ADC4229 (Application Report SLAA545 – Oct 2012), section 2.1 discusses a method of splitting the clock domain of the ADC from that of the FPGA. This method allows the two clock domains to remain at the same required rate of 204.8 MHz but they may be out of phase with each other. The clock rate would be maintained with the aid of a PLL (phase locked loop) which will also require the input data to be buffered. This method may provide a way to meet the timing requirements.

The design specifications require the data to be output via USB to a host computer. It was originally believed that this could be accomplished by averaging the results of two successive frames of data and then transmitting to the host. The USB2.0 specification indicates that in high speed mode, USB2.0 should be able to achieve a rate of 480 M bit/s. This is fine if drivers are available for the host PC to connect to the FPGA at that rate. The only drivers found to exist were USB UART drivers which will not meet the requirements. The design team does not have the expertise required to develop a driver that will meet the specification. Part of the reason for the requirement was for testing and evaluation purposes. As a workaround for testing, a Xilinx tool called Chipscope was implemented. This tool allowed for moderate testing of the design.

The design was not able to output data in 256 – 1024 bins. This would correlate to FFT point sizes of 512 – 2048. The final successful design uses an FFT point size of 256. The team was limited by a combination of FPGA hardware resources and the chipscope testing method. While it is possible to implement a 512 point FFT version of this design on this hardware, to reduce hardware resources, the FFT results must be output in bit reverse order. This means that the data would be presented to chipscope in a non-linear frequency bin order (ie X0, X256, X128, …, X512), making evaluation difficult. For more information on bit reversal refer to "LogiCORE IP Fast Fourier Transform v8.0" datasheet.

During the implementation of the Simulink design, the team discovered that the hardware resources required did not agree with predicted values from the datasheets. The datasheets reported values

derived from the Xilinx XC6SLX150T FPGA. It was expected that the XC6SLX45T FPGA would require the same resources but it was found to be otherwise. The datasheet predicted 16 DSP slices would be required for a 1024 point, streaming, bit reversed order, 16 bit data width, convergent rounding FFT. When generating design, it was found to require 24 DSP slices. The resource that was found to most limit the design was LUTs (Look-up tables). By moving most of the RAM to block RAM instead of distributed RAM the team was able to get the design to build.

With more optimization, time, and testing, the team is confident a 512-point FFT version of this design can be implemented on this board and meet all required specifications. The team recommends using a data transport other than the requested USB. The Ethernet platform available on the SP605 evaluation board will allow for a much larger bandwidth eliminating the need for sample averaging.

Section 10

10.1 References

[1] “LogiCORE IP Fast Fourier Transform v7.1” web.March 1, 2011 <http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf>.

[2] " Spartan-6 Family Overview" *xilinx*. Web. 3 Dec. 2012. <http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf>.

[3] " Spartan-6 FPGA Data Sheet:DC and Switching Characteristics" *xilinx*. (2011): Web. 3 Dec. 2012. <http://www.xilinx.com/support/documentation/data_sheets/ds162.pdf>.

[4] Wang, “FFT implementations”, January 2011.

[5] “LogiCORE IP Fast Fourier Transform v8.0” datasheet http://www.xilinx.com/support/documentation/ip_documentation/ds808_xfft.pdf

11.1 Appendix

```
% Implementation of a Fast Fourier Transform (FFT)

f1=20; % frequency 1 (Hz)
f2=70; % frequency 2 (Hz)
f3=35; % frequency 3 (Hz)

fs=1000; % sampling frequency (samples/second)

L=1; % length of signal (seconds)

Ts=1/fs; % sampling period (seconds)

t=[0:Ts:L]; % discrete time domain

x=0.2*cos(2*pi*f1*t)+0.15*cos(2*pi*f2*t)+0.3*cos(2*pi*f3*t); % sampled signal

N=length(x); % number of samples

sigma=0.1;

noise=sigma*randn(1,N);

x=x+noise;

plot(t,x)

title('Time Domain Signal Representation');
xlabel('t (s)');
ylabel('x(t)');

X=fft(x);

f=[0:N-1]/L; % discrete frequency domain

Xm=((2*abs(X)/N).^2)./2; % power spectrum

Xm=10*log10(1000*Xm); % convert to dBm

figure(2)
plot(f(1:(N-1)/2), Xm(1:(N-1)/2)) % plot half of the power spectrum

title('Fast Fourier Transform (FFT)');
xlabel('f (Hz)');
ylabel('Power (dBm)');
```

VHDL CODE FOR LOG BLOCK:

VHDL CODE:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--Final Port mapping
entity db_converter is
   Port ( DB_INPUT : in  STD_LOGIC_VECTOR (15 downto 0);
          clk : in  STD_LOGIC;
          ce : in STD_LOGIC := '1';
          DB_CONVERTED : out  STD_LOGIC_VECTOR (7 downto 0));
end db_converter;

architecture Behavioral of db_converter is

signal six_N1: STD_LOGIC_VECTOR (7 downto 0);
signal X1 :STD_LOGIC_VECTOR (4 downto 0);
signal six_log_value1 : STD_LOGIC_VECTOR (3 downto 0);
signal delayed_six_N1 :  STD_LOGIC_VECTOR (7 downto 0);
component six_log_two
            port(
                        X : in  STD_LOGIC_VECTOR (4 downto 0);
                        clk: in  STD_LOGIC;
                        ce: in  STD_LOGIC := '1';
                        six_log_value : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

component db
```

```
        port(
                DB_INPUT : in  STD_LOGIC_VECTOR (15 downto 0);
    clk : in  STD_LOGIC;
                ce : in STD_LOGIC := '1';
    six_N : out  STD_LOGIC_VECTOR (7 downto 0);
                X: out  STD_LOGIC_VECTOR (4 downto 0));
end component;

component adder
        port(
                six_log_value : in  STD_LOGIC_VECTOR (3 downto 0);
        six_n : in  STD_LOGIC_VECTOR (7 downto 0);
        db_out : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component dflipflop
   Port ( six_N : in  STD_LOGIC_VECTOR (7 downto 0);
                        clk : in  STD_LOGIC;
                        ce :in  STD_LOGIC := '1';
        delayed_six_N : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

begin

U1: db
```

```
port map(
                DB_INPUT => DB_INPUT,
    clk => clk,
                ce => ce,
    six_N => six_N1,
                X => X1);

U2: six_log_two

port map(
                X => X1,
                clk => clk,
                ce => ce,
                six_log_value => six_log_value1 );

U3: adder

port map(
                six_log_value => six_log_value1,
    six_n => delayed_six_N1,
    db_out => DB_CONVERTED );

U4: dflipflop
```

```
port map(
                six_N => six_N1,
                clk => clk,
                ce => ce,
     delayed_six_N => delayed_six_N1);
end Behavioral;
----------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity dflipflop is
    Port ( six_N : in  STD_LOGIC_VECTOR (7 downto 0);
                        clk : in  STD_LOGIC;
                        ce :in  STD_LOGIC := '1';
          delayed_six_N : out  STD_LOGIC_VECTOR (7 downto 0));
end dflipflop;

architecture Behavioral of dflipflop is
begin
process(six_N,clk)
begin
if(clk'EVENT AND clk = '1') then
 delayed_six_N <= six_N;
end if;

end process;
```

```
end Behavioral;

----------------------------------------------------------------------------------------------

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder is

   Port ( six_log_value : in  STD_LOGIC_VECTOR (3 downto 0);

          six_n : in  STD_LOGIC_VECTOR (7 downto 0);

          db_out : out  STD_LOGIC_VECTOR (7 downto 0));

end adder;


architecture Behavioral of adder is

begin

process( six_log_value, six_n)

begin


db_out <= six_n + ("0000" & six_log_value);


end process;

end Behavioral;

----------------------------------------------------------------------------------


library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
```

```
entity six_log_two is
   Port ( X : in  STD_LOGIC_VECTOR (4 downto 0);
                        clk: in  STD_LOGIC;
                                ce: in  STD_LOGIC := '1';
         six_log_value : out  STD_LOGIC_VECTOR (3 downto 0));
end six_log_two;

architecture Behavioral of six_log_two is
begin
process(X, clk)
begin

            if(clk'EVENT AND clk = '1') then
                        if   (X = "00000") then six_log_value <= "0000";
                        elsif(X = "00001") then six_log_value <= "0001";
                        elsif(X = "00010") then six_log_value <= "0001";
                        elsif(X = "00011") then six_log_value <= "0010";
                        elsif(X = "00100") then six_log_value <= "0010";
                        elsif(X = "00101") then six_log_value <= "0011";
                        elsif(X = "00110") then six_log_value <= "0011";
                        elsif(X = "00111") then six_log_value <= "0011";
                        elsif(X = "01000") then six_log_value <= "0100";
                        elsif(X = "01001") then six_log_value <= "0100";
                        elsif(X = "01010") then six_log_value <= "0101";
```

```
                elsif(X = "01011") then six_log_value <= "0101";
                elsif(X = "01100") then six_log_value <= "0110";
                elsif(X = "01101") then six_log_value <= "0110";
                elsif(X = "01110") then six_log_value <= "0110";
                elsif(X = "01111") then six_log_value <= "0111";
                elsif(X = "10000") then six_log_value <= "0111";
                elsif(X = "10001") then six_log_value <= "0111";
                elsif(X = "10010") then six_log_value <= "1000";
                elsif(X = "10011") then six_log_value <= "1000";
                elsif(X = "10100") then six_log_value <= "1000";
                elsif(X = "10101") then six_log_value <= "1001";
                elsif(X = "10110") then six_log_value <= "1001";
                elsif(X = "10111") then six_log_value <= "1001";
                elsif(X = "11000") then six_log_value <= "1010";
                elsif(X = "11001") then six_log_value <= "1010";
                elsif(X = "11010") then six_log_value <= "1010";
                elsif(X = "11011") then six_log_value <= "1011";
                elsif(X = "11100") then six_log_value <= "1011";
                elsif(X = "11101") then six_log_value <= "1011";
                elsif(X = "11110") then six_log_value <= "1011";
                else six_log_value <= "1100";
                end if;
        end if;
end process;
end Behavioral;
```

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

entity db is
   Port ( DB_INPUT : in  STD_LOGIC_VECTOR (15 downto 0);
          clk : in  STD_LOGIC;
                        ce : in STD_LOGIC := '1';
          six_N : out  STD_LOGIC_VECTOR (7 downto 0);
                        X: out  STD_LOGIC_VECTOR (4 downto 0));
end db;

architecture Behavioral of db is
begin

process(DB_INPUT , clk)
variable DB_INPUT1 : STD_LOGIC_VECTOR (15 downto 0);
begin
          DB_INPUT1 := DB_INPUT + 1;
          if(clk'EVENT AND clk = '1') then
                    if(DB_INPUT1(15) = '1') then
                    six_N <= "10110100";
                    X <= DB_INPUT1 (14 downto 10);
```

```
elsif(DB_INPUT1(14) = '1') then
six_N <= "10101000";
X <= DB_INPUT1 (13 downto 9);
elsif(DB_INPUT1(13) = '1') then
six_N <= "10011100";
X <= DB_INPUT1 (12 downto 8);
elsif(DB_INPUT1(12) = '1') then
six_N <= "10010000";
X <= DB_INPUT1 (11 downto 7);
elsif(DB_INPUT1(11) = '1') then
six_N <= "10000100";
X <= DB_INPUT1 (10 downto 6);
elsif(DB_INPUT1(10) = '1') then
six_N <= "01111000";
X <= DB_INPUT1 (9 downto 5);
elsif(DB_INPUT1(9) = '1') then
six_N <= "01101100";
X <= DB_INPUT1 (8 downto 4);
elsif(DB_INPUT1(8) = '1') then
six_N <= "01100000";
X <= DB_INPUT1 (7 downto 3);
elsif(DB_INPUT1(7) = '1') then
six_N <= "01010100";
X <= DB_INPUT1 (6 downto 2);
elsif(DB_INPUT1(6) = '1') then
```

```
six_N <= "01001000";
X <= DB_INPUT1 (5 downto 1);
elsif(DB_INPUT1(5) = '1') then
six_N <= "00111100";
X <= DB_INPUT1 (4 downto 0);
elsif(DB_INPUT1(4) = '1') then
six_N <= "00110000";
X <= DB_INPUT1 (3 downto 0) & '0' ;
elsif(DB_INPUT1(3) = '1') then
six_N <= "00100100";
X <= DB_INPUT1 (2 downto 0) & "00";
elsif(DB_INPUT1(2) = '1') then
six_N <= "00011000";
X <= DB_INPUT1 (1 downto 0) & "000";
elsif(DB_INPUT1(1) = '1') then
six_N <= "00001100";
X <= DB_INPUT1 (0) & "0000";
elsif(DB_INPUT1(0) = '1') then
six_N <= "00000000";
X <= "00000";
else
six_N <= "11000000";
X <= "00000";
end if;
end if;
```

```
end process;
end Behavioral;
```

VHDL code for MAGNITUDE BLOCK

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
entity mag is
        Port ( realpart : in  STD_LOGIC_VECTOR (15 downto 0);
        imaginary : in  STD_LOGIC_VECTOR (15 downto 0);
        enable : in  STD_LOGIC;
        clk : in  STD_LOGIC;
        ce : in  STD_LOGIC;
      magnitude : out  STD_LOGIC_VECTOR (15 downto 0));
end mag;
architecture Behavioral of mag is
begin
process(clk,realpart,imaginary, enable)
variable real1:STD_LOGIC_VECTOR (15 downto 0);
variable imaginary1:STD_LOGIC_VECTOR (15 downto 0);
begin
real1:=abs( signed(realpart));
```

```
imaginary1:=abs( signed(imaginary));
if(clk'EVENT AND clk = '1') then
        if(enable = '1') then
                if(real1 >= imaginary1) then
                        magnitude <= real1 + ('0'&(imaginary1(15 downto 1)));
                else
                        magnitude <= imaginary1 + ('0'&(real1(15 downto 1)));
                end if;
         end if;
end if;
end process;
end Behavioral;
```