2013

FPGA-BASED IMPLEMENTATION OF DUAL-FREQUENCY PATTERN SCHEME FOR 3-D SHAPE MEASUREMENT

Brent Bondehagen
University of Kentucky, brentbondehagen@gmail.com

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained and attached hereto needed written permission statements(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine).

I hereby grant to The University of Kentucky and its agents the non-exclusive license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless a preapproved embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's dissertation including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Brent Bondehagen, Student

Dr. J. Robert Heath, Major Professor

Dr. Zhi David Chen, Director of Graduate Studies

FPGA-BASED IMPLEMENTATION OF DUAL-FREQUENCY PATTERN SCHEME FOR 3-D SHAPE MEASUREMENT

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Electrical Engineering in the
College of Engineering
at the University of Kentucky

By

Brent Bondehagen

Lexington, Kentucky

Co-Directors: Dr. J. Robert Heath, Associate Professor of Electrical Engineering
and Dr. Daniel L. Lau, Associate Professor of Electrical Engineering

Lexington, Kentucky

2013

ABSTRACT OF THESIS

FPGA-BASED IMPLEMENTATION OF DUAL-FREQUENCY PATTERN SCHEME FOR 3-D SHAPE MEASUREMENT

Structured Light Illumination (SLI) is the process where spatially varied patterns are projected on to a 3-D surface and based on the distortion by the surface topology, phase information can be calculated and a 3D model constructed. Phase Measuring Profilometry (PMP) is a particular type of SLI that requires three or more patterns temporarily multiplexed. High speed PMP attempts to scan moving objects whose motion is small so as to have little impact on the 3-D model. Given that practically all machine vision cameras and high speed cameras employ a Field Programmable Gate Array (FPGA) interface directly to the image sensors, the opportunity exists to do the processing on camera. This thesis focuses on the design, implementation, testing, and evaluation of a camera-projector system to implement a PMP dual-frequency scheme for 3-D shape measurement on a single FPGA chip. The processor architecture is implemented and tested using the Xilinx Spartan 3 FPGA chip on an Opal Kelly development board. The hardware is described using VHDL and Verilog Hardware Description Languages (HDLs).

KEYWORDS: Structured Light Illumination, Phase Measuring Profilometry, 3-D Shape Measurement, Hardware Description Language, Processor Architecture

Brent Bondehagen

May 3, 2013

FPGA-BASED IMPLEMENTATION OF DUAL-FREQUENCY PATTERN SCHEME FOR 3-D SHAPE MEASUREMENT

By

Brent Bondehagen

Dr. J. Robert Heath
Co-Director of Thesis

Dr. Daniel L. Lau
Co-Director of Thesis

Dr. Zhi David Chen
Director of Graduate Studies

May 3, 2013
Date

ACKNOWLEDGMENTS

I wish to thank, first and foremost, my parents for their unconditional support, both financially and emotionally throughout my academic studies.

I would like to thank my advisors, Dr. Daniel Lau and Dr. Robert Heath for their encouragement, guidance and support throughout the duration of my research work.

In addition, I would like to thank my committee member, Dr. Hank Dietz for his assistance and expertise.

Last, but not least, I want to thank my wife, Huiyu, for her love and support. Her encouragement and confidence in me kept me optimistic and motivated.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Three dimensional (3-D) shape measurement is an active area of research [1] where data of an object's surface is collected and a digital model with depth data can be constructed. Such technology has a myriad of potential applications in nearly all areas of science and industry such as medical imaging, product inspection, security surveillance, computer vision, entertainment, etc. [3].

There are many different technologies for 3-D measurement purposes. There are contact techniques, where the object must be physically probed to get the data, and non-contact techniques where triangulation or time-of-flight are used. This thesis focuses on the design, development, testing and evaluation of a hardware prototype and computer architecture implemented in FPGA technology that implements an algorithm to calculate data for 3-D reconstruction.

1.1 PMP Structured Light Illumination

One such non-contact technique using triangulation is Structured Light Illumination (SLI) where light patterns are projected on to a 3-D surface and information based on the distortion by the surface topology is collected. These patterns can be spots, stripes or other geometric shapes [5]. Phase Measuring Profilometry (PMP) is a popular

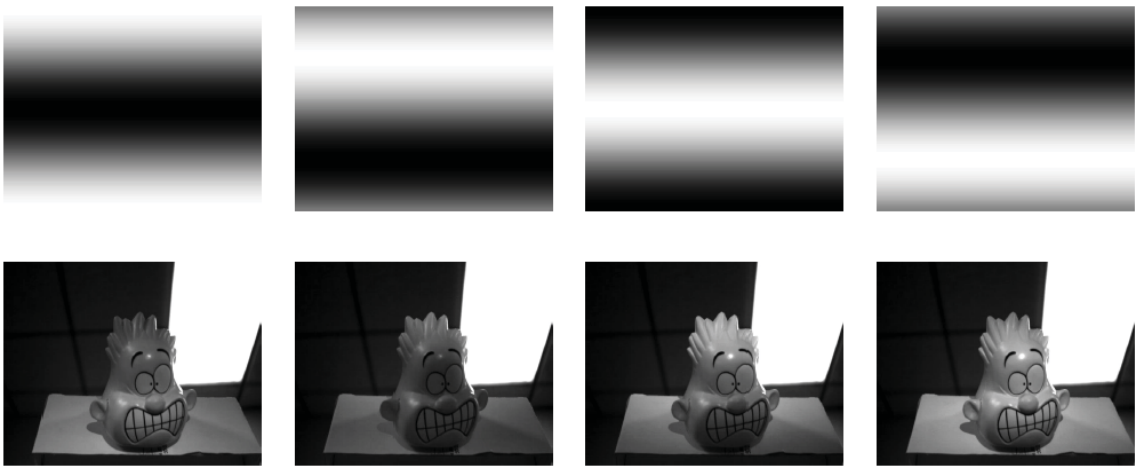


Figure 1.1: Projected PMP patterns with $N = 4$, $f = 1$, $A^p = 127.5$ and $B^p = 127.5$ (top) with Corresponding Patterns Shown on Object (bottom) as Presented in K. Liu.

method of SLI due to its high reliability and high accuracy[4]. In this technique multiple phase shifted patterns are projected onto a 3-D surface and the depth data is computed based on the phase distortion observed by the camera. In comparison to other SLI techniques, PMP is known for its high accuracy, robustness to the environmental illumination and target texture, simple implementation, and fast point matching in 3D reconstruction [5]. As explained in Kai *et al.* [2], the following PMP patterns are described as

$$I_n^p(x^p, y^p) = A^p(x^p, y^p) + B^p(x^p, y^p)\cos(2\pi f y^p - \frac{2\pi n}{N}) \tag{1.1}$$

where I_n^p is the intensity of the pixel, (x^p, y^p) is the coordinates of a pixel in the projector, A^p and B^p are some constants, f is the frequency of the sine wave, n represents the phase-shift index, and N is the total number of phase shifts. Figure 1.1 shows example PMP patterns.

For the reconstruction, the camera captures each pattern projected on an object

and distorted based on the surface topology expressed in the following equation as

$$I_n^c(x^c, y^c) = A^c(x^c, y^c) + B^c(x^c, y^c)\cos(\phi(x^c, y^c) - \frac{2\pi n}{N}) \tag{1.2}$$

where (x^c, y^c) is the coordinate of a pixel in the camera and $I_n^c(x^c, y^c)$ is the intensity of the pixel. Pixel coordinates will be removed henceforth for simplification. The term A^c is the average pixel intensity in the pattern set:

$$A^c = \frac{1}{N}\sum_{n=0}^{N-1} I_n^c \tag{1.3}$$

such that the image of A^c is equal to the intensity of the frame of the scene. The term B^c is the intensity modulation from a given pixel derived from I_n^c as:

$$B^c = \frac{2}{N}\left\{\left[\sum_{n=0}^{N-1} I_n^c \sin\left(\frac{2\pi n}{N}\right)\right]^2 + \left[\sum_{n=0}^{N-1} I_n^c \cos\left(\frac{2\pi n}{N}\right)\right]^2\right\}^{0.5} \tag{1.4}$$

where B^c indicates the amplitude of the sinusoid reflecting off of a point on the object surface. B^c is chosen equal to the magnitude of k=1 Discrete Fourier Transform (DFT) coefficient. For pixels where I_n^c is constant or not affected by the projected sinusoid patterns, B^c will be close to zero so B^c is employed as a shadow or noise filter. Figure 1.2 shows an example scene with A^c and B^c images.

For reliable pixels with a large B^c, ϕ represents the phase value of the captured sinusoid pattern derived as:

$$\phi = \tan^{-1}\left[\frac{\sum_{n=0}^{N-1} I_n^c \sin\left(\frac{2\pi n}{N}\right)}{\sum_{n=0}^{N-1} I_n^c \cos\left(\frac{2\pi n}{N}\right)}\right] \tag{1.5}$$

With the phase values calculated and triangulation with the projector, depth information can be calculated. A system diagram can be seen in Figure 1.2.

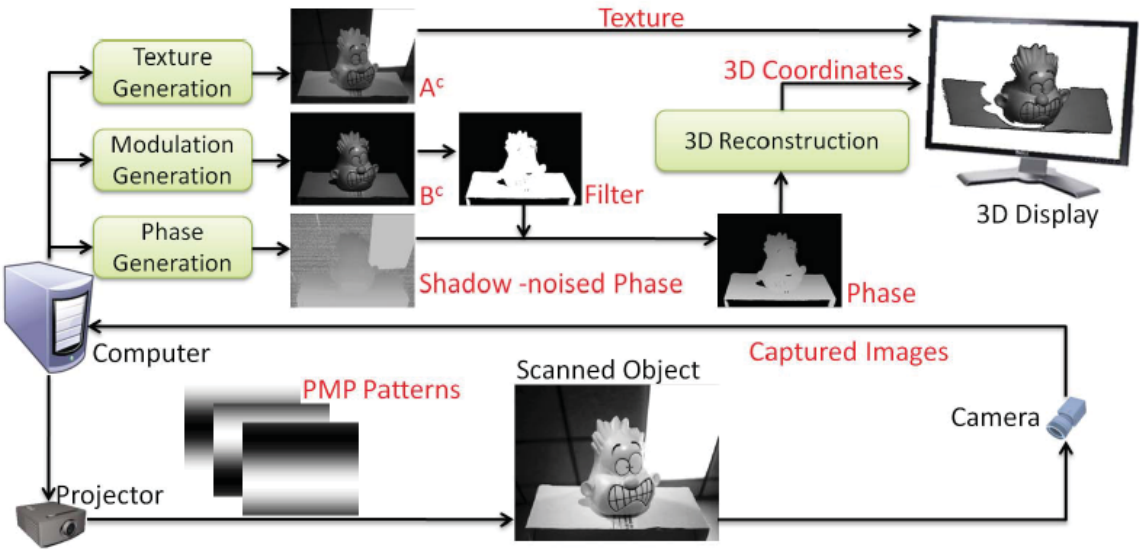


Figure 1.2: PMP System as Presented by L. Kai *et al.*

1.1.1 Dual Frequency Scheme

In order to minimize the effects of sensor noise while minimizing the expense of performing phase unwrapping, Kai *et al.* propose a dual-frequency pattern defined as:

$$I_n^p = A^p + B_1^p \cos(2\pi f_u y^p - \frac{2\pi n}{N}) + B_2^p \cos(2\pi f_h y^p - \frac{4\pi n}{N}) \tag{1.6}$$

where I_n^p is the intensity of a pixel in the projector, A^p, B_1^p, and B_2^p are constants to make the value of I_n^p between 0 and 255 for a 8-bit depth projector, f_u is the unit frequency of the sine wave, f_h is the high frequency of the sine wave equal to 1, n represents the phase-shift index, and N is the total number of phase shifts. A sample dual frequency pattern and its cross-section is shown in Figure 1.3.

Using the dual frequency patterns, Eq. 1.2 then becomes

$$I_n^c = A^c + B_1^c \cos(\phi_u - \frac{2\pi n}{N}) + B_2^c \cos(\phi_h - \frac{4\pi n}{N}) \tag{1.7}$$

where I_n^c is the intensity of the pixel in the camera. The term A^c is still the average pixel intensity across the pattern set as defined by Eq. 1.3. B_n^c is the intensity

modulation of a pixel corresponding to ϕ_h and derived from I_n^c as:

$$B_m^c = \frac{2}{N}\left\{\left[\sum_{n=0}^{N-1} I_n^c \sin\left(m\frac{2\pi n}{N}\right)\right]^2 + \left[\sum_{n=0}^{N-1} I_n^c \cos\left(m\frac{2\pi n}{N}\right)\right]^2\right\}^{0.5} \tag{1.8}$$

with $m = 1$ where B_1^c can be thought of as the amplitude of the sinusoid reflecting off of a point on the object surface corresponding to ϕ_h. Similar to the traditional PMP described above B_1^c is used as a noise filter and B_2^c for $m = 2$ is the intensity modulation corresponding to ϕ_u. For reliable pixels with a large B_1^c, the phase-pair (ϕ_u, ϕ_h) is then derived as :

$$(\phi_u, \phi_h) = \left(\tan^{-1}\left[\frac{\sum_{n=0}^{N-1} I_n^c \sin\left(\frac{2\pi n}{N}\right)}{\sum_{n=0}^{N-1} I_n^c \cos\left(\frac{2\pi n}{N}\right)}\right], \tan^{-1}\left[\frac{\sum_{n=0}^{N-1} I_n^c \sin\left(\frac{4\pi n}{N}\right)}{\sum_{n=0}^{N-1} I_n^c \cos\left(\frac{4\pi n}{N}\right)}\right]\right) \tag{1.9}$$

where ϕ_h represents the wrapped phase value of the captured pattern and ϕ_u represents the base phase used to unwrap ϕ_h. These are also equal to the phase value of the k=1 and k=2 DFT coefficients.

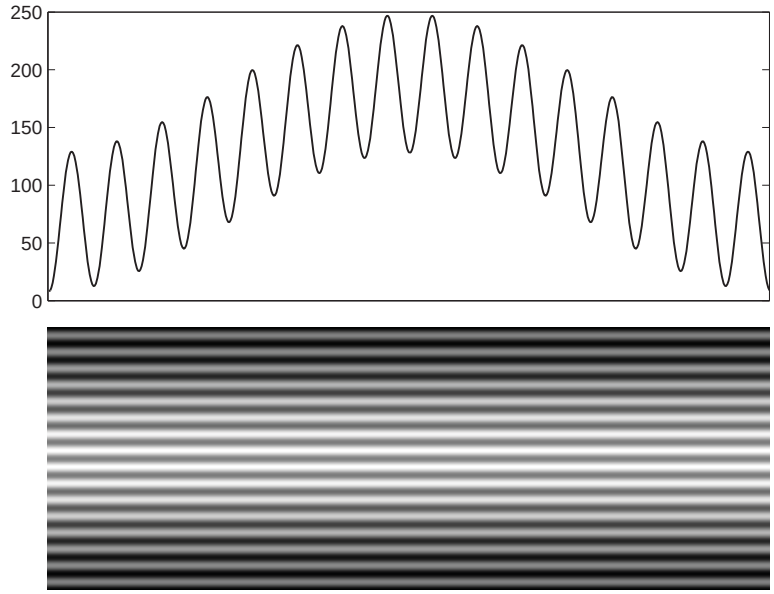


Figure 1.3: Cross-section of a Dual-Frequency Pattern (top) and its Corresponding Pattern (bottom).

The inverse tangent function returns a value within the interval $(-\pi, \pi]$. If the

phase exceeds this range, as is the case of frequencies higher than 1, it will be wrapped, or repeated, around so it remains in this range thereby introducing 2π jumps. Therefore, the wrapped phase needs to be unwrapped. The high frequency phase is the robust, noise resilient phase but has phase ambiguity due to the jumps. The unit frequency phase is the noisy phase but doesn't have the 2π jump. The higher frequency phase, ϕ_h can be unwrapped using the unit-frequency phase ϕ_u as a reference thereby removing wrapped phase ambiguities. Because the high and unit frequencies are projected together as opposed to being projected separately, the acquisition speed is faster and more suitable for real-time applications.

With the phase ϕ, and camera pixels (x^c, y^c), triangulation can be used and 3-D world coordinates can be calculated.

1.1.2 Lookup Table Calculations

In addition to the novel dual-frequency pattern scheme described in the subsection above, Kai *et al.* introduced the idea of using a lookup table (LUT) for real-time video reconstruction. The arctangent and square root functions are computationally expensive so double precision float values were stored into a LUT giving high precision results and higher frame rates at low computational cost.

1.2 Contribution of Thesis

Given that practically all machine vision cameras and high speed cameras employ a Field Programmable Gate Array (FPGA) chip interface directly to the image sensors and that such cameras are used for high-speed capture of PMP patterns, the oppor-

tunity exists to move some or all of the the processing from the PC to the FPGA interfacing the camera. This allows for the ability to read frames from the image sensor and perform calculations simultaneously. Custom parallel computation logic can also be implemented in the FPGA .

This thesis focuses on the design and implementation of a camera-projector system which implements a PMP dual-frequency scheme for 3-D shape measurement on a single FPGA chip that could be used in high-speed cameras. The architecture calculates phase and amplitude data. The hardware prototype performance will depend greatly on the limitations of the FPGA chip, peripherals, and interfaces. Therefore, the focus of this thesis is to present the computer architecture and design of a working system with the available hardware.

1.3 Previous Research on FPGA Implementation on 3-D Scanning Algorithms

There have been a number of papers written for implementing 3-D scanning algorithms in FPGA technology. Y. Oike *et al.* [30] presented a real-time 3-D imaging system based on the light-section method with VGA pixel resolution. The FPGA performs the sensor control, the light projection, and range data pre-processing (triangulation). S. Lee *et al.* [29] present a real-time 3-D camera based on infrared structured light for robots. An FPGA is used to interface the Digital Mirror Device (DMD) and CCD camera module. A second FPGA is used to implement the Hierarchical Orthogonal Coding for depth images. B. Hong *et al.* [28] present a DMD, CMOS sensor and FPGA on a single board. The CMOS sensor runs at 500 Hz with

range information frame rate of 17 Hz. Gray code structured light is used. A. Peter *et al.* [27] present an efficient algorithm for determining phase using the CORDIC function in a time of flight range imaging system. J. Weingarten, G. Gruener, and R. Siegwart present a time-of-flight sensor with a FPGA interface that calculates phase for four phase shifted patterns and filters by amplitude threshold for robot navigation. S. Bellis and W. Marnane [31] present an FPGA implementation of a pipelined CORDIC arctangent unit for a 3D camera system with four phase shifted patterns. Magnitude data is also used to null pixels when the contrast is low.

In the research and developed prototype presented in this paper, the dual-frequency PMP scheme is presented giving more robust data with little expense due to unwrapping. In previous PMP using FPGA research, four phase shifted patterns are used for simplicity so a DFT algorithm is not necessary. In this thesis, an architecture for eight phase shifted patterns using the FFT algorithm is developed with the ability to easily modify it for a different number of patterns. A simplified algorithm to calculate magnitude is used as well. The prototype system described here allows for an external projector to be connected via a VGA interface.

1.4 Thesis Objectives

The objectives of this thesis are as follows:

- Study the research done by Kai et al. and describe a hardware architecture design to implement the calculation of the phase for the dual-frequency algorithm.

- Capture frames from a CMOS image sensor and write frames into SDRAM at 60Hz.
- Project Structured Light patterns on to an object by driving a VGA board.
- Design and build a prototype system controlled by a single FPGA that can calculate dual-frequency pattern phase and magnitude values.
- Use the magnitude data to as a shadow or noise filter for the phase data.
- Compare results of the FPGA implementation with results from a Matlab script.
- Suggest possible improvements to the system and architecture.

1.5 Thesis Outline

This thesis consists of twelve chapters.

Chapter 1 introduced Structured Light Illumination and the algorithm. The contribution and objectives of this thesis were explained.

Chapter 2 explains the FPGA architecture and considerations to take when designing for implementation on an FPGA.

Chapter 3 gives an overview of the hardware system.

Chapter 4 discusses the VGA standard, the built VGA board, and the implementation of a VGA controller.

Chapter 5 explains how images are captured and written into the SDRAM.

Chapter 6 provides an understanding of the overall phase pipeline architecture that was designed.

Chapter 7 discusses the Fourier Transform and its implementation in the FPGA.

Chapter 8 explains the CORDIC core use and implementation.

Chapter 9 describes how the two phases are combined into one phase.

Chapter 10 deals with the GUI Application developed.

Chapter 11 provides the results and timing analysis of the system.

Chapter 12 concludes the thesis and suggests possible improvements.

Chapter 2

FPGA Design

2.1 Spartan 3

The Xilinx Spartan-3 family of FPGAs are designed for high volume applications with a high price-performance ratio. The Spartan 3 is suited for consumer electronic applications such as digital televisions, display equipment and home networking [11]. Programmability, low-cost, improved energy efficiency and performance make the FPGA a viable alternative to ASICs in many applications such as this one.

The Spartan 3 family architecture consists of the following programmable functional elements :

- **Configurable Logic Blocks (CLBs)** are the primary resource in the FPGA for implementing combinational and synchronous logic. A CLB consists of four slices where a slice consists of two Look-up Tables (LUTs) for logic or memory and storage elements for flip-flops or latches. CLBs are interconnected in a two-dimensional array. LUTs can be used to implement distributed RAM for register files and FIFOs.

- **Input/Output Blocks (IOBs)** provide a programmable input, output and bidirectional interface between the internal logic and the I/O pins.

- **Block RAM** is synchronous 18Kbit blocks. It can be combined to form RAM blocks of greater width and depth. It has a lower access latency and stores more data using less resources compared to distributed RAM.

- **Multiplier Blocks** have dedicated multipliers that accept two 18-bit binary numbers.

- **Digital Clock Manager(DCM)**s utilize a Delay-Locked Loop (DLL). It can phase-shift output signals with respect to its input and can eliminate clock-skew by aligning its output clock with an input clock so there is no phase difference. It can also multiply and divide clocks to create a new frequency. A DCM is used in the design to phase align the PLL generated clock that is used by the SDRAM and shared with the FPGA internal fabric clock.

2.2 FPGA Design Flow

When designing a system targeting FPGA technology, Computer Aided Design (CAD) tools such as the Xilinx ISE Design Suite are used to take a design description and implement it to an FPGA. The input to the CAD tool is a Hardware Description Language (HDL) such as the popular Verilog and VHDL (Very High Speed Hardware Description Language) HDLs. With the design in HDL code, the CAD tool synthesizes the code. Synthesis translates the HDL into a netlist of the primitive components and their connections. In the Placement step, the primitive components of the netlist are mapped to a physical logic block. For instance, a NAND gate could be implemented with one of the thousands of LUTs in the logic blocks in a Spartan

3. If the component is placed intelligently, the connection could be faster and easier for the router tool. In the routing step, decisions on how each signal gets from where it is to its destination are made. With the large number of possible paths this part can take the longest. Lastly, a FPGA programming file is generated consisting of a bit stream controlling every programmable element in the chip.

2.3 Core Cells

Core cells, or cores, are modules that have been designed and tested for a specific function. There are cores for processors, memory interfaces, signal processing, and other functionality. Rather than focusing on developing such modules, an core can be used to speed up development and build larger systems. Cores are divided into three categories: soft cores, firm cores, and hard cores.

Soft cores are in the form of synthesizable RTL logic or a gate-level netlist. Firm cores are commonly gate-level netlist for placement and routing. Xilinx offers firm cores that are mapped to Xilinx macros which the CAD tool can convert to lower level primitives. Hard cores contain layout information and cannot be changed. [8]. Xilinx and Altera both provide soft-core processors optimized for their FPGAs. Hard core processors have performance and power benefits over soft cores. Xilinx incorporated PowerPC hard cores in its high-end Virtex-4 FPGAs. ARM architecture cores are now included in some of the latest FPGAs [26]. Often cores are subject to intellectual property rights such as patents and copyrights so they are licensed to users; these are called IP cores. Xilinx offers parameterized IP cores with its ISE design software that are generated using the Xilinx Core Generator Wizard. Such cores are optimized to

the specific architecture so they effectively use available resources.

2.4 Binary Data Representation

Two common ways to represent numbers in binary are fixed-point and floating point numbers. Fixed-point arithmetic is simple so it is often used on systems where resources are limited such as an FPGA. A floating point number can represent a wide range of numbers with fewer bits than fixed-point but floating point arithmetic uses more resources.

2.4.1 Fixed Point

Fixed-point numbers make it easy to express fractional numbers. A fixed point number has a specific number of bits for the integer and a specific number for the fractional part of the number (to the right of the fixed point). The 'fixed-point' is the same as the decimal point of the base 10 number system and refers to the fact that the binary point location does not change. One notation to represent a fixed point number is Qm.f where there is a sign bit, m-1 integer bits and f fraction bits. For example, a Q3.29 number has one sign bit, two integer bits, and 29 fractional bits in a 32 bit word. The precision of a fixed point number is based on the number of fractional bits. With f fractional bits, there is a scaling factor of 2^{-f} and the minimum representable value is $-2^{m-1}/2^f$ and the maximum is $(2^{m-1}-1)/2^f$. A fixed-point number is simply an integer number that is scaled to a specific factor. For example, a binary number of "01110000" is equal to 112 but in Q2.6 form, it is equal to $112/2^6 = 1.75$. The number is scaled down by 2^6.

2.4.2 Floating Point

Unlike the fixed point number, floating point does not reserve a specific number of bits for the integer or fractional part. It has a sign bit, a certain number of bits for the actual number called the mantissa, or significand, and a specific number of bits for the exponent that tells where within the significand the decimal place is located. A floating point number is similar to scientific notation in that it is a normalized number.

Chapter 3

System Overview

Structured light illumination systems commonly use a projector to cast patterns on to an object or scene, a camera to capture the scene, and a PC to read in the frames from the camera and calculate the data needed to reconstruct a 3-D model. A fully functional hardware prototype was built with an FPGA that interfaces with a VGA interface to control the projection of the patterns, a camera sensor running at 60 Hz to capture frames of a scene, and a USB interface to transfer the calculated data to a PC. Figure 3.1 shows a high-level diagram of the system.

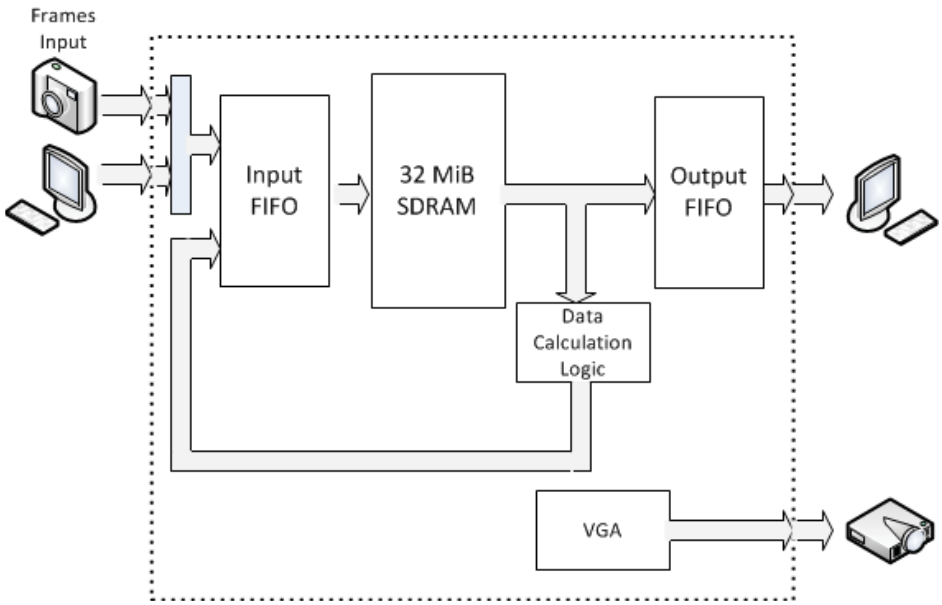


Figure 3.1: An Overall Diagram of the System

Patterns that are stored in the memory are projected onto a scene by a projector

via a VGA interface. The camera captures a specified number of frames of the scene and writes them to an input First-In First-Out memory (FIFO). When the FIFO has enough data, a block of data is written to memory. After the frames are captured, they are processed and phase data for reconstruction is calculated and stored back to memory. On request, the frames captured and calculated phase data are transferred over USB to the host computer.

The following sections serve to introduce the hardware that was utilized for the prototyping of this project. Design decisions relating to the available hardware and limitations are discussed.

3.1 Opal Kelly development board

An off-the-shelf FPGA development board (XEM3010-1500) from Opal Kelly Incorporated was chosen for this project. It provides a complete system for prototype development with a Spartan 3 FPGA chip. The board includes 32 MByte SDRAM, PROM for configuration storage, USB 2.0 interface for communication, Multi-PLL for clock management, and an ample number of user I/O ports in a small form factor. The board is distributed with libraries for Opal Kelly's own Application Programmer's Interface (API) in C, C++, C#, Ruby, Python, and Java. The API is used to communicate with the board, transfer data, and expedite development.

3.1.1 FPGA

The development board has a 1500K gate Spartan-3 Xilinx FPGA chip. It has a documented 3,328 CLBs with 576Kb block RAM and up to 208Kb of distributed

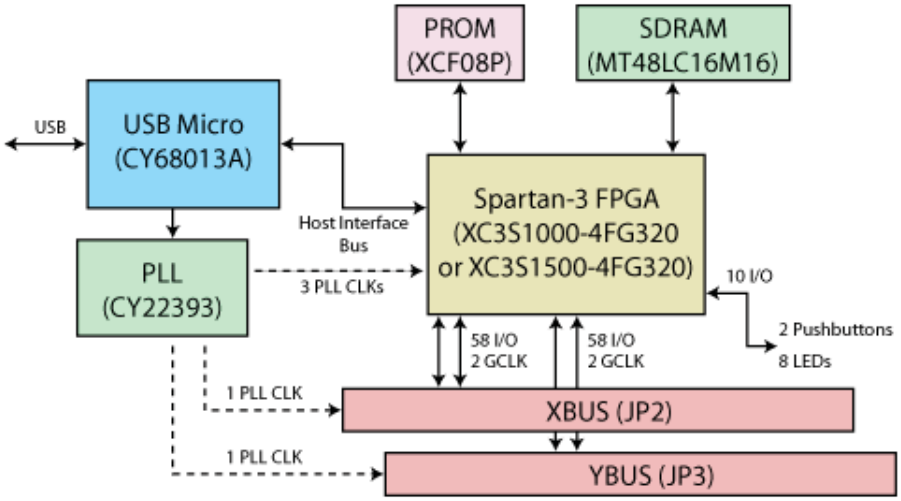


Figure 3.2: Functional Block Diagram of Opal Kelly Development Board. [20]

RAM. The Spartan-3 is a low-cost family and offers just enough resources for this application. The family has been discontinued and no longer is recommended for designs. It has been replaced with the Spartan-6 and Artix-7 families [11]. Due to timing constraints and memory usage, a higher performance FPGA family with more block RAM and lower fabric latency would be suggested for future improvements.

3.1.2 SDRAM

The Opal Kelly board has a 32-MByte SDRAM (Micron MT48LC16M16). It runs at a clock rate of 100MHz. The banks are organized as 8192 rows by 512 columns by 16 bits. Read and write accesses to the SDRAM are burst-oriented and a full page (512 x 16 bits) is written or read during an access [18]. One thing to note is that it is Single Data Rate (SDR) synchronous SDRAM. This means that only one word of data is transmitted per clock cycle. The SDR SDRAM reduces system performance. This is unlike a double data rate (DDR) SDRAM which allows data transfer on the rising and falling edge of the clock. Using DDR memory in the future would allow

for a higher memory bandwidth.

3.2 Leopard Imaging WVGA Camera Board

This board by Leopard Imaging incorporates an Aptina MT9V034 1/3 inch CMOS digital image sensor which supports CS-Mount or M12 lens. It supports a frame rate of up to 60Hz with a 752 by 480 resolution. A 60 Hz or higher frame rate is ideal so as to reduce any motion in the scene as multiple frames must be captured.

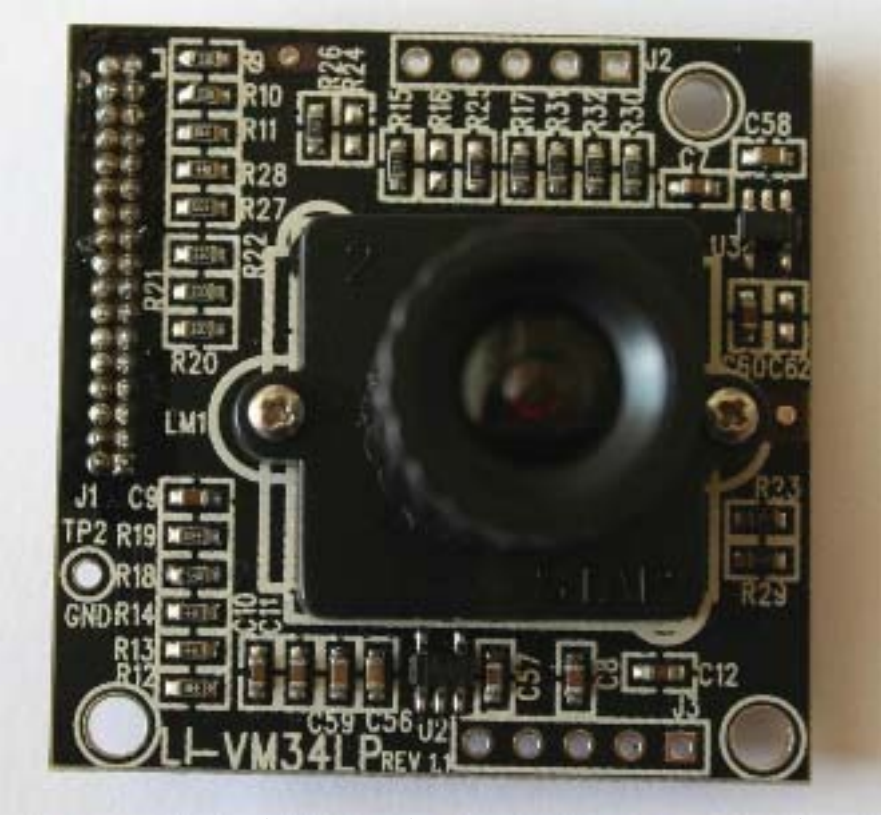


Figure 3.3: The Leopard Imaging LI-VM34LP Board.

3.3 Digital to Analog Converter

The ADV7125 is a triple high speed video digital-to-analog converter on a single monolithic chip. It is designed for applications such as image processing and digital

video systems. It has three 8-bit video DACs with complementary outputs, a high impedance, analog output current source and a standard TTL interface. Additional video control signals, composite SYNC and BLANK signals, are available and there is a power save mode.

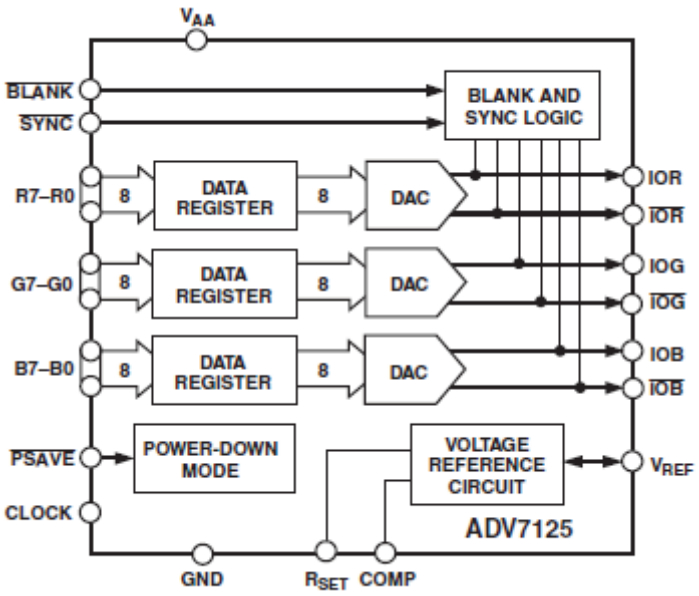


Figure 3.4: ADV7125 Functional Block Diagram. [17]

Chapter 4

VGA Module

This chapter gives overview of the VGA standard, VGA hardware, and HDL modules are described.

4.1 VGA Standard

Video Graphics Array (VGA) is a video display analog standard that is mostly used with computer monitors. VGA uses separate wires for the five following signals:

- Vertical Sync (VSYNC): a synchronization signal used to start a new frame
- Horizontal Sync (HSYNC): a synchronization signal used to start a new line
- Red, Green, and Blue: 0.7V peak-to-peak analog level that controls the color. 0.7V is full intensity and 0V is black.

Each frame is made up of a number of horizontal lines and each line is made up of pixels. VGA is not interlaced so each line is transmitted in order from top to bottom and the pixels from left to right. The VSYNC signal determines the frame rate and the HSYNC and VSYNC determine the screen resolution by defining where each line and frame starts and ends. Lines begin with an active video region where RGB values for the pixels are output followed by a blanking region where black pixels are output.

During the blanking region, a horizontal sync pulse is output. The blanking interval before the sync is referred to as the front-porch and after the sync interval is the back-porch. A line consists of pixels controlled by the horizontal sync signal where as the frame is made up of lines controlled by the vertical sync. The vertical sync signal is similar to that of the horizontal sync with blanking regions consisting of lines [6]. Figure 4.1 shows a sample timing of sync signals with labeled regions.

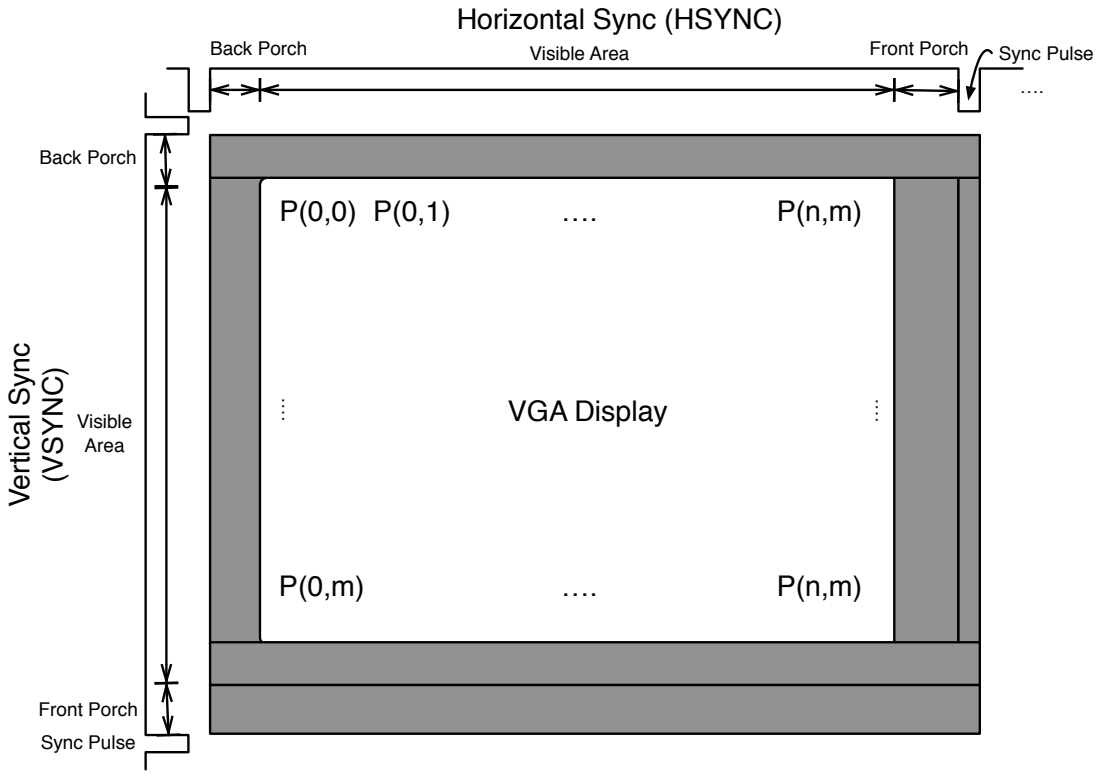


Figure 4.1: VGA Display Area and Blanking Regions

4.2 PCB VGA Board

The ADV7125 digital-to-analog converter chip is used to convert the digital signal from the FPGA to an analog signal for the color signals for video. A PCB board was

designed and built for the chip. The board includes multiple decoupling capacitors to reduce noise pickup and resistors for doubly terminated 75 Ω connections To operate at VGA voltage levels, a 5V supply, clock input, and the internal 1.23V power supply are used. The clock input rate is such that refresh rate would be 60 Hz. It has 0.1" (2.54mm) headers for connection with many prototyping development boards and holes for a standard VGA connector. A plug-in gray-scale board was also designed and built so that if required, only 8 data pins are needed instead of 24 data pins for the RGB colors.

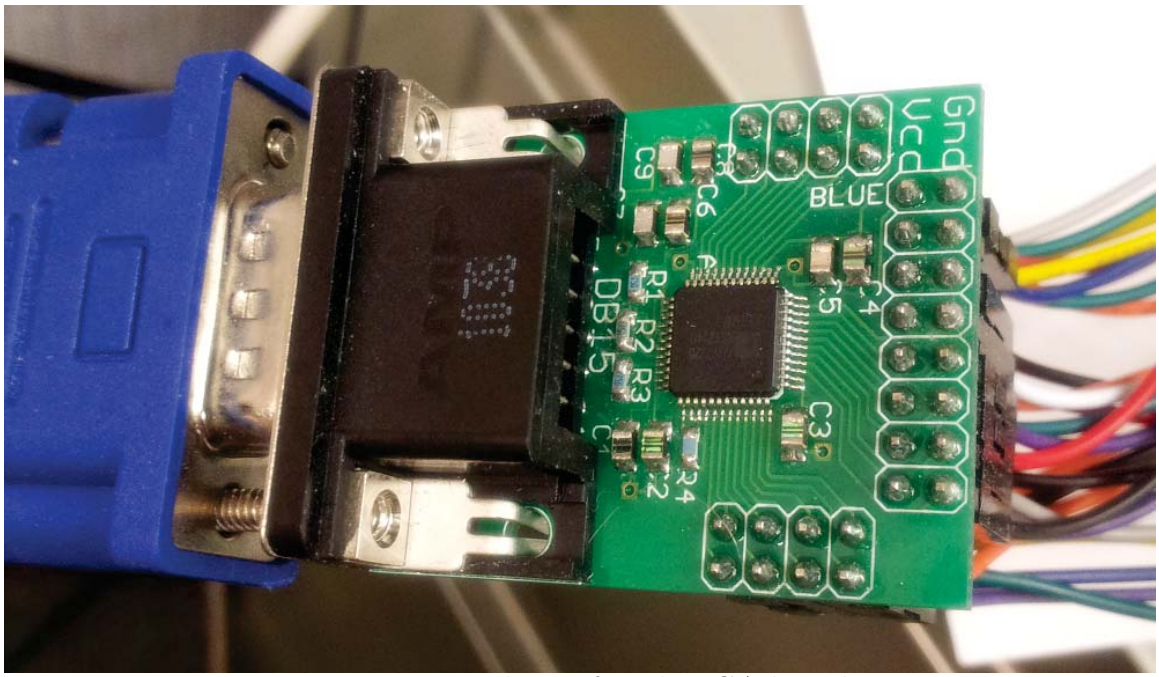


Figure 4.2: Photo of Built VGA board

4.3 VGA HDL Implementation

The VGA module's tasks are to supply the DAC chip with digital color data and output horizontal sync, vertical sync, and blank signals. The module has a memory with discrete value sine wave data that is accessed and outputs the pixel data for the

sine-wave patterns to be projected. HDL code was written in VHDL to interface with the VGA board and output phase-shifted patterns to be projected onto a scene.

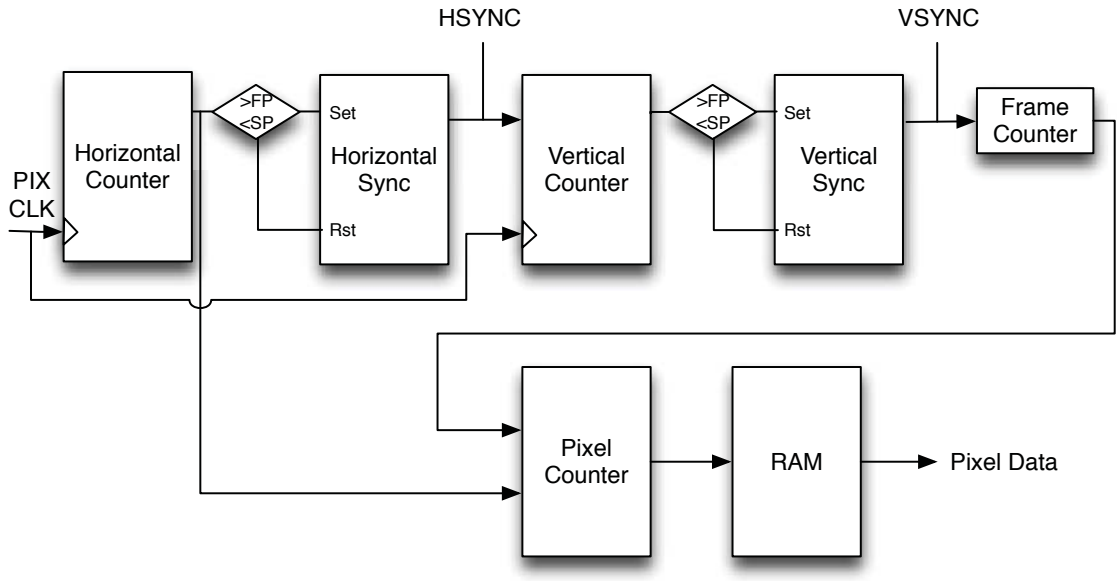


Figure 4.3: VGA HDL Module Data Flow Diagram

4.3.1 Sync Signals

The horizontal and vertical sync signals output value is based on the value of synchronous counters. When the counter value is between where the front porch ends and the sync pulse ends, the polarity of the signal is inverted. Table 4.1 gives timing information for 800x600 resolution running at 60Hz. The horizontal counter increments one for every pixel on the rising edge of the clock and will count to 840 pixels at which the sync pulse will start (inverting the polarity) and lasts 128 pixels until the back porch region. This is shown in Figure 4.3 as a comparison of the horizontal counter with the front porch time (greater than 800+40=840 from the Timing Information table) and the sync pulse time (less than 800+40+128=968). If the counter is in this range, then the horizontal sync is logic high, else it is logic low. The horizon-

tal counter counts the number of pixels (1 per clock cycle) and the vertical counter counts the number of horizontal lines (1 per 1056 pixels).

Table 4.1: Timing Data for 800x600 resolution running at 60Hz

Pixel Clk = 40MHz	Vertical Timing		Horizontal Timing	
Resolution (lines/pixels)(uS)	600	15840	800	20
Front Porch (lines/pixels)(uS)	1	26.4	40	1
Sync Pulse (lines/pixels)(uS)	4	105.6	128	3.2
Back Porch (lines/pixels)(uS)	23	607.2	88	2.2
Whole Period (lines/pixels)(us)	628	16579	1056	26.4

4.3.2 Camera Signals

In order to capture the same order of patterns each time, a signal is sent when the frame number is equal to a specified value. This tells the camera controller that it can start capturing frames. Also, when there is a new frame projected (as evidenced by a vertical sync pulse), a trigger signal is sent. If a camera sensor is in snapshot mode, this will trigger the camera to capture a frame. This signal was delayed for an experimentally determined number of clock cycles to avoid a frame change during the exposure period.

4.3.3 Pixel Output

To output the dual-frequency patterns that were introduced in Chapter 1, a dual-port Block RAM was instantiated with the contents initialized to unit frequency sine wave data. The pixels of a horizontal line have the same value and for each new line, the RAM address is increased to display a new pixel value. One port is used to output a unit frequency sine wave and the other is for the higher frequency sine wave. These

pixel values are added together giving a pixel value for the dual-frequency pattern. To get a higher frequency sine wave using a unit-frequency sine wave, each new horizontal line increments the pixel pointer by the frequency value where as the unit-frequency is simply incremented by one. For each new frame, the pointers are set to an offset for the phase-shift. When an address pointer goes out of bounds by becoming too large, it is simply wrapped around to the beginning of the periodic sine wave. Figure 4.4 shows an example of dual-frequency patterns generated using Eq. 1.7.

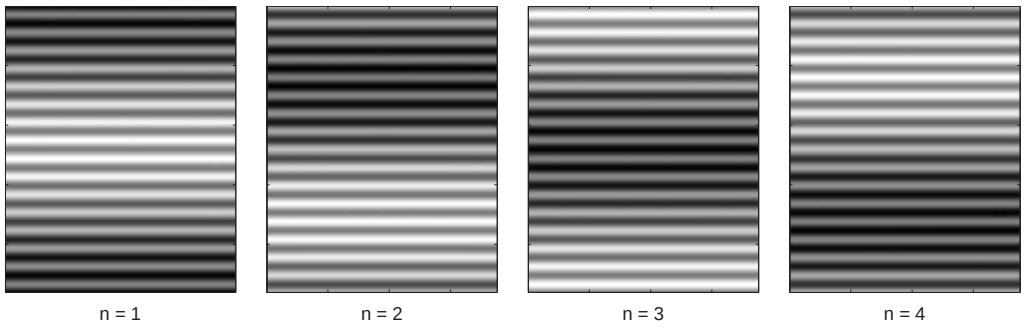


Figure 4.4: Dual-Frequency Patterns for N=4 with $f_h = 16$

Chapter 5

Image Acquisition

In this chapter, the implementation of HDL modules to interface with the imaging sensor and data read from it are discussed. Frames can alternatively be sent to the FPGA from a PC for testing or alternative applications.

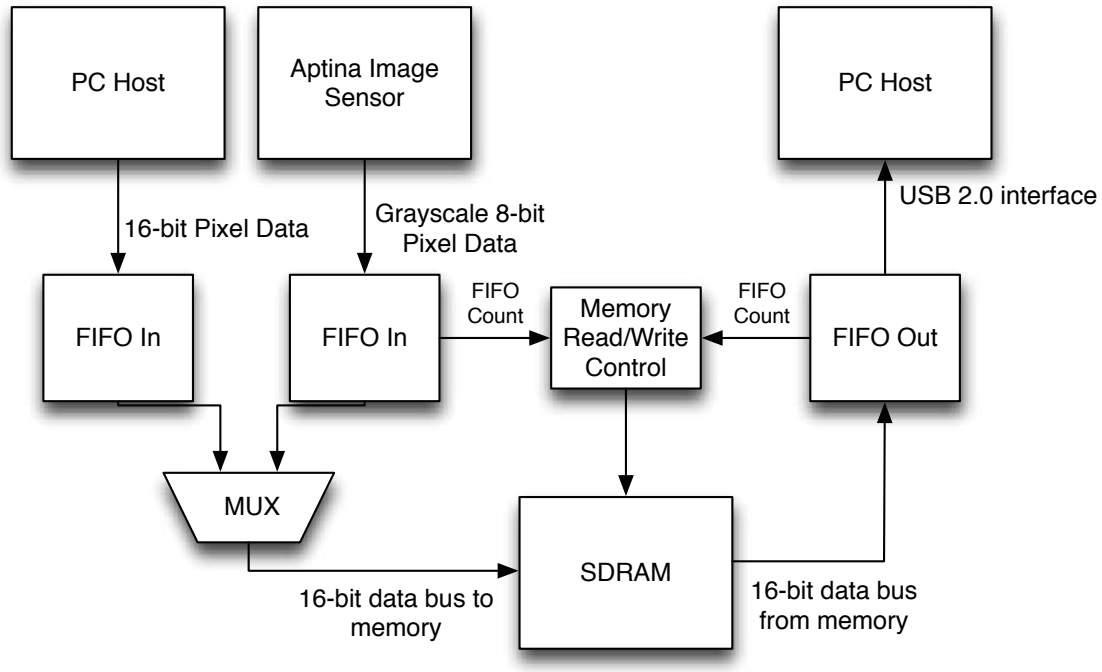


Figure 5.1: Image Aquisition Data Flow Diagram

5.1 System Overview

The tasks of this system are to configure the camera sensor's registers, read pixel data and store it into RAM and lastly output it to a PC where it can be read. Figure 5.1

shows a data flow diagram of the system. The camera employed is an Micron Aptina MT9V034 752 by 480 CMOS array that runs at a typical 27MHz system frequency and has up to a 60Hz frame rate. The operation of the sensor is controlled by writing to registers using a two-wire interface.

5.2 Imaging Sensor

5.2.1 Camera Shuttering

Digital cameras capture images by by converting light that hits a photosensitive sensor into an electrical signal. The amplitude of that signal depends on the intensity and duration of light. To control the duration of light, a shutter is used. This can be mechanically by blocking light to the sensor, using an on-sensor electrical shutter or both. A rolling shutter in a digital camera does not capture a whole image at one instant. Rather, different areas of an array are exposed to light at different times. The process is as follows: a row of an array resets the current values in the array, the exposure period starts and lastly reads the data out. This time is called the integration period and allows the simultaneous readout of pixels and exposure. An issue with this type of shutter is it can introduce distortions motion artifacts when taking pictures of moving objects due to different exposure times of rows. This is not acceptable for real-time applications. Also, intensity flicker from the projector and light sources could result in unexpected pixel values for different areas depending on when the exposure was. To prevent this, a global, or snapshot, shutter is ideal. A global shutter exposes all the pixels at one instance and read out happens after

exposure [23].

The Micron image sensor has a TrueSnap^{TM} global shutter where the signal charge is transferred into an analog memory where it is isolated from any photoelectric signals effectively ending the exposure [16].

5.2.2 Modes of Operation

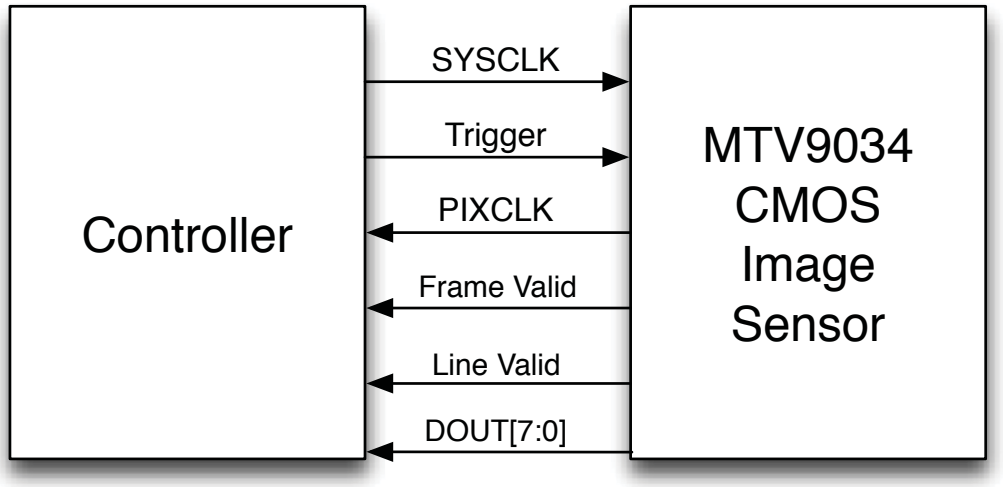


Figure 5.2: Interface Signals Between Controller and Image sensor

The Aptina MTV9034 comes with three different modes of operation: master, snapshot, and slave. In master mode, the readout timing is generated by the chip. Readout can happen during or after the integration time. Readout of the prior frame during the integration time is the fastest allowing a 60 Hz frame rate. Snapshot mode is a mode where an input trigger signal initiates the capture of a frame and the readout happens after. This mode was determined to have a maximum frame rate of 30 Hz. Figure 5.2 shows the signals required for the snapshot mode. Lastly, in slave mode the exposure and readout timing are provided by external signals.

5.3 Pixel Data

The MT9V034 ADC has a resolution of 10-bits per pixel that are output in parallel every pixel clock period. Pixel data is raw and grayscale so no Bayer filter processing is required. The pixel clock is an inverted version of the master clock of the system which is used as the clock to store the data into a FIFO. Frame Valid and Line Valid signals are asserted when the pixel data output is valid. Figure 5.3 shows an example of how the timing of the signals. When pixels from a frame are about to be output the Frame Valid signal is asserted.

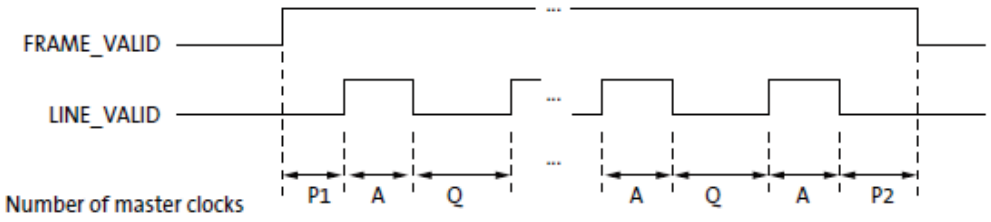


Figure 5.3: Frame Valid and Line Valid Signals [16]

When Line Valid is asserted, pixel values are being read out and stored into an input FIFO on the rising edge of the pixel clock. Although the MT9V034 has a resolution of 10-bits per pixel, 8-bit pixel data was decided as enough for now so the two least significant bits of the 10-bit pixel data are truncated.

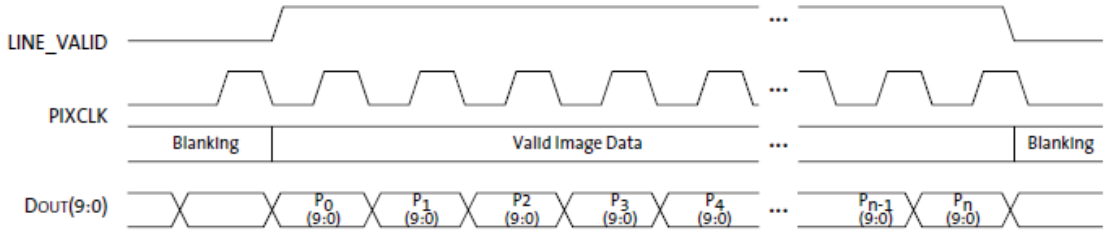


Figure 5.4: Timing Example of Pixel Data [16]

5.3.1 Two-Wire Interface

To change the operation of the chip, registers are written and read through a two-wire serial interface bus. One wire is the clock and the other is the data line. In this project, registers for operation mode, gain settings and exposure timing were set. The protocol to write to the slave device, the CMOS image sensor, from the master device, the FPGA, is as follows:

1. A start bit is sent by the master by pulling the data line low while the clock is high.

2. 7 bits of the slave address and 1 bit of direction (read or write) are sent.

3. The slave acknowledges the address by sending back an acknowledgement bit.

4. The master then sends 16 bits of data, 8 bits at a time followed by an acknowledgment from the slave each time.

5. The register address is incremented so the next 16 bits will be written to the next register or a stop bit is sent by making the data line high while the clock is high.

Open source code to implement a two-wire protocol for the CMOS sensor was found and modified for the the specified CMOS sensor which has 8-bit addresses and 16-bit data with an ack bit between the data. [7]. A ROM (Read-only Memory) is initialized with the register addresses and values to be written at startup. Each memory location is read and sent to the I2C state-machine to write the register.

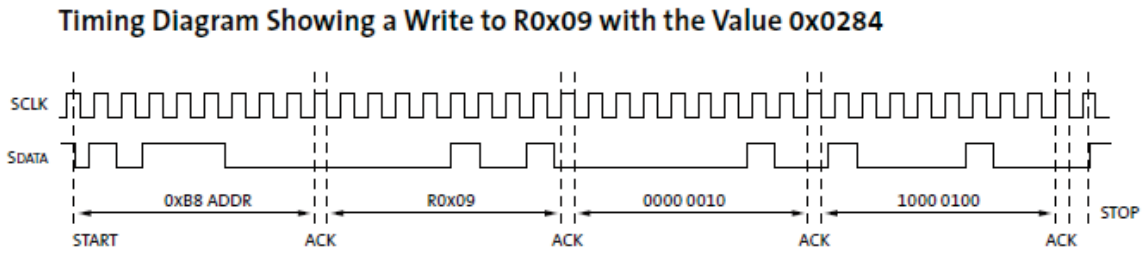


Figure 5.5: Example 16-bit Write Sequence for Aptina MT9V034.

When a final acknowledgement is received, the next memory location is read until all the data in the ROM has been written to the camera registers

5.3.2 Alternative Cameras

For demonstration purposes the above sensor was used chosen for the system. It should be noted that other CMOS VGA sensors can employed with minimal changes to the code. One possible alternative is the Cypress LUPA-300 with a 600x400 pixel resolution, electronic global shutter, and maximum of 250 frames-per-second at full resolution.

Off-the-shelf high-speed cameras with accessible FPGAs are also available for purchase. One such company that makes such cameras is OptoMotive, mechatronics Ltd. which offers high-speed cameras with a FPGA catering to real-time high-speed image processing applications. The OptoMotive Cameleon camera uses the same Micron Aptina sensor with a Xilinx Spartan-3E FPGA and 64MB of DDR SDRAM. It has a USB 2.0 connection with general purpose I/O pins that could be used to drive a a VGA port. The company offers a range of camera packages with higher resolution sensors and more capable FPGAs with reference designs and example PC software.

Figure 5.6: OptoMotive Cameleon BaseBoard [19]

5.4 Input Image FIFOs

A FIFO (First-In First-Out) memory is similar to a distributed or block RAM except that there is no input for the address. The read and write addresses are kept track of internally. When a write is performed, the data is stored in the memory and the write counter automatically increments. When a read is performed, the data stored in the memory is put on the output port and the read address is incremented. In this way, as the name implies, the first data that is written to the FIFO is the first data to be read out. Depending on the read and write counter values, it can be determined if the FIFO is filled or empty.

The FIFO acts as a cache to the SDRAM. When it has enough data (512x16 bits) for a SDRAM page, a memory write is activated. The SDRAM runs faster than both PC and camera writes so there is no worry about overrun as long as the FIFO has a depth greater than 512. Because of this, a 16-bit word FIFO with a depth of 1024 is used. The depth of the FIFO is equal to the number of elements that can be written

to. A depth of 1024 means that the address is 10 bits (2^{10}).

Two input FIFOs are instantiated for both the PC input and the camera input. The FIFOs are dual port allowing for simultaneous reads and writes. Reads and writes depend on different clock domains. The camera outputs 8-bit pixels so the FIFO has a read width of 8-bits and the memory has a read width of 16-bit words. The PC has a read width of 16-bits and a write width of 16-bits. Writes to the FIFO happen on the positive edge of the separate clocks from the camera and the PC. Reads happen on the positive edge of the system clock which is running at 100MHz.

5.5 SDRAM Transfer Controller

This module controls the transfers between the FIFOs and the DRAM. The FIFOs act as a cache to the SDRAM that holds at least a full page, 512 words, of memory while the camera writes to or the PC reads from the FIFOs. Rather than write a byte at time, which is slow and inefficient, reads and writes are burst-oriented meaning a whole row of data is read or written at one time. For example, the SDRAM chip used has its banks organized as 8192 rows by 512 columns by 16 bits. Therefore, each memory access touches 512 by 16 bits. The Transfer Controller checks the input and output FIFO statuses (how much data is in the FIFO). If it finds that an input FIFO has at least a page of memory, a memory write is initiated and data is read from the FIFO. If a read is requested and an output FIFO has at least enough space for a page, then a memory read is initiated. The Transfer Controller also increments the row address after each write and read and accepts inputs to set the row address location. The imaging sensor and PC transfers are slower than the DRAM so there

is no fear of over or underrun of the FIFO.

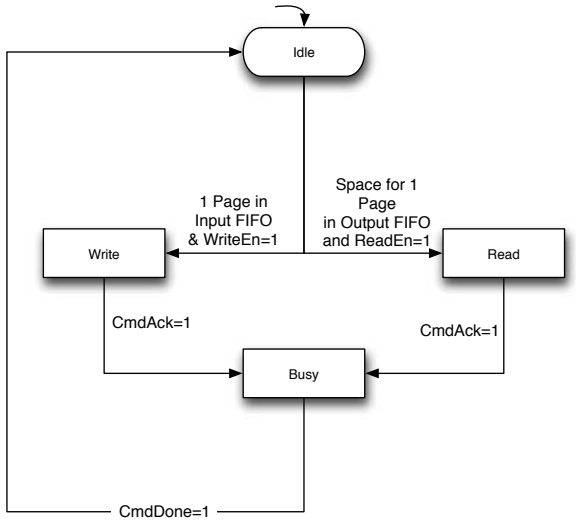


Figure 5.7: Memory Controller Flow Diagram

5.6 Image Aquisition Controller

Figure 5.8 shows a flow diagram of the process to acquire frames from the camera and store them into a FIFO. Each of the rectangular boxes correspond to stages in the controller state machine. The SDRAM transfer controller is working in parallel when a FIFO is filled with a page of data, a page is written to the SDRAM. After each frame, the FIFO is flushed to SDRAM so that the results are block aligned. This continues until all N frames have been captured and stored in to SDRAM.

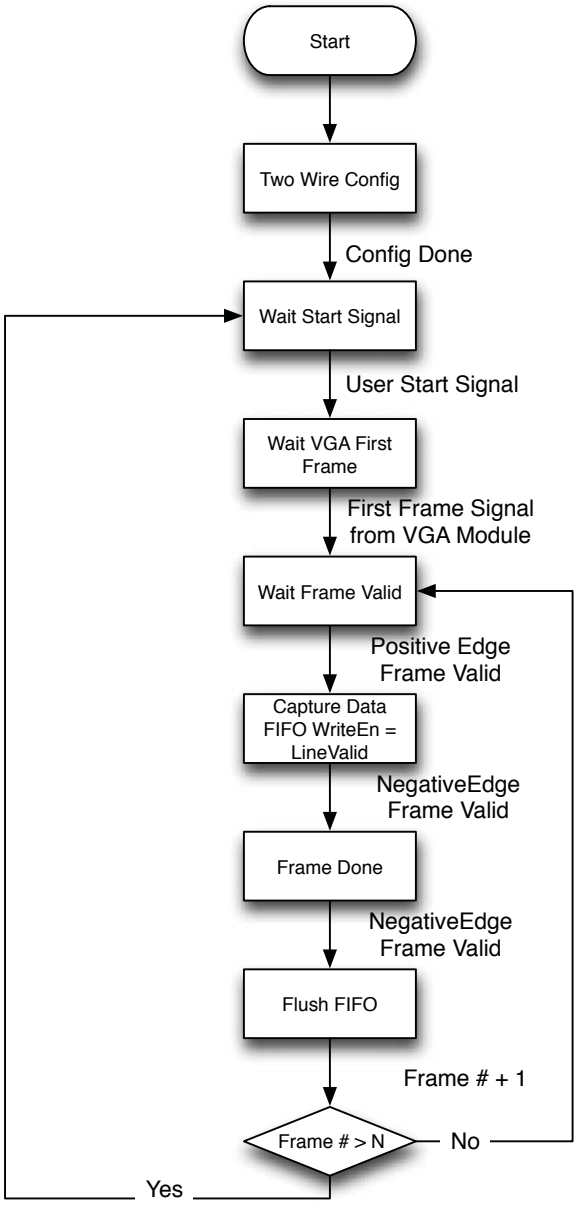


Figure 5.8: Image Acquisition Flow Diagram

Chapter 6

Phase Calculation Pipeline Design

6.1 Introduction

In previous chapters, the design of the architecture to write frames from the imaging sensor to the on-board SDRAM was discussed. With the frames stored in memory, they now need to be processed and the phase calculated. This section centers around Eq. 1.9 which calculates the phase-pair (ϕ_u, ϕ_h). It is rewritten here for clarity:

$$(\phi_u, \phi_h) = \left(\tan^{-1} \left[\frac{\sum_{n=0}^{N-1} I_n^c \sin\left(\frac{2\pi n}{N}\right)}{\sum_{n=0}^{N-1} I_n^c \cos\left(\frac{2\pi n}{N}\right)} \right], \tan^{-1} \left[\frac{\sum_{n=0}^{N-1} I_n^c \sin\left(\frac{4\pi n}{N}\right)}{\sum_{n=0}^{N-1} I_n^c \cos\left(\frac{4\pi n}{N}\right)} \right] \right)$$

where ϕ_h represents the wrapped phase value of the captured pattern and ϕ_u represents the base phase used to unwrap ϕ_h.

What is apparent is that the inside of the tan^{-1} function, is the Discrete Fourier Transform (DFT), second, K=1, and third, K=2, coefficients. Lastly, the inverse tangent of the imaginary over the real result is taken, giving the phase value. In this chapter, the overall architecture of the phase calculation operations are discussed. In the next chapters, the individual modules that perform the DFT and Inverse Tangent are described in detail.

6.2 Architecture for N=4 Patterns

In chapter 1, Equation 6.1 was introduced as the single-frequency PMP phase and rewritten below.

$$\phi = \tan^{-1}\left[\frac{\sum_{n=0}^{N-1} I_n^c \sin\left(\frac{2\pi n}{N}\right)}{\sum_{n=0}^{N-1} I_n^c \cos\left(\frac{2\pi n}{N}\right)}\right] \tag{6.1}$$

If we let N=4, this equation can be simplified such that

$$\phi = \tan^{-1}\left[\frac{I_1^c - I_3^c}{I_0^c + I_2^c}\right] = \tan^{-1}\left[\frac{V}{U}\right] \tag{6.2}$$

Now the argument to the inverse tangent is simply made up of a subtraction and addition operator. Because of its simplicity, this case was first designed. Let the result of the operation in the numerator be V and the denominator be U. When reading from the RAM, as long as we keep track of the frame number (determined by the location in memory) then the data read out can be multiplexed to an Arithmetic Logic Unit (ALU) to perform the addition, the U ALU, or to perform a subtraction operation, the ALU V, and then stored in a respective memory.

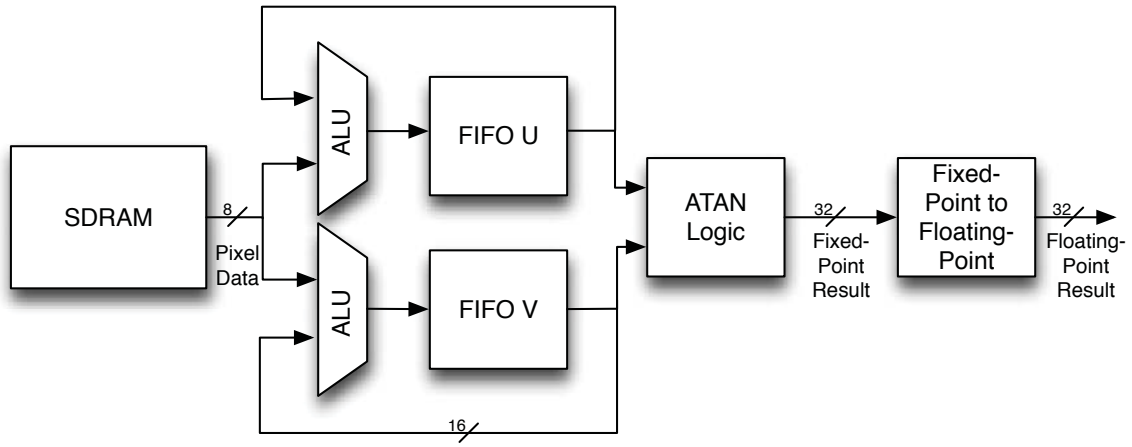


Figure 6.1: Phase Calculation Dataflow Diagram for N=4 Patterns

Remember that SDRAM reads only happen in pages so it is only possible to read data from one frame at a time and the FPGA does not have enough resources

to form a distributed or block RAM in the FPGA to hold a whole image therefore calculations are performed one page at a time. A FIFO interface was designed for Synchronous Write, Asynchronous Read Distributed RAM. The asynchronous read ability allows feedback of the data in memory to the ALU. The initial block of data is written directly to the FIFO and then that data is fed back into the ALU for the next frame data. The designed FIFO also has a address reset so that when the first block of data is written to the FIFO, the read and write address pointers are reset to the beginning of the FIFO. This allows the first element of data to be fed back into the ALU and therefore, the first element is overwritten with the original data $\pm$ the data from the first block of the next frame. The following list explains the process:

1. Read the first block of data from frame 1, I_0^c.
2. Because it is the first block, it is passed through the ALU by adding 0 to it.
3. Store the result in the U FIFO.
4. Read the first block of data from frame 2, I_1^c and repeat Step 2
5. Store the result in the V FIFO.
6. Reset the FIFO addresses so the read and write address go back to zero
7. Read the first block of data from frame 3, I_2^c
8. The ALU adds the I_0^c data that is present in the FIFO to I_2^c and overwrites the FIFO memory with the result.
9. Do that above for the V FIFO but subtract.
10. The first block is done and the data transferred to the next module to perform the inverse tangent operation.
11. Continue the above steps for the next block of memory.

The SDRAM read data width is 16-bits and pixel values are 8-bits so the ALU hardware is duplicated to allow two pixel operations to be performed in parallel.

Although not discussed, the K=2 coefficient for N=4 patterns can be calculated similarly with addition and subtraction.

This design works and has been implemented and tested. It is implemented using simple single-cycle operations like addition, subtraction or shifting and is fast because there is no FIFO on the direct output of the SDRAM. The pixels are processed as they are output from the RAM. Multiple clock cycle operations like multiplication and division can be accomplished provided they are pipelined.

There are, however, some limitations with this architecture. This architecture works for straight line computations with a specific order of operands and operations but is limited for more advance computation. FPGA on-chip memory is saved by only using two FIFOs but because asynchronous reads are necessary, distributed RAM is required. Distributed RAM takes up FPGA resources like look-up tables which also implement logic. In comparison, Block RAM is faster and is built in to the chip so even if it is used, it does not affect the utilization of logic resources like CLBs.

6.3 Architecture for N patterns

As discussed in the last section, the design for N=4 has a number of limitations. Because Block RAM is dedicated and does not affect utilization of CLBs, it is okay to make use of the FPGA on-chip memory as long as BlockRAM capacity is not exceeded. According to the Xilinx data sheet, the FPGA chip used has a BlockRAM storage capacity of 576·1024 bits, or 72 KiloBytes. A block of SDRAM memory takes up 512·16bits = 1KiloByte so the FPGA has enough to store up to 72 blocks using on-chip memory. If we increased the number of patterns to N=8 with a block of memory

for each frame, then there is still ample space left for input and output FIFOs, VGA ROM, and other memory structures used in the design. This architecture has an acceptable latency delay that the previous architecture does not due to the FFT module.

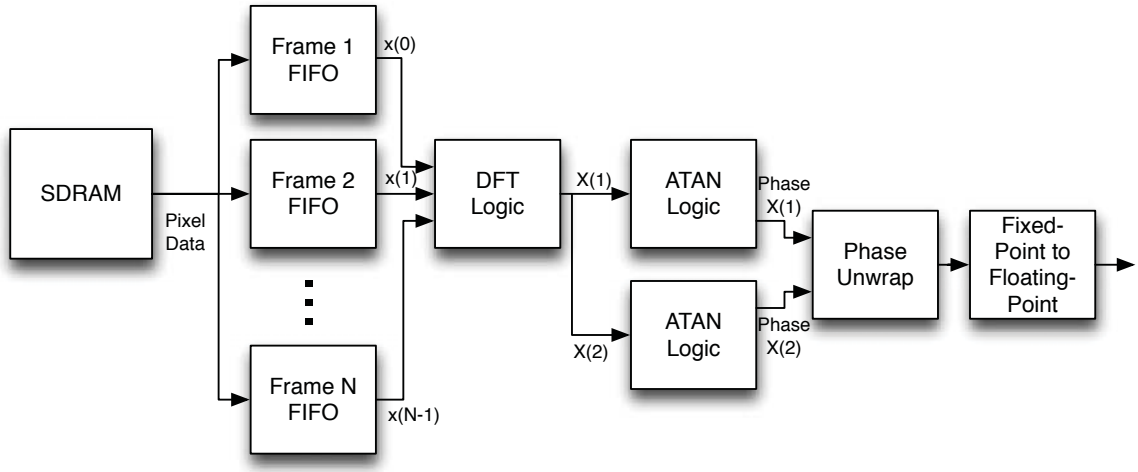


Figure 6.2: Phase Calculation Dataflow Diagram for N Patterns

Figure 6.2 shows the architecture implemented to perform the phase calculation. It shows that each frame has a FIFO that can hold a page of SDRAM memory. The process of reading pages from the SDRAM is similar to the architecture for N=4 patterns where data is processed page by page. So the same page number of each frame is read to its corresponding FIFO. Once all the FIFOs are filled, then the pixels are sent in parallel to the FFT to implement the N-point DFT to get the two required coefficients.

The DFT Logic, Inverse Tangent Logic, Phase Unwrapping and Fixed-Point to Floating Point modules should be viewed as one pipeline. Each has a input dv (data valid) and rdy (output data read) signal that feeds into the next. Once data is read from the FIFO and input to the DFT logic, the pipeline runs without any external

control logic or state machine.

6.4 State Diagram

Figure 6.3 shows a flow diagram of the process to take the frames from memory and send the data to the phase calculation logic. Each rectangular block represents a state in the state machine. An important thing to keep track of the memory address. The controller controls reading the first page of each frame, the second page of each frame, and so on. An offset address of 353 is needed to jump from one frame to the next. In addition to the offset, the page number needs to be kept. These two numbers are added making up the memory address.

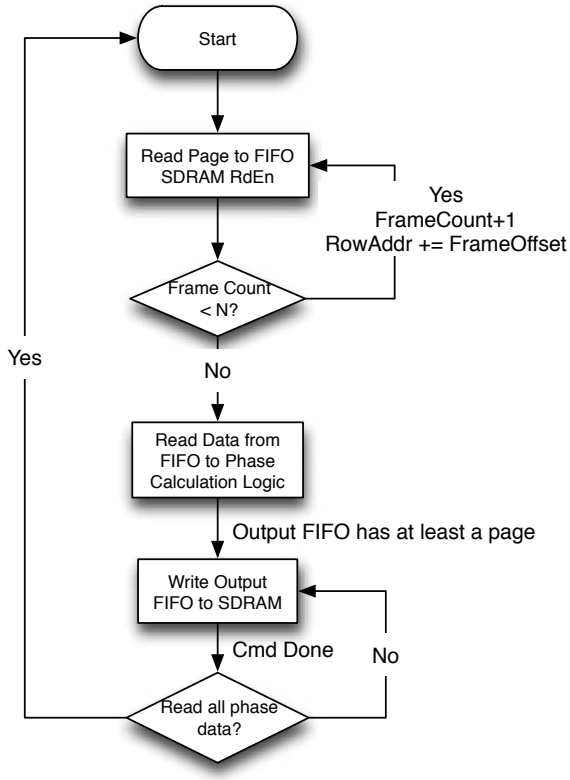


Figure 6.3: Phase Data Flow Controller

Chapter 7

Digital Fourier Transform Module Design

In this chapter, the techniques used to calculate the Discrete Fourier Transform module of the Phase Calculation pipeline is described. The FFT algorithm is used to retrieve the coefficients which are passed to the inverse tangent function to retrieve the two phases. Lastly, the unit phase is used to unwrap the high frequency phase.

7.1 Fast Fourier Transform Theory

The Fourier Transform is a mathematical transform which allows decomposition of a signal in its sinusoidal components. It is commonly used in signal processing by converting the input is a time, or spatial, domain signal into the frequency domain. A signal can be real-world analog signal that is converted to the digital domain and its frequency components can be calculated using the Discrete Fourier Transform (DFT). The DFT of a discrete-time signal x[n] is given by:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{j2\pi kn}{N}} \text{ where } k = 0, 1, 2, ..., N-1 \text{ and } x(nT) = x[n]. \tag{7.1}$$

Letting $W_n = e^{\frac{-j2\pi}{N}}$, the equation can be written as

$$X(k) = \sum_{n=0}^{N-1} x[n] W_N^{nk}. \tag{7.2}$$

The W term is called the twiddle factors.

This equation can be written out as

$$X(0) = x[0]W_n^0 + x[1]W_N^{0*1} + ... + x[N-1]W_n^{0*(N-1)}$$

$$X(1) = x[0]W_n^0 + x[1]W_N^{1*1} + ... + x[N-1]W_n^{1*(N-1)}$$

$$X(k) = x[0]W_n^0 + x[1]W_N^{k*1} + ... + x[N-1]W_n^{k*(N-1)}$$

To solve the equation requires N^2 multiplications and $(N-1)*N$ additions resulting in $\mathcal{O}(N^2)$ arithmetical operations. This algorithmic complexity is very inefficient and not useful for practical applications.

To improve this a Fast Fourier Transform (FFT) algorithm can be used to reduce the complexity to $\mathcal{O}(N\log\ N)$. The most common FFT algorithm is the Cooley-Tukey algorithm which works by recursively decomposing a set of data into smaller sets of data and taking the DFT until a single 2-point DFT is produced (for a radix-2 FFT) [24]. A radix-2 FFT decomposes a power of 2 element set of data in to two interleaved transforms of size N/2 with each recursive stage.

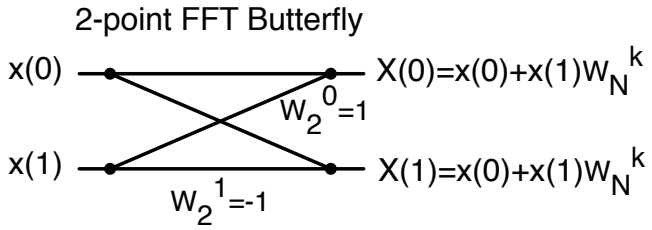


Figure 7.1: 2-point FFT

This is done by dividing it into even(2n) and odd(2n+1) sequences and rewriting Eq. 7.3 as:

$$X[k] = \sum_{n=0}^{N/2-1} x[2n]W_N^{2nk} + \sum_{n=0}^{N/2-1} x[2n+1]W_N^{(2n+1)k} \tag{7.3}$$

With the identity,

$$W_N^{2nk} = e^{\frac{-j2\pi}{N}2nk} = e^{\frac{-j2\pi}{N/2}nk} = W_{N/2}^{nk} \tag{7.4}$$

then the equation can be rewritten as:

$$X[k] = \sum_{n=0}^{N/2-1} x[2n]W_{N/2}^{nk} + \sum_{n=0}^{N/2-1} x[2n+1]W_{N/2}^{nk} \tag{7.5}$$

where the two sums are N/2 point DFTs of the even and odd sets. Continuing the process, the equations can be divided into N/4 point DFTs until only 2 point DFTs are left [22].

A two-point DFT can be shown graphically using a butterfly diagram shown in Figure 7.1 which is the most simple unit of a radix-2 FFT with two inputs and two outputs. Each path goes only from left to right and the values along the path are multiplied by the input, and at the nodes both lines are added together. To form the individual components of a four-point DFT, two 2-point DFTs are summed together.

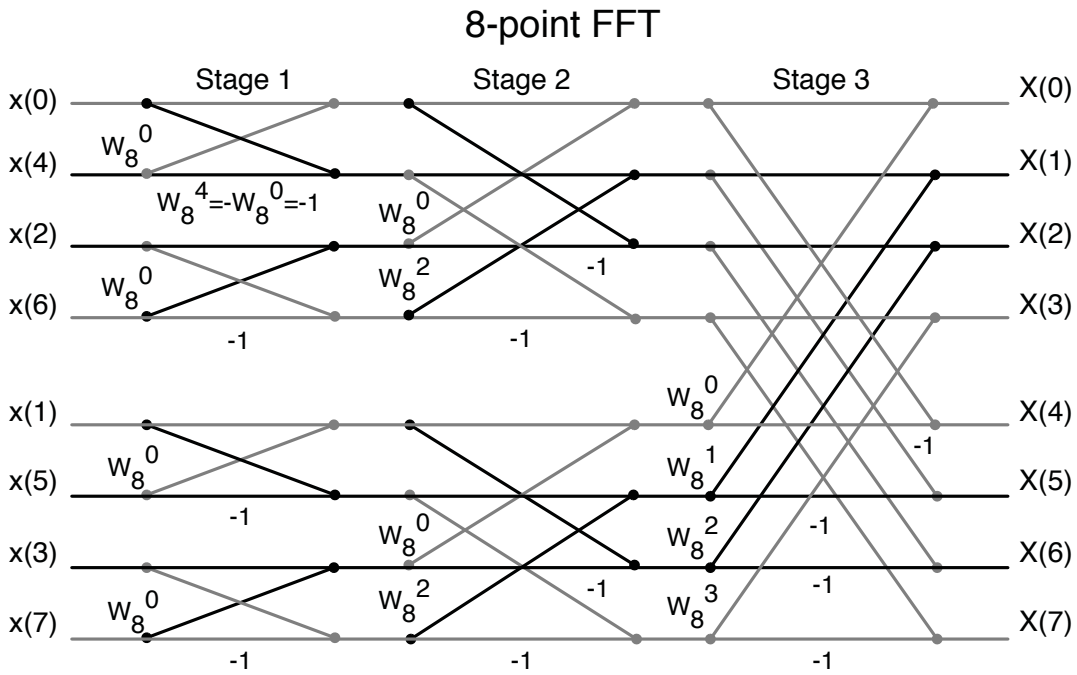


Figure 7.2: 8-point Radix-2 FFT

As shown in Eq. 1.3, the second and third coefficients are required to calculate the unit and higher frequency phase. As a result, the other coefficients can be ignored

and the FFT calculations simplified. Figure 7.2 shows the paths required to compute the second and third coefficient in black for N=8 patterns and the gray paths can be ignored. Tracing through the stages and calculating the FFT gives the following equations:

$$\begin{aligned} X(1) =&\big(x(0) - x(4)\big) + \big(x(2) - x(6)\big)W_8^2 + \\ &\{\big(x(1) - x(5)\big) + \big(x(7) - x(3)\big)\}W_8^1 \end{aligned} \tag{7.6}$$

$$X(2) = \big(x(0) + x(4)\big) - \big(x(2) + x(6) + \{\big(x(3) + x(7)\big) - \big(x(1) - x(5)\big)\}W_8^2 \tag{7.7}$$

7.2 FFT HDL Implementation

7.3 Introduction

The Cooley-Tukey FFT is a very common algorithm used in digital signal processing. As a result, it is not hard to find configurable cores. Xilinx offers an IP Core with its CORE Generator software to implement the FFT with transform sizes from 2^m where $m = 3 - 16$. The Xilinx core architecture receives a START signal and begins loading the serially. The results are also output serially [13]. This is reasonable for an application that takes in a signal at discrete time intervals but is inefficient for this application where all the data is available from the beginning. Furthermore, with only two coefficients needed, some of the circuitry is used. To make up for these facts, an FFT module was designed with parallel 8-point input and parallel output of real and imaginary values for the second and third coefficient.

7.4 Single Cycle Design

The equations to calculate the coefficients, Eq. 7.6 and 7.7, can be simplified by the following steps. Substituting the values of the the twiddle factors

$$W_8^0 = W_8^8 = cos(0^\circ) - jsin(0^\circ) = 1 - j0$$

$$W_8^1 = W_8^9 = cos(45^\circ) - jsin(45^\circ) = 0.707 - j0.707$$

$$W_8^2 = W_8^1 0 = cos(90^\circ) - jsin(90^\circ) = 0 - j1$$

gives

$$X(1) = \big(x(0) - x(4)\big) + \big(x(2) - x(6)\big)(-j) + \big(x(1) - x(5)\big) + \big(x(7) - x(3)\big)(0.7 - 0.7j)$$

$$X(2) = \big(x(0) + x(4) - (x(2) + x(6)\big) + \big((x(3) + x(7)) - ((x(1) - x(5)\big)(-j)$$

which can be broken into its real and imaginary parts:

$$X_r(1) = x(0) - x(4) + \big(x(1) - x(5) + (x(7) - x(3)\big) \cdot 0.707$$

$$X_i(1) = x(6) - x(2) - \big(x(1) - x(5) - (x(7) - x(3)\big) \cdot 0.707$$

$$X_r(2) = x(0) + x(4) - (x(2) + x(6))$$

$$X_i(2) = x(3) + x(7) - (x(1) + x(5))$$

As can be seen, the results of some additions and subtractions can be reused.

The FFT module takes in eight 8-bit inputs and has four 16-bit outputs for the $X_r(1)$, $X_i(1)$, $X_r(2)$ and $X_i(2)$ terms. When a design is synthesized, the Xilinx ISE Project Tools give timing reports. These allow a designer to analyze the latency of the slowest path in the design.

```
Delay:               25.750ns (Levels of Logic = 26)
  Source:            in1<0> (PAD)
  Destination:       out1i<15> (PAD)
```

The above snippet from the timing report shows the path of the module which the greatest latency which is 25.750ns. The system frequency can be set appropriately to meet this timing but will lower the performance of the system. A goal of the system is to keep the system clock frequency at 100MHz, a 10ns period, so in order to meet this timing, the module was pipelined.

7.5 Pipelined Design

Pipelining breaks the module down into smaller tasks where the output of one task is the input of the next one. In this way, tasks can be executed in parallel. Also, when a pipeline is filled and kept that way by a stream of data, valid data is available every clock cycle leading to higher throughout of the system. For example, the timing report shows that the operation with the most delay is the multiplication so this is the focus of the pipelining. Figure 7.3 show a data flow diagram of the design 8-point FFT. Pipeline registers are placed on the input and output, after the first addition and after the multiplication. The module takes 8-bit inputs as each pixel is 8-bits. The initial addition and subtraction gives a 9-bit result. This prevents losing any data due to overflow when adding. Because the pixel values have a range of 0 to 255, the subtraction operation will not use the most significant bit but it is kept so that one module can be used for both addition and subtraction. The second stage adds and subtract the result of the first stage to give the value that will be multiplied by the twiddle factor. The twiddle factor equal to 0.707 is converted to a 16 bit number

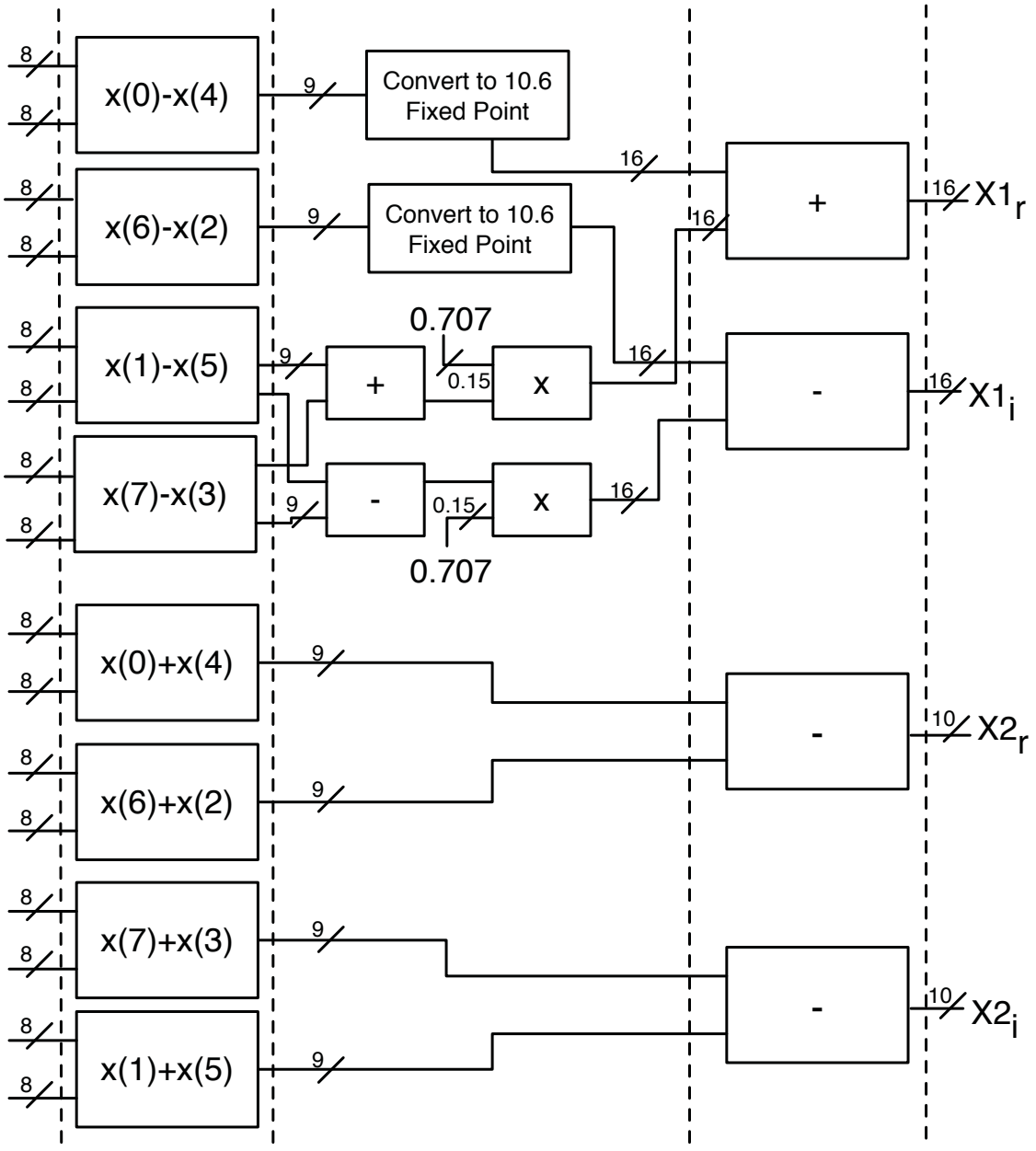


Figure 7.3: Data Flow Diagram of the Simplified 8-point FFT with Pipeline Registers

by multiplying $0.707 \cdot 2^{16}$ giving a hex value of 0x5a82 which is a Q0.16 number without the sign bit. The most significant bit is zero so it can be ignored. The 15-bit twiddle factor is then multiplied by the 10-bit result from the second stage resulting in a 25-bit Q10.15 number. This number is then truncated to become a Q10.6 fixed point number. To implement the multiplier, a Xilinx Core generated with its CORE

Generator software was used. Figure 7.4 shows a few of the settings for the multiplier. The estimated resource usage is shown in the left column and the software suggests to use 3 pipeline stages. With the multiplier broken up in to 3 stages, the resulting number of total pipeline stages is six.

In this design input data is latched in on the rising edge of the clock when the nd (new data) signal is asserted. When the data has gone through the pipeline, the dv (data valid) signal is asserted and the data can be used. The dv signal is implemented with a simple shift register with a width equal to the clock cycle delay of the pipeline. At each clock cycle, the contents of the shift register are shifted right and nd signal is shifted in.

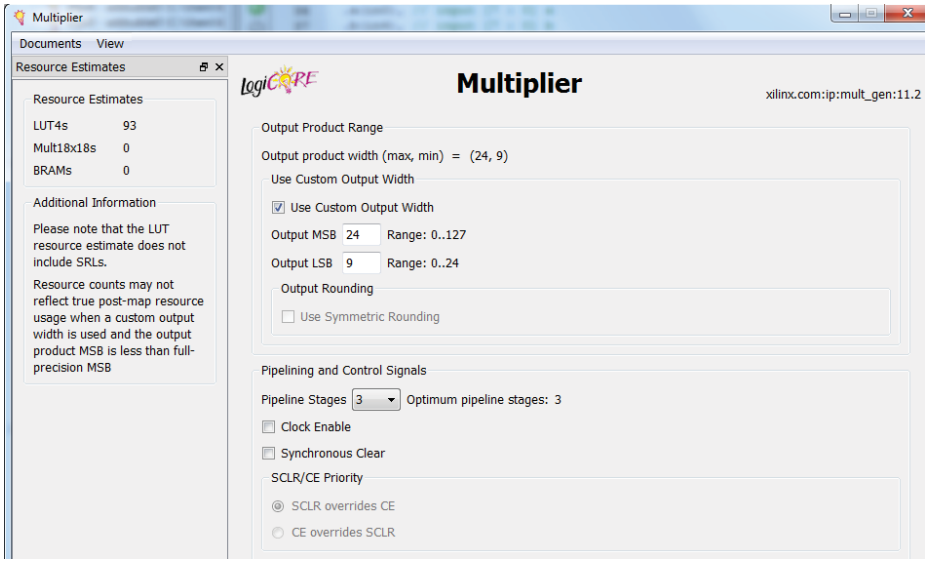


Figure 7.4: Xilinx Multiplier IP Core Settings

7.6 Pipelining Results

By pipelining the design into six stages, a decrease in 80% of the latency was achieved. It should also be noted that the single cycle design used a Complex Multiplier Xilinx

Core which requires three multiplication results as opposed to the one here that only needs two. Also, the Xilinx Multiplier Core has a configuration option for multiplication by the constant which further simplifies the implementation of the multiplier.

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 5.042ns (Maximum Frequency: 198.334MHz)
   Minimum input arrival time before clock: 4.643ns
   Maximum output required time after clock: 7.430ns
   Maximum combinational path delay: No path found
```

This snippet from the timing reports shows that the system frequency of 100MHz is well under the maximum frequency of the module.

Chapter 8

Inverse Tangent Module Design

8.1 Introduction

In the Kai *et al.* paper, quick and accurate phase calculations like the inverse tangent are made possible by a look-up table (LUT). It is estimated that for phase values of the N=3 case of the dual frequency equations, approximately 2MB (512·512·64 bits) of memory is necessary. This is too big to store in the on-chip BlockRAM and although with a 32 MB RAM a look-up table is a possibility, is actually inefficient on an FPGA system. The reason is because RAM reads can take many clock cycles. If we want to get the phase data for even one pixel then we would have to get the phase values and read a whole page out of memory. This could be mitigated by using a cache or other technique but if we can find a hardware efficient algorithm with good enough accuracy and single clock cycle latency, then the performance will increase.

Luckily, there are several hardware efficient algorithms to calculate trigonometric functions. It is possible to use series or polynomial approximation but even better is a CORDIC algorithm for low level hardware. CORIC stands for **CO**ordinate **R**otation **DI**gital **C**omputer. The CORDIC algorithm was developed by Jack Volder in 1959 to replace an analog resolver for measuring degrees of rotation on an B-58 bomber's navigation computer and what was needed was a real time algorithm with

little hardware [9].

What makes the CORDIC algorithm fast and efficient is the fact that it requires no multiplication hardware. Binary numbers are only multiplied or divided by factors of 2^n which are simply shifts n places to the right or the left, respectively. Also, the operations are limited to addition and subtraction, comparing two numbers to see if larger or smaller, shifts, and table look-ups. These are all cheap and fast operations in comparison to multiplication and division. [10].

8.2 CORDIC Theory

The CORDIC algorithm works by performing a vector rotation as a sequence of smaller rotations. The vector rotation equation rotates a point $P = (x, y)$ through an angle of θ degrees. The resulting rotated point in the Cartensian plane has coordinates where

$$x^{'} = xcos\theta - ysin\theta \tag{8.1}$$

$$y^{'} = xsin\theta - ycos\theta \tag{8.2}$$

This can be rearranged as:

$$x^{'} = cos\theta \cdot [x - ytan\theta] \tag{8.3}$$

$$y^{'} = cos\theta \cdot [y + xtan\theta] \tag{8.4}$$

If the each rotation angles are limited to $tan(\theta) = \pm 2^{-i}$ where i is the iteration number, then the tangent term is expressed as a shift. The cosine term $cos(tan^{-1}2^{-1}) = 1/\sqrt{1 + 2^{-2i}}$ which is a scale constant k_i. The product of these k terms is the gain

Table 8.1: CORDIC Algorithm Iterations for $tan^{-1}(2.5)$

i	n	x	y	2^{-n}	Phase Shift °	Cum. Angle
		2.000000	5.000000			0.000
1		5.000000	-2.000000		-90	-90.000
2	0	7.000000	3.000000	1.000000	45	-45.000
3	1	8.500000	-0.500000	0.500000	-26.565051	-71.565
4	2	8.625000	1.625000	0.250000	14.0362435	-57.529
5	3	8.828125	0.546875	0.125000	-7.1250163	-64.654
6	4	8.862305	-0.004883	0.062500	-3.5763344	-68.230
7	5	8.862457	0.272064	0.031250	1.78991061	-66.440
8	6	8.866708	0.133588	0.015625	-0.8951737	-67.335
9	7	8.867752	0.064317	0.007813	-0.4476142	-67.783
10	8	8.868003	0.029677	0.003906	-0.2238105	-68.007
11	9	8.868061	0.012357	0.001953	-0.1119057	-68.119
12	10	8.868073	0.003697	0.000977	-0.0559529	-68.175
13	11	8.868075	-0.000633	0.000488	-0.0279765	-68.203
14	12	8.868075	0.001532	0.000244	0.01398823	-68.189
15	13	8.868075	0.000449	0.000122	-0.0069941	-68.196
16	14	8.868075	-0.000092	0.000061	-0.0034971	-68.199
17	15	8.868075	0.000179	0.000031	0.00174853	-68.197
18	16	8.868075	0.000043	0.000015	-0.0008743	-68.198
19	17	8.868075	-0.000024	0.000008	-0.0004371	-68.199

of the system. The gain can be calculated in advance by the number of iterations and accounted for by multiplying the result with the inverse of the gain. Removing the scale constant k_i and introducing a third equation for z which accumulates the angles gives:

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i} \tag{8.5}$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i} \tag{8.6}$$

$$z_{i+1} = z_i - d_i \cdot tan^{-1}(2^{-i}) \tag{8.7}$$

where $d_i = \pm 1$ depending on the direction of rotation.

The CORDIC algorithm can easily understood with an example. Let $x = 2$ and $y = 5$ and finding $tan^{-1}(5/2)$. The gain value mentioned above is ignored because we

are only interested in phase.

1. Check if y is greater than or less than zero. If less than, rotate by -90°otherwise rotate by 90°.

2. Continue to rotate by $tan^{-1}(2^{-n})$ from 0 to n. Each time evaluate y and decide which direction to rotate. Accumulate the degrees rotated in total.

The steps in the process can be seen in Table 8.1. It is clear from the table that the more iterations completed, decided the number of bits of precision in hardware, the closer the cumulative angle value gets to the actual phase value while y and θ_i approaches zero.

The iterative process is not only simple but it turns out to be accurate with enough iterations. As proven in [10], the error made in computing $n+1$ iterations is no more than $1/2^n$. For a 32-bit output precision implementation $1/2^{32} = 2.32 \times 10^{-10}$ or approximately 8 decimal places of accuracy.

8.3 CORDIC Implementation

The architecture uses a Xilinx IP CORDIC 4.0 core generated using the Xilinx Core Generator software. A CORDIC core could be designed for this project or an open source core found online but a fully tested Xilinx IP Core is preferred. Xilinx cores are optimized for the FPGA architecture so they can have a high performance and efficiently utilize the resources.

The CORDIC core can implement the following equation types:

- Rectangular and Polar Conversion

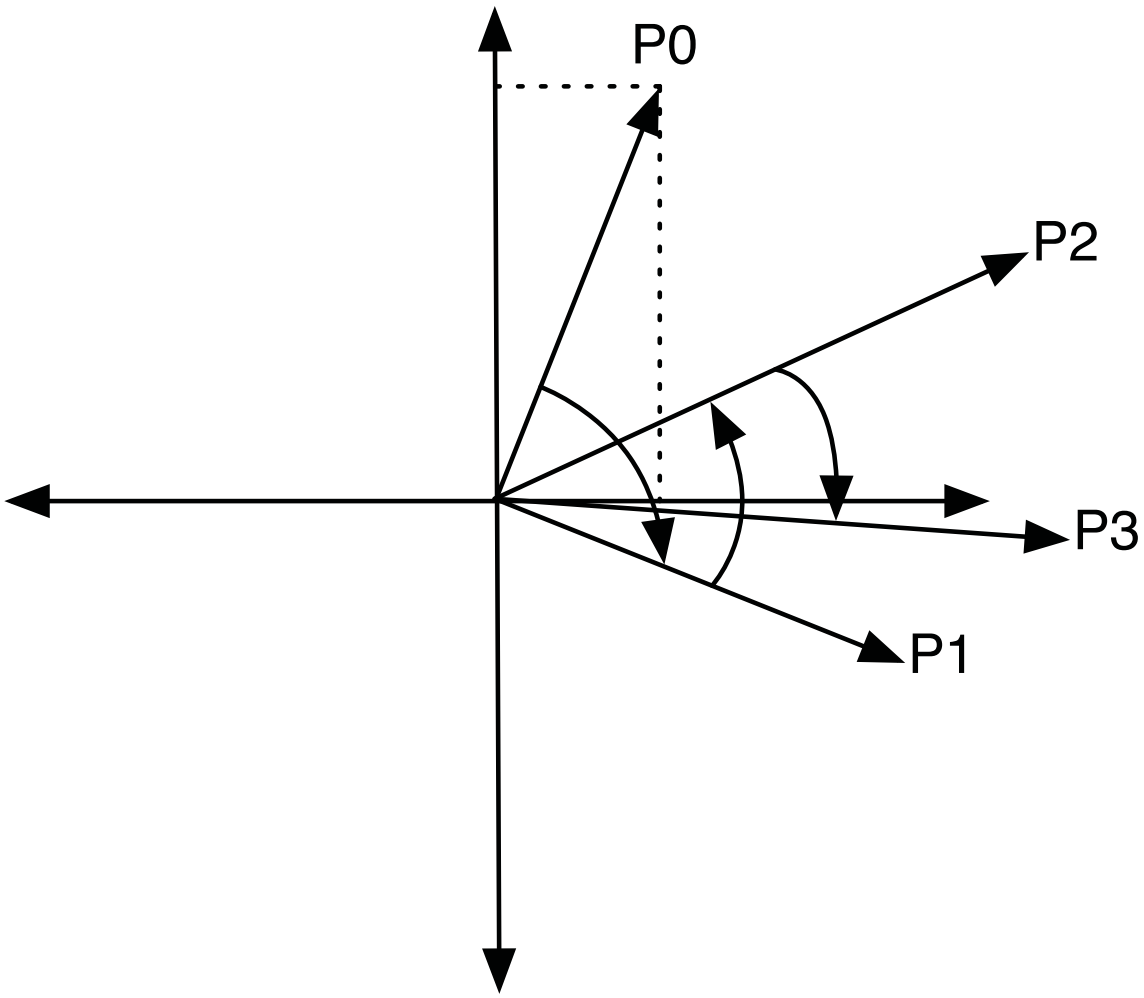


Figure 8.1: A Diagram Illustrating the Rotations in the CORDIC Algorithm

- Trigonometric
- Hyperbolic
- Square Root

A block diagram of the CORDIC core is presented in Figure 8.2. The inputs used for the inverse tangent operation are the X, Y, ND and the outputs used are Phase output and the RDY signal. ND is logic high when new data is on the input ports. RDY is logic high when data is ready on the output. The coarse rotation option

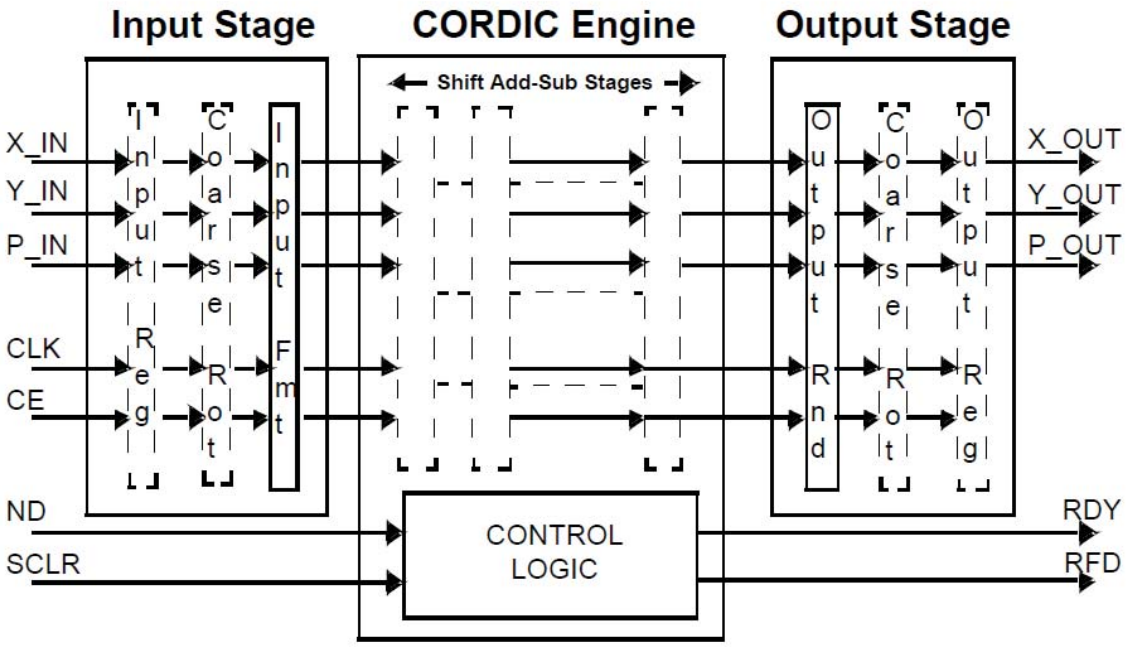


Figure 8.2: Xilinx CORDIC Core Block Diagram [14]

extends the Phase range from the first quadrant, $-\pi/4$ to $\pi/4$, to the full circle $-\pi$ to π .

As the CORDIC core specifications sheet presents, the Inverse Tangent operation has the following range for inputs and outputs. The input vectors are in Q2.N format

Table 7: **ArcTan**

Signal	Range	Description
X_IN	-1 <= X_IN <=1	Input X Coordinate
Y_IN	-1 <= Y_IN <=1	Input Y Coordinate
PHASE_OUT	-Pi <= Phase Out <= Pi	Output Angle

Figure 8.3: Xilinx ArcTan I/O and Ranges [14]

with an implied sign bit and single integer bit. The output angle, expressed in radians is a fixed-point 2's complement number in Q3.N format with an implied sign bit and two integer bits. An example of the input and output is below with a width set to

10 bits:

$$X_{in} = 0010110000 = 00.10110000 = 0.6875 \tag{8.8}$$

$$Y_{in} = 0010000000 = 00.10000000 = 0.5000 \tag{8.9}$$

$$P_{out} = 0001010100 = 000.1010000 = 0.625 \tag{8.10}$$

The core has two architecture configurations : parallel with single-cycle throughput and Word Serial. Word Serial uses a single shift-addition/subtraction stage and with the output feeding back to the input. It minimizes device utilization but has a latency of N clock cycles for an N bits of accuracy on the output.

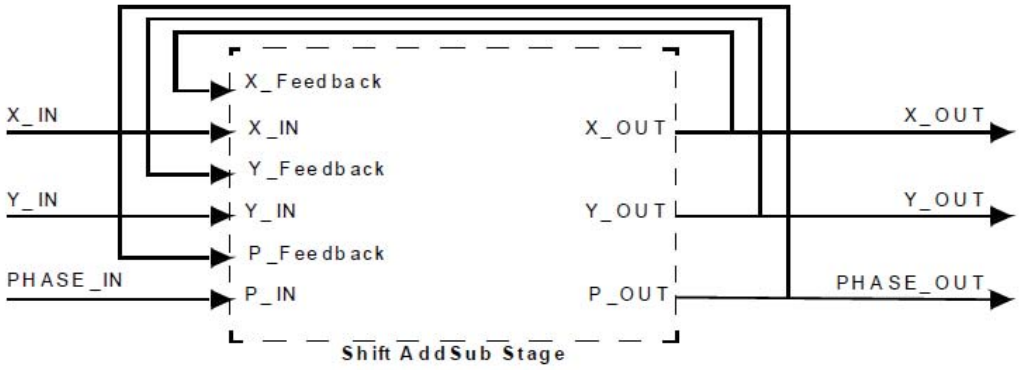


Figure 8.4: CORDIC Word Serial Architecture [14]

The architecture configuration that is used is a parallel architecture configuration. It can perform with single-cycle data throughput but uses more resources because it has a shift-addition/subtraction operation for each bit of accuracy.

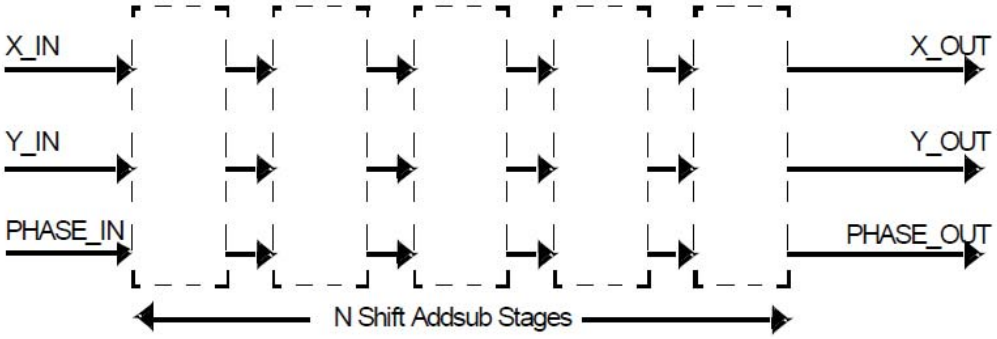


Figure 8.5: CORDIC Parallel Architecture [14]

8.4 Quantization Error

The CORDIC core has two sources of quantization error. The Output Quantization Error due to Input Quantization is from the uncertainty of the 1/2 LSB on the inputs. Because the inverse tangent depends on the ratio Y/X, a small X input can greatly affect the phase output. To lessen the effect of this quantization error, values are scaled up or zero padded. As seen in the FFT module, the multiplication operands take a 10-bit input and the final result is truncated to a 16-bit output. The result was not truncated any further to reduce loss of precision and lessen this quantization error. The other source of quantization error is due to the internal precision. As we saw in the CORDIC example above, the more angle iterations performed, the more precise the rotations were and hence the better the accuracy. By increasing the output result bit precision, this source of quantization error can be minimized.

8.5 Comparison of CORDIC and Matlab ATAN2

To check the accuracy, the output of the CORDIC core was compared to the Four-quadrant inverse tangent, ATAN2(y,x), function in Matlab.

CORDIC cores generated with an 8-bit input (Q1.6) and 8 (Q2.5), 16 (Q2.13), and 32 (Q2.29) bit outputs were simulated and the results compared. Figure 8.6 shows a section of the results from the simulation where x=32 and y was changed from -64 to 64. The 8-bit output with its low precision does not have a very accurate result in comparison to the 16-bit and 32-bit results.

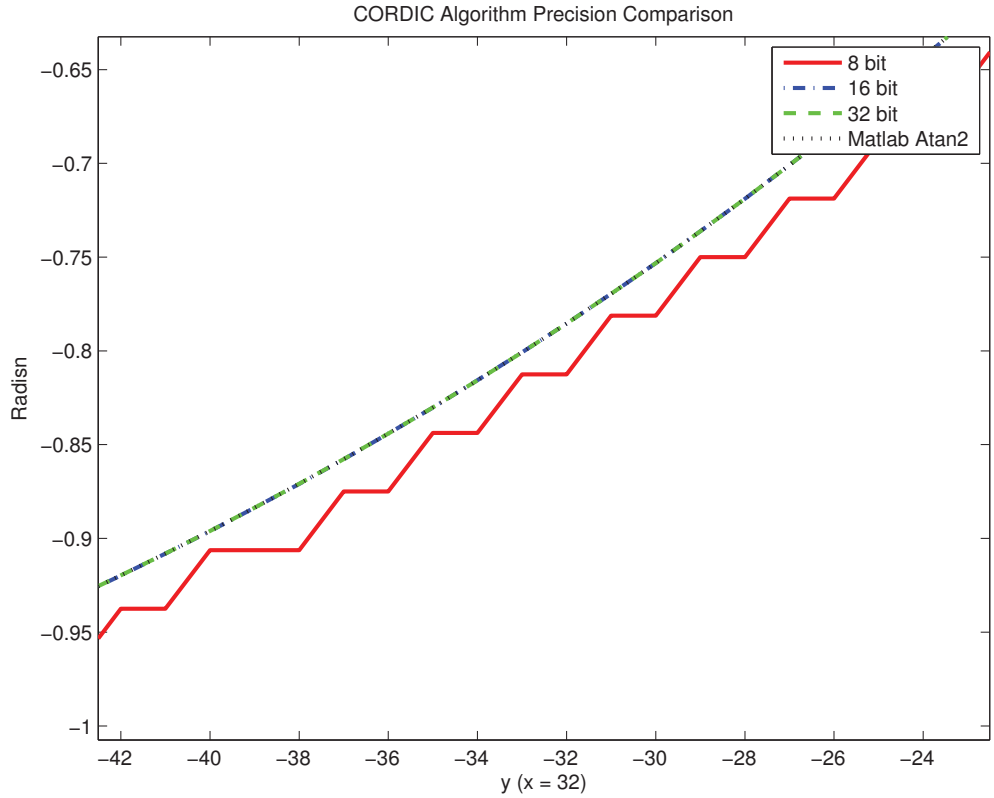


Figure 8.6: Comparison of CORDIC Core with Different Output Precisions

The mean of the difference of the double-precision floating point result from Matlab and the CORDIC cores gives 0.0141 radians for the 8-bit output, 5.29×10^{-5} radians for 16-bit output and 9.24×10^{-10} radians for 32-bit output. These numbers are roughly equivalent to $2^{-(nbits-2)}$.The 32-bit output is deemed accurate enough for this application.

During testing, a case where the CORDIC core generates a significantly different value from the Matlab ATAN2 function was found. When the y value is equal to zero and x<0, the Matlab result is π and the CORDIC result is $-\pi$. This is not an issue and is discussed in the Results chapter.

8.6 Results

The CORDIC core was generated using the Xilinx Core Generator software. The CORDIC algorithm takes 16 bit inputs for X and Y and has a 32-bit output. 32-bit precision on the output means that the minimum fractional representable is $2^{-29} = 1.86 \times 10^{-9}$ giving $log_{10}(2^{29}) = 8.72 \approx 8$ fractional decimal digits. This is comparable to single-precision floating number precision with about 7 fractional decimal digits of accuracy.

Chapter 9

Phase Unwrapping

9.1 Introduction

The inverse tangent function will return a value in the range $(-\pi, \pi]$ If the phase exceeds this range, as is the case of frequencies higher than 1, it will be wrapped, or repeated, around so it remains in this range. This introduces unwanted 2π jumps. With the pair of frequencies obtained the higher frequency phase ϕ_h needs to be "unwrapped" in order to find an absolute phase thereby getting rid of the remove ambiguity.

In order to unwrap the high frequency using a spatial approach, an algorithm needs to count the number of periods from one side to the other and adding or subtracting multiplies of 2π. Another way is to use a unit frequency as a reference to unwrap phase with 2π jumps. This method is less computationally intense and does not suffer the problem of discontinuities. Because the high and unit frequencies are projected together, as opposed to being projected separately, the acquisition speed is faster and more suitable for real-time applications.

The following pseudo-code was given to unwrap the higher phase with a unit frequency:

```
unitPhase = calculated phase between 0 and 2PI from k=1 term
```

```
highPhase = calculated phase between 0 and 2PI from k=2 term
N = the high frequency value
highPhaseScaled = highPhase/N;
temp1 = N*(unitPhase - highPhaseScaled) / (2PI);
// unit phase is now discrete steps from 0 to N-1
temp2 = round(temp1);
combinedPhase = 2PI * temp2 / N + highPhaseScaled;
```

As can be seen from the pseudo-code, the unit phase is subtracted from the scaled down high frequency phase and the result is scaled resulting in discrete steps from 0 to N-1. The round function rounds to the nearest integer which removes the noise from the unit phase. The robust high frequency phase that is scaled down by N is then added and the result is scaled down to fit in the range $(0, 2\pi]$. Figure 9.1 shows graphically how the combined phase looks like even with noise or distortion on the unit frequency. If the input phases are on the range of $-\pi$ to pi as is the case with the CORDIC core, then the phases are shifted to range between 0 and 2π.

9.2 Implementation of Phase Unwrapping on FPGA

The pseudo-code has division and multiplication operations and the constant π. Although possible to implement, the pseudo-code performance can be increased by simplifying the code to only use shifts, addition, subtraction, comparisons, and a look-up table. The multiplication and division of the N term can be done with shifts if $N = 2^n$. In the temp1 term, it is not necessary to scale it to 0 to N if the round function is rewritten.

The division of 2π is removed, making the range of 0 to (N-1)*2PI. To account for this the round functionality was changed. The round function rounds an integer up if the fractional component is greater than or equal to 0.5 and rounds down otherwise.

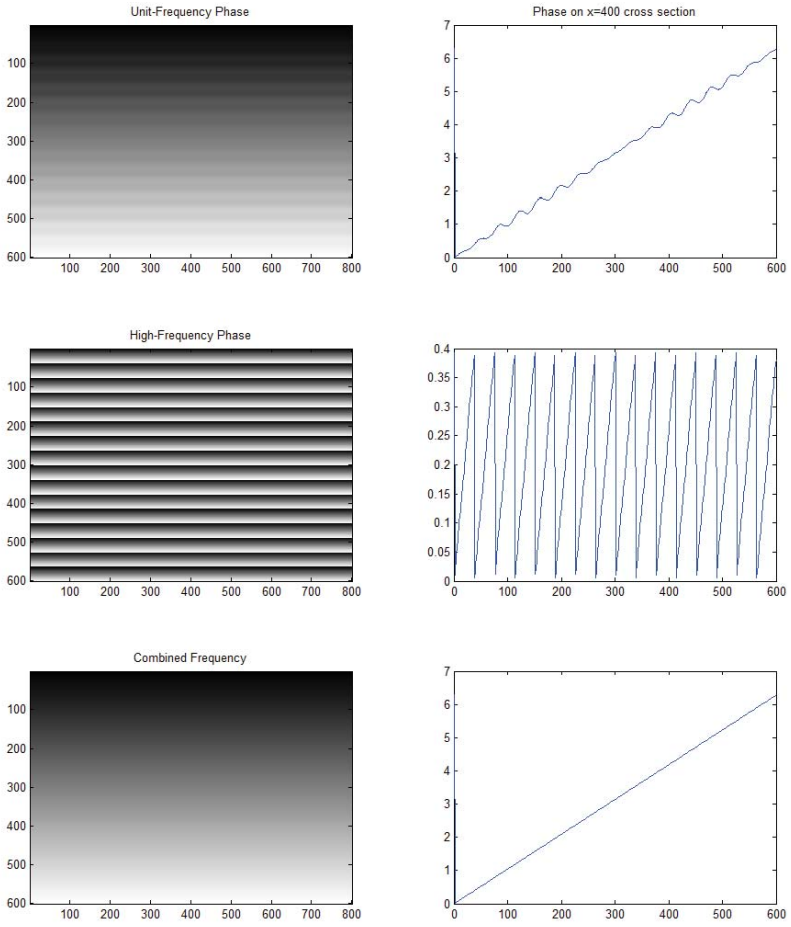


Figure 9.1: Unit Phase Unwrapping a High Frequency Phase. The Unwrapped Phase is at the Bottom.

If the round function is written such that 0.5 is changed to π and the nearest integer is a multiple of 2π then the removal of the division of 2π is accounted for. This is implemented by comparing if the value is less than odd multiples of π. If the comparison evaluates true, the result is rounded down the the even multiple of pi. For instance, 3.5 is greater than π but less than 3π so it is rounded to 2π. After the round, the result is divided (shifted) to be between 0 and 2π and added to the already scaled down high frequency phase value.

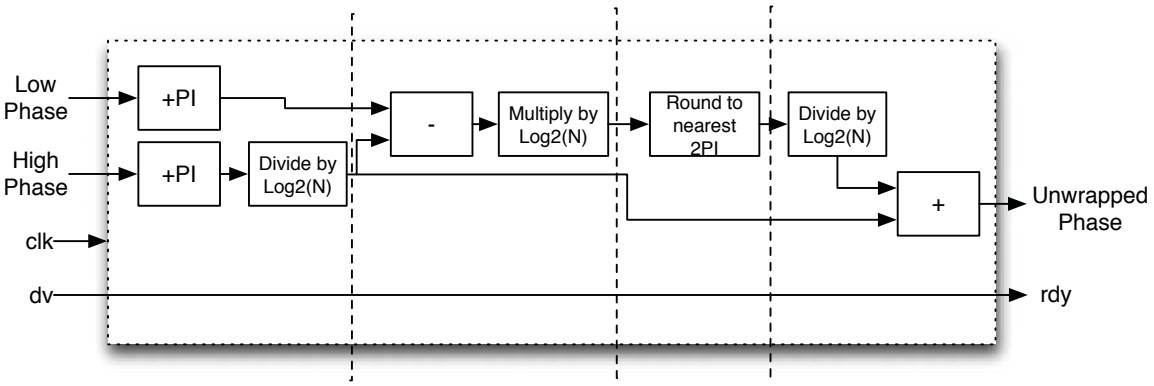


Figure 9.2: Data Flow Diagram Showing the Phase Unwrap Module

9.3 Conversion to Floating Point

Throughout the phase calculation pipeline, fixed-point numbers have been used. Fixed-point arithmetic is a simple representation and cheap to compute and therefore used in a lot of digital signal processors. The phase data needs to be read and a model constructed by a PC and PCs commonly use the IEEE 754 floating point standard. Therefore, a core to convert from a 32-bit Q3.29 number to single precision floating point is instantiated.

The core used is a Xilinx LogiCore IP Floating-Point Operator v5.0 [15]. It provides floating-point arithmetic on an FPGA. One of the operations it is capable of is conversion from fixed-point to floating point. The inputs are the fixed point data, a nd (new data) signal that is connected to the unwrap module's output data rdy signal and has a clock. The outputs are a 32-bit single precision floating point number, which is connected to the Ouput FIFO's data in port, and an output data rdy signal, which is connected to the Output FIFOs write enable port.

Chapter 10

Thresholding of Phase Data

In the previous chapters, the implementation of an architecture to calculate phase data was described. As mentioned in the introduction, the magnitude of k=1 DFT coefficient, B^c, can be used to indicate the amplitude of the sinusoid reflecting off of a point on the object surface. For pixels where I_n^c is constant or not affected by the projected sinusoid patterns, B^c will be close to zero so B^c can be employed as a shadow or noise filter.

The magnitude of a complex number calculated as $\sqrt{X_r^2 + X_i^2}$ where X_r and X_i are the real and imaginary components of the FFT coefficient. The magnitude can be complicated to calculate but because magnitude is used as a filter, full precision is not necessary. To make the calculations considerable simple with a minor loss in accuracy, a well known technique that is described in [25] was used. The algorithm estimates the magnitude as:

$$Mag \approx MAX(|X_r|, |X_i|) + 0.5 \cdot MIN(|X_r|, |X_i|) \tag{10.1}$$

Using this equation results in an average error of -0.086 which is acceptable for filtering. The result is then compared to a set threshold value and if the magnitude is less than the set threshold, the pixel value is zeroed and shows up as a black pixel.

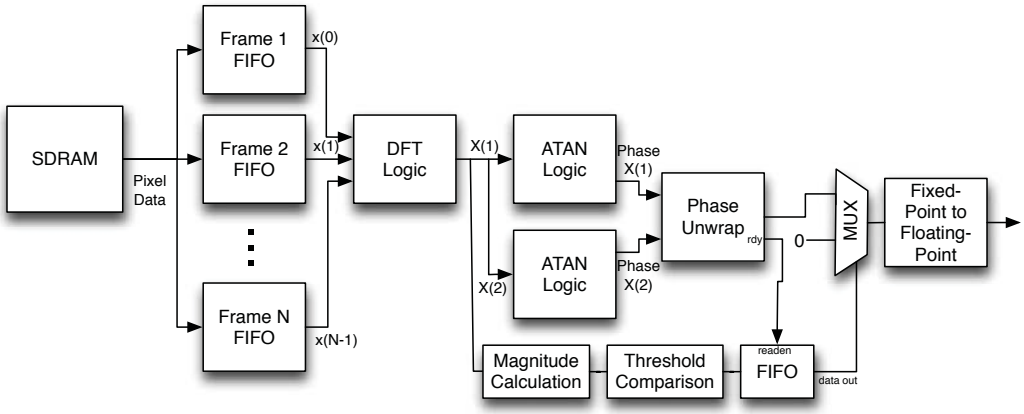


Figure 10.1: Phase Calculation Dataflow Diagram with Thresholding

10.1 Implementation

The absolute value of real and the imaginary parts of k=1 coefficient from the FFT is first taken by checking if the most significant bit is 1, a negative number. If so, then the absolute value is taken. The two numbers are then compared. The minimum number is shifted right one, dividing it by two, and the result is added to the maximum number. The result is then compared to a set threshold and the binary result (1 = greater than threshold, 0 = less than threshold) is stored in a first-word fall-through FIFO. When the phase unwrap data is ready, the FIFO data is read and the phase data is zeroed if the FIFO output is a 0 before sending phase data to be converted to a floating-point number. Figure 10.1 shows the phase dataflow diagram with thresholding modules added.

Chapter 11

GUI Application

A GUI application using the Qt libraries and the Opal Kelly C++ API to communicate with the FPGA board was designed and written. This application recognizes when any Opal Kelly board is plugged in and will dynamically update a combo-box with the type of board. It is possible to plug in several boards and then select which one the user wants to use. The Opal Kelly software ships with their FrontPanel software to program the FPGA board but this is not necessary because programmable capability was built it into the designed Qt application. When the "Capture" button is pressed, a sequence of 8 frames from the CMOS image sensor is captured and written into the SDRAM. In addition to data from the camera, binary data of images can be sent from the PC for processing. To send data, the "Send Image" button is clicked and up to 8 images can be stored in the SDRAM. The ability to take external data assists with testing the device and provides an alternative for calculating phase. The "Show Image" button will read and show the 8 images from the SDRAM. Click the "Phase Data" button and the frames in memory will be used to calculate the phase data. The read frame data and phase data is saved as a binary file in the directory of the application. The sent and captured images are shown in eight boxes on the GUI. The application can display video data as it receives phase data. The threshold level

for filtering of phase data can be set in the application.

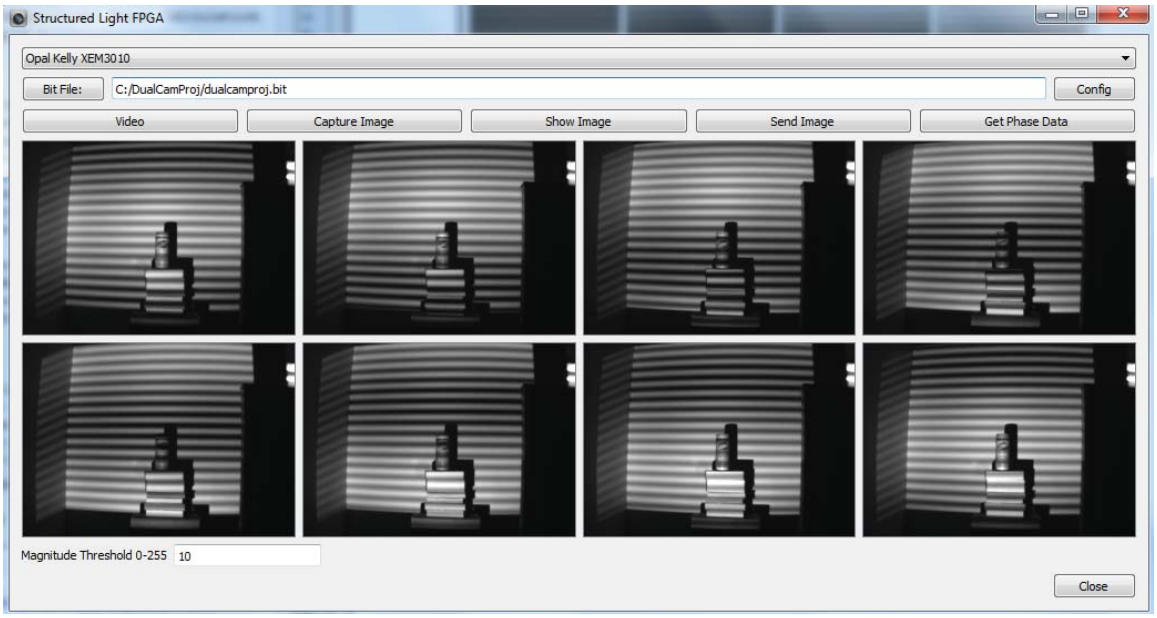


Figure 11.1: Screenshot of the GUI Application.

Chapter 12

System Analysis and Results

12.1 Introduction

This chapter provides information about the system architecture performance and results. The modules of the HDL code were tested with simulation test benches before they were experimentally tested on the FPGA as a system.

The architecture design works as described and is able to calculate the unit and high frequency phase properly and combine the two into a single phase value. The results were compared to a Matlab implementation using sample test data. The system can project patterns and take pictures of the scene.

12.2 Hardware Prototype

Figure 12.1 shows the hardware prototype built for this project. The adapter board connects the camera board and VGA board to the FPGA breakout board. A pen is shown for scale.

12.3 Results

To first test if the system is working, dual-frequency images were generated and its phase calculated with Matlab and the FPGA and the results were compared. Figure

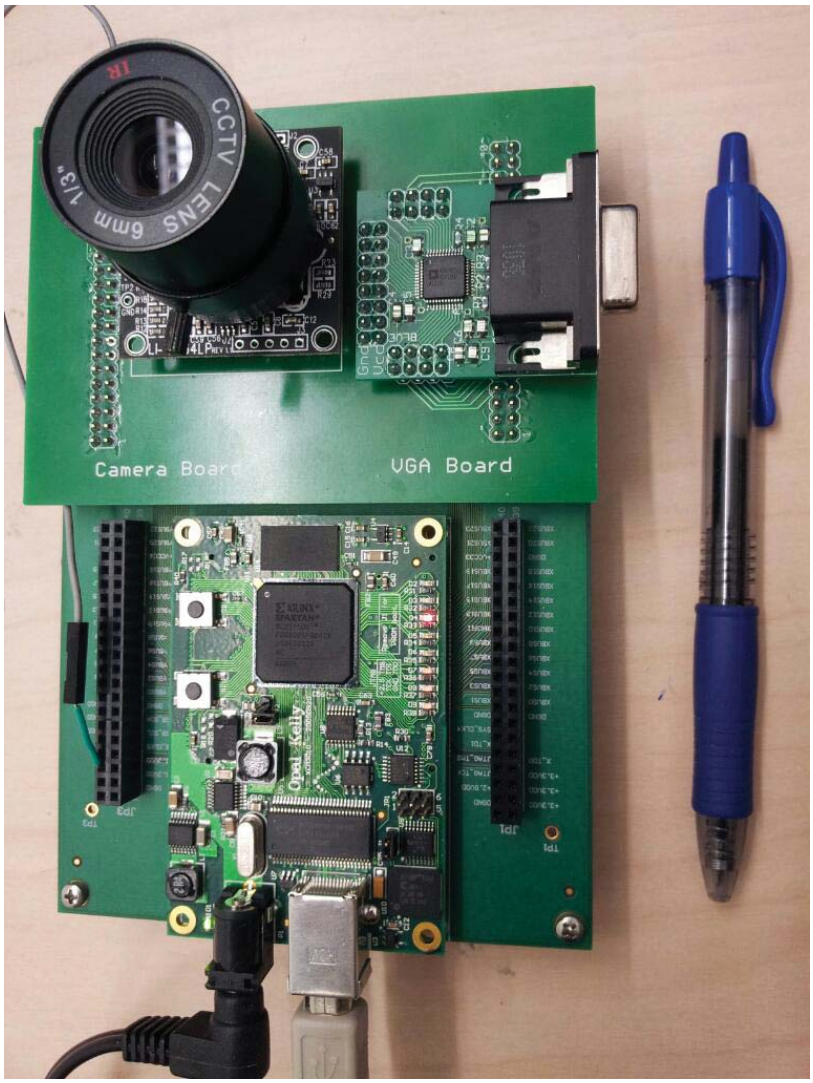


Figure 12.1: The Hardware Prototype

12.2 shows eight dual-frequency patterns that were used as test input.

The results of the phase calculation are shown in Figure 12.3 The phase calculations look identical. To further test the FPGA phase calculation, random data was generated and the results were compared.

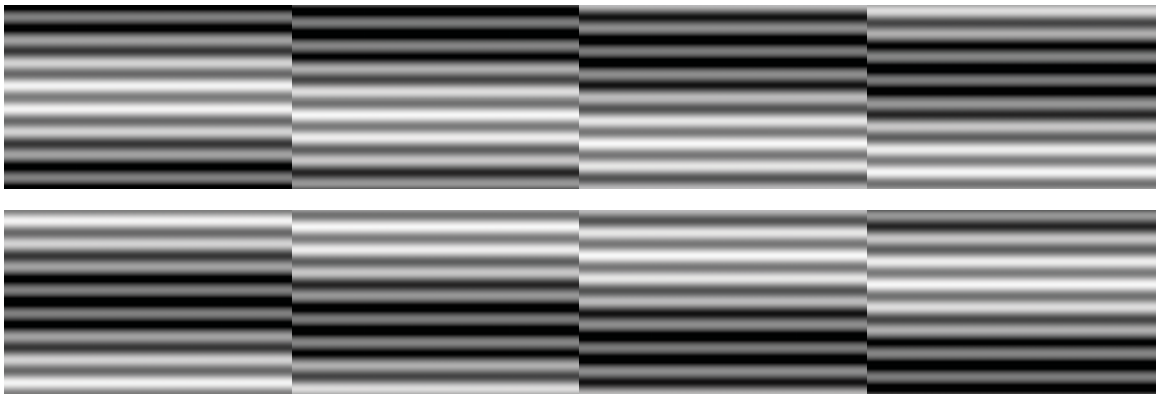


Figure 12.2: Eight Dual Frequency Patterns where $A^p = 128$ and $B^p = 60$

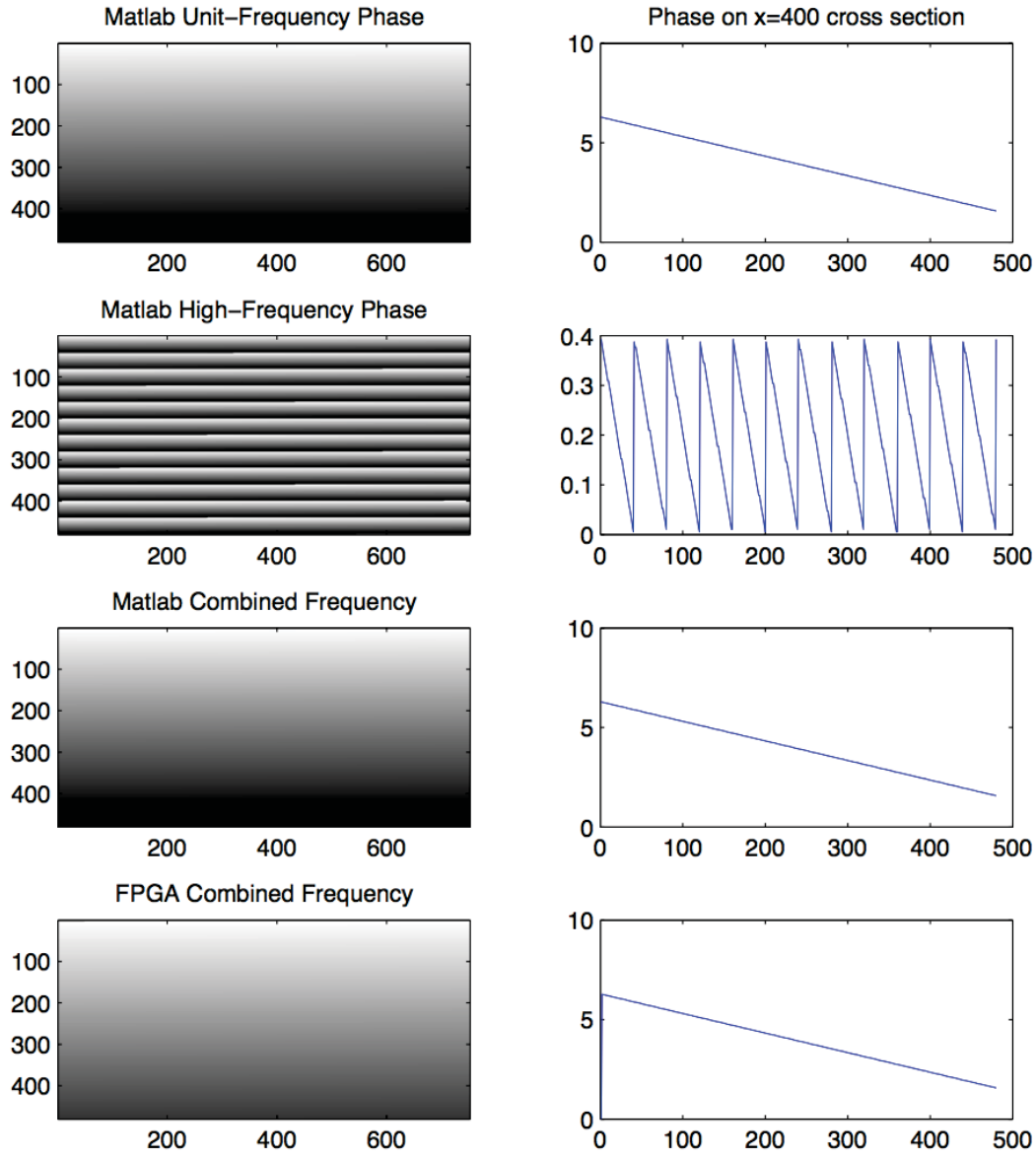


Figure 12.3: Phase Data for N=8

Figure 12.4 shows the difference between the double-precision floating point Matlab phase result and the FPGA phase result on a pseudorandom numbers input test. Most of the difference results is on the order of 10^{-7} or smaller. The difference of -6.28 is due to the difference between the CORDIC result and the ATAN2 function as discussed in Section 8.5. These pixels are expect to be at the top or bottom edge of the projected pattern and will often by filtered out. The elements where the difference of 0.3927 is due to a lower precision (32-bit) in the round function circuitry for phase unwrapping. These are numbers that are right on the edge of either being rounded up or down. These differences are deemed acceptable for this application.

difphase <480x752 double>

	687	688	689	690	691	692	693	694	695	696	697	698
450	1.4383e-07	-1.8297e-07	-1.0964e-07	-1.3768e-08	-3.5391e-09	5.1645e-08	-1.1212e-07	1.7062e-07	2.6861e-08	-2.2319e-07	5.6835e-09	-1.1220e-08
451	4.0436e-08	-1.4906e-08	-3.1060e-08	-4.4014e-08	-2.6700e-08	-6.9674e-09	-5.2948e-08	-3.2542e-09	4.1521e-08	1.1443e-07	-2.1877e-07	-2.2038e-07
452	-2.2217e-08	1.0388e-07	1.9393e-07	-4.4476e-08	1.9672e-08	-9.0922e-09	-1.3261e-08	-1.2147e-07	1.6794e-08	-1.1030e-07	-5.4750e-09	-6.6977e-08
453	-3.3212e-09	-1.9870e-07	1.1320e-07	-2.1084e-07	-1.1412e-07	-8.5366e-08	1.0205e-08	5.6867e-08	-7.1654e-08	-4.5061e-09	1.9859e-07	8.1060e-08
454	-5.0899e-08	-1.3337e-07	3.9375e-08	2.6255e-08	1.4092e-07	2.2086e-07	2.6723e-08	-3.0428e-08	-1.7140e-07	-2.5955e-08	-1.3064e-08	-2.4829e-08
455	-5.3794e-08	8.8129e-09	-3.1633e-08	1.0846e-09	4.1177e-08	-9.7013e-08	-3.3121e-08	-3.5620e-08	-4.0599e-08	6.5944e-08	-8.6230e-10	3.7644e-08
456	5.4604e-09	1.1066e-07	4.9490e-08	9.8358e-10	9.2438e-08	-3.5254e-08	-2.3790e-09	3.7668e-08	1.7612e-07	8.5203e-08	3.2474e-09	1.1460e-07
457	4.4683e-08	6.8843e-08	3.4828e-08	2.3117e-07	-1.2539e-07	4.2938e-09	-3.6353e-09	4.6196e-08	5.8288e-08	-1.2990e-08	3.2515e-08	-4.0226e-08
458	-3.8919e-08	3.9875e-08	4.7022e-09	-9.9382e-08	-2.8235e-08	8.9761e-08	-6.1540e-08	-1.1139e-07	5.1099e-08	-4.3624e-08	-7.7339e-09	-1.8426e-08
459	-1.0313e-07	-0.3927	-2.0030e-08	3.7470e-08	1.9211e-08	-2.3886e-09	-4.1665e-08	9.2314e-08	1.1175e-07	-2.2848e-07	-4.7560e-08	6.8821e-09
460	4.9686e-09	9.3731e-08	-3.4715e-08	2.2166e-07	3.7956e-08	-4.6479e-08	-4.6135e-08	5.5669e-09	-1.1056e-07	5.5094e-08	-1.1817e-08	2.4425e-08
461	2.3129e-07	2.1469e-07	-2.0612e-07	7.0002e-08	-4.6739e-08	-7.1486e-09	9.0303e-08	-1.5190e-07	2.0539e-07	8.2110e-08	1.1412e-07	2.0892e-07
462	-9.3909e-08	9.5196e-08	7.3485e-08	-9.6282e-08	2.0044e-07	-8.7009e-08	-7.6201e-09	1.1786e-08	-1.6880e-07	9.1735e-08	6.6305e-08	2.5711e-10
463	-3.6818e-08	5.3542e-08	-2.3316e-08	1.6187e-08	-1.5197e-07	1.9223e-08	-1.4199e-07	-7.0181e-08	-6.5924e-10	2.7160e-08	-2.4430e-08	-1.6152e-09
464	-8.9347e-08	2.3193e-09	-6.2832	8.1696e-08	-4.1322e-08	-1.0073e-07	-9.5756e-08	-6.0950e-08	3.8996e-08	-2.3896e-08	-1.6694e-07	-1.1609e-07
465	-1.8081e-08	5.3160e-09	1.0808e-07	4.6536e-09	9.4403e-08	-2.5964e-08	1.6987e-07	-8.0796e-08	2.2191e-07	9.3568e-08	-7.8292e-08	1.1237e-07
466	2.3656e-07	-4.7919e-08	-6.8955e-08	3.7461e-08	3.1285e-09	3.0115e-08	-7.6686e-08	-3.2598e-08	7.3260e-09	-1.4583e-08	5.3578e-08	-1.3273e-08
467	4.9895e-08	-1.0690e-07	-3.2538e-08	4.1706e-08	2.5186e-08	2.6255e-08	-5.6077e-08	4.4427e-08	7.9918e-09	2.5629e-09	9.5129e-08	-3.3087e-08
468	4.7072e-08	-4.5298e-08	-1.0928e-08	-8.0518e-09	6.8132e-09	4.3917e-08	1.3643e-07	4.9203e-08	-1.5214e-07	-3.1566e-08	-7.0344e-08	1.0140e-07
469	-2.3071e-07	9.6025e-08	-2.0215e-07	-1.2726e-07	1.8104e-09	1.8450e-08	-2.1926e-07	-1.5432e-07	-3.7285e-08	-1.0399e-07	3.8412e-08	-2.1447e-07
470	1.2435e-08	-1.8322e-07	-2.3140e-08	-4.9068e-08	-1.0588e-08	-1.6783e-07	8.0600e-08	1.1125e-08	7.7834e-08	9.9716e-08	1.0955e-07	-1.1788e-08
471	-1.9308e-08	-3.7872e-10	8.1269e-08	-8.9619e-09	4.9093e-08	1.2360e-07	-1.5886e-07	2.5660e-08	-5.2452e-09	-2.6779e-08	1.7824e-08	-3.9503e-08
472	7.9573e-08	2.6794e-08	-7.3974e-08	-7.0491e-08	-1.8373e-08	-3.3109e-08	-6.4619e-08	-1.0621e-07	5.7729e-08	3.5559e-08	-8.6812e-08	1.8764e-08
473	8.7767e-08	1.1734e-07	4.6551e-08	3.2965e-08	4.0156e-09	-1.0496e-07	1.9156e-09	-2.4516e-08	-1.0641e-07	1.1009e-08	9.6879e-08	4.9173e-09
474	1.5611e-07	8.1408e-09	-2.2032e-08	-8.4251e-09	1.1887e-07	2.3821e-07	2.9115e-08	-2.1832e-09	-4.2594e-08	-4.4116e-08	1.4460e-07	-5.0113e-08
475	-5.0608e-08	-3.9902e-09	-1.2056e-07	6.1057e-09	-2.2190e-09	-2.3885e-09	-1.8286e-09	-8.9697e-09	-1.5594e-08	4.3845e-09	1.7408e-07	-2.1288e-07
476	4.7945e-08	-6.5647e-08	4.9817e-08	-1.9958e-07	1.1954e-07	4.0774e-08	-2.7532e-08	2.2946e-07	-4.4727e-08	-7.7582e-08	-2.2674e-07	1.3175e-07
477	0.3927	-6.8462e-09	2.9853e-08	-3.2600e-08	1.7215e-07	-4.0068e-08	-4.9525e-08	-2.7119e-08	7.7677e-08	2.5152e-08	7.7647e-08	-7.3713e-09
478	1.7684e-08	9.3805e-08	-2.0598e-08	2.2788e-08	-6.1628e-08	2.2880e-07	2.2385e-07	-3.4255e-09	8.1701e-09	-9.7775e-08	-1.4417e-07	-3.1936e-08
479	-1.7407e-07	2.0528e-08	-3.8389e-08	1.4095e-08	1.8570e-07	2.0507e-07	2.2411e-07	2.3019e-08	2.2705e-08	6.4312e-08	4.1108e-09	5.6497e-08
480	-1.9947e-07	1.5755e-08	4.6037e-08	1.2003e-08	-3.0778e-08	1.5884e-07	1.7461e-08	-2.6439e-08	1.0659e-07	-1.9396e-08	-1.0265e-07	-6.4245e-08
481												

Command Window

```
>> difphase = combinedPhase - fpgaphase;
```

Figure 12.4: Uniformly Distributed Pseudorandom Numbers Test. Difference between Combined Phase Result from Matlab and FPGA.

Figure 12.5 shows the results of an actual capture using the built projector-camera system before calibration. The unit-frequency phase shows crosstalk of the patterns. The expected result is a smooth gradient like that shown on Figure 12.3.

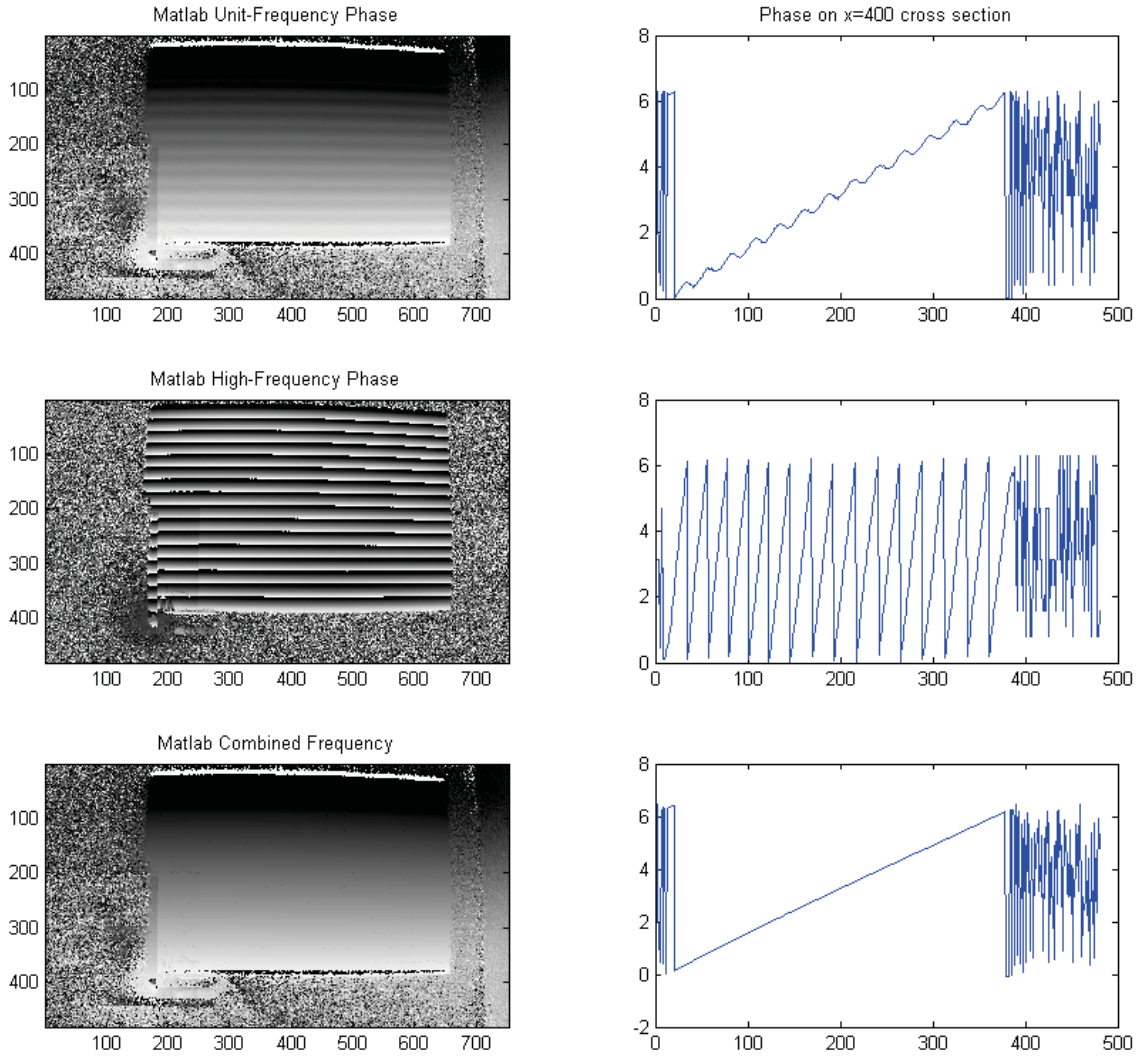


Figure 12.5: Capture of Projected Patterns Showing Crosstalk

It was hypothesized that this was a result of the non-linear gamma of the projector which is common on display equipment. To determine the response curve, a set of 256 uniform gray-scale patterns ranging from 0 to 255 were projected on to a surface and a image captured for each pattern. If $I^p(x, y)$ is the intensity of the pattern to be

projected and $I^c(x, y)$ is the value of a given pixel pixel of the captured image then:

$$I^c(x, y) = H(I^p(x, y)) \tag{12.1}$$

where H is the non-linear monotonic response function giving the correspondence of the pixels value and the luminance from the projector. With H determined, a compensated $I^{p'}$ can be computed with the equation:

$$I^{p'}(x, y) = H^{-1}(I^p(x, y)) \tag{12.2}$$

Figure 12.6 shows a non-linear response curve with I^p on the x-axis and I^c on the y-axis. A small section of each image was sampled and the mean taken giving the value I^c.

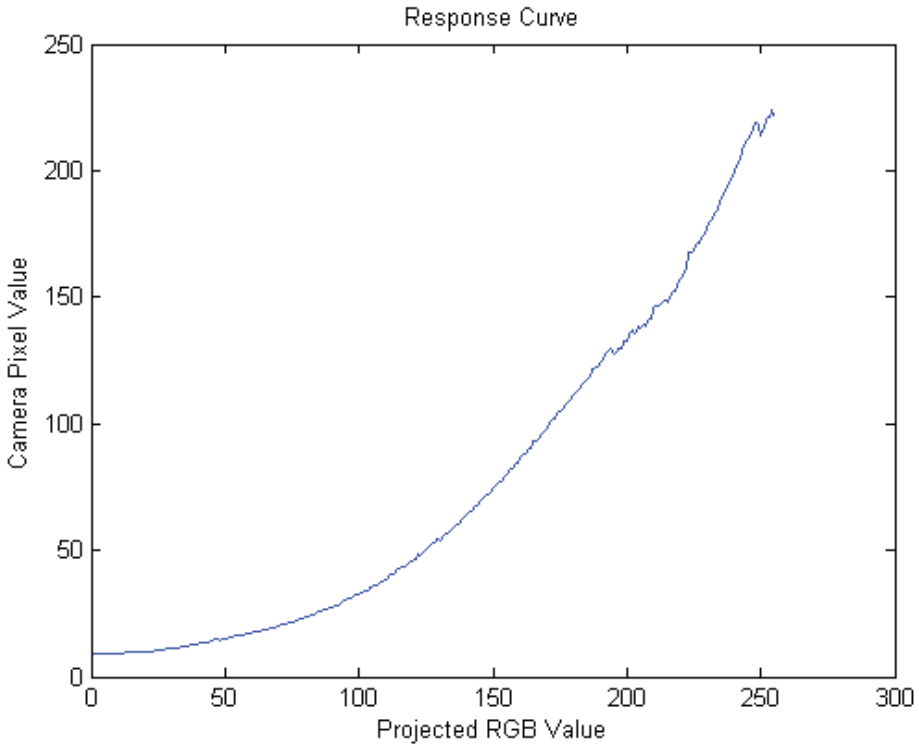


Figure 12.6: Projector-Camera Response Curve

Figure 12.7 shows a fixed linearized response curve. A look-up table ROM was

generated with the Xilinx core generator software and initialized with the inverse response function H^{-1} values. The summation of the unit frequency and high frequency values, is the data input to the ROM and the output is the compensated projector pixel value.

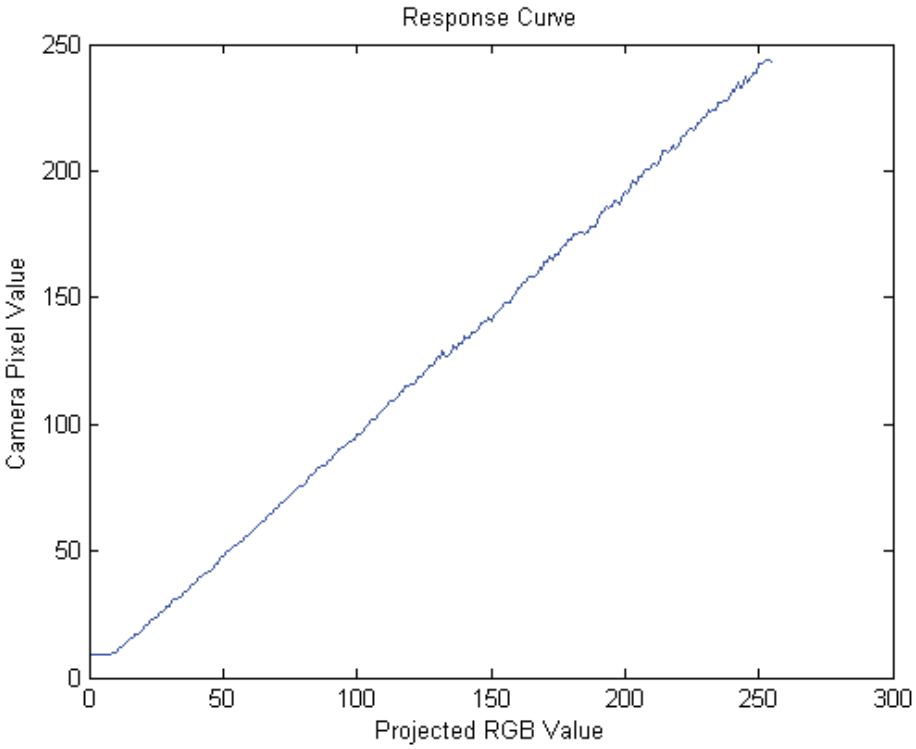


Figure 12.7: Projector-Camera Linearized Response Curve

With the response curve now linearized, the patterns were captured again giving a cleaner unit-phase without the crosstalk as shown in Figure 12.8.

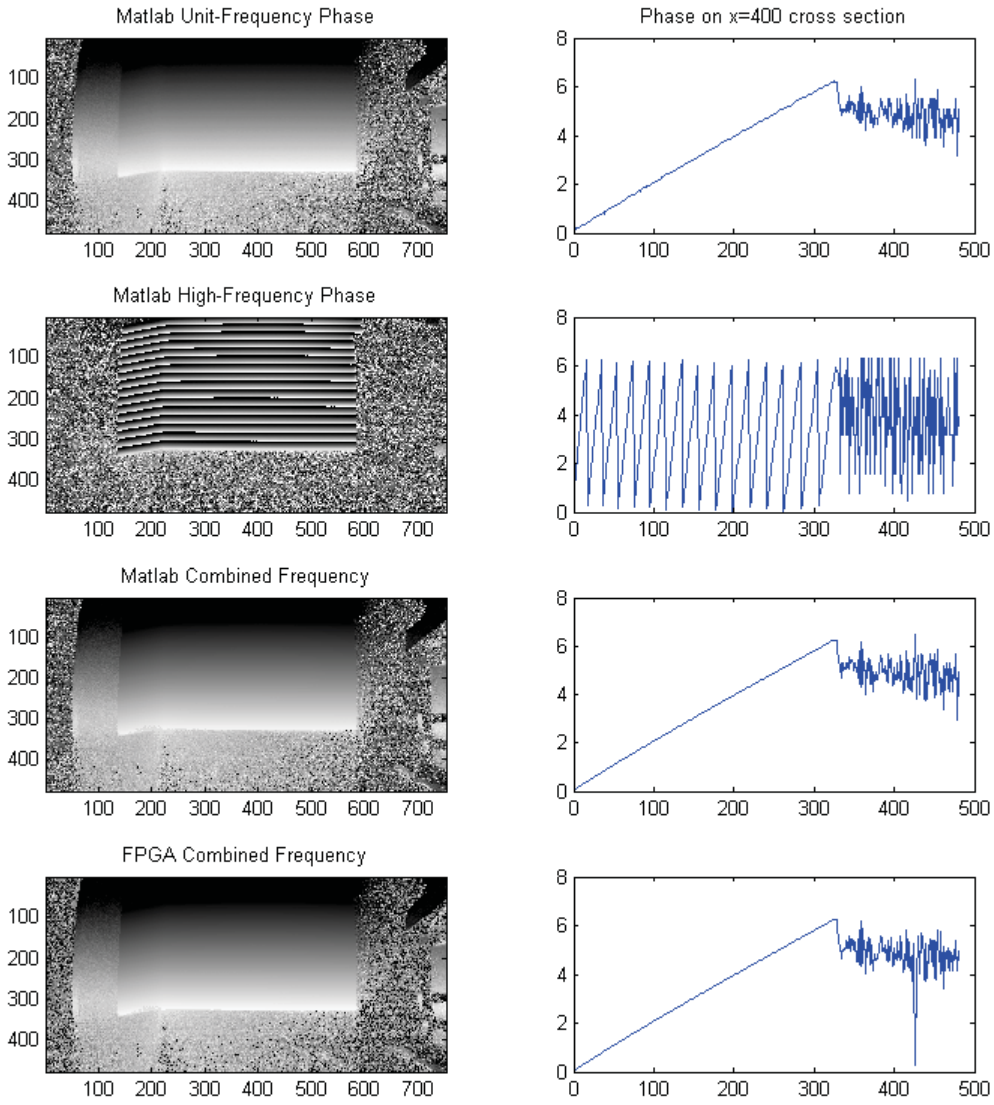


Figure 12.8: Capture of Projected Patterns

The above images show noise in the areas where the pattern is not being projected. Setting a threshold value for the magnitude will remove the noise leaving only pixels with 3-D data. Figure 12.9 shows a scene with an object. The top image is the magnitude image. This is used as a filter and could also be used as a texture in the resulting 3-D construction. The second image shows phase data without thresholding and the third image shows phase data with thresholding.

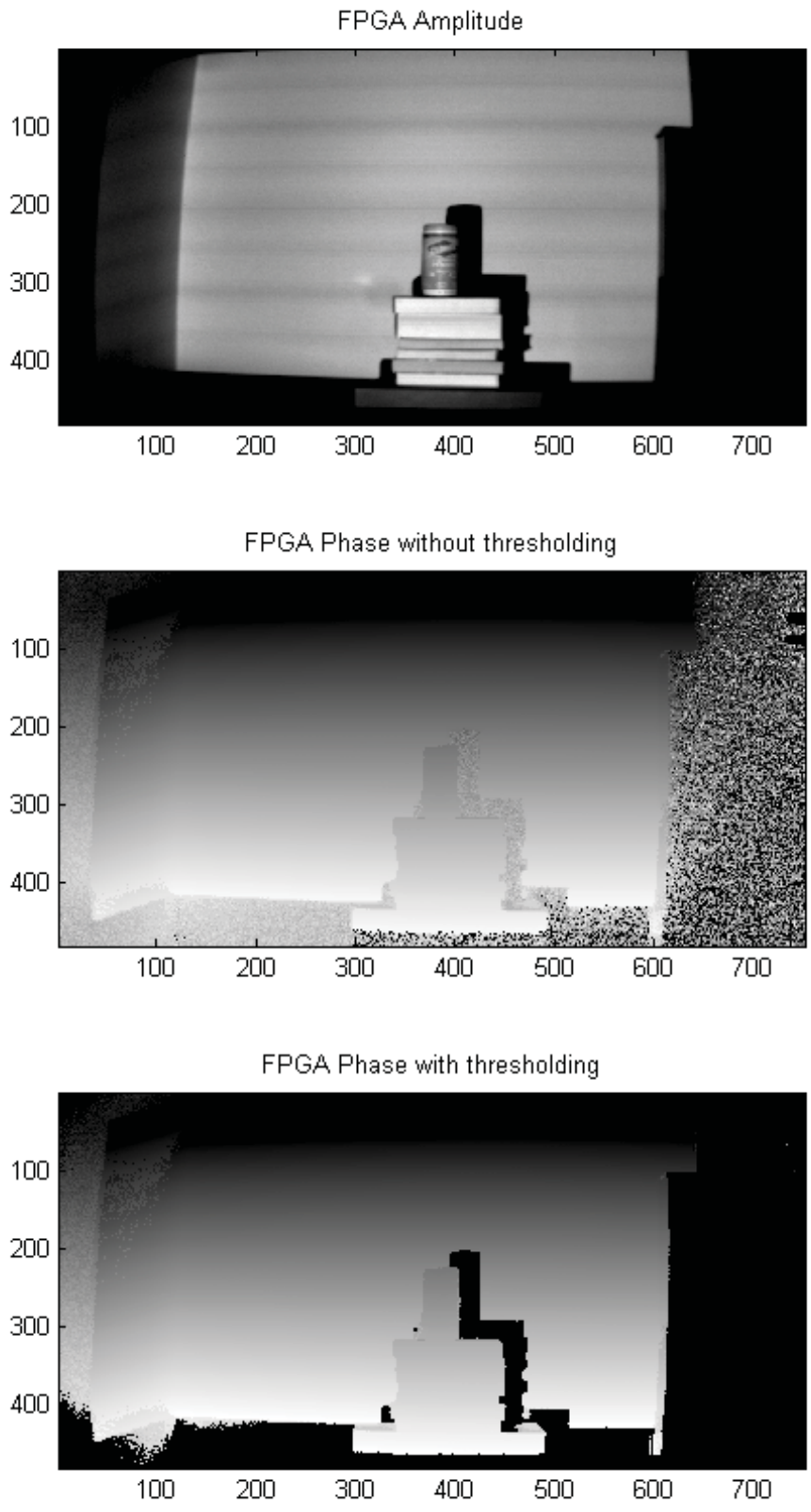


Figure 12.9: Magnitude and Phase with and without thresholding

12.4 Timing Analysis

12.4.1 Image Aquisition

The camera module operates at 27MHz with an 8-bit interface. The FPGA system clock is 100Mhz. A frame is 360960 bytes. If the camera runs at 60 Hz, a frame is written every 16.7ms to the FIFO. Because the SDRAM runs at 100MHz, faster than the data coming in, the FIFO is never filled. A frame takes up 352.5 blocks of memory so the last block of a frame is flushed from the FIFO after a frame is written so as to keep the frames block aligned in memory. The block aligned frames can be accessed with an offset of 353 rows. In total, to write 8 frames from the camera module, it takes about 134ms (7.5Hz).

12.4.2 Phase Calculation

The data from SDRAM first needs to be read into the Frame FIFOs, a page at a time. The latency between supplying a column address and receiving the data is specified as 3 clock cycles. To read one page, the FIFOs take 512·8 clock cycles. With a 100MHz clock, the time it takes to read all pages of 8 frames from SDRAM to a FIFO is 14.54ms or 69Hz. This is a memory bandwidth of 1.58Gb/s. There is also the effect of refresh and efficiency that is ignored.

On the phase calculation side, the first block of frame data for each of the eight frames are read, passed to the Phase Calculation pipeline. Because the memory has a data bus length of 16-bits, two pixels of data comes out every clock cycle. In this architecture, the pipeline is duplicated so that both pixels can be processed in

parallel. In total, the Phase Processing pipeline has a delay of 51 clock cycles. With 512 words per page, a page takes 563 clock cycles and there are 353 total pages. With a 100Mhz clock, the data for all 8 frames are processed in 1.987ms or 503Hz.

Once the frames are in SDRAM, all 8 frames are read and phase data is calculated in 16.5ms giving a rate of 61Hz per 8 frames. The Phase Calculation Pipeline is significantly faster than image acquisition allowing for a future improvement of continuous capture of frames.

12.4.3 Improving Performance

With the above timing analysis, this section will discuss what is necessary for a higher performing system.

Every second, the camera running at 60Hz with 360960 bytes per frame writes 21.6 MB to the RAM. As found in the section above, the memory bandwidth of the RAM is 1.58Gb/s or 198.7 MB/s, well above the camera bandwidth. It is theoretically possible to output phase images up to 61 frames per second (assuming the output USB transfer rate is fast enough). There are, however, several limitations in the current hardware and architecture. The first is that ideal phase image frame rate is calculated as the frames rate of the camera divided by the number of patterns. Therefore, a 60Hz frame rate camera with 8 patterns has a maximum phase image frame rate of 7.5 Hz. In the current design, frames are captured, then phase is calculated sequentially, then frames are captured again. It would be ideal to have the frames from the camera continuously written to RAM so that none are missed. This requires coordination of writes and reads to RAM. A simple way to do this would

be to write N frames to RAM and afterwards read the N frames for processing while keeping the data coming from the camera in a FIFO memory on the FPGA. With the current hardware, in the time that it takes to read 8 frames from memory for processing, 314 kB will be written to the FIFO which is more than is available on the FPGA. To overcome this, a higher memory bandwidth is necessary. The FPGA only has 72kB of block RAM space and the majority of it is already being used.

In comparison to the Opal Kelly XEM3010-1500 which is used in this project, the Opal Kelly XEM6010-LX45 cites a maximum bandwidth of 10Gb/s and 261kB of on-chip Block RAM. Given the maximum bandwidth and ignoring latency delays, reading 8 frames of data from RAM drops from 14.54ms to 2.31 ms. This requires only a 50kB FIFO. This board would allow for simultaneous capture and processing of the data.

Lastly, the output architecture would need to be rewritten. Currently, when the phase and magnitude data is calculated, it is written back to the RAM. This would need to be changed such that the resulting data is sent directly over the USB interface. Each 8-bit pixel has a 32-bit result for both phase and magnitude resulting in a total of 2.8 MB per result. For a USB 2.0 interface, Opal Kelly's test indicate tests up to 38 MB/s although this depends on a number of factors [21]. If 8-patterns are used at 60Hz resulting in phase and magnitude data every 7.5 Hz then a transfer rate of atleast 21 MB/s is needed. Opal Kelly also offers USB 3.0 boards with rates of 300 MB/s.

12.5 HDL Simulations

12.5.1 Fast Fourier Transform

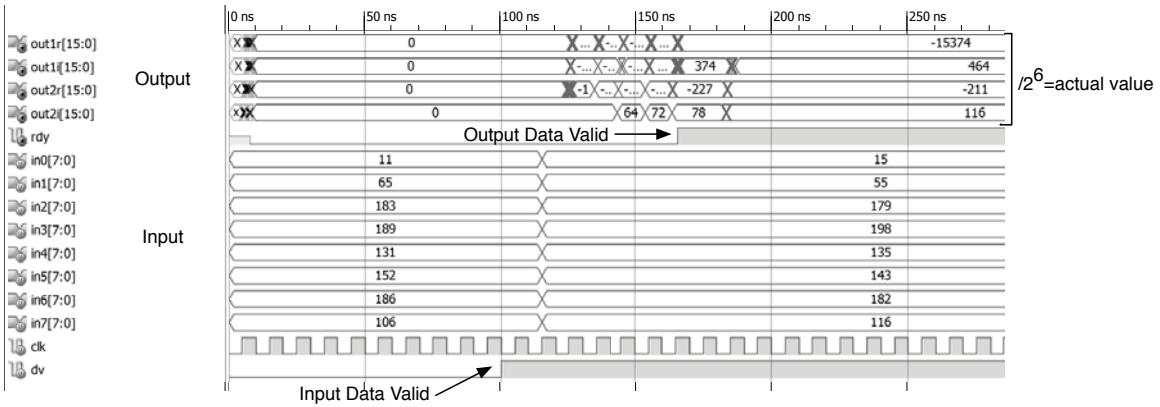


Figure 12.10: FFT Simulation

12.5.2 Phase Calculation Pipeline

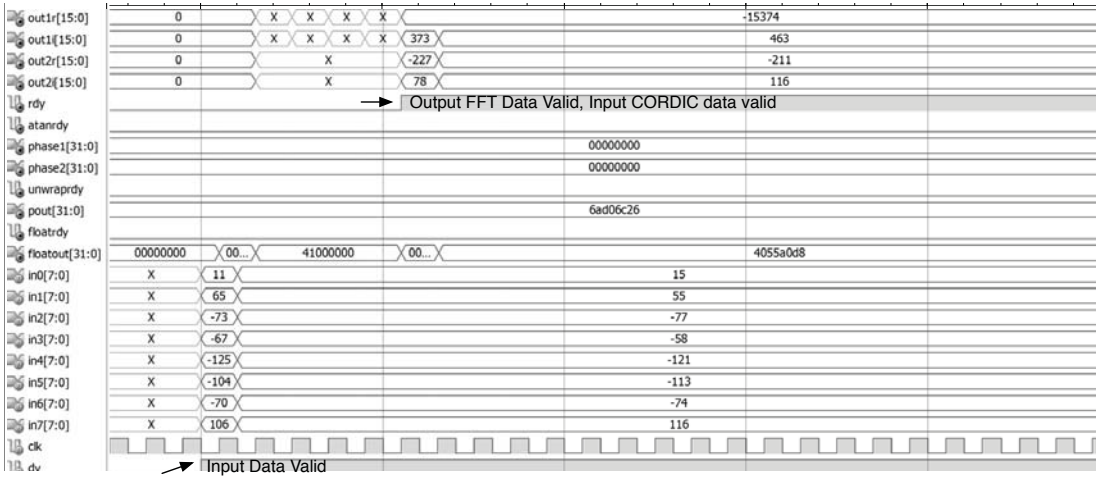


Figure 12.11: Phase Calculation Pipeline Part 1

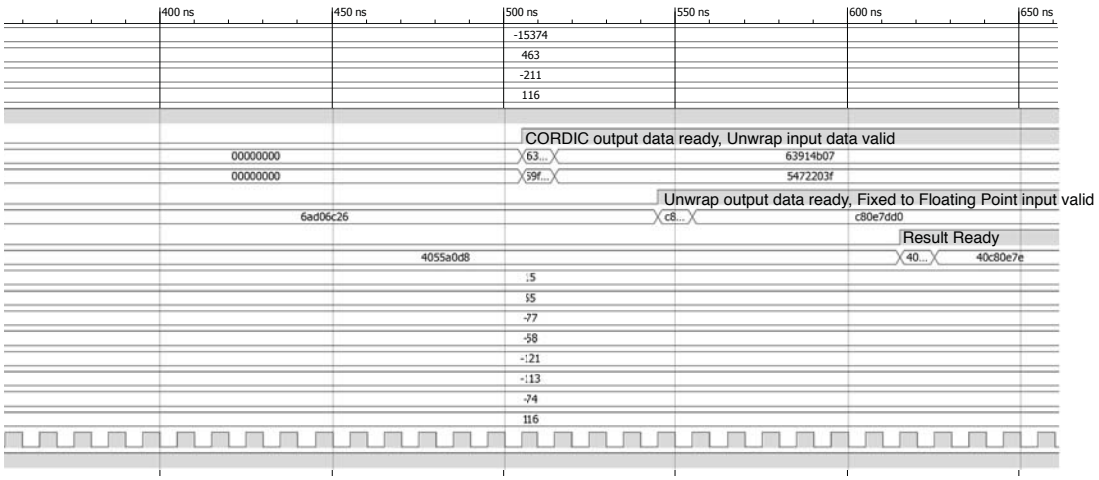


Figure 12.12: Phase Calculation Pipeline Part 2

Chapter 13

Conclusion

This thesis provided a detailed explanation of hardware and software techniques used to design, implement, and validate a computer architecture to implement a dual-frequency structured light illumination algorithm. The research accomplished the objectives outlined in Chapter 1. Output phase data from the system was tested and evaluated to be correct. Section 13.2 shows the FPGA device utilization of slices is 85%. The HDL code was designed with Xilinx ISE 13.4 and tested using the Xilinx ISim software.

13.1 Contributions

- The calculation of the accurate phase and magnitude data for the dual-frequency algorithm was designed and implemented in a Spartan-3 FPGA chip.
- Able to control the output of a light projector using the VGA standard and read in frames from a CMOS image sensor. The two are synchronized.
- A prototype FPGA-Camera-Projector SLI system was designed and built Design and build a prototype system controlled by a single FPGA. Performance was maximized with available hardware.

- The entire system was simulated, tested experimentally, and validated with results from the algorithm running on a PC.

13.2 Future Work

The performance of the system can be improved with architecture and hardware modifications and upgrades. The architecture can be modified from push-to-start camera operation to continuous video capture such that reading in frames from the camera and processing of frames in memory can happen simultaneously. The output architecture will also have to be modified so that the result is not written back to RAM but sent directly to a PC via USB interface.

Hardware can be improved with a camera with a higher frame rate, such as a LUPA-300, will help achieve the goal of making SLI more real-time. A higher performance FPGA will allow for lower latency times, a faster clock, and more logic resources. A RAM with a higher bandwidth will allow more throughput. Details of the required hardware specifications are discussed in section 12.4.3.

To achieve a smaller form factor, a board can be built that cuts the size in half by putting the FPGA prototype board on the back of the adapter board with the camera and VGA interface on the top.

Appendix

FPGA Resources Used

```
Selected Device : 3s1500fg320-4

 Number of Slices:                    11347  out of  13312    85%
 Number of Slice Flip Flops:          18284  out of  26624    68%
 Number of 4 input LUTs:              19976  out of  26624    75%
    Number used as logic:             19460
    Number used as Shift registers:     324
    Number used as RAMs:                192
 Number of IOs:                         121
 Number of bonded IOBs:                 119  out of    221    53%
    IOB Flip Flops:                      91
 Number of BRAMs:                        23  out of     32    71%
 Number of GCLKs:                         5  out of      8    62%
 Number of DCMs:                          1  out of      4    25%
```

Unpipelined 8 point FFT Timing Report

```
Speed Grade: -4

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: 25.750ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default path analysis
  Total number of paths / destination ports: 11612104 / 64
-------------------------------------------------------------------------
Delay:               25.750ns (Levels of Logic = 26)
  Source:            in1<0> (PAD)
  Destination:       out1i<15> (PAD)

  Data Path: in1<0> to out1i<15>
                                Gate     Net
```

```
Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
----------------------------------------  ------------
 IBUF:I->O             2   0.821   1.216  in1_0_IBUF (in1_0_IBUF)
 begin scope: 'x1mx5'
 LUT3:I0->O            1   0.551   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:S->O            1   0.500   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:CI->O           1   0.064   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:CI->O           1   0.064   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:CI->O           1   0.064   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:CI->O           1   0.064   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:CI->O           1   0.064   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:CI->O           1   0.064   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:CI->O           0   0.064   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 XORCY:CI->O          14   0.904   1.382  U0/xst_addsub/i_baseblox.i_baseblox_
 end scope: 'x1mx5'
 begin scope: 'compmult'
 begin scope: 'blk00000003'
 LUT2:I1->O            1   0.551   0.000  blk000000c1 (sig0000014b)
 MUXCY:S->O            0   0.500   0.000  blk000000c0 (sig00000148)
 XORCY:CI->O           9   0.904   1.124  blk000000bd (sig0000007f)
 MULT18X18:B17->P22    1   4.214   1.140  blk00000007 (sig0000008b)
 LUT2:I0->O            1   0.551   0.000  blk00000051 (sig000000fb)
 MUXCY:S->O            1   0.500   0.000  blk00000050 (sig000000f8)
 XORCY:CI->O           1   0.904   1.140  blk0000004c (pi(14))
 end scope: 'blk00000003'
 end scope: 'compmult'
 begin scope: 'out1radd'
 LUT2:I0->O            1   0.551   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 MUXCY:S->O            0   0.500   0.000  U0/xst_addsub/i_baseblox.i_baseblox_
 XORCY:CI->O           1   0.904   0.801  U0/xst_addsub/i_baseblox.i_baseblox_
 end scope: 'out1radd'
 OBUF:I->O                 5.644          out1r_15_OBUF (out1r<15>)
----------------------------------------
Total                     25.750ns (18.947ns logic, 6.803ns route)
                                   (73.6% logic, 26.4% route)
```

Pipelined FFT Timing Report

```
Timing Summary:
---------------
Speed Grade: -4
```

```
   Minimum period: 5.042ns (Maximum Frequency: 198.334MHz)
   Minimum input arrival time before clock: 4.643ns
   Maximum output required time after clock: 7.430ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 5.042ns (frequency: 198.334MHz)
  Total number of paths / destination ports: 4143 / 781
-------------------------------------------------------------------------
Delay:               5.042ns (Levels of Logic = 19)
  Source:            x1mx5_p_x7mx3mult/blk000000bd (FF)
  Destination:       x1mx5_p_x7mx3mult/blk00000032 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: x1mx5_p_x7mx3mult/blk000000bd to x1mx5_p_x7mx3mult/blk00000032
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDRE:C->Q             1   0.720   1.140  blk000000bd (sig00000019)
     LUT2:I0->O            1   0.551   0.000  blk00000030 (sig00000035)
     MUXCY:S->O            1   0.500   0.000  blk0000002f (sig0000001f)
     MUXCY:CI->O           1   0.064   0.000  blk0000002d (sig0000002c)
     MUXCY:CI->O           1   0.064   0.000  blk0000002b (sig0000002d)
     MUXCY:CI->O           1   0.064   0.000  blk00000029 (sig0000002e)
     MUXCY:CI->O           1   0.064   0.000  blk00000027 (sig0000002f)
     MUXCY:CI->O           1   0.064   0.000  blk00000024 (sig00000030)
     MUXCY:CI->O           1   0.064   0.000  blk00000021 (sig00000031)
     MUXCY:CI->O           1   0.064   0.000  blk0000001e (sig00000032)
     MUXCY:CI->O           1   0.064   0.000  blk0000001b (sig00000033)
     MUXCY:CI->O           1   0.064   0.000  blk00000018 (sig00000034)
     MUXCY:CI->O           1   0.064   0.000  blk00000015 (sig00000020)
     MUXCY:CI->O           1   0.064   0.000  blk00000012 (sig00000021)
     MUXCY:CI->O           1   0.064   0.000  blk0000000f (sig00000022)
     MUXCY:CI->O           1   0.064   0.000  blk0000000c (sig00000023)
     MUXCY:CI->O           1   0.064   0.000  blk0000000a (sig00000024)
     MUXCY:CI->O           1   0.064   0.000  blk00000008 (sig00000026)
     MUXCY:CI->O           1   0.064   0.000  blk00000006 (sig00000028)
     XORCY:CI->O           1   0.904   0.000  blk00000003 (sig00000059)
     FD:D                      0.203          blk00000032
```

```
    ----------------------------------------
    Total                      5.042ns (3.902ns logic, 1.140ns route)
                                       (77.4% logic, 22.6% route)
```

FPGA Pin Mappings

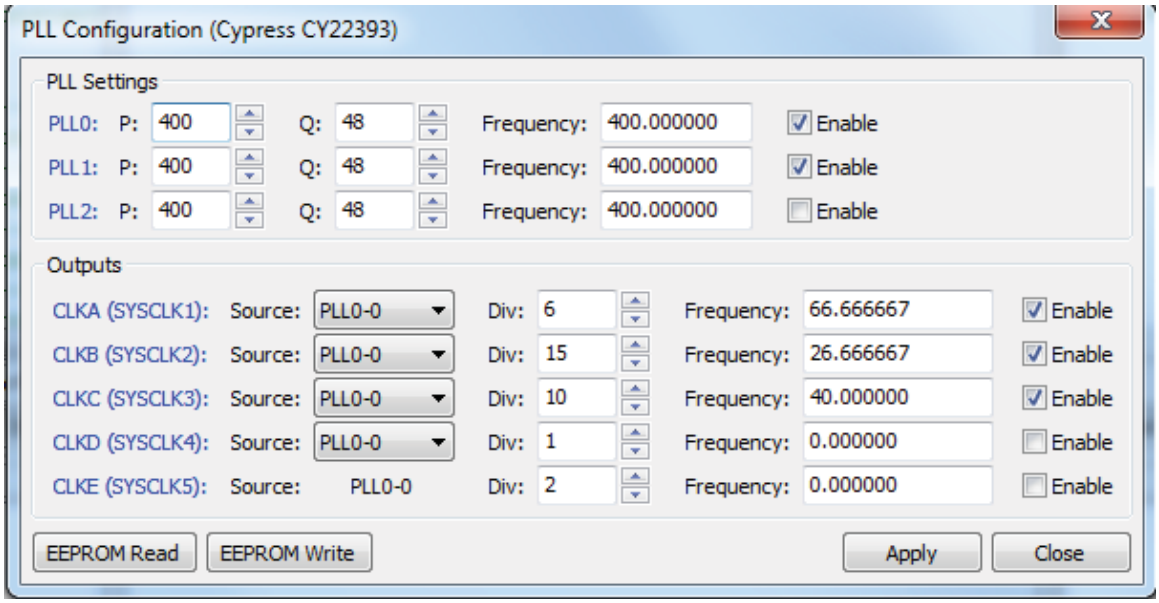
```
#------------------------------
# FrontPanel Host Interface pins
#------------------------------
NET "hi_in<0>" LOC = "N10";
NET "hi_in<1>" LOC = "V2";
NET "hi_in<2>" LOC = "V3";
NET "hi_in<3>" LOC = "V12";
NET "hi_in<4>" LOC = "R8";
NET "hi_in<5>" LOC = "T8";
NET "hi_in<6>" LOC = "V8";
NET "hi_in<7>" LOC = "V7";
NET "hi_out<0>" LOC = "V10";
NET "hi_out<1>" LOC = "V11";
NET "hi_inout<0>" LOC = "T7";
NET "hi_inout<1>" LOC = "R7";
NET "hi_inout<2>" LOC = "V9";
NET "hi_inout<3>" LOC = "U9";
NET "hi_inout<4>" LOC = "P11";
NET "hi_inout<5>" LOC = "N11";
NET "hi_inout<6>" LOC = "R12";
NET "hi_inout<7>" LOC = "T12";
NET "hi_inout<8>" LOC = "U6";
NET "hi_inout<9>" LOC = "V5";
NET "hi_inout<10>" LOC = "U5";
NET "hi_inout<11>" LOC = "V4";
NET "hi_inout<12>" LOC = "U4";
NET "hi_inout<13>" LOC = "T4";
NET "hi_inout<14>" LOC = "T5";
NET "hi_inout<15>" LOC = "R5";
NET "hi_muxsel"      LOC = "R9";
NET "i2c_sda"        LOC = "R13" | PULLUP;
NET "i2c_scl"        LOC = "U13" | PULLUP;

#System Clock
```

```
NET "clk1"          LOC = "N9";
NET "sdram_clk" TNM_NET="TNM_clk1";
TIMESPEC "TS_clk1" = PERIOD "TNM_clk1" 7.0 ns HIGH 50%;

#------------
# Peripherals
#------------
NET "sdram_cke" LOC = "F8";
NET "sdram_cas_n" LOC = "E11";
NET "sdram_ras_n" LOC = "D12";
NET "sdram_we_n" LOC = "E7";
NET "sdram_cs_n" LOC = "E8";
NET "sdram_ldqm" LOC = "D9";
NET "sdram_udqm" LOC = "A9";
NET "sdram_a<0>" LOC = "A15";
NET "sdram_a<1>" LOC = "A16";
NET "sdram_a<2>" LOC = "B15";
NET "sdram_a<3>" LOC = "B14";
NET "sdram_a<4>" LOC = "D11";
NET "sdram_a<5>" LOC = "B13";
NET "sdram_a<6>" LOC = "C11";
NET "sdram_a<7>" LOC = "A12";
NET "sdram_a<8>" LOC = "A11";
NET "sdram_a<9>" LOC = "D10";
NET "sdram_a<10>" LOC = "A17";
NET "sdram_a<11>" LOC = "B10";
NET "sdram_a<12>" LOC = "A10";
NET "sdram_ba<0>" LOC = "C12";
NET "sdram_ba<1>" LOC = "A14";
NET "sdram_d<0>" LOC = "C4";
NET "sdram_d<1>" LOC = "D5";
NET "sdram_d<2>" LOC = "C5";
NET "sdram_d<3>" LOC = "D6";
NET "sdram_d<4>" LOC = "D7";
NET "sdram_d<5>" LOC = "C7";
NET "sdram_d<6>" LOC = "C8";
NET "sdram_d<7>" LOC = "D8";
NET "sdram_d<8>" LOC = "B9";
NET "sdram_d<9>" LOC = "A8";
NET "sdram_d<10>" LOC = "A7";
NET "sdram_d<11>" LOC = "B6";
NET "sdram_d<12>" LOC = "A5";
NET "sdram_d<13>" LOC = "B5";
NET "sdram_d<14>" LOC = "A4";
NET "sdram_d<15>" LOC = "B4";
```

```
NET "led<0>"        LOC = "V14";
NET "led<1>"        LOC = "U14";
NET "led<2>"        LOC = "T14";
NET "led<3>"        LOC = "V15";
NET "led<4>"        LOC = "U15";
#NET "led<5>"        LOC = "V16";
#NET "led<6>"        LOC = "V17";
#NET "led<7>"        LOC = "U16";

NET "uclk" LOC = "P10"; #for projector
NET "uclk_out" LOC = "T16";

NET "VGAR<1>"  LOC = "D18" ;
NET "VGAR<0>"  LOC = "E15"  ;
NET "VGAR<3>"  LOC = "D17"  ;
NET "VGAR<2>"  LOC = "D16"  ;
NET "VGAR<5>"  LOC = "C18" ;
NET "VGAR<4>"  LOC = "C17"  ;
NET "VGAR<7>"  LOC = "B18" ;
NET "VGAR<6>"  LOC = "C16" ;

NET "VGAG<7>"  LOC = "G18" ;
NET "VGAG<6>"  LOC = "G16" ;
NET "VGAG<5>"  LOC = "F17" ;
NET "VGAG<4>"  LOC = "G15" ;
NET "VGAG<3>"  LOC = "E18" ;
NET "VGAG<2>"  LOC = "F15" ;
NET "VGAG<1>"  LOC = "E17";
NET "VGAG<0>"  LOC = "E16" ;

NET "VGAB<1>"  LOC = "H13" ;
NET "VGAB<0>"  LOC = "J14" ;
NET "VGAB<3>"  LOC = "H14" ;
NET "VGAB<2>"  LOC = "J15" ;
NET "VGAB<5>"  LOC = "J18" ;
NET "VGAB<4>"  LOC = "G14" ;
NET "VGAB<7>"  LOC = "J17";
NET "VGAB<6>"  LOC = "F14" ;

NET "VGAHS" LOC = "H16";
NET "VGAVS" LOC = "H18";
NET "VGACLK"  LOC = "H17";
NET "enableVideoOUT" LOC = "H15";
```

```
NET "din<2>" LOC = "H2";
NET "din<3>" LOC = "H4";
NET "din<4>" LOC = "G1";
NET "din<5>" LOC = "G3";
NET "din<6>" LOC = "F2";
NET "din<7>" LOC = "G4";
NET "din<8>" LOC = "E1";
NET "din<9>" LOC = "F4";
NET "line_valid" LOC = "F10";
NET "frame_valid" LOC = "E10";
NET "pix_clk" LOC = "E9"; //output from camera
NET "rst"        LOC = "D3";
NET "scl"      LOC = "E4" | PULLUP;
NET "sda"      LOC = "D1" | PULLUP;
NET "triggerCam" LOC = "C2";
NET "cam_clk" LOC = "P9"; //input to cam clock
NET "cam_clk_out" LOC = "F9";
```

Opal Kelly PLL Settings



Opal Kelly PLL Settings

Bibliography

[1] V. Valla, "Optimal pmp techniques for static and dynamic 3d data acquisition," Ph.D. dissertation, University of Kentucky, Lexington, KY, 2006. http://uknowledge.uky.edu/gradschool_diss/348/

[2] K. Liu *et al.*, "Dual-frequency pattern scheme for high-speed 3-D shape measurement," *Opt. Express*, vol. 18, no. 5, pp. 5229-5244, 2010.

[3] K. Liu, "Real time 3-d reconstruction by means of structured light illumination," Ph.D. dissertation, University of Kentucky, Lexington, KY, 2010. http://uknowledge.uky.edu/gradschool_diss/81/

[4] X. Su, G. von Bally, and D. Vukicevic, "Phase-stepping grating profilometry: utilization of intensity modulation analysis in complex objects evaluation," *Opt. Commun.*, vol. 98, pp. 141-150, 1993.

[5] Y. Wang, *et al.* "Multicamera phase measuring profilometry for accurate depth measurement," *Proc. of SPIE Sensors and Systems for Space Applications*, pp. 655509, May 2007.

[6] M. Hinner, "VGA Timings". http://martin.hinner.info/vga/vga.html/

[7] FPGA-Cam. "Interfacing low cost fpga papilio platform to the ov7670 sensor." https://code.google.com/p/fpga-cam/

[8] R. K. Gupta and Y. Zorian, "Introducing Core-Based System Design," *IEEE Design and Test of Computers*, vol. 14, no. 4, pp 15-25, Oct-Dec 1997.

[9] R. A. Andraka, "Survey of CORDIC algorithms for FPGA based computers," *FPGA '98 Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pp 191-200, Feb 1998.

[10] A. Sultan, "CORDIC: How Hand Calculators Calculate," *The College of Mathematics Journal*, vol. 40, no. 2, pp 87-92, March 2009.

[11] Xilinx, Inc. "Spartan-3 FPGA Family Data Sheet DS099," October 2012.

[12] C. Guan, G. Hassebrook and D.L. Lau. "Composite structured light pattern for three-dimensional video," *Optics Express*, vol. 11, no. 5, February 2003.

[13] Xilinx, Inc. "LogiCORE IP Fast Fourier Transform v7.1," March 2011.

[14] Xilinx, Inc. "LogiCORE IP CORDIC v4.0," March 2011.

[15] Xilinx, Inc. "LogiCORE IP Floating-Point Operator v5.0," March 2011. http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf

[16] Aptina Imaging. "1/3-Inch Wide-VGA CMOS Digital Image Sensor MT9V034," October 2008.

[17] Analog Devices. "CMOS, 330MHz High Speed Video DAC ADV7125," 2002.

[18] Micron Technology Inc. "SDR SDRAM 256Mb: x4, x8, x16," January 2010.

[19] OptoMotive Ltd. "Cameleon Baseboard." http://www.optomotive.com/products/cameleon-baseboard

[20] Opal Kelly Inc. "XEM3010 User's Manual," November 2009. http://assets00.opalkelly.com/library/XEM3010-UM.pdf

[21] Opal Kelly Inc. "Front Panel User's Manual," January 2013. http://assets00.opalkelly.com/library/FrontPanel-UM.pdf

[22] N. Dahnoun. "Fast Fourier Transform (FFT) (Theory and Implementation)," *Texas Instruments*, 2004.

[23] Kodak. "Shutter Operations for CCD and CMOS Image Sensors," December 2003. http://www.isgchips.com/pdf/Shutter_Operations_Kodak_App_Note.pdf

[24] dspGuru. "Fast Fourier Transform (FFT) FAQ." http://www.dspguru.com/dsp/faqs/fft

[25] dspGuru. "DSP Trick: Magnitude Estimator." http://www.dspguru.com/dsp/tricks/magnitude-estimator

[26] W. Wong. "Understanding FPGA Processor Interconnects," *Electronic Design*, July 2012. http://electronicdesign.com/fpgas/understanding-fpga-processor-interconnects

[27] A. Peter *et al.* "Efficient FPGA implementation of homodyne-based time-of-flight range imaging," *J Real-Time Image Proc*, vol. 7, no. 1, March 2010.

[28] B. Hong *et al.* "A Real-time Compact Structured-light based Range Sensing System," *Journal of Semiconductor Technology and Science*, vol. 12, no. 2, June 2012.

[29] S. Lee *et al.* "A Real-Time 3D IR Camera Based on Hierarchical Orthogonal Coding," *Proceedings of 2006 IEEE International Conference on Robotics and Automation*, May 2006.

[30] Y. Oike *et al.* "Real-Time and High-Resolution 3-D Imaging System Using Light-Section Method and Smart CMOS Sensor," *Proceedings of Sensors IEEE*, vol1, 2003

[31] S. Bellis and W. Marnane. "A CORDIC Arctangent FPGA Implementation for a High-Speed 3D-Camera System," *R. W Hartenstein and H. Grunbacher*, 2000.

[32] J. Weingarten, G. Gruener, and R. Siegwart. "A State-of-the-Art 3D Sensor for Robot Navigation," *Proceedings of 2004 IEEE International Conference on Intelligent Robots and Systems*, September 2004.

Vita

Brent Bondehagen was born in Louisville, Kentucky. He received his Bachelor of Science degree in Computer Engineering from the University of Kentucky in 2011. He worked as a technical intern at General Electric Energy in Louisville, Kentucky while an undergraduate student. He participated in an engineering exchange program with Nagoya University in Nagoya, Japan for a year.