ENHANCED RAW IMAGE CAPTURE AND DEBLURRING

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science
in
Electrical Engineering

by

Ruiwen Zhen

Robert L. Stevenson, Director

Graduate Program in Electrical Engineering
Notre Dame, Indiana
April 2013

© Copyright by

Ruiwen Zhen

2013

All Rights Reserved

ENHANCED RAW IMAGE CAPTURE AND DEBLURRING

Abstract

by

Ruiwen Zhen

Camera motion blur is a common problem in low-light imaging applications. Over the year numerous algorithms have been developed to deblur the image and provide a visually pleasing capture of the sensed scene. Many of these algorithms require that the blur kernel is known a priori which limits their application in many real-world situations. Recently, inertial sensors have been utilized in an attempt to estimate a blur kernel that can then be incorporated into deblurring algorithms to provide improved results. However, the effectiveness of these algorithms has been limited by lack of access to unprocessed raw image data obtained directly from the image sensor.

In this thesis, we build an digital imaging system for the acquisition of raw image data in conjunction with 3-axis acceleration data. From the acceleration data camera motion during image capture can be estimated to provide information about blurring of the sensed scene. This blur kernel is used in a maximum a posteriori estimation algorithm to deblur the raw image data. Experiments demonstrate that the accelerometer-based deblurring algorithm on raw image data can generate improved results.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I wish to thank my advisor Dr. Stevenson for his patience, guidance and understanding. His encouraging and constructive feedback helped me make continuous progress on my study and research. I am also indebted to Professors Ken Sauer and Patrick Flynn for their advice.

I would also like to extend my appreciation to many friends. Thank Han Chen for his some insight on FPGA-based design, thank Jonathan Simpkins for the help on PSF estimation and thank Terasic engineers for their immediate reply to my emails.

Finally, I am grateful to my parents and my grandparents. Their love provided me inspiration and was my driving force. I owe them everything and wish I could show them just how much I love and appreciate them. I would like to dedicate this work to my grandma who left us last year. I hope that this work makes her proud.

CHAPTER 1

INTRODUCTION

1.1 Background

Motion blur is a common problem in digital imaging that occurs when there is relative motion between the camera and the scene being captured. The degree of motion blur present in an image is a function of both the characteristics of the motion as well as the integration time of the sensor (mainly exposure time). Motion blur may be caused by camera motion or it may be caused by object motion within the scene [1]. Except for intentional blur, camera motion blur due to shaky hand shooting makes up most cases of the motion blur. It is particularly problematic in low-lighting imaging, which typically requires long integration time to acquire images with acceptable signal-to-noise levels. The commercial approach for reducing such camera motion blur is image stabilization (IS) which uses mechanical means to dampen camera motion by offsetting lens elements or translating the sensor [23]. However, such hardware remedy has its limitations, as it can only compensate for motion of a very small extent and speed [37]. In recent years, deblurring the image offline as another choice has received increasing attention.

The motion blur process usually can be modeled as the convolution of a sharp image and a blur kernel (or Point Spread Function PSF). Therefore, the image deblurring problem consists of two tightly coupled sub-problems: PSF estimation and non-blind image deconvolution. Non-blind deconvolution problem has been addressed since 1972 Richardson, and many classical algorithms have appeared, such as

Richardson-Lucy algorithm [33][29], Wiener filter and constrained least squares filter. In order to suppress ringing artifacts and restore image features effectively, various image priors and regularization schemes have been proposed, such as Gaussian prior, sparse image prior [26], and natural image statistics [36]. However, these approaches can only deal with a small amount of image noise that can be well approximated by a Gaussian distribution. For handling impulsive noise such as salt-and-pepper noise, Bar et al. [11] adopted a Laplacian distribution for noise and Cho et al. [13] detected outliers by marginalization and masked out them from the deconvolution process.

Non-blind deconvolution algorithms can effectively work if the PSF is already given. However, this is not true in many real situations. Thus, to use the same non-blind deconvolution algorithm, PSF estimation is a key step in obtaining the optimal sharp image. Generally, the methods of PSF estimation can be divided into two categories. The first category is blind estimation, which estimates the blur kernel directly from the acquired image. Fergus et al. [16] recovered a blur kernel by using a natural image prior on image gradients in a variational Bayes framework. Shan et al. [36] incorporated spatial parameters to enforce natural image statistics using a local ringing suppression step. Joshi et al. [24] detected edges and predicted the underlying sharp edges that created the blurred observations. Each pair of predicted and blurred edges can give information about the PSF. Levin et al. [27] gave a nice overview of the blind deblurring techniques mentioned above. Common to all of them is that they assume spatial invariance for the blur. Levin et al. showed that spatial invariance is often violated, as it is only valid in limited cases of camera motion. For example, camera rotation can lead to spatially-varying blur kernels. Whyte et al. [43] considered rotations about three axes and described deblurring using three basis vectors. Gupta et al. [20] adopted a similar approach, but replaced rotations about x and y axes by translations. The work in the first category sometimes can achieve very good results, but their effectiveness is limited to specific situations.

Another category of PSF estimation tries to overcome the drawbacks that exist in the first category by using information about the camera motion from other sources. One possibility is to acquire a pair of images: one correctly exposed but blurred and one underexposed (noisy) but sharp image [46][1]. Another possibility is to attach an auxiliary high-speed camera of low resolution to estimate the PSF, such as Ben-Ezra and Nayar [31] and Wu et al. [44]. Though these possibilities can estimate PSF accurately, they are complex to realize and cost much due to the necessity of another camera. Instead, Joshi et al. [23] integrated inexpensive gyroscopes and accelerometers with a camera to estimate a blur function from the camera's acceleration and angular velocity during an exposure. Their method is completely automatic, handles per-pixel, spatially-varying blur and outperforms the current leading image-based methods. However, they assumed the depth is constant over the scene and used a naive assumption on the direction of the gravity. Moreover, they defined the bound for the accelerometer drift on the order of a few millimeters during a long exposure (the longest is 0.5s). If longer exposure time is used, the drift error is larger and their method requires much more computing. Bae et al. [10] proposed solutions to the problems of Joshi's work. They used a depth sensor (Microsoft Kinect) to estimate the scene depth map besides using inertial sensors to measure the camera motion. In addition, they regarded the displacement due to invalid gravity estimation and the drift error as a single variable per axis and estimated them based on a reference kernel. Šindelář et al. [37] simplified the work of Joshi et al. and have realized this deblurring algorithm on smart phones.

From the recent image deblurring methods reviewed in this section, it is obvious that the joint hardware and software approach, especially utilizing inexpensive inertial sensors for PSF estimation, can yield better results with lower cost. The work proposed in this thesis is similar to this approach.

1.2 Motivation

All of the example images in Joshi's, Bae's and Šindelář's work either are synthesized or come from commercial cameras, such as a single-lens reflex camera or phone camera. The image sensor in the majority of them is a Bayer Pattern sensor. It is a CMOS or CCD sensor covered by Bayer color filter array, which lets the sensor detect colors. Each pixel in the array responds to either red, green or blue light and there are 2 green sensitive pixels for each red and blue pixel. There are more green pixels because the eye is more sensitive to green [9]. The pattern of the color filter array is shown in Figure 1.1.

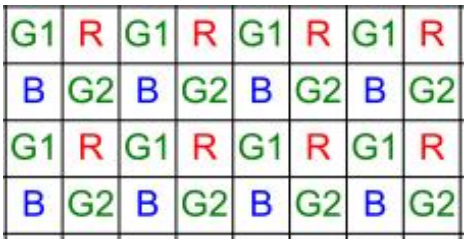


Figure 1.1. Bayer Pixel Color Pattern [39]

Raw image data is the output from the original red, green and blue sensitive pixels of the image sensor after passing through A/D converter. Depending on the camera's circuitry either 12 or 14 bits of data are recorded, corresponding to 4096 brightness levels or 16384 brightness levels. Many commercial cameras can save this raw data along with a header file which contains all of the camera settings, including sharpening level, contrast and saturation settings, color temperature/white balance and so on. The image is not changed by these settings, and they are simply tagged onto the

raw image data [32]. Even more common is images saved in JPEG format. The raw data needs to go through steps including Raw-to-RGB conversion, modification by camera settings and compression (Figure 1.2) [9]. The raw data holds exactly what the image sensor records in maximum resolution and thus PSF models the blurring process more accurately considering that the PSF describes the redistribution of light from a point source across a local neighborhood.

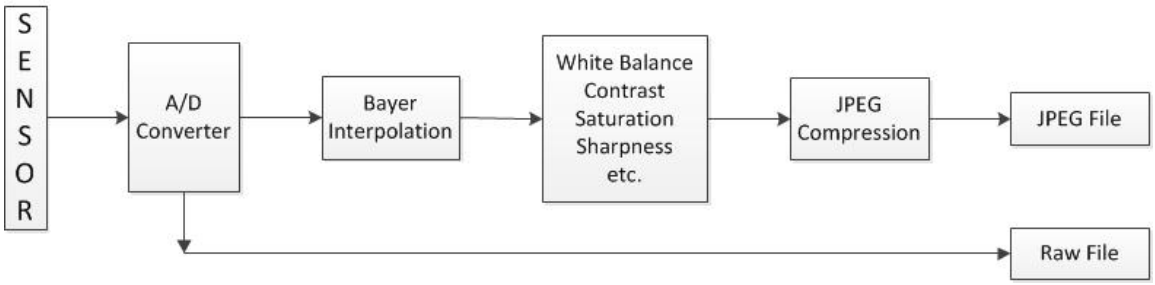


Figure 1.2. Conversion from Raw Data to JPEG Format

Another reason for desire to obtain raw image data is the electronic shutter mechanism. This mechanism takes images by scanning the rows of the sensor twice in the scan order. On the first scan, each row is released from reset, starting the exposure. On the second scan, the row is sampled, processed and returned to the reset scan. The exposure for any row is therefore the time between the first and second scans. Each row is exposed for the same duration, but at slightly different point in time. The timing for the shutter mechanism is shown in Figure 1.3. So if inertial sensor data is recorded during an exposure, the motion of each row in the raw data can be exactly calculated by shifting those measurements.

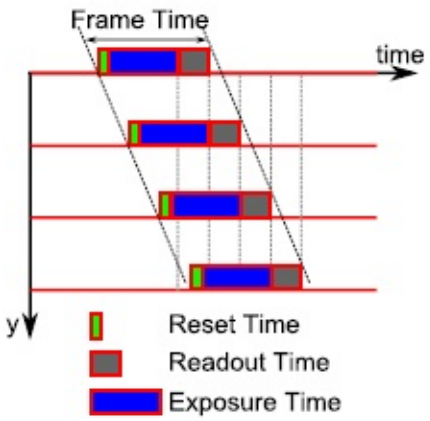


Figure 1.3. Timing for Electronic Shutter Mechanism [19]

Therefore, both PSF estimation from inertial sensor data and non-blind deconvolution algorithms can perform better on raw image data than other common format images. In fact, the desire to use raw data for deblurring frequently appears in papers. Joshi et al. [24] hoped to access raw camera images instead of using the demosaicked color values to compute more accurate per-channel PSFs. Šindelář et al. [37] owed the imperfect quality of deblurring result to the lack of control over camera hardware, such as no manual exposure settings, no access to raw data and so on. However, it is difficult to access raw image data because the raw files have been encoded through the file headers by camera manufacturers. Every manufacturer provides software for decoding and processing their camera's raw files [32]. Besides the functions provided in the software, other algorithms or functions cannot be applied on raw image data.

1.3 Problem Statement and Approach

This thesis aims at building an imaging system to access raw image data and inertial sensor data so as to remove the blur caused by camera motion. This system

will be developed on the Altera FPGA board VEEK-MT, in which the TRDB-D5M models the CMOS image sensor, the LCD screen displays the taken image and the accelerometer ADXL345 records the 3-axis acceleration data. Then post-capture processing will be performed to estimate the PSF and the latent sharp image. The final result of the method proposed in this thesis should be able to remove camera motion blurs existed in the raw images with little ringing artifacts. As the first step of this research, some assumptions are given to simplify the deblurring process.

1.4 Contribution

The most significant original work in this thesis is building an FPGA-based imaging system which can capture raw image data and acceleration data at the same time. In addition, the image prior deconvolution algorithms are used to test the validity of the captured raw image data and PSF estimation given acceleration data. The final deblurred result has proven that our method is effective and can perform better than advanced blind deblurring algorithms.

1.5 Organization

Chapter 2 will introduce Altera development platform VEEK-MT and FPGA-based development flow which consists of hardware development flow and software development flow. In addition, corresponding development tools, Quartus II and Nios II SBT GUI, are also introduced.

Chapter 3 will discuss the hardware frame developed in Quartus II. The design and function of each module is described in detail. This hardware frame basically completes transferring the raw image data from the CMOS image sensor to SDRAM and provides a hardware platform for the software development. Chapter 4 will illustrate the software frame developed in Nios II SBT GUI. The implementation of

each software layer is given. This software frame realizes obtaining the acceleration data and raw image data at the same time.

Chapter 5 will model the motion blur process, calculate camera motion curve based on acceleration data, and estimate the PSF. The adopted non-blind deblurring algorithms are described as well. Chapter 6 will use a raw image obtained from the hardware platform as an example to show the result of every step given in Chapter 5. The final deblurred image is also compared with that of others' deblurring algorithms.

Chapter 7 will draw conclusions about our work in this thesis and address future work.

CHAPTER 2

DEVELOPMENT PLATFORM AND DEVELOPMENT FLOW

This chapter introduces the Video and Embedded Evaluation Kit - Multi-touch (VEEK-MT) FPGA development board and examines the embedded SOPC system development flow.

2.1 VEEK-MT FPGA Development Board

The Video and Embedded Evaluation Kit - Multi-touch (VEEK-MT) is a comprehensive design environment with everything embedded that developers need to create multimedia applications. The reference designs in the supplied CD-ROM can be used to quickly architect, develop and build complex embedded SOPC systems. This fully integrated kit adopts a DE2-115 development board targeting the Cyclone IV E FPGA and combines a capacitive LCD color touch screen, a 5-megapixel digital image sensor, an ambient light sensor as well as a 3-axis accelerometer through HSMC port of the DE2-115 (Figure 2.1). The key features of each module are listed below for reference. More details are provided in the related manuals [40][41][39][8][30].

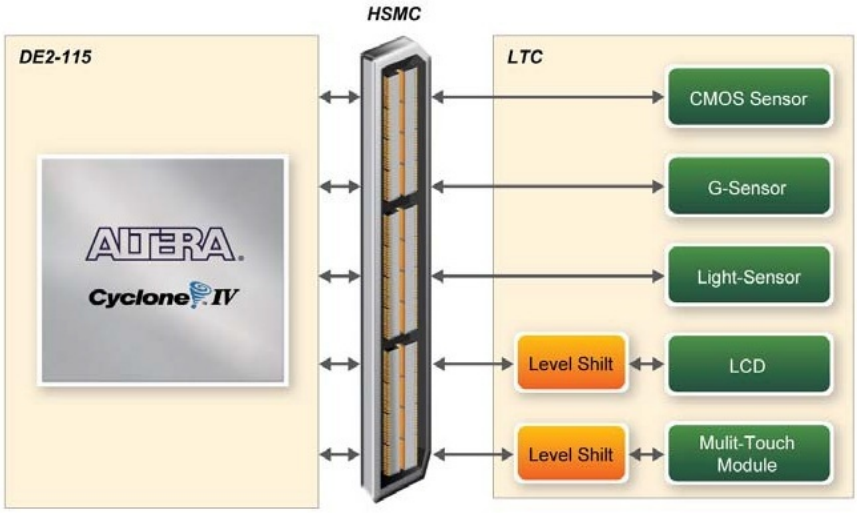


Figure 2.1. Block Diagram of VEEK-MT [40]

2.1.1 DE2-115 Development Board

The DE2-115 board has many features that allow users to implement a wide range of designed circuits. The following hardware is provided on the DE2-115 board:

- Cyclone IV EP4CE115 FPGA
- USB Blaster for both JTAG and Active Serial (AS) programming
- Memory Devices: 128MB SDRAM, 2MB SRAM, 8MB Flash with 8-bit mode
- Switches and Indicators: 18 switches and 4 push-bottons, 18 red and 9 green LEDs, 8 seven-segment displays
- Three 50MHz oscillator clock inputs
- SD Card Socket: support SPI and 4-bit SD mode
- High Speed Mezzanine Card (HSMC): support voltage levels 3.3/2.5/1.8/1.5V
- Other features: CODEC, TV decoder, Ethernet Ports, USB, RS232 Port etc.

2.1.2 LCD Touch Screen

This module is equipped with an 7-inch amorphous-TFT-LCD panel (Thin Film Transistor Liquid Crystal Display) and a Touch Controller. The panel offers resolution of 800 × 480 and supports 24-bit parallel RGB interface (R,G,B sub-pixels are arranged in vertical stripes). As for multi-touch function, the Touch Controller can convert X/Y touch coordinates to corresponding digital data and output these data through a serial port interface I^2C.

2.1.3 Digital Image Sensor

TRDB_D5M is a 5-megapixel digital image sensor that provides an active imaging array of 2592 × 1944. It features low-noise CMOS imaging technology that achieves CCD image quality. In addition, it incorporates sophisticated camera functions and parameters which can be programmed by the user through a simple two-wire serial interface I^2C.

2.1.4 Digital Accelerometer

ADXL345 is a small, thin, ultralow power, 3-axis accelerometer with high resolution (13-bit) measurement at up to $\pm 16g$ (maintaining $3.9mg/LSB$ sensitivity in all g ranges). Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I^2C digital interface. This accelerometer measures the static acceleration of gravity as well as dynamic acceleration resulting from motion or shock. In addition, several special sensing functions are provided, such as activity/inactivity monitoring, single tap/double tap detection and free-fall detection. These functions can be mapped individually to either of two interrupt output pins. Moreover, a 32-level first in, first out (FIFO) buffer can be used to store data to minimize host processor activity and lower overall system power consumption.

2.2 Development Flow and Tools

The embedded SOPC system design consists of hardware development and software development, which are implemented by Quartus II Design Software and Nios II Embedded Design Suite respectively. Note that for VEEK-MT FPGA device (Cyclone IV E), 10.0 or later versions are necessary [40]. We select version 11.0 as our development platforms. The basic development flow (mainly for our design) is shown in Figure 2.2.

2.2.1 Hardware Development Flow

The Altera Quartus II Design Software provides a complete, multi-platform design environment that easily adapts to specific hardware development. It includes solutions for all phases of FPGA and CPLD design through the easy-to-use graphical user interface [2]. The left branch in Figure 2.2 represents the Quartus II-based hardware design flow. A detailed description is given below:

1. Design Entry:

We use Hardware Description Language (Verilog HDL), MegaWizard Plug-In Manager and SOPC Builder to build system-level design. Megafunctions are parameterizable functional blocks. The MegaWizard Plug-In Manager allows one to create custom megafunctions which can be instantiated in design files. The SOPC Builder can help implement customized Nios II system. In this software package, one can configure the Nios II soft processor, select the desired standard I/O cores, and incorporate the user designed I/O peripherals. Then the SOPC Builder generates the HDL code for the customized Nios II system and also generates the .sopcinfo file that contains system configuration information. This code is combined with other HDL code to form the top-level HDL description of the complete hardware. In addition, initial design constraints should be specified through Assignment editor and Settings dialog box.

2. Compilation:

The compilation process consists of analysis and synthesis, fitting, timing analysis and assembling. This process realizes functions such as checking the syntax of HDL code, transforming HDL constructs to gate-level components, deriving the layout inside FPGA chip, performing timing analysis and finally producing programming files (.sof for JTAG programming, .pof for AS programming).

3. Programming and Debugging:

In this step, the configuration file is downloaded into the target device. The SignalTap II Logic Analyzer can be used for debugging. This logic analyzer captures real-time signal behavior and supervises the interactions between hardware and software in the system design. The Quartus II software allows one to select signals to capture, when signal capture starts, and how many data samples to capture. After stopping the analysis, the acquired data is displayed as waveforms [6].

2.2.2 Software Development Flow

The right branch in Figure 2.2 represents the software design flow. The Altera Nios II Embedded Design Suite (EDS) is used for software development. It can be accessed by the SBT (Software Build Tools) command-line interface, by the SBT GUI, or by the IDE GUI. The SBT GUI is selected to develop the software in this design. The SBT GUI is based on the Eclipse open development environment and supports creating, modifying, building, running and debugging C programs targeted for a Nios II system. A Nios II software project contains two major parts: user applications and BSP (Board Support Package). The former is the user's programs which include user I/O drivers and user high-level functions, and the latter is the support codes for a specific Nios II configuration. Note that the BSP is based on the information from the .sopcinfo file generated by SOPC Builder. The code from the two parts are complied and linked into a single software image (.elf) and loaded into

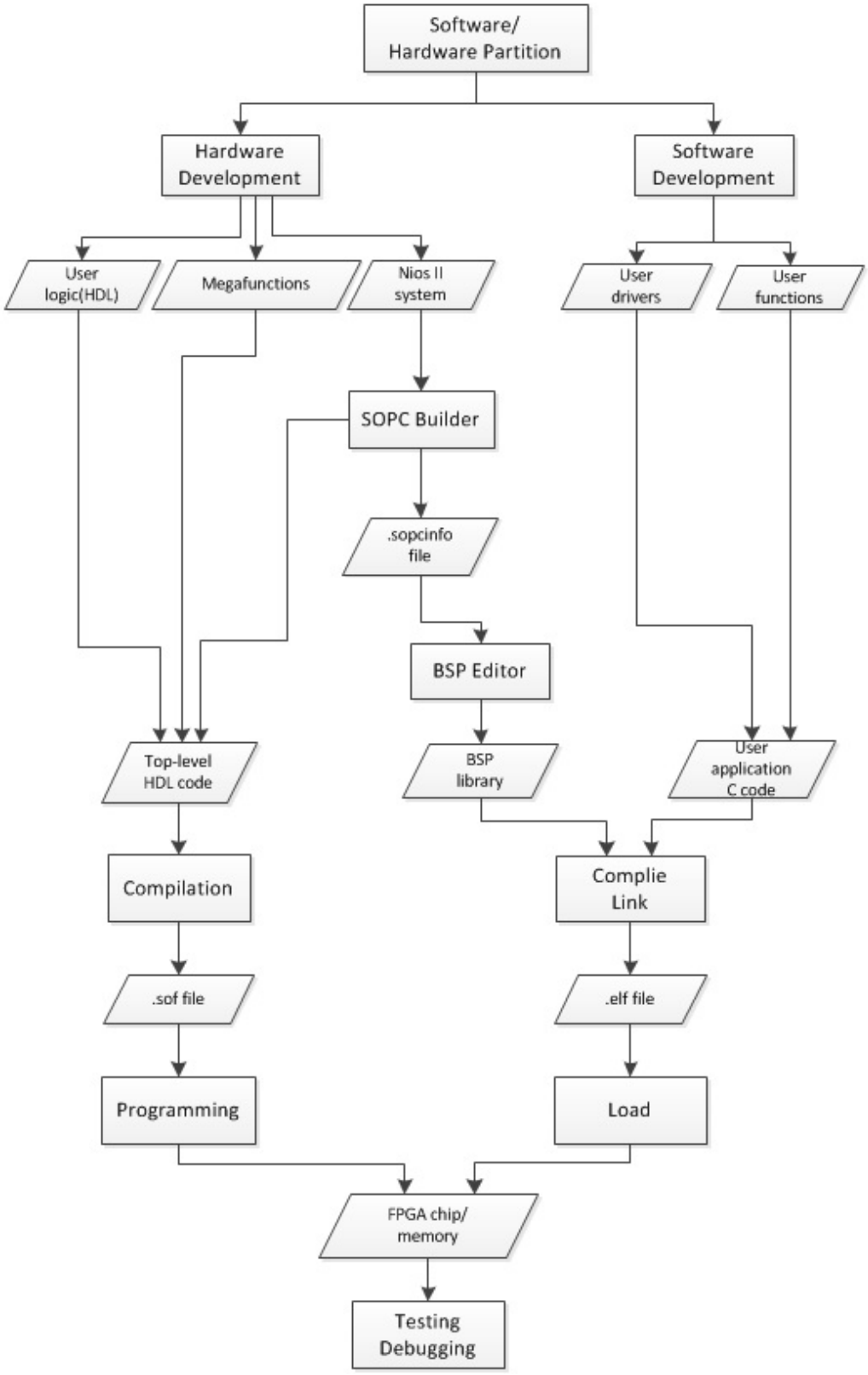


Figure 2.2. Development Flow of Embedded SOPC System

Nios II system's main memory [14].

2.3 Conclusion

The all-in-one embedded solution offered on the VEEK-MT provides an ideal hardware platform to collect image and motion data. The CMOS image sensor is utilized to capture the Bayer Pattern raw data, and the 3-axis accelerometer samples acceleration data for estimating motion during image exposure time. The LCD screen displays what the image sensor captures so as to verify the raw image data is correct. Based on this idea, the corresponding hardware and software can be developed following the method decribed in Section 2.2.

CHAPTER 3

HARDWARE DEVELOPMENT

The hardware frame of the embedded SOPC system is developed in Quartus II 11.0. It can be partitioned into two parts: user logic (Verilog HDL) and customized Nios II system (SOPC Builder). In each part, the top-down approach is used to divide it into functional modules for implementation. Figure 3.1 shows the complete block diagram of the hardware frame. The design and function of each module is described in detail below.

3.1 User Logic

The red block[1] in Figure 3.1 contains logic modules realized by Verilog HDL and Megafunctions. It is separated from the Nios II system and does not interact directly with the soft processor. The working principle of this part is similar to a digital camera. After the CMOS Image Sensor is triggered by signal from Signal Controller (belonging to the Nios II system), it starts to capture and output image data streams and the CMOS Sensor Data Capture module extracts the valid pixel data streams based on the synchronous signals from the CMOS Image Sensor. The data streams are generated in a Bayer Pattern. Then it is sent to two modules, a Bayer Pattern Data Buffer module which does nothing to the data but buffer, and a RAW2RGB module which converts the Bayer Pattern data to RGB data streams. After that, the Multi-Port SDRAM Controller acquires and writes the Bayer Patter data and RGB

[1]It is based on the camera reference design in the supplied CD-ROM of VEEK-MT.

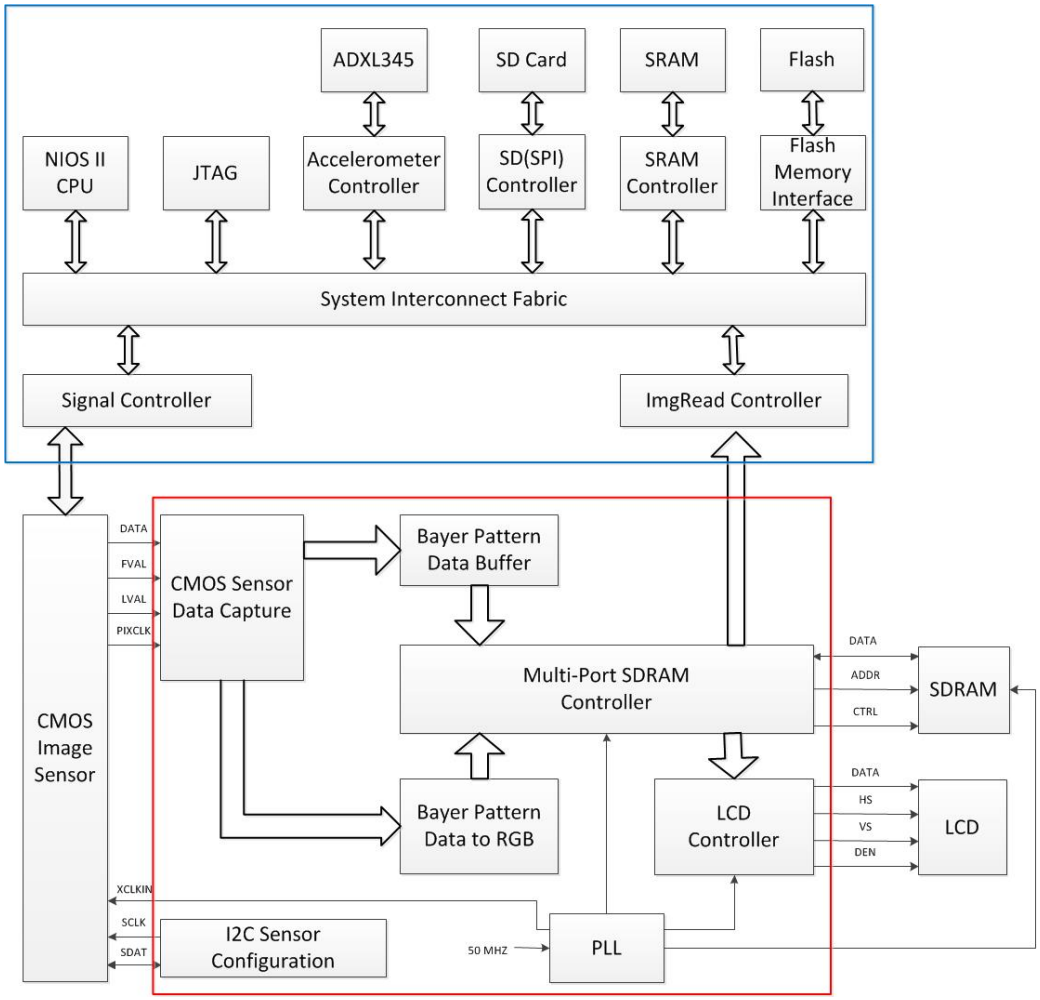


Figure 3.1. Block Diagram of the Hardware Frame

data streams into SDRAM performing as a frame buffer. Finally, the LCD Controller and ImgRead Controller (belonging to the Nios II system) separately fetch the RGB data and Bayer Pattern data from the buffer. The RGB data is displayed on the LCD panel continuously and the Bayer Pattern data will be sent to the Nios II processor. Here the I^2C Sensor Configuration module sets the registers of the CMOS Image Sensor via I^2C interface and the PLL module provides related clock signals. The paths for data streams are marked on Figure 3.2.

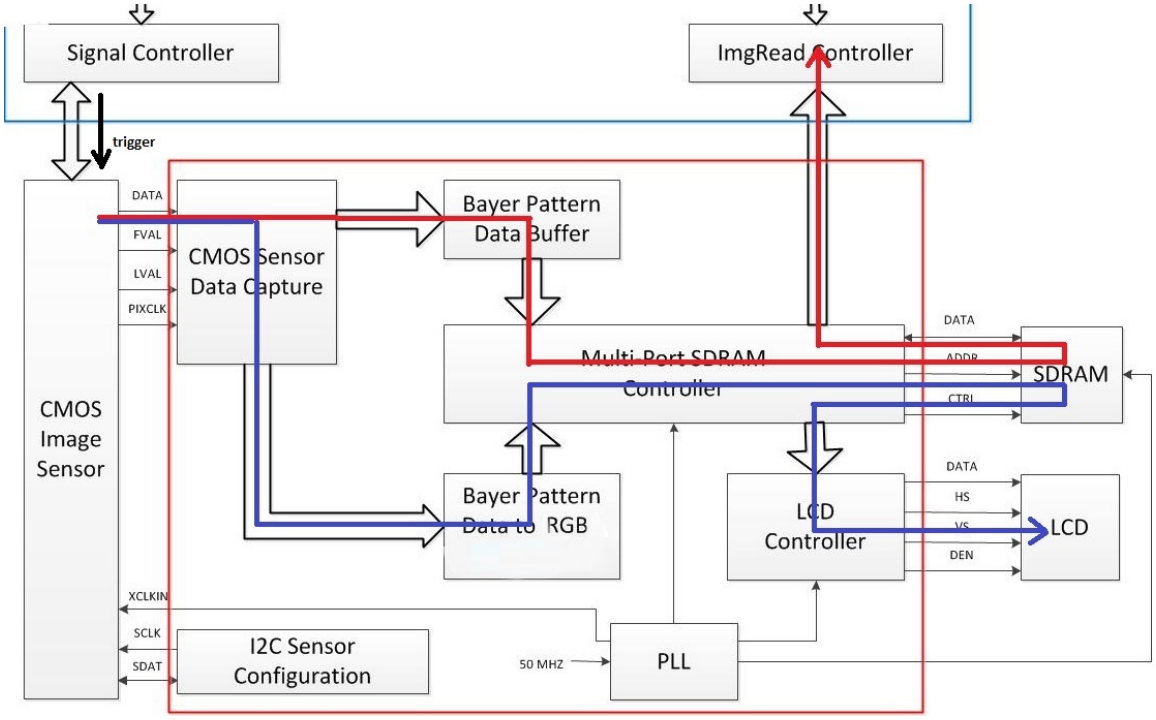


Figure 3.2. Paths for Data Streams

3.1.1 PLL Module

The goal of this module is to generate stable clock frequencies for other modules. It adopts the ALTPLL (Phase-Locked Loop) MegaWizard interface provided by Altera to specify the PLL circuitry in the FPGA device. This is because compared to adding another oscillator source, this method can easily convert 50MHZ oscillator input to other stable frequencies with low cost. The ALTPLL MegaWizard interface is configured by the MegaWizard Plug-In Manager. Figure 3.3 shows the PLL instance in our design.

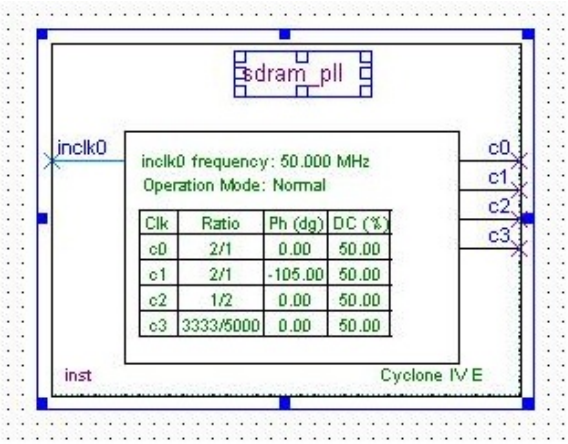


Figure 3.3. PLL Instance

In the figure above, *inclk0* is connected to one of the three 50MHZ oscillator inputs. There are four output clock frequencies: *c0* (100MHZ) is the frequency used by the SDRAM Controller; *c1* (100MHZ, -3ns phase shift) is the frequency used for the SDRAM; *c2* (25MHZ) is connected to the input clock pin *XCLKIN* of the CMOS Image Sensor; *c3* (33MHZ) is the clock frequency for both the LCD Controller and

the LCD itself. Note that the SDRAM documentation suggests to use -3 ns shift for the SDRAM clock (i.e., the rising edge of the SDRAM clock is ahead of the rising edge of the controller by 3ns). The other features configured in the MegaWizard interface include device speed, PLL type and operation mode.

3.1.2 I^2C Sensor Configuration Module

The 5-megapixel CMOS Image Sensor supports a simple two-wire serial interface based on the I^2C bus protocol. This module utilizes the serial interface to set register values of the CMOS Image Sensor. The I^2C bus consists of two bidirectional lines, *sda* (serial data) and *scl* (serial clock). Each device on the I^2C bus has a unique address and can operate as either a transmitter or receiver. One device on the bus functions as the master and other devices function as slaves. The master generates the clock on the *scl* signal and also initiates and terminates the data transfer. The master and the designated slave place data on or retrieve data from the *sda* signal [14]. In this module, the bus is connected between the FPGA chip, which acts as a master, and the CMOS Image Sensor[2], which acts as a slave. And the slave address is 7'b1011101.

The basic timing diagram of a typical data transfer is shown in Figure 3.4. Both lines are high when the bus is idle. The master initiates a transfer by creating the start condition, in which *sda* changes from high to low while the *scl* is high. It then generates the clock signal on *scl* and sends the slave's 7-bit address plus a direction bit (write is 0 and read is 1) on *sda*. Next based on the type of transaction, either the master or the slave places data on *sda*. The data must be stable when *scl* is high and thus the change can only occur when *scl* is low. The transfer is done on a byte-by-byte basis. Each byte is followed by an acknowledge bit in the ninth clock cycle. The number of bytes in a transfer is unrestricted. After completion, the master

[2]The I^2C bus is connected to the *SDATA* and *SCLK* pins of the image sensor.

terminates a transfer by creating the stop condition, in which *sda* changes from low to high while *scl* is high. After the transaction is done, the devices on the bus must wait for a small turnaround time before initiating another transaction [14].

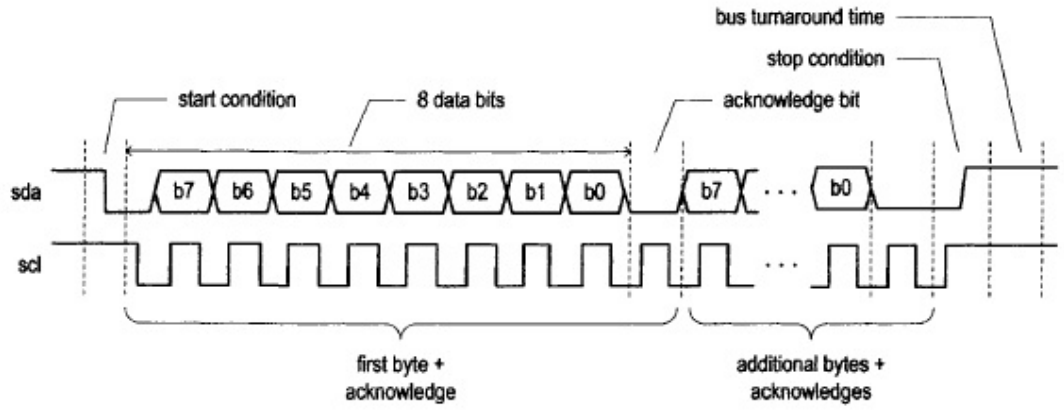


Figure 3.4. Basic Timing Diagram of I^{2}C Data Transfer [14]

In the I^{2}C Sensor Configuration module, only a write operation is performed. In addition to the slave address plus direction bit, configuring the internal registers of the sensor requires sending an 8-bit register ID (i.e., register address) and 16-bit register data value. So the register write operation is composed of several parts: start, slave address plus direction bit, register address, one byte of data, one byte of data and stop. Figure 3.5 gives the complete sequence of writing an internal register. As for reading an internal register, there is slight difference and this will be discussed in the software frame part[3].

[3]In the software programs of ADXL345, both writing and reading internal registers are needed.

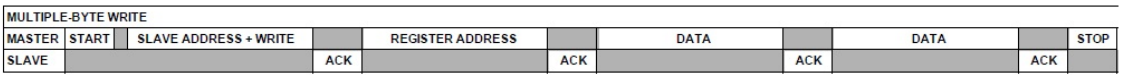


Figure 3.5. Complete Sequence of Writing an Internal Register [8]

This module is implemented through an FSMD (Finite State Machine and Datapath) which generates proper *scl* and *sda* signals according to the timing sequence in Figure 3.5. Here we list the important register values set by this module in Table 3.1.

From the register values, we can obtain some results: (1) The image starts from coordinates (530,480) and the image size is 800×480. Note that the Column_Start and Row_Start must be set to an even number and the Column_Size and Row_Size must be set to odd numbers. (2) The pixel clock is generated through PLL in the CMOS Image Sensor. Its value can be determined by the formula: $PIXCLK = XCLKIN \times$ PLL_m_Factor)/((PLL_n_Divider+1) × (PLL_p1_Divider+1)) = 25MHZ. (3) Read Mode 2 sets row mirror for the objective of a better visual effect. (4) Other important register values will be illustrated in other modules.

TABLE 3.1

REGISTER VALUES SET IN I^{2}C SENSOR CONFIGURATION MODULE

Register Address	Register Value	Description
0x001	16'h01E0	Row_Start = 480
0x002	16'h0212	Column_Start = 530
0x003	16'h03BF	Image Row_Size = 480
0x004	16'h063F	Image Column_Size = 800
0x005	16'h0000	Horizontal Blanking = 0
0x006	16'h0019	Vertical Blanking = 25
0x009	16'h05E5	Shutter Width Lower = 1509
0x011	16'h1805	PLL_m_Factor = 24, PLL_n_Divider = 5
0x012	16'h0003	PLL_p1_Divider = 3
0x01E	16'h4313	Read Mode 1
0x020	16'h8000	Read Mode 2
others	default	No

3.1.3 CMOS Sensor Data Capture Module

The clock frequency of this module is *PIXCLK*. Its main function is to extract the valid pixel data streams based on the synchronous signals *FVAL* (Frame Valid) and *LVAL* (Line Valid) from the CMOS Image Sensor. Valid image data from the sensor is surrounded by horizontal blanking and vertical blanking, as shown in Figure 3.6.

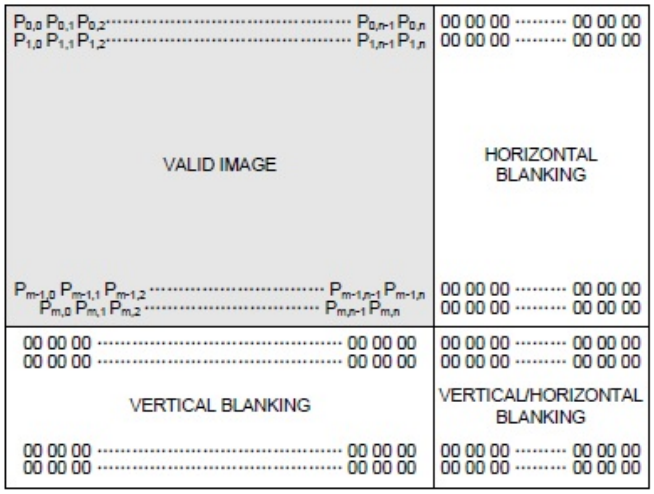


Figure 3.6. Spatial Illustration of Image Readout [39]

The image data is read out in a progressive scan. For each *PIXCLK* cycle, one 12-bit pixel datum outputs on the *DOUT* pins of the sensor. When both *FVAL* and *LVAL* are asserted, the pixel is valid. In vertical blanking region, *FVAL* is negated. In horizontal blanking region, *LVAL* is negated. Figure 3.7 displays part of the output data timing waveforms (from SignalTap II Logic Analyzer). The *oDVAL* signal is the output signal of this module, representing the time when pixel data is valid. Note that there only exists some delay between *oDVAL* and *LVAL*[4].

[4]The default situation is for *LVAL* to be negated when *FVAL* is negated. So except for some delay, *oDVAL* signal is just *LVAL* signal.

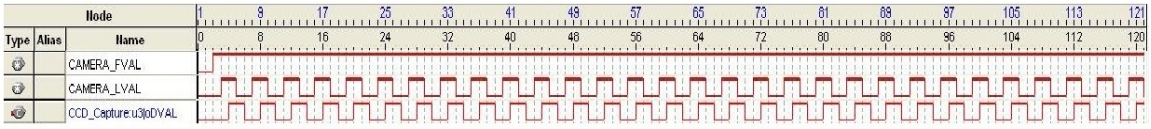


Figure 3.7. Output Data Timing Waveforms

In addition, this module counts the number of frames based on the falling edge of *FVAL* signal and displays it on the seven-segment displays. Meanwhile, the X/Y coordinates of each valid pixel in one image is also output for RAW2RGB module use.

3.1.4 RAW2RGB Module & Bayer Pattern Data Buffer Module

The raw data or Bayer Pattern data has been introduced in Section 1.2. This RAW2RGB module converts the Bayer Pattern data from the CMOS Image Sensor to RGB data format so that the image can be displayed on LCD. We already know that each pixel in Bayer Pattern format is one of the three colors, red, blue and green. However, RGB data format requires that each pixel has red, blue and green three colors which are added together in some way to reproduce a broad array of colors. The method adopted in this module is to extract four adjacent pixels in a square (*R*, *G1*, *G2*, *B*) and reproduce a pixel with $oR = R$, $oG = (G1+G2)/2$, $oB = B$, as shown in Figure 3.8.

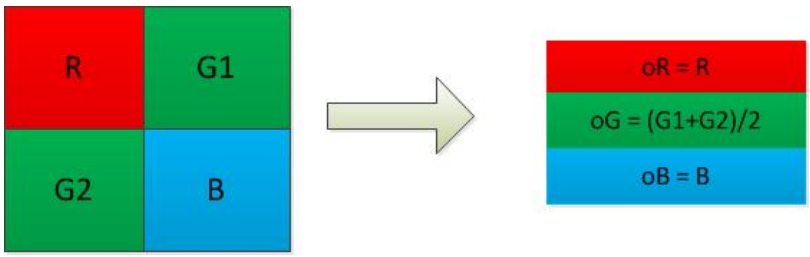


Figure 3.8. The Algorithm of RAW2RGB

The ALTSHIFT_TAPS megafunction is used to realize this method. This megafunction is a parameterized shift register with taps. The taps provide data outputs at certain points in the shift register chain, but the tap points must be evenly spaced. The space between taps, the number of taps and the width of the shiftin and shiftout ports can be configured through the MegaWizard Plug-In Manager [3]. In order to extract four Bayer Pattern pixels in a square, TAP_DISTANCE is 800, NUMBER_OF_TAPS equals to 2 and WIDTH_OF_SHIFT is set to be 12[5]. Figure 3.9 displays tapping data from the shift register.

[5]Recall that each row in the Bayer Pattern format has 800 pixels and each pixel is 12 bit. We need to extract pixel values from two rows at the same time.

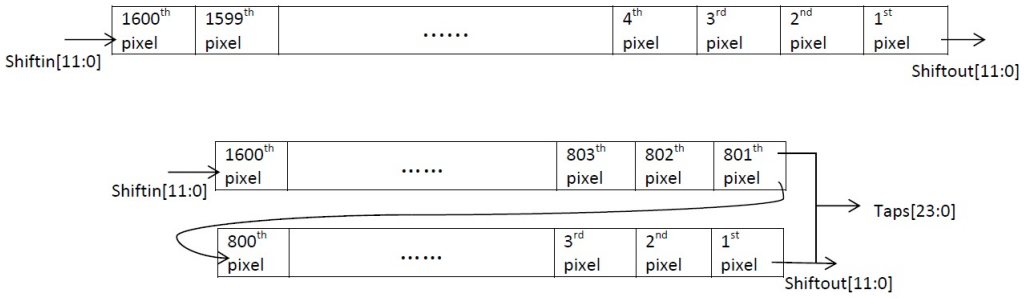


Figure 3.9. Line Buffer

At the beginning, the shift register is empty. At each pixel clock cycle, one pixel is shifted into the register. When the first row of the raw data is totally in the register, only *Taps[11:0]* has pixel data. Both the *Shiftout* port and *Taps[23:12]* output 0. Then the existing data in the shift register chain are shifted right. Until the second row of the raw data is also totally in the register, *Taps[23:0]* can give the desired data. So the output signal *oDVAL* of the RAW2RGB module for valid RGB data should be delayed for time of two rows compared to the input valid pixel signal *iDVAL* (Figure 3.10). In the next pixel clock cycle, the 1^{st} pixel is shifted out. At the same time, *Taps[23:12]* (i.e., 1^{st} pixel) and *Taps[11:0]* (i.e., 801^{th} pixel) are given to signals *WData0* and *WData1* respectively. Now the 2^{nd} pixel becomes the 1^{st} pixel and the 802^{th} pixel becomes the 801^{th} pixel. During the following clock cycle, the values of *WData0* and *WData1* are given to another two signals *WData0_d* and *WData1_d* (d for delay) and *WData0* and *WData1* obtain new values, the 2^{nd} pixel and 802^{th} pixel. This process continues as long as *oDVAL* is high. The four signals *WData0*, *WData1*, *WData0_d*, *WData1_d* are ordered like Figure 3.11. Note that i represents the column (X) coordinate and j represents the row (Y) coordinate. Thus the X axis direction is from right to left and the Y axis direction is from top to bottom.

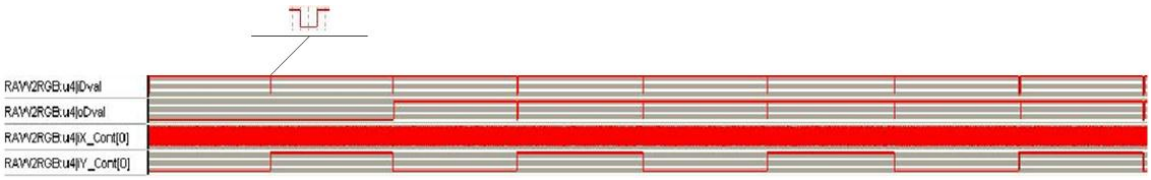


Figure 3.10. Important Signal Waveforms in RAW2RGB Module

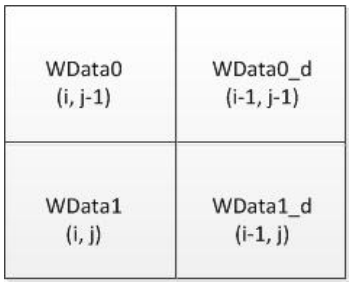


Figure 3.11. The Order of *WData0*, *WData1*, *WData0_d*, *WData1_d*

Once the four adjacent pixels in a square have been obtained, the pixel order must be determined, i.e., figure out which piece of data is *R*, *G1*, *G2* and *B*. In the document, it mentions the pixel (10,50) is *G1*. Based on the Row_Start, Column_Start, Row_Size, Column_Size and row mirror mode[6], the first row to be readout is *B, G2, B, G2...* and the second row to be readout is *G1, R, G1, R....* In addition, as mentioned in Subsection 3.1.3, the CMOS Sensor Capture Module outputs

[6]Row mirror means that the readout order of the rows are reversed. In Figure 3.12, we put the last row in the first row place, so as to keep the readout direction consistent with the Y axis direction.

the coordinates of each valid pixel. Based on the last bit of X/Y coordinates (i.e., *iX_Cont[0]* and *iY_Cont[0]*), each pixel in the square can be determined. The waveforms for *iX_Cont[0]* and *iY_Cont[0]* are referenced to Figure 3.10. With *WData0*, *WData0_d*, *WData1* and *WData1_d* arranged as Figure 3.11, the four possible orders of *R, G1, G2, B* are shown in Figure 3.12. In this figure, 0 and 1 represent the values of *iX_Cont[0]* and *iY_Cont[0]*. When *WData0*, *WData0_d*, *WData1* and *WData1_d* is the condition in the red block, the input coordinates is the X/Y coordinates of the second pixel in the third row (i.e., now $iX\_Cont[0] = 1$, $iY\_Cont[0] = 0$). Thus it can be concluded: If $iY\_Cont[0] = 0$ and $iX\_Cont[0] = 1$, $oR = WData1$, $oG = (WData0 + WData1\_d)/2$, $oB = WData0\_d$. Similarly, other conclusions are obtained. The final result is displayed in Table 3.2.

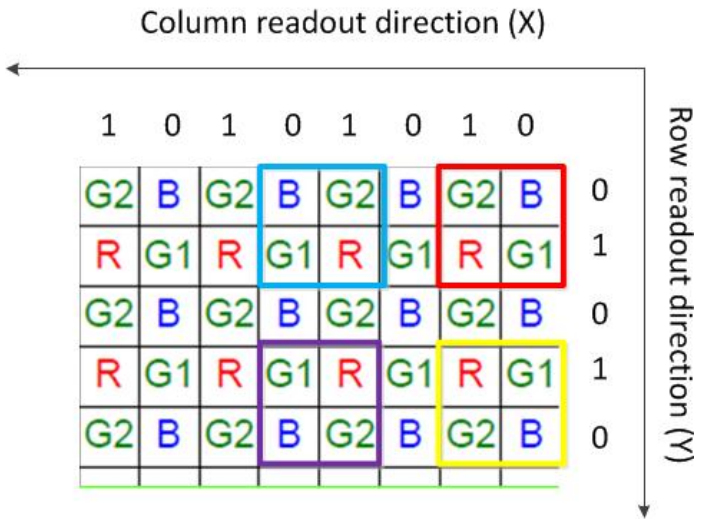


Figure 3.12. Four Possible Orders of *R, G1, G2, B*

TABLE 3.2

CONCLUSIONS FOR FOUR ORDERS OF *R*, *G1*, *G2* AND *B*

Condition	*iX_Cont[0]*	*iY_Cont[0]*	*oR*	*oG*	*oB*
Red Block	1	0	*WData1*	$\frac{WData0+WData1\_d}{2}$	*WData0_d*
Blue Block	1	1	*WData1_d*	$\frac{WData0\_d+WData1}{2}$	*WData0*
Yellow Block	0	0	*WData0*	$\frac{WData0\_d+WData1}{2}$	*WData1_d*
Purple Block	0	1	*WData0_d*	$\frac{WData0+WData1\_d}{2}$	*WData1*

Another module, Bayer Pattern Data Buffer module, does nothing to the Bayer Pattern data streams but buffer, i.e., synchronize with the RAW2RGB module in case of data overlap in SDRAM. The same ALTSHIFT_TAPS megafunction with the same parameters is used in this module again. Though it doesn't need the taps function any more, 1600-pixel buffer (shift register) is still needed. At the beginning, pixels are shifted into the register chain and *Shiftout* port just outputs 0. When the first two rows of the raw data are both in the register, the *Shiftout* port begins to output the first pixel of the first row. So similarly, the output valid pixel signal should be delayed for time of two rows compared to the input valid pixel signal. Under such condition, the Bayer Pattern data from this module and the RGB data from RAW2RGB module can be output synchronously into SDRAM[7].

[7]Note that the resolution from the two modules are the same 800 × 480.

3.1.5 SDRAM Controller

The VEEK-MT features 128MB of SDRAM, implemented using two 64MB S-DRAM devices (IS42S163200). Each device consists of separate 16-bit data lines connected to the FPGA, and they share control and address lines. Therefore, the two SDRAM devices cannot be used separately, but used as one SDRAM with 32-bit data lines. The SDRAM is internally configured as four banks and each bank is organized as 8192 rows by 1024 columns. Pins $BA0$ and $BA1$ select bank address, pins $A0-A12$ is the row address input and pins $A0-A9$ is the column address input[8]. All operations of the SDRAM is controlled by pins $\overline{CS}$, $\overline{RAS}$, $\overline{CAS}$ and $\overline{WE}$ [22]. The main operations implemented by this SDRAM Controller module are Initialization, Read/Write, Refresh, Precharge.

The initialization process is:

1. After the power is applied , wait INIT_PER[9] = 24000 clock cycles (the clock for SDRAM Controller is 100MHZ) until the clock is stable with pin *DQM* high and pin *CKE* high.
2. A 200us delay is required prior to issuing any command (the manual only requires 100us).
3. Once the delay is satisfied, a Precharge command should be applied. All banks must be precharged.
4. At least eight Auto Refresh cycles must be performed. The interval between two refresh cycles is 200us.
5. The SDRAM is ready for mode register programming. The mode register is used to define the specific mode of operation of the SDRAM, including the selection of a burst length, a burst type, a CAS latency, an operation mode and a write burst mode. In this design, burst length is Full Page (SDR_BM = 3'b111), burst mode is sequential (SDR_BT = 1'b0), CAS latency mode is 3 (SDR_CL = 3'b011).

The procedure to perform Read/Write operations is:

[8]The address pins are $A0-A12$, so row address and column address share the address lines.

[9]The parameters mentioned below are all set in a file Sdram_Params in the Camera demo of VEEK-MT.

1. A row in one bank should be opened first. This is accomplished via the Active command. The minimum time interval between successive Active commands to different banks is defined by SC_RRD = 7 clock cycles.
2. After opening a row, s Read or Write command is issued to that row, subject to the SC_RCD = 3 clock cycles.
3. The delay between the registration of a Read command and the available of the first piece of output data is CAS latency (SDR_CL = 3'b011).

Other operations are also described below:

Refresh: SDRAM is a volatile memory so Auto Refresh is necessary to hold data. The manual states that this command must be executed at least 8192 times for every 64ms. So refreshing one row of each bank cost 7.8125us, approximating 1024 clock cycles (REF_PER = 1024).

Precharge: When the controller needs to access a different row after one row has been opened, the SDRAM should first return to idle state. This is Precharge command, and pin $A10$ determines whether one or all banks are precharged.

The main operations mentioned above are realized through a FSM which is displayed in Figure 3.13.

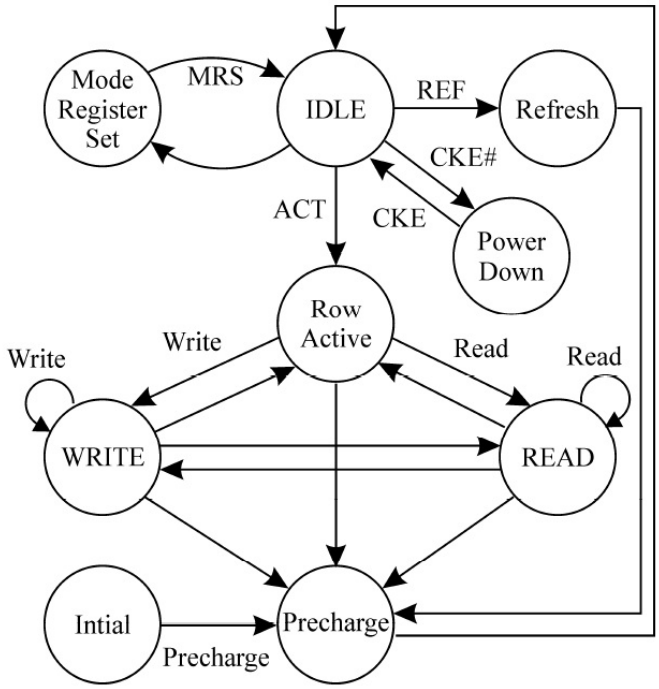


Figure 3.13. State Transition Diagram of the SDRAM Controller [22]

Another feature of the SDRAM Controller is the use of FIFO. The data streams from the RAW2RGB module and the Bayer Pattern Data Buffer module are written into SDRAM in *PIXCLK* clock frequency (i.e., 25MHZ). Similarly, the LCD Controller module and ImgRead Controller module read data from SDRAM in clock frequencies 33MHZ and 25MHZ respectively. However, SDRAM and SDRAM Controller work on 100MHZ clock frequency. To avoid problems caused by different clocks, a FIFO (Figure 3.14) must be added as a buffer [45]. There are four FIFOS in the SDRAM Controller, two for writing and the other two for reading. The FIFO megafunction provided by Altera [7] is adopted to set up dual-clock, 32-bit dual port asynchronous FIFO memory with 256 words. The 32-bit port is wide enough for writing or reading data in our design. In RAW2RGB module, for each Bayer Pat-

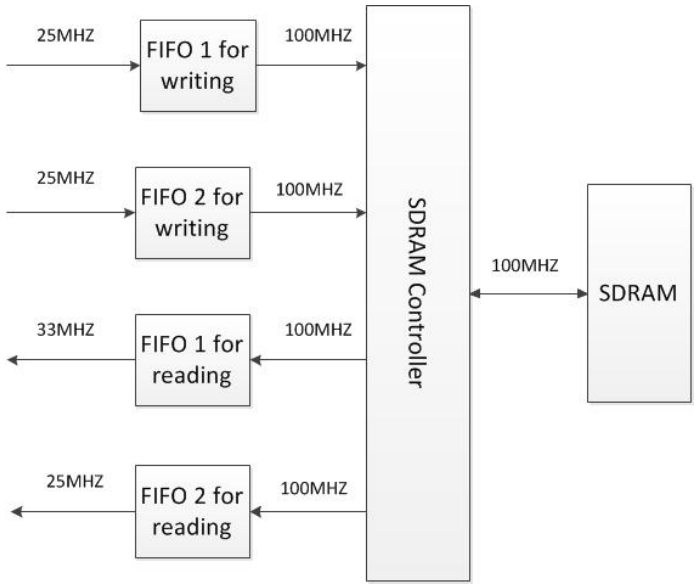


Figure 3.14. Four FIFOS in the SDRAM Controller

tern pixel (12-bit), RGB three color pixels are generated and each one is still 12-bit. However, the LCD only requires 8-bit for each color of RGB. So the number of bits for RGB data from RAW2RGB module can be reduced to 8. Now for each pixel with RGB three colors only $8 \times 3 = 24$ bits is needed, smaller than the port width of FIFOs. In this case, both the RGB data (24-bit) and the Bayer Pattern data (12-bit) can be written into and read from the FIFOs.

In addition, another thing that should be paid attention to is the frequency of S-DRAM and SDRAM Controller. The SDRAM clock frequency should provide enough bandwidth for FIFOs to write data into SDRAM [45]. In our design, the frequency of reading data from FIFOs is 33MHZ + 25MHZ = 58MHZ, and SDRAM works on clock frequency 100MHZ. Thus there is 42MHZ bandwidth that can be provided for FIFOs to write data into SDRAM. Though the total frequency for writing FIFOs is 25MHZ + 25MHZ = 50 MHZ, the left 42MHZ still can satisfy our needs. Enough bandwidth can ensure the data for writing will not be lost.

3.1.6 LCD Controller

This module generates data enable signal (*DEN*), horizontal synchronization signal (*HS*) and vertical synchronization signal (*VS*) for LCD to display images correctly. The *HS* signal (Figure 3.15) consists of four parts, pulse, back porch, valid period and front porch. The pulse in *HS* means the data of a new line starts to be sent, back porch and front porch fields are blank regions for flyback and valid period is the time when data is valid. The *VS* signal is similar to *HS* except that every pulse in *VS* means a new image will be sent (Figure 3.16). *DEN* signal (Figure 3.17) implies the coordinates of the current pixel data is in the active region of LCD (i.e., It is high only when *HS* and *VS* are both in the valid period.).

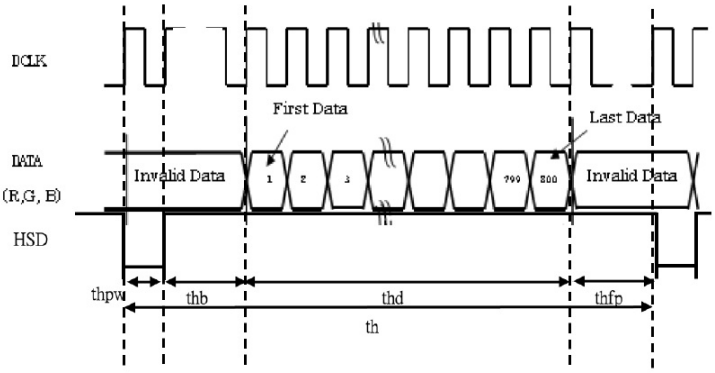


Figure 3.15. *HS* Timing [30]

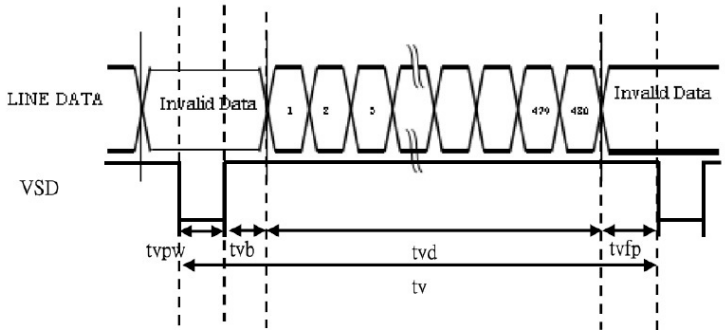


Figure 3.16. *VS* Timing [30]

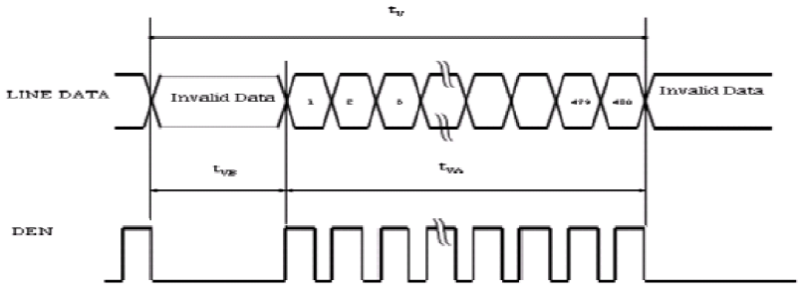


Figure 3.17. *DEN* Timing [30]

The timing sequences in figures above should satisfy the specifications in the data sheet of LCD [30]. The related parameters are listed Table 3.3.

TABLE 3.3

PARAMETERS FOR LCD DISPLAY TIMING

Parameter	Value
Dot Clock	33MHZ
HS pulse width	30 clocks
HS back porch	16 clocks
HS valid period	800 clocks
HS front porch	210 clocks
HS period	1056 clocks
VS pulse width	13 *HS* periods
VS back porch	10 *HS* periods
VS valid period	480 *HS* periods
VS front porch	22 *HS* periods
VS period	525 *HS* periods

The idea to implement this module is very simple. At each dot clock cycle, one pixel is sent to LCD. So based on the number of dot clock cycles, the X/Y coordinates of the current pixel can be obtained. When the X coordinate is between 45 and 844, the Y coordinate is between 23 and 502, *DEN* is high. The pulse in *HS* and *VS* signals can also be generated according to the coordinates.

3.2 Customized Nios II System

The blue block in Figure 3.1 is the customized Nios II system built by SOPC builder. It utilizes the Avalon interconnect structure to connect the Nios II soft processor and various I/O peripherals to provide a hardware environment for future software development. Altera provides the soft core of the processor and a collection of frequently used I/O peripherals, such as JTAG UART core, PIO core and Flash memory interface [4]. For certain specialized I/O functions, a pre-designed core may not exist and we must design the hardware from scratch or existed cores to satisfy our needs. In this Nios II system, the SPI Controller, Aceelerometer Controller, Signal Controller and ImgRead Controller are created as custom I/O soft cores. With such configuration, most work of the complex SD card operation, accelerometer operation and so on can be left to software programs. Figure 3.18 shows this Nios II system structure again for convenience.

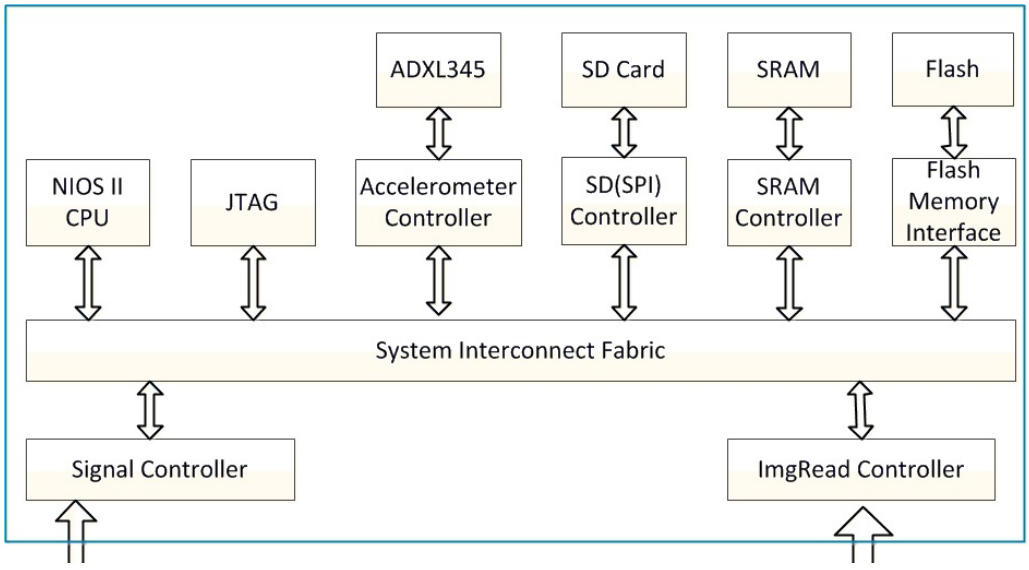


Figure 3.18. Customized Nios II System Structure

3.2.1 Nios II Processor

Nios II is a soft-core processor targeted for Altera's FPGA devices. As opposed to a fixed prefabricated processor, this soft-core processor is described by HDL codes and then mapped onto FPGA's generic logic cells. Thus it can be configured and tuned by adding or removing features to meet performance or cost goals. This approach offers more flexibility.

There are three basic versions of Nios II in the SOPC Builder [14]:

- Nios II/f: The fast core is designed for optimal performance. It has a 6-stage pipeline, instruction cache, data cache, and dynamic branch prediction.
- Nios II/s: The standard core is designed for small size while maintaining good performance. It has a 5-stage pipeline, instruction cache, and static branch prediction.
- Nios II/e: The economy core is designed for optimal size. It is not pipelined and contains no cache.

In our Nios II system, the size of Nios II processor is not a problem so Nios II/f is selected for optimal performance. The configuration dialog is displayed in Figure 3.19. In addition, we also need to specify the memories and locations of the reset vector and exception vector. A typical system usually adopts a nonvolatile memory module for the reset code. Thus the Flash memory is selected as the reset vector memory. Since our software program requires a relatively large amount of memory, SRAM is adopted as the exception vector memory. The other settings for our Nios II processor is default. Note that a level-1 JTAG debug module is used in the default setting. For specific characteristics of the processors and related options, please refer to Altera's tutorial [4].

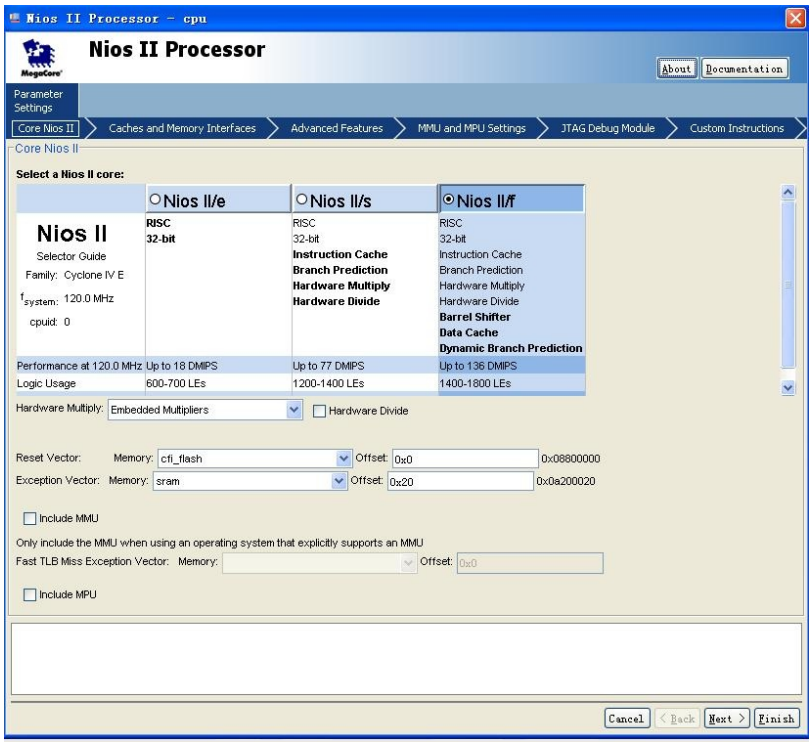


Figure 3.19. Nios II Processor Dialog

3.2.2 JTAG UART Core

The JTAG UART (Universal Asynchronous Receiver and Transmitter) core (Figure 3.20) with Avalon interface provides a method to communicate serial character streams between a host PC and the board. On one side, the Nios II processor communicates with the core by reading and writing control and data registers. To increase the performance and regulate data transmission, a write FIFO buffer and a read FIFO buffer are also included in this core. On the other side, the core uses the JTAG circuitry built into Altera FPGA and provides host access via the JTAG pins on the FPGA [4]. The host PC can connect to the FPGA via any Altera JTAG download cable, such as the USB-Blaster cable.

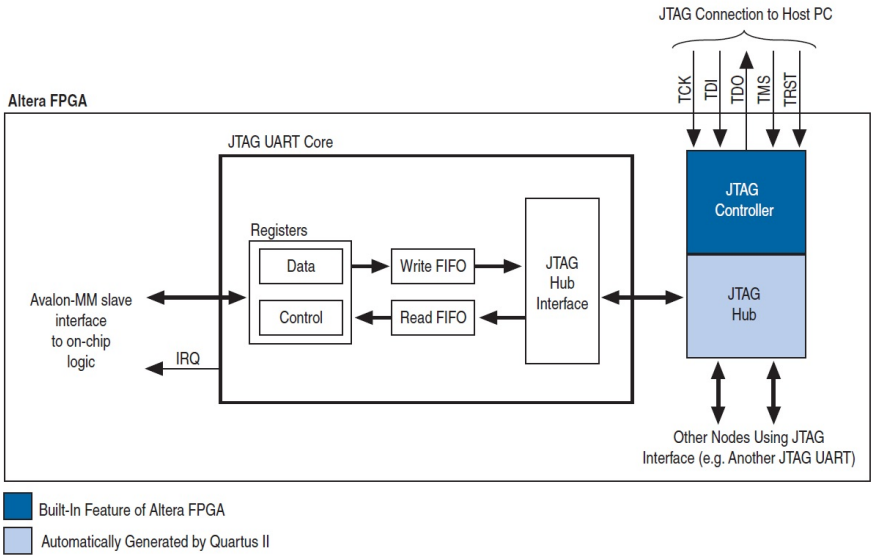


Figure 3.20. JTAG UART Core Block Diagram [4]

The software support for the JTAG UART core is provided by Altera as well. For the Nios II processor, device drivers are provided in the HAL (Hardware Abstraction Layer) system library, allowing software to access the core using the ANSI C Standard Library functions, such as getchar() and printf() [5]. For the host PC, Altera provides JTAG terminal software NIOS EDS that manages the connection to the target, decodes the JTAG data stream, and displays characters on screen. The connection between a host PC and an Nios II system containing a JTAG UART core is shown in Figure 3.21.

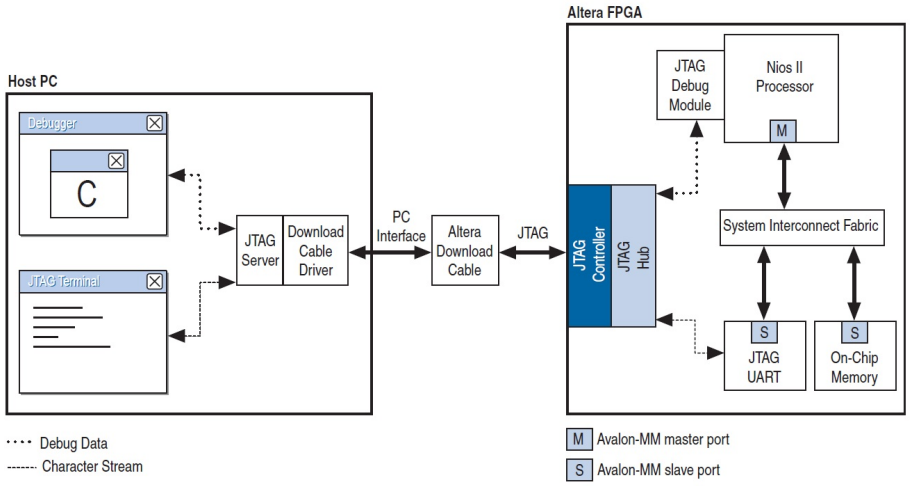


Figure 3.21. Host-Target Connection [4]

The JTAG UART core in our Nios II system is used to debug the software program. For example, print out acceleration data or write image data into a file of the host PC. In the SOPC Builder, this core is configured in default settings.

3.2.3 Flash Memory Interface

For VEEK-MT flash memory that complies with the Common Flash Interface (CFI) specification [41], Altera provides a CFI Controller core with Avalon interface allowing one to easily connect the Nios II system to the flash memory [4]. As stated in the Nios II processor, the flash in our design works as the memory for reset vector.

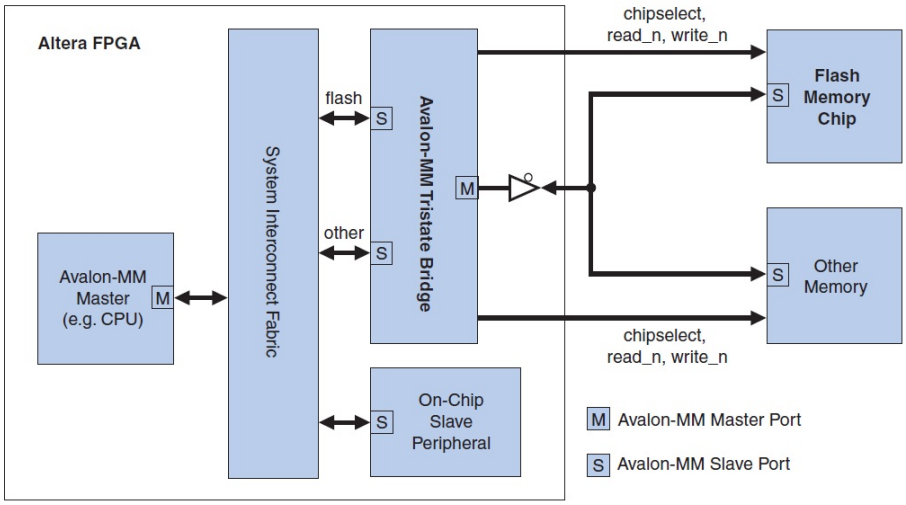


Figure 3.22. Block Diagram of the CFI Controller [4]

As shown in Figure 3.22, the Avalon interface for flash memory is connected through an Avalon Memory-Mapped (Avalon-MM) tristate bridge. The CFI Controller core is simply an Avalon-MM tristate slave port. The tristate bridge creates an off-chip memory bus that allows the flash chip to share address and data pins with other memory chips but provides separate chipselect, read, and write pins. The slave port (i.e., CFI Controller) is capable of Avalon-MM tristate slave read and write transfers. Therefore, besides CFI Controller core, the Avalon-MM tristate bridge also needs to be integrated into our Nios II system. In the configuration dialog of CFI Controller (Figure 3.23), the address width is 23 bits and the data width is 8 bits so flash memory capacity is 8MB. In addition, there are also options specifying the timing requirements for read and write transfers within the flash memory. Referring to the specifications of the flash memory, we can obtain the needed timing values, including $Setup = 60$ns, $Wait = 160$ns and $Hold = 60$ns.

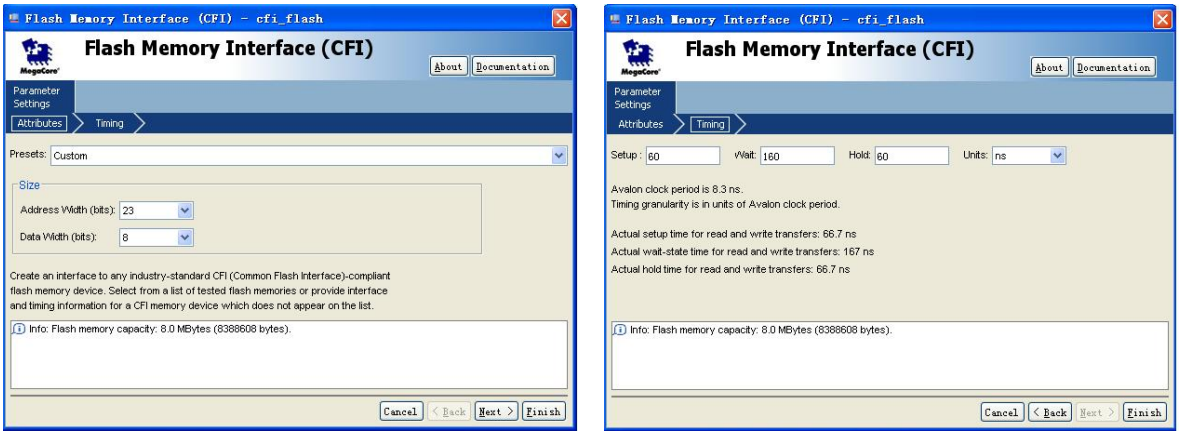


Figure 3.23. Configuration Dialog of the CFI Controller

3.2.4 SRAM Controller

The VEEK-MT board has an IS61WV102416BLL device, which is a $1024K \times 16$ SRAM module. This device has a 20-bit address bus, $A0-A19$, a 16-bit bidirectional data bus, $I/O0 - I/O15$, and five control signals, $\overline{CE}$ (chipselect), $\overline{OE}$ (output enable), $\overline{WE}$ (write enable), $\overline{LB}$ (lower byte control), and $\overline{UB}$ (upper byte control). The SRAM in our system is used for providing the exception vector memory.

The SRAM Controller provided by Altera is a circuit used to access the SRAM device. It generates proper control signals, issues the address, and places and retrieves data according to the SRAM's timing specifications. The SRAM is asynchronous, so it does not contain a clock signal and its operation is based on the duration and level of the address, data, and control signals. This greatly simplifies the design of SRAM Controller. In the HDL design file, the program is just mapping the Avalon-MM slave interface signals to the SRAM's pins. Then the SOPC component can be created based on the HDL file, with the Avalon-MM slave interface configured to

match the timing characteristics of the SRAM. As shown in Figure 3.24, the timing parameters are best set in timing units ns rather than clock cycle such that the SRAM Controller can be used in any clock frequency without modifying the timing parameters.

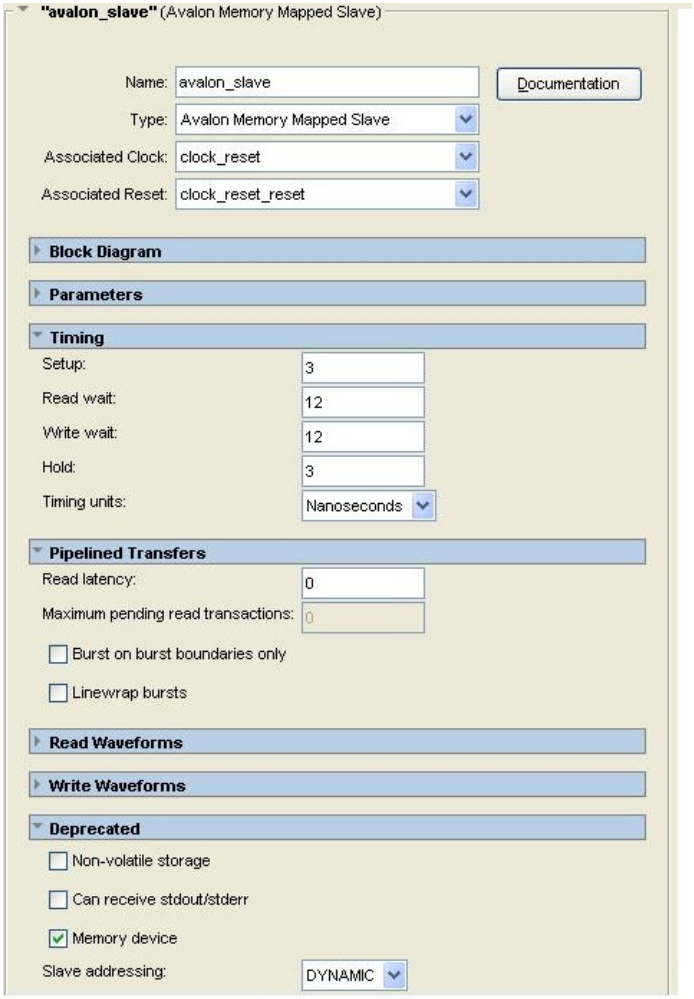


Figure 3.24. Timing Configuration of the SRAM Controller

3.2.5 Accelerometer Controller

The accelerometer ADXL345 is accessible through either a SPI (3- or 4-wire) or I^2C digital interface. The I^2C communication mode is selected in this design. As introduced in the I^2C Sensor Configuration module, the I^2C bus between FPGA chip (Accelerometer Controller) and the accelerometer ADXL345 requires a simple 2-wire connection, *sda* (serial data) and *scl* (serial clock). In addition, it is required that the $\overline{CS}$ be connected to high level and that the *ALT ADDRESS* pin be connected to either high level or GND (Ground) in the I^2C mode. With the *ALT ADDRESS* pin high, the 7-bit I^2C slave address is 0x1D. Otherwise, the alternate I^2C address is 0x53 [8]. In our system, *ALT ADDRESS* pin is assigned 1'b0, so the slave address is 0x53. The I^2C mode connection diagram is given in Figure 3.25.

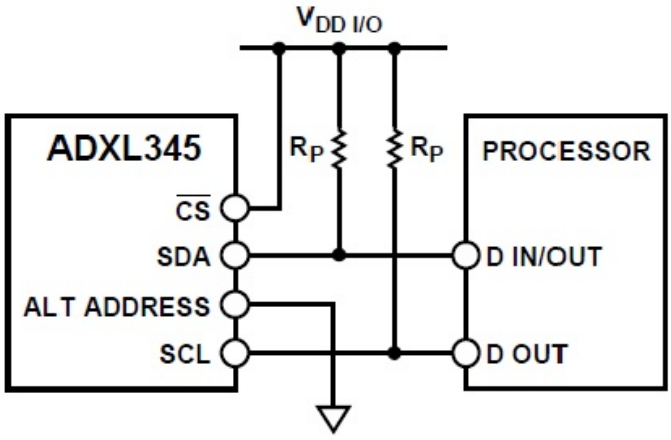


Figure 3.25. I^2C Mode Connection Diagram [8]

The Accelerometer Controller in the Nios II system is used to provide an interface between Avalon interconnect fabric and the ADXL345. The simple PIO core provided

by Altera is adopted here to build the controller. The PIO core works as general I/O port, connecting either to on-chip user logic or to off-chip external devices. Besides that, the PIO core can be configured to perform more sophisticated tasks, such as capturing edge of the input signal and generating interrupts [14]. The conceptual diagram of a full-featured PIO core is shown in Figure 3.26.

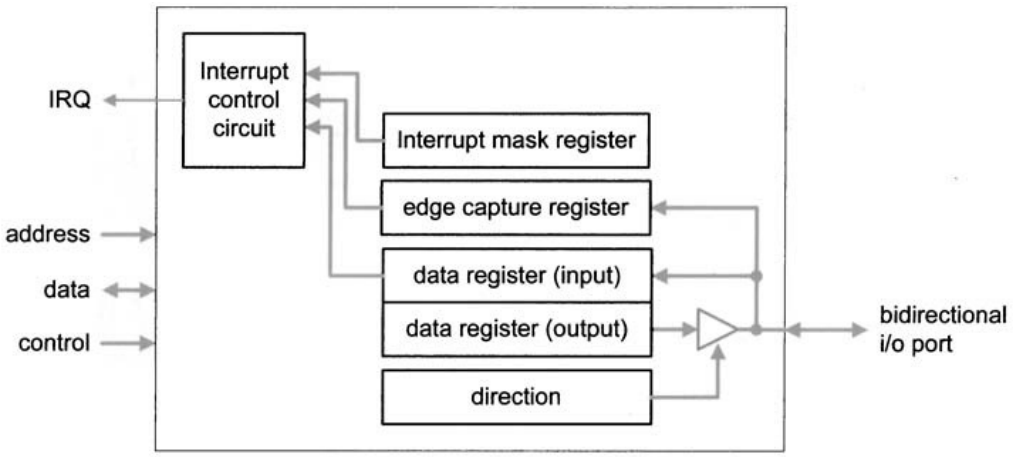


Figure 3.26. Conceptual Diagram of a Full-featured PIO Core

The Accelerometer Controller consists of three PIO cores, one connecting to *SCL* pin, one connecting to *SDA* pin, and the third one connecting to the two interrupt pins *INT1* and *INT2* of the accelerometer. The PIO core for serial clock *scl* is set to be 1-bit width, output direction[10], output reset value 0, not enable individual bit setting/clearing[11]. The PIO core for serial data *sda* is configured to be 1-bit width, bidirectional, output reset value 0, not enable individual bit setting/clearing, no

[10]The FPGA acts as the master in I^2C bus, so it generates the clock.

[11]This PIO core is 1-bit width, so it does not matter whether this option is turned on or not.

edge capture, no interrupt. The third one for interrupts input is 2-bit width, input direction, synchronously falling edge capture, enable bit-clearing for edge capture register, IRQ generation triggered by input signal edge. It is already known that many functions of this accelerometer is mapped to either of the two interrupt pins. Therefore, the edge capture and IRQ generation functions in the third PIO core are turned on to help us capture the transition edge of the interrupt signal, and even generate the IRQ signal to Nios II processor. In the software, the processor controls and communicates with these PIO cores via a set of registers. The register map will be discussed in the software part.

3.2.6 Signal Controller

In the I^2C Sensor Configuration module, the register 0x01E is set to be the value 16'h4313, i.e., invert trigger, ERS mode, snapshot mode, strobe enable, first trigger as the strobe start, second trigger as the strobe end. The ERS mode is actually the electronic shutter mechanism. The snapshot mode means that frames are output one at a time, with each frame initiated by the signal on *TRIGGER* pin of the sensor. Thus this trigger signal can work as a start point to collect the raw data and the acceleration data synchronously. In default, the low level on the *TRIGGER* pin[12] triggers frames. For convenience, the invert trigger option is turn on such that the high level will initiate frames. One PIO core in the Signal Controller is connected to the *TRIGGER* pin of the sensor, so as to control the start of one frame by writing registers of the PIO core. The configuration for this PIO core is 1-bit width, output direction, output reset value 0.

After one frame is initiated, the CMOS Image Sensor is exposed row-by-row.

[12]The *TRIGGER* pin is level-sensitive, so multiple frames can be output by holding the *TRIGGER* pin at the triggering level.

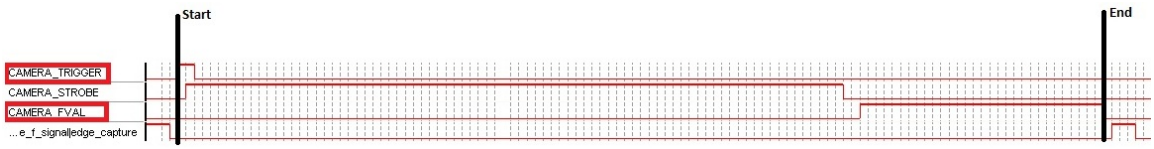


Figure 3.27. Basic Timing of One Frame

When the first row ends its exposure[13], the *FVAL* signal[14] changes from 0 to 1, which means the start of the data readout. Thus, the falling edge of the *FVAL* signal marks the end of the data readout and the end of this frame as well. Another PIO core in the Signal Controller is connected to the *FVAL* pin of the sensor so that by capturing the falling edge of this signal we can obtain the end time of the frame. In this case, the PIO core for *FVAL* signal is set to be 1-bit width, input direction, synchronously falling edge capture.

In conclusion, the Signal Controller can help us to control the start of one frame and capture the end of one frame through software programs. The timing for one frame is shown in Figure 3.27. The rising edge of the *STROBE* signal represents the beginning of the exposure (called first trigger), and its falling edge (or almost the rising edge of the *FVAL* signal) marks the start of the readout (called second trigger).

3.2.7 ImgRead Controller

It is already known that the Bayer Pattern Data Buffer module writes the raw image data into SDRAM through one port of the SDRAM Controller. To read out these raw image data and send them to the Nios II processor for further software processing, the ImgRead Controller is designed as an interface between the SDRAM Controller and the Avalon interconnect fabric. Figure 3.28 shows the input and

[13]The specific timing has been introduced in Chapter 1.

[14]The *FVAL* and *LVAL* signals have been discussed in the CMOS Sensor Data Capture Module.

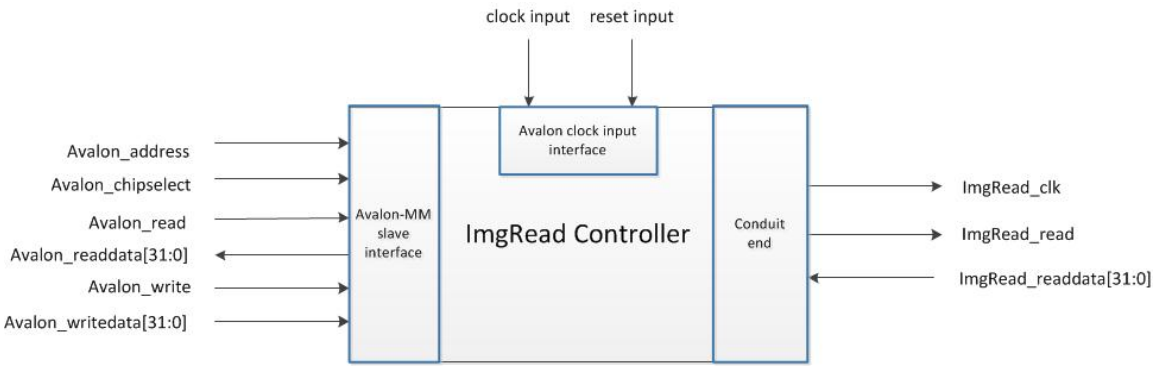


Figure 3.28. Input and Output Ports of ImgRead Controller

output ports of this controller.

To integrate a component into the Nios II system, the interface compatible with the Avalon specification must be provided in the design file. These interfaces, known as Avalon interfaces, can accommodate various communication needs and connect various components. As shown in Figure 3.28, the ImgRead Controller contains Avalon-MM slave interface, Avalon clock input interface and Avalon conduit interface.

The Avalon clock input interface receives the clock and reset signals configured in the SOPC Builder. The Avalon-MM slave interface defines an address-based master-slave connection. The Avalon-MM master (Nios II processor) uses an address to identify an Avalon-MM slave (ImgRead Controller) and reads data from or writes data to the slave [14]. The signals defined by Avalon-MM slave interface are:

- *Avalon_read*: it is a 1-bit signal asserted in a read transfer.
- *Avalon_write*: it is a 1-bit signal asserted in a write transfer.
- *Avalon_chipselect*: it is a 1-bit signal asserted when the slave device is selected. The Avalon interconnect fabric automatically decodes the base address (slave address) to generate the chipselect signal, so the *Avalon_chipselect* is 1 or 0, not an address.
- *Avalon_readdata[31:0]*: it is the data provided by a slave in a read operation.

The width is 32-bit because every pixel of the raw image data from SDRAM is 32-bit.

- *Avalon_writedata[31:0]*: it is the data written to a slave in a write operation.
- *Avalon_address*: it is used to specify an offset relative to the base address. Each value identifies a memory location in the register map.

In addition, because the Avalon-MM slave interface is a synchronous interface and all transactions are based on the clock input signal, a set of timing properties should also be specified for the Avalon-MM slave interface. The key properties [14] are:

- *readWaitTime*: it controls the length of the read signal. It allows us to prolong the read signal to accommodate a slow I/O device in a read operation.
- *writeWaitTime*: it controls the length of the write signal. It allows us to prolong the write signal to accommodate a slow I/O device in a write operation.
- *setupTime*: it specifies the time interval between the assertion of the address and data signals and the assertion of the read or write signal.
- *holdTime*: it specifies the time interval between the deassertion of the address and data signals and the deassertion of the write signal.
- *readLatency*: it specifies the time interval between the assertion of the read signal and the availability of data.

The Altera Avalon-MM Interface specifications state that the fundamental write and read transfer that can capture data in a single clock cycle is generally appropriate for on-chip peripherals. The ImgRead Controller is on the same FPGA chip with the Nios II processor, so *readWaitTime* and *writeWaitTime* are 0 if the timing unit is clock cycle. Then other timing properties are also set to 0 because both the read and write operations are completed in one clock cycle. The timing diagram of the ImgRead Controller is given in Figure 3.29.

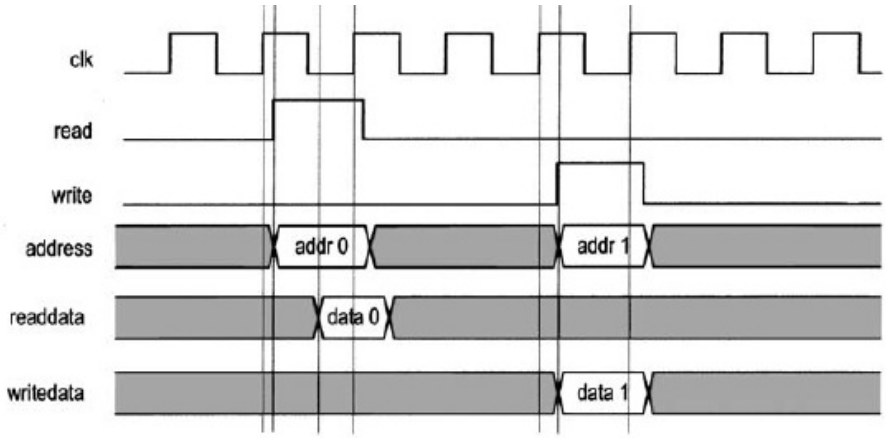


Figure 3.29. Timing Diagram in ImgRead Controller [14]

The third interface, Avalon conduit interface, communicates with the SDRAM Controller. It generates the clock signal *ImgRead_clk* and read enable signal *ImgRead_read* for one reading port of the SDRAM Controller, and extracts the raw image data from the data lines *ImgRead_readdata[31:0]*.

With these Avalon interfaces, the design of the ImgRead Controller becomes is just signal mappings. At the negative edge of *reset* input, reset *ImgRead_read* to 0 and *ImgRead_readdata[31:0]* to high impedance. Then at each positive edge of the *clock* input signal, check whether *Avalon_chipselect* is 1 and *Avalon_read* is 1. If so, assign *ImgRead_read* 1 and send the raw image data on *ImgRead_readdata[31:0]* to *Avalon_readdata[31:0]*. The clock on *ImgRead_clk* is the clock frequency specified for this controller in the SOPC Builder (25MHZ), i.e., map the clock input to *ImgRead_clk*. Based on this HDL design file, the SOPC component ImgRead Controller can be created by specifying the timing parameters given above.

3.2.8 SD (SPI) Controller

SD (secure digital) card is a memory card widely used for massive storage. In this design, the SD card is utilized to store the raw image data and acceleration data. A block diagram of the SD card is shown in Figure 3.30. It consists of a 9-pin interface, a card controller, a memory interface and a memory core. The 9-pin interface allows the exchange between a connected system and the card controller. The controller can read/write data from/to the memory core using the memory core interface. In addition, several internal registers are provided to store the state of the card [34].

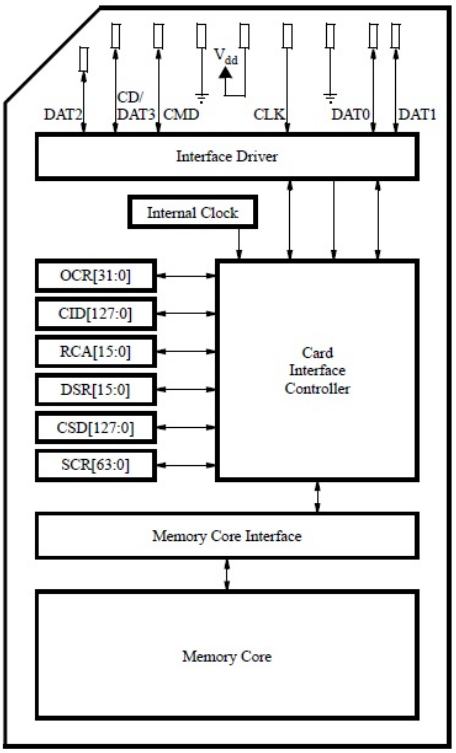


Figure 3.30. Block Diagram of SD Card [34]

To interface with an SD card, the VEEK-MT features an SD card socket, on which pins labeled as *CLK*, *CMD*, *DAT0*, and *CD/DAT3* are connected to the FPGA. The data exchange between the FPGA and the SD card can adopt one of the two modes: SD mode or SPI (Serial Peripheral Interface) mode. The SD mode is a proprietary format and uses four lines for data transfer. The SPI is an open standard for serial interfaces and is widely used in embedded applications [14]. We selects SPI mode for SD card communication. Therefore the SD Controller in this Nios II system mainly implements the SPI protocol (Figure 3.31).

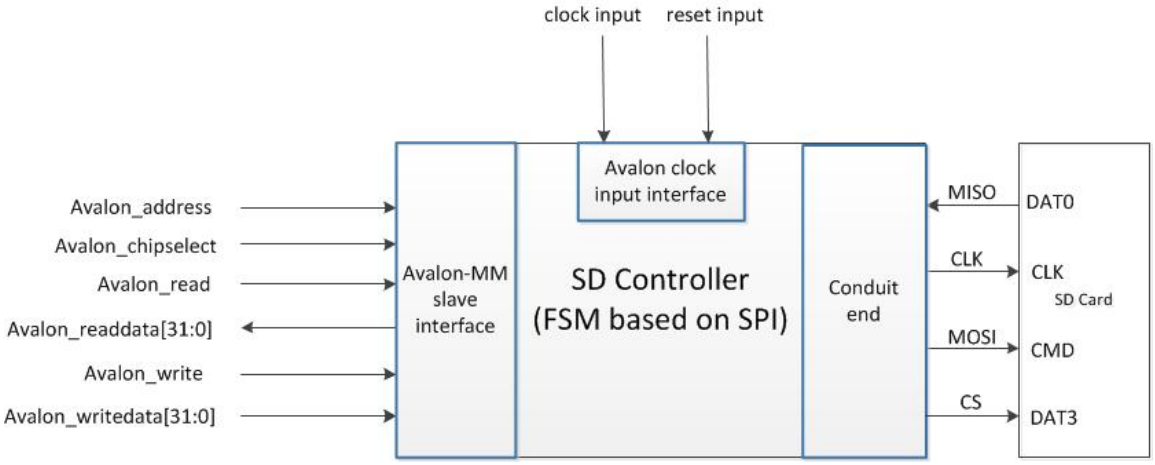


Figure 3.31. Input and Output Ports of SD Controller

The SPI comprises four wires, clock (*CLK*), Master-Out Slave-In (*MOSI*), Master-In Slave-Out (*MISO*) and chip select (*CS*). The clock signal *CLK* is generated by the master to synchronize the exchange of data. The *MOSI* line is used by the master to send commands and data to the slave, while the *MISO* line is used by the slave to respond to commands and send data back to the master. The fourth line *CS* enables or

disables the slave device [14]. As shown in Figure 3.31, these four lines are connected to the pins *CLK*, *CMD*, *DAT0* and *DAT3* of SD card respectively. For illustrating the data transfer in SPI, it is assumed that there are two 8-bit shift registers, one in the master (FPGA chip or SD Controller) and one in the slave (SD card), shown by Figure 3.32. At the beginning of the operation, both the master and slave load data into the registers. Then at each *CLK* cycle, data in both registers is shifted to the left by one bit. After eight *CLK* cycles, eight data bits are shifted and the master and slave have exchanged register values. This operation can be interpreted that the master writes data to and reads data from the slave simultaneously, which is known as full-duplex operation. Note that for the SPI operation in SD card, the data are read at the rising edge and changed at the falling edge of the *CLK* signal [14].

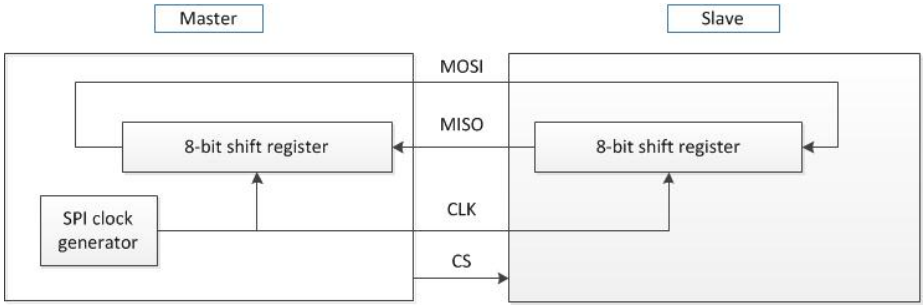


Figure 3.32. Data Transfer in SPI

The design approach of the SD Controller is similar to that of the I^2C Sensor Configuration module. The controller is basically an FSM that generates the *CLK* signal, shifts a data bit into an input buffer at the rising edge of *CLK*, and shifts a data bit out from an output buffer at the falling edge of *CLK*. The data is transferred

byte-by-byte. The sketch of the FSM is shown in Figure 3.33. The *CLK0* and *CLK1* states represent the low and high portions of the *CLK* signal. The FSM circulates in these two states eight times to transfer a byte. Wrapping this FSM design with the Avalon interfaces, we can create the SOPC component and iintegrate it into the Nios II system.

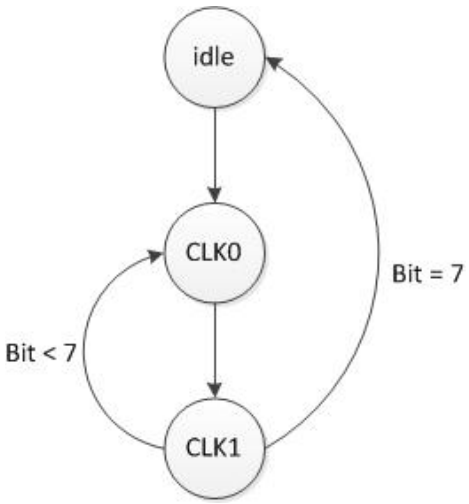


Figure 3.33. FSM of SD Controller

3.2.9 Other SOPC Components

The other components included in the Nios II system are:

- System ID core: it is a unique 32-bit value based on an SOPC design, like a signature. It can be used to maintain the consistency between the hardware configuration file (.sof) and software image (.elf).
- Interval Timer core: it can be configured to fit different timing needs. In this system, it helps us to obtain the current time stamp.
- KEY PIO core: it is connected to the *KEY[3]* on the VEEK-MT so as to receive the key press signal.

3.3 Conclusion

In this chapter, the hardware frame, including user logic and customized Nios II system, is discussed in detail. The user logic part realizes capturing the raw data from the CMOS Image Sensor, displaying the RGB data on LCD screen and meanwhile sending the raw data to the Nios II system. The Nios II system built by the SOPC Builder provides a hardware platform for further software processing in Nios II SBT GUI.

CHAPTER 4

SOFTWARE DEVELOPMENT

The software frame of the embedded SOPC system is developed through Nios II SBT GUI. It contains two parts: user applications and BSP (Board Support Package), as shown in Figure 4.1. This chapter will discuss each layer of the software frame in detail.

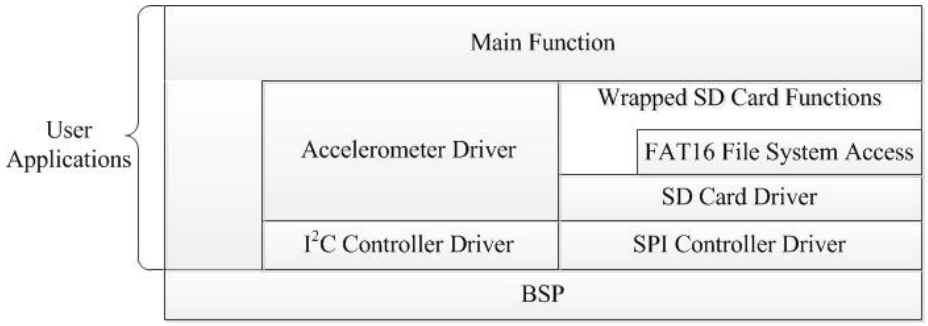


Figure 4.1. Block Diagram of the Software Frame

4.1 BSP

The software frame is based on the hardware platform provided by the Nios II system. As described in Section 2.2, the processor and I/O configuration is recorded in the .sopcinfo file when the Nios II system is created in the SOPC Builder. During the software project initialization, the BSP Editor in Nios II SBT GUI examines

.sopcinfo file and builds up a BSP library to support the user applications. The BSP library consists of three directories (drivers, HAL and obj) and a collection of system-level files [14], in which several headers are important for our software project.

- system.h (system-level file): provides automatically generated base address and interrupt request number of each I/O device.
- io.h (HAL): provides enhanced I/O register read and write macros.
- alt_types.h (HAL): provides explicitly defined (width and format) low-level data types.
- sys/alt_timestamp.h (HAL): provides API functions to return the current value of the timestamp counter.
- sys/alt_alarm.h (HAL): provides the driver for the interval timer core.
- sys/alt_irq.h (HAL): provides API functions to manage interrupt requests.
- sys/alt_dev.h (HAL): defines the alt_dev structure which is a collection of function pointers. These functions are called in response to application accesses to the HAL file system.

In addition, the settings of the BSP library should be adjusted according to the design requirements. Especially the stderr, stdin, stdout, sys_clk_timer, and timestamp_timer fields should be checked to ensure that the proper I/O modules are selected.

4.2 User Applications

With the BSP library, the user application programs realize the specific functional requirements including reading the raw image data out of SDRAM, obtaining acceleration data from 3-axis accelerometer and finally saving these data into a SD card. They are composed of basic device drivers and high-level user functions, as shown in Figure 4.1. The device drivers, SPI Controller Driver and I^2C Controller Driver, are a collection of routines based on low-level I/O transactions to control the basic operation of SD card and accelerometer. On top of them are high-level user functions

including SD Card Driver, FAT16 File System and Accelerometer Driver. The Main Function is a sequence of high-level functions and basic I/O transactions to meet the application needs.

4.2.1 SD Card Operation

The lowest layer for SD card operation is the SPI Controller Driver consisting of routines to receive and transmit data via the SPI bus[1]. These routines just write and read the corresponding registers in the SPI Controller, indirectly completing the data exchange with the SD card. The specific tasks of this driver include checking whether the SPI Controller is idle (ready), receiving and transmitting 8-bit data via the SPI bus, setting up the SD card chip select signal and setting the SPI clock divisor [15].

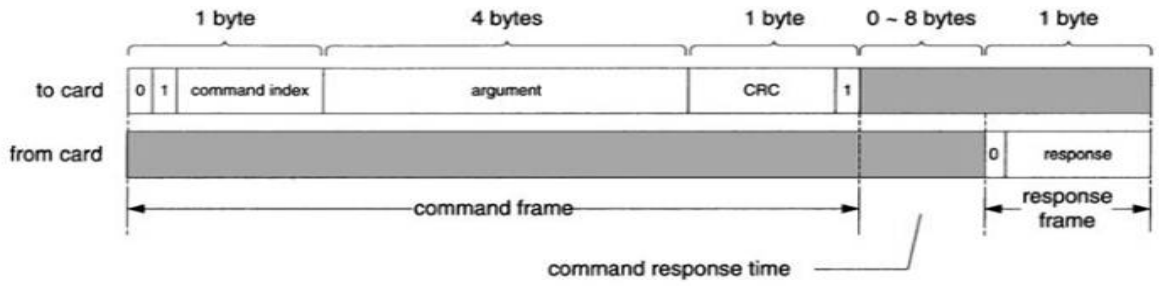


Figure 4.2. Basic Timing Diagram of the SD Card Protocol [14]

On top of the SPI Controller Driver, the communication between the SPI Controller and the SD card should comply with SD card protocol [34]. First, the SPI Controller (or Nios II processor) issues a command frame. Then the SD card pro-

[1]Subsection 3.2.8 has introduced the SPI bus.

cesses this command within a certain amount of time (known as command response time) and finally responds with a response frame. The timing diagram of the basic protocol is given in Figure 4.2. As shown in this figure, the command frame consists of six bytes. The first byte is command, which is started with 01 and followed by the 6-bit command index (usually appears as CMDxx and ACMDxx in SD card documentation). The next four bytes are the argument field that can accommodate up to 32 bits of information. The last byte consists of the 7-bit CRC code and a final stop bit 1. The command response time is between zero and eight bytes. During this interval, the SPI Controller continues generating the clock signal and the SD card sends bytes filled with all 1's. As for response frames, there are several formats of response frames. The most common one is the R1 response, which contains one byte. The MSB is always 0 and the seven LSBs indicate various conditions and error status:

- bit 7: always 0
- bit 6: parameter error
- bit 5: address error
- bit 4: erase sequence error
- bit 3: command CRC error
- bit 2: illegal command
- bit 1: erase reset
- bit 0: idle state

The other response frames in the implementation are R3 and R7 formats. These formats contain five bytes, in which the first byte is the same as R1 format and the remaining four bytes return certain status information.

The commands involved in the SD Card Driver are summarized in Table 4.1 [14], which lists the command index , argument usage, response type and its basic function. The complete command set can be found in the SD card documentation.

TABLE 4.1

BASIC SD CARD COMMANDS

Command Index	Argument	Response Types	Description
CMD0	[31:0]: stuff bits	R1	reset SD card to idle state
CMD8	[31:12]: reserved(0) [11:8]: voltage [7:0] check pattern	R7	send SD card interface condition
CMD17	[31:0]: address	R1	read a single block
CMD24	[31:0]: address	R1	write a single block
CMD55	[31:0]: stuff bits	R1	start an application command
CMD58	[31:0]: stuff bits	R3	read OCR register
ACMD41	[30]: HCS other: reserved(0)	R1	send HCS bit and initialize card

The SD Card Driver follows the protocol and consists of functions to send a command, initialize the card, and read and write a sector. The function to send a command completes transmitting a 6-byte command frame (1-byte command index, 4-byte argument, and 1-byte CRC), waiting for up to 8-byte delay, and retrieving the response frame. The other functions utilize this basic function to send a sequence of commands corresponding to their purposes. The steps of initializing the card are:

1. The master holds the *MOSI* and *CS* lines high for at least 74 cycles over *CLK* line and then SD card enters the SD mode and is in the idle state.
2. The master holds *CS* low and send the CMD0 command to force the SD card to

switch to the SPI mode and the SD card responds with R1 0x01.

3. The master issues the CMD8 command (argument 0x000001aa) to verify that the SD card interface can operate within the host's supplied voltage range 2.7-3.6V and the SD card should respond with R7 0x01000001aa.

4. The master issues the ACMD41 command to send the host capacity support information (bit 30 is 1 if the host supports high capacity SD cards) and activate the card's initialization process (several hundred milliseconds). The host should repeatedly issue this command until the R1 response is 0x00.

5. The master uses the CMD58 command to retrieve the contents of the internal register OCR in SD card to check whether the card is a standard capacity SD card or a high capacity SDHC card. The SD card should respond with R3, the first byte is 0x00 and the remaining four bytes are the contents of the OCR register.

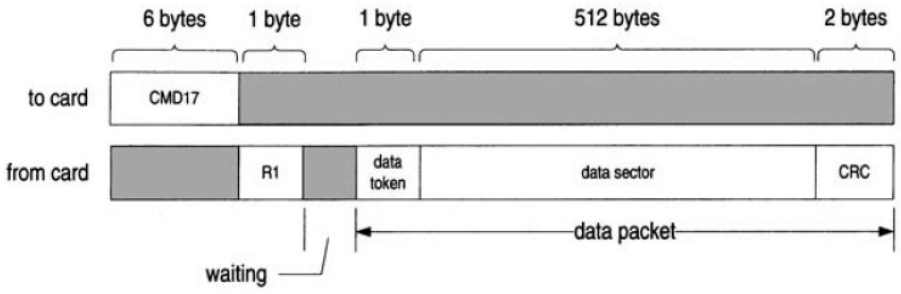


Figure 4.3. Basic Timing Diagram of a Single Block Read Operation [14]

Compared to the initialization process, the reading and writing operations are relatively easier. Note that SD cards transfer data in blocks or called sectors and each sector is 512 bytes. As shown in Figure 4.3, the reading process is:

1. The master sends the CMD17 command (argument is the starting location of the data) and the SD card responds with R1 0x00.

2. The SD card requires some time to process the request.

3. The SD card transmits the data packet with the data token 0xFE.

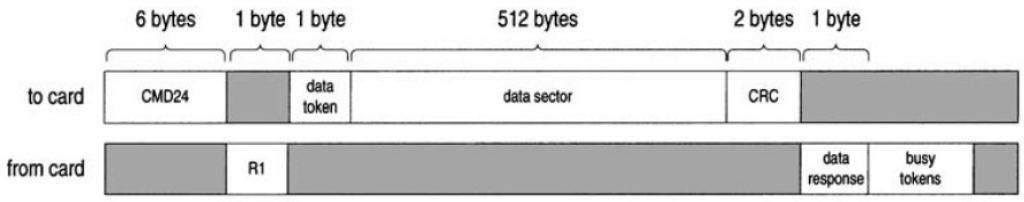


Figure 4.4. Basic Timing Diagram of a Single Block Write Operation [14]

The writing process in Figure 4.4 is:

1. The master sends the CMD24 command (argument is the starting location of the data) and the SD card responds with R1 0x00.
2. The master transmits the data packet with data token 0xFE.
3. The SD card acknowledges the packet with a data response token 0x05 and then starts the write operation, i.e., issues a continuous stream of 0x00 busy tokens on the *MISO* line.

The software layers described above only deal with the access of physical I/O storage device. To handle file management, a simple file system is needed to map logical files to physical storage. Two commonly used file systems for SD cards are FAT16 and FAT32, which supports storage size up to 4GB and 2TB respectively. Since the SD card used in the design is 2GB, FAT16 file structure is adopted. The simplified layout of the FAT16 file system is shown in Figure 4.5.

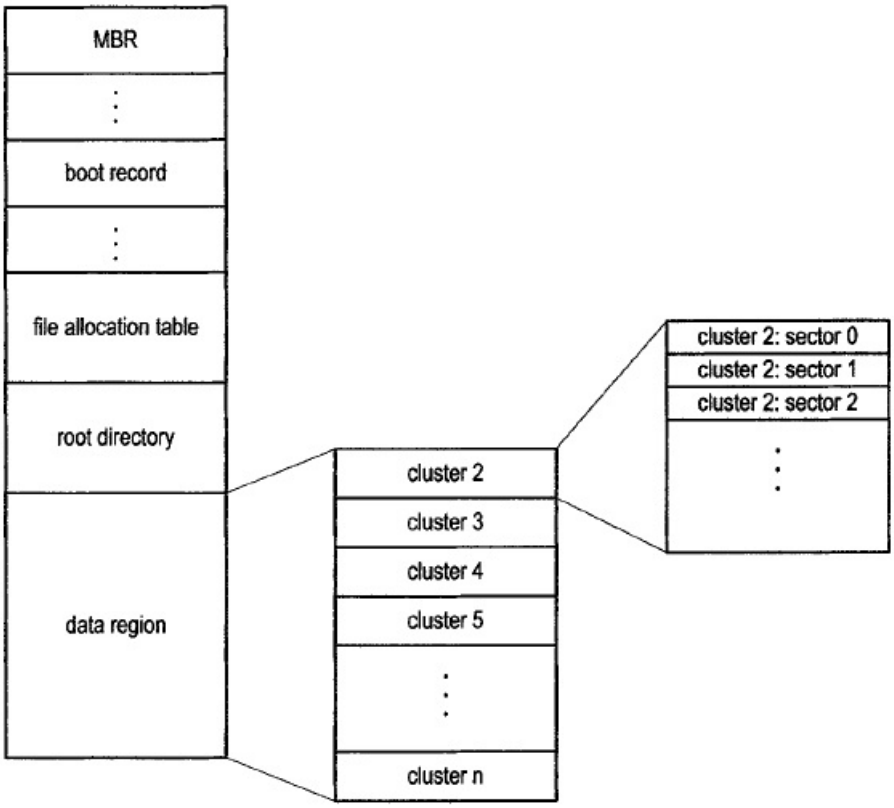


Figure 4.5. Simplified Layout of a FAT16 File System

The MBR (Master Boot Record), which is located in sector 0, uses a partition table to maintain the information of each partition. Though the FAT16 system can support up to four partitions, only the first partition is considered in this design. The starting location of the first partition, as well as the boot record sector of the first partition, can be found at the offset address 0x1C6 of MBR. This boot record contains the basic information of the file system in the first partition, such as the starting location and size of the root directory, file allocation table, and data region. The root directory maintains information for files on root level. Each file is represented by a 32-byte entry, which consists of fields for the name, extension, attributes, date

and time of creation, the starting cluster of the file data, and the size of file. The actual file data is stored in the data region. This region is organized as a collection of clusters, which usually contains multiple sectors. A file can take many clusters. Instead of allocating the file to one large consecutive chunk of storage space, the file system uses a linked list to thread the clusters together. The linked list is realized by a lookup table, in which the index of the table represents the current record and its content is the next link [14]. For example, a file is allocated to clusters 4,5,8 and 9 and the lookup table is given in Figure 4.6. This table is referred to as the file allocation table.

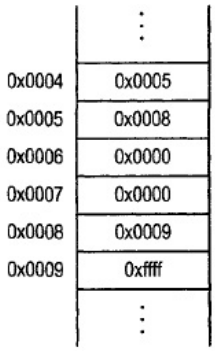


Figure 4.6. Portion of File Allocation Table

Based on this basic structure of FAT16 file system, the necessary file access routines, including initializing the FAT16 file system, opening a file, writing a file and closing a file, can be derived.

The procedure to initialize the FAT16 file system is:

1. The master reads sector 0 and checks the file system type at the offset address 0x1C2 of MBR. For a FAT16 file system, the value in this location should be 0x04,

0x06 or 0x0E.

2. The master fetches the boot record of the first partition. The boot record location can be found at the offset address 0x1C6 of MBR.
3. The master checks the boot record signature 0x55AA at its offset address 0x1FE.
4. The master extracts the key parameters from the boot record, such as the starting sector of the root directory, starting sector of the file allocation table, starting sector of the data region, number of file entries in the root directory, number of sectors per cluster, number of clusters in the data region.

To facilitate the implementation of other routines, a data structure is used to maintain basic file information, to keep track of the operation status, and to provide buffering space [14]. This data structure is defined as:

```
typedef struct file_descriptor
{
  char* name; // file name and extension
  int rdir_index; // entry index in root directory table
  int size; // size of file in byte
  alt_u16 cls0; // starting cluster number
  alt_u16 clsi; // current data cluster number
  int sect; // current sector
  int idx; // current index in sector
  int seek; // current position in file
  char data_buffer[512]; // data sector buffer
  int fat_sector; // current fat sector number
  char fat_buffer[512]; // fat sector buffer
  int open; // file is properly opened
} file_dp_type;
```

By creating an instance *fd* of this data structure, the process of opening a file can be expressed as:

1. The master loads the root directory sector by sector to *fd->data_buffer* according to the information obtained from the initialization process.
2. The given file name *fd->name* is compared with the file name of each file entry in the current root directory sector until a match is found.
3. The master loads this file's first file allocation table sector to buffer *fd->fat_buffer*.
4. The master initializes other information in *fd*, including *fd->open* =1.

The steps of writing a file are:

1. This file is assumed to be opened successfully.
2. The master calculates the number of clusters and sectors needed for the data to be written.
3. The master loads this file's last file allocation table sector to buffer *fd->fat_buffer*, modifies the content of the last cluster number to an available cluster number and writes this sector back to the file allocation table. This operation continues until all needed clusters are allocated.
4. The file entry in the root directory is updated.
5. The host loads a data block into *fd->data_buffer*, updates *fd->idx* and *fd->seek*, and writes the data in the buffer into the corresponding sector in the cluster. Then *fd->sect* and next *fd->clsi* are updated and the host writes a new data block. This operation continues until the host finishes writing all data.

If there is no other operations of this file, the host should close it by modifying the open flag *fd->open* to 0.

Finally, the routines in the SD Card Driver and FAT16 File System can be combined and wrapped to provide a simpler application interface for the Main Function,

4.2.2 Accelerometer Operation

The I^2C bus protocol and the procedure of writing a register via I^2C bus have been introduced in the I^2C Sensor Configuration module (Subsection 3.1.2). This protocol is also adopted for the communication between the Nios II processor and the accelerometer. The Accelerometer Controller in Subsection 3.2.5, which consists of three PIO cores, has provided the I^2C connection between the FPGA chip and

the accelerometer. To realize data exchange, the routines that comply with the basic timing of the I^2C bus protocol should be derived in the software.

To illustrate the implementation of I^2C Controller Driver, the register map of a PIO core is given first [4]. The Nios II processor controls and communicates with a PIO core via a set of registers. The offsets of all the registers in the address space is called the register map, as shown in Figure 4.7. Address offset 0 is for the input and output data register. The other registers are optional according to the configurations in the SOPC Builder. Address offset 1 is for the direction register if the port is bidirectional. Address offset 2 is for the interrupt mask register if the PIO is configured to include the interrupt function. Setting a bit in this register to 1 enables the interrupt request for the corresponding bit of the PIO input port. Address offset 3 is for the edge capture register if included. A bit is set to 1 when an edge is detected in the corresponding bit of the PIO input port. Writing 1 to the register clears its content to 0. The registers of address offset 0 and 1 are used in the I^2C Controller Driver. Other registers will be involved in the Main Function.

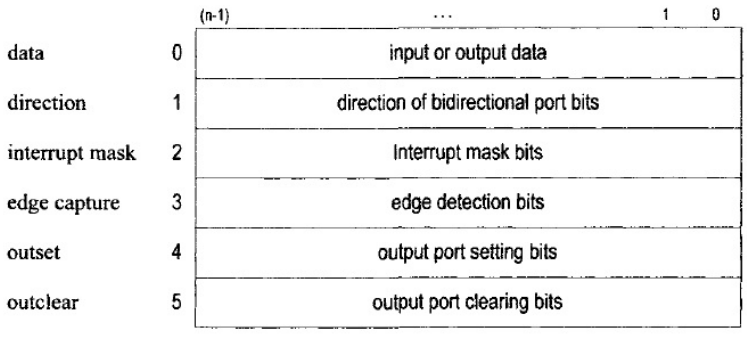


Figure 4.7. Register Map of a PIO Core [14]

The lower routines in the I^2C Controller Driver mainly realize the basic timing of I^2C bus protocol (Figure 3.4), including creating start condition, stop condition, reading 8-bit data and sending *ACK*, and writing 8-bit data and receiving *ACK*. For example, the start condition, in which *SDA* changes from high to low while *SCL*[2] is high, can be created as below.

1. The master writes 1 in the direction register of the *SDA* PIO core so as to make this core as an output port
2. The master writes 1 in the data register of *SDA* PIO core and then writes 1 in the data register of *SCL* PIO core. 5ns delay is needed for setting up the signal.
3. The master writes 0 in the data register of *SDA* PIO core. After 5ns delay, it writes 0 in the data register of *SCL* PIO core. 5ns delay is needed again.

Other lower routines are similar to the start condition routine by operating the data registers and direction registers of the *SDA* PIO core and *SCL* PIO core according to the timing requirement. Then these routines are called in order to complete reading or writing an internal register of the accelerometer, as shown in Figure 4.8. The most complex one is reading multiple-byte data from the starting register address and the specific procedure to realize this function is:

1. The master creates start condition.
2. The master writes 8-bit data, which is 7-bit slave address 0x53 plus bit 0, and receives *ACK*.
3. The master writes 8-bit data, which is the register address, and receives *ACK*.
4. Repeat step 1.
5. Repeat step 2.
6. The master reads 8-bit register value and sends *ACK*. This step is repeatedly called until the second last byte.
7. The master reads the last 8-bit register value and sends *NACK*.

[2]The three PIO cores in the Accelerometer Controller are connected to pins *SDA*, *SCL*, *INT1* and *INT2* of the accelerometer, so these PIO cores are called based on these pin names.

8. The master creates stop condition.

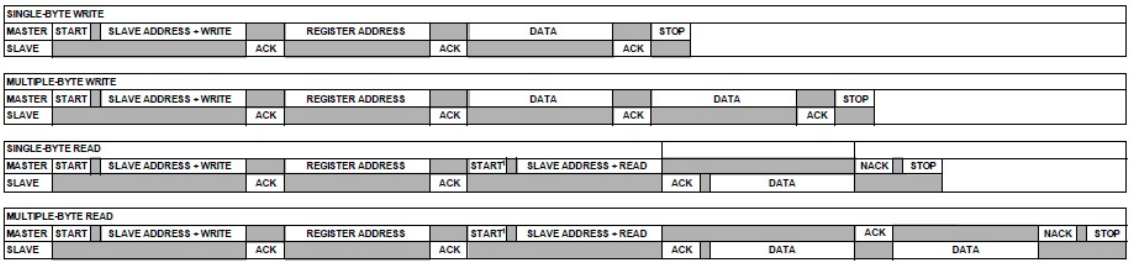


Figure 4.8. The Timing Diagram of Reading and Writing Internal Registers [8]

On top of the I^2C Controller Driver, the Accelerometer Driver layer mainly performs initializing the accelerometer and obtaining the acceleration data from it. At the beginning of the initialization, the master reads the register value at the offset address 0x00 (Device ID) to verify that the physical connection is correct. Then the master waits 1.1ms and configures related registers in order [42]. Generally, the registers POWER_CTL (address 0x2D) and INT_ENABLE (address 0x2E) are set in the end. Table 4.2 lists the register values configured in sequence during the initialization.

TABLE 4.2

REGISTER VALUES CONFIGURED DURING ACCELEROMETER INITIALIZATION

Register Address	Register Value	Description
0x31 DATA_FORMAT	0x0B	13-bit resolution at range $\pm 16g$
0x2C BW_RATE	0x0C	400HZ ODR (Output Data Rate)
0x2E INT_ENABLE	0x00	disable all interrupts
0x2F INT_MAP	0x80	map interrupts to INT1 and INT2
0x01 FIFO_CTL	0x01	bypass mode and sample number is 1
0x2E INT_ENABLE	0x83	enable corresponding interrupts
0x2D POWER_CTL	0x08	enter into measure mode

Before the needed acceleration data is sampled, calibration is needed to achieve greater accuracy. A simple method of calibration is to measure the offset while assuming that the sensitivity is the typical value. The offset can be automatically accounted for by using the offset registers OFSX, OFSY and OFSZ (0x1E, 0x1F, 0x20). This results in the data acquired from the DATA registers already compensating for the offset. The method to measure offset [8] is:

1. Put the accelerometer with z-axis in $1g$ field and x-axis, y-axis in $0g$ field.
2. Take the average of a series of samples and save them in X_0g, Y_0g and Z_1g. The number of samples is 1000.
3. Subtract the ideal sensitivity $256LSB/g$ from Z_1g to obtain Z_0g.
4. Calculate the values in the offset registers. First note that a negative value is placed in the offset register to eliminate a positive offset and vice versa for a negative offset. Second, the scale factor of DATA registers is $3.9mg/LSB$ but the

scale factor of offset register is $15.6mg/LSB$, so the values in the offset register should be 1/4 of the measured offset. Third, the offset registers contain 8-bit twos complement values. Therefore, the final values set in the offset registers are:

$$X_{OFFSET} = [-Round(X_{0g}/4)]_{twos-complement}$$
$$Y_{OFFSET} = [-Round(Y_{0g}/4)]_{twos-complement}$$
$$Z_{OFFSET} = [-Round(Z_{0g}/4)]_{twos-complement}$$

Now the acceleration data can be obtained through reading registers DATAX0, DATAX1, DATAY0, DATAY1, DATAZ0 and DATAZ1 (0x32∼0x37). DATAX0 and DATAX1 hold the output data for the x-axis, DATAY0 and DATAY1 hold the output data for the y-axis, and DATAZ0 and DATAZ1 hold the output data for the z-axis. The output data is twos complement, with DATAx0 as the least significant byte and DATAx1 as the most significant byte where x represents X, Y, or Z [8]. Note that the acceleration data of X, Y and Z should be read out together, that is to perform a multiple-byte read of all registers, to prevent a change in data [18]. After obtaining the register values, we can combine the DATAx0 and DATAx1 to get the final 13-bit acceleration data of x-axis, y-axis and z-axis.

4.2.3 Main Function

The top layer utilizes the lower layers' routines and basic I/O operations to realize triggering one image via key press, reading out acceleration data during the image exposure time and saving them into the SD card, and finally reading out the raw image data from SDRAM and writing the data into the SD card. The flow of the Main Function is:

1. Initialization:
    - Initialize the accelerometer and perform calibration
    - Register the interrupt request of the key so as to include it in the lookup table of the exception handler
    - Initialize SD card and FAT16 file system
2. Recording acceleration data

- Obtain 100 acceleration samples when this PFGA board is stationary, and save the data into the SD card in TXT format. These data are left for image deblurring use.
- Switch the accelerometer from measure mode to standby mode and reconfigure the FIFO mode and interrupt mapping. The accelerometer provides a 32-level FIFO. It is capable of holding up to 32 sample sets of data and each sample set of data consists of one x-axis sample, one y-axis sample and one z-axis sample. Figure 4.9 shows the representation of the FIFO. When the data registers are read, the data in FIFO[0] is obtained and then removed from the FIFO, allowing the rest of the stack to shift one level closer to the data registers. Then next is FIFO[1]. Before this step, the accelerometer operates in bypass mode and disables the FIFO. However, the speed to read out data from data registers cannot satisfy the 400Hz ORD of the accelerometer. Thus, a lot of samples are lost. To obtain more acceleration samples, the FIFO operation should be changed from bypass mode to FIFO mode. In addition, the watermark interrupt is mapped to *INT2* pin so as to supervise whether the number of entries in FIFO is over the sample number configured in FIFO_CTL register.

	X-AXIS		Y-AXIS		Z-AXIS	
OUTPUT FILTER	DATAX0	DATAX1	DATAY0	DATAY1	DATAZ0	DATAZ1
FIFO[31]	DATAX0[31]	DATAX1[31]	DATAY0[31]	DATAY1[31]	DATAZ0[31]	DATAZ1[31]
FIFO[30]	DATAX0[30]	DATAX1[30]	DATAY0[30]	DATAY1[30]	DATAZ0[30]	DATAZ1[30]
⋮	⋮	⋮	⋮	⋮	⋮	⋮
FIFO[2]	DATAX0[2]	DATAX1[2]	DATAY0[2]	DATAY1[2]	DATAZ0[2]	DATAZ1[2]
FIFO[1]	DATAX0[1]	DATAX1[1]	DATAY0[1]	DATAY1[1]	DATAZ0[1]	DATAZ1[1]
FIFO[0]	DATAX0[0]	DATAX1[0]	DATAY0[0]	DATAY1[0]	DATAZ0[0]	DATAZ1[0]
DATA REGISTER (0x32 TO 0x37)						

Figure 4.9. FIFO Buffer Representation [18]

- Wait for the key press. If the key is pressed, the interrupt service function is called to trigger one image and the accelerometer is switched from standby mode to measure mode. In Subsection 3.2.6, a PIO core in the Signal Controller is connected to the *TRIGGER* pin of the CMOS Image Sensor to control the start of one frame. Therefore, writing 1 in the data register of this PIO core can initiate one image. The *TRIGGER* pin should not stay high too long. Otherwise, more than one image will be triggered.
- Continuously read out the acceleration data and the INT_SOURCE (0x30)

register until the falling edge of the *FVAL* signal is detected. Another PIO core in the Signal Controller is connected to the *FVAL* pin of the image sensor to detect the end of one image by the edge detection register. The reason of reading the INT_SOURCE register is to check whether the watermark interrupt has happened or not.

- Read out the samples left in the FIFO and combine them with the obtained acceleration samples in the previous steps. The combined data are saved into the SD card in TXT format.

3. Reading out image data

   - Create and open files raw.dat and capture.bmp in the SD card. The raw.dat is used to record the raw image data and the capture.bmp record the color image converted from the raw data.
   - Write BMP header into capture.bmp
   - Read out the raw image data from SDRAM and convert them into RGB data at the same time, and then write these data into raw.dat and capture.bmp respectively. The ImgRead Controller in Subsection 3.2.7 is designed to read out the raw data from one port of the SDRAM Controller. This Controller has a data register of 32-bit. Through continuously reading this register, 800×480 raw data pixels can be obtained. Note that the readout data is 32-bit but one raw pixel is only 12-bit. Thus, the 32-bit data is converted to 16-bit data and saved into raw.dat. The process to obtain the RGB data is similar to the RAW2RGB module in Subsection 3.11.
   - Close raw.dat and capture.bmp after completing writing data.

4.3 Conclusion

The software frame is based on the Nios II system built in SOPC Builder and realizes obtaining the acceleration data and the raw image data at the same time. This chapter illustrates the functions and implementations of each layer in the software frame. For details about SD card and accelerometer ADXL345 in this chapter, please refer to related manuals and documentations.

CHAPTER 5

RAW IMAGE DEBLURRING USING ACCELERATION DATA

This chapter provides an overview of our approach which removes camera motion blur from the raw image by using the acceleration data. The first several sections discuss about how to estimate the blur kernel from the acceleration data and the last section introduces the non-blind deconvolution algorithms adopted in this design.

5.1 Image Blur Model

As mentioned in the Chapter 1, a typical model for image blurring is the linear convolution model, where the blurred image B is assumed to be unblurred image I convolved with a blur kernel K of size $(2M+1) \times (2M+1)$ plus Gaussian noise N:

$$B(x,y) = \sum_{i=-M}^{M} \sum_{j=-M}^{M} I(x-i, y-j)K(x,y,i,j) + N(x,y) \tag{5.1}$$

where the coordinate indices x, y, i, j are defined on the scale of the pixels on the image sensor. The blur kernel K is commonly referred to as the Point Spread Function (PSF), because it describes the redistribution of light from a point source across a local neighbourhood in the blurred image. In this case, I is an isolated point of light, the blurred image B is a shifted copy of the K corresponding to the location (x, y) of the original point of light. The coordinate $(i, j) = (0, 0)$ is referred to as the "anchor point" of the PSF, and it's the element of the PSF that acts on the ideal image point location in the output image.

The blur kernel in Equation 5.1 can depend on (x,y) (spatially-varying blur), or can be constant across the image plane (spatially-invariant blur). When the kernel K is spatially-invariant, the image blur model can be simplified to:

$$B = I * K + N \tag{5.2}$$

where " $*$ " denotes convolution, K is a spatially-invariant kernel (PSF) and $N \sim \mathcal{N}(0, \sigma^2)$.

5.2 Camera Motion Blur

In our work, the blur is restricted to camera motion blur. To connect the camera motion with the blur kernel K, we first assume a standard perspective projection $\Pi : \mathbb{R}^3 \to \mathbb{R}^2$ that transforms a three-dimensional (3-D) point $[x, y, z]$ in the observed scene to a two-dimensional (2-D) location $[x', y']$ in the image plane [37]:

$$\Pi([x, y, z]^T) = [x', y']^T \tag{5.3}$$

If camera motion occurs, projection of the scene point $[x, y, z]$ at time t within the exposure time is given by:

$$C(t, x', y') = \Pi(R(t)[x, y, z]^T + [T_x(t), T_y(t), T_z(t)]^T) \tag{5.4}$$

where R and T are 3-D camera rotation matrix and translation vector, respectively, that define the camera pose at time t. The rotation matrix $R(t)$ is given by three rotation angles $\theta_x(t)$, $\theta_y(t)$, $\theta_z(t)$. $C(t, x', y')$ is dependent not only on time t but also on the initial projection location (x', y'). The first reason is that the translations for all points is not the same when the camera is rotating though the pure rotation at all points is the same. The second reason is the electronic rolling shutter mechanism,

which leads to slightly difference of the motion between different rows. The third reason is that the scene depth is spatially-varying. Figure 5.1 displays the camera motion model. Note that the camera optical axis is always identical with the z-axis.

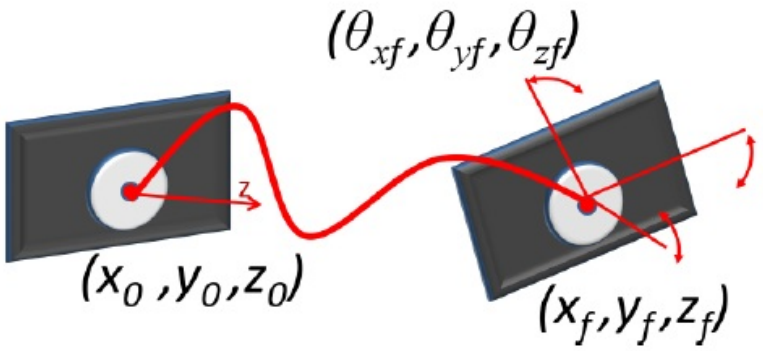


Figure 5.1. Camera Motion Model [21]

The resulting curve $C(t, x', y')$ makes up the trajectory of a trace that is left on the sensor by a point light source. Assuming a constant illuminance over the exposure time, the light energy emitted from the point is distributed evenly (with respect to time) over the curve $C(t, x', y')$. This effectively gives us a time parametrization of a PSF for a given point on image plane. Let $h(i, j)$ denote the PSF of an ideal light point from $[x, y, z]$ displayed at $[x', y']$ when the camera has no motion. Thus, the blur kernel $K(x', y', i, j)$ is obtained by integrating $h(i, j)$ over the camera motion curve $C(t, x', y')$ with respect to time t [37].

Some assumptions are used to simplify this process. First, camera rotation is ignored here because only 3-axis accelerometer is used to measure camera motion. In addition, the motion along z-axis is also assumed minimal. Therefore, only translation

along x-axis and y-axis are considered in our work. Figure 5.2 shows the geometrical constructions to determine the motion curve on image plane from camera translation along x-axis.

Figure 5.2. Simple Geometrical Constructions [21]

If the scene depth is assumed constant as well, the motion curve $C(t, x', y')$ becomes:

$$C(t, x', y') = [x', y']^T + M * [T_x(t), T_y(t)]^T \tag{5.5}$$

where $[x', y']^T$ is the initial position on the image plane and M is the magnification of the camera. Second, as for image blur, it is not necessary to consider the absolute motion of the camera, only the relative motion and its effect on the image [23]. Now $C(t, x', y')$ is:

$$C(t, x', y') = M * [T_x(t), T_y(t)]^T \tag{5.6}$$

Third, the motion curve (or blur) is spatially-invariant if the motion blur is not so large or we only focus on a small region of the image plane. Thus,

$$C(t) = M * [T_x(t), T_y(t)]^T \tag{5.7}$$

M can be obtained through experiments and $T_x(t), T_y(t)$ can be calculated from the acceleration data. Then the blur kernel K can be finally figured out by integrating the lens blur PSF $h(i, j)$ over the motion curve $C(t)$.

5.3 Camera Position Calculation

As discussed in the previous section, camera motion blur is dependent on $T_x(t)$ and $T_y(t)$, i.e., camera x-axis and y-axis translations at each time point. In this section, we describe how to recover translations given measurements from the accelerometer.

The measured acceleration is the sum of the acceleration due to translation of the camera, centripetal acceleration due to rotation, the tangential component of angular acceleration, and gravity, as shown in Equation 5.8 [23]. The description for each symbol in this equation is given in the Table 5.1.

$$\vec{m}_t^t = {}^tR^i(\vec{a}_t^i + \vec{g}^i + (\vec{\omega}_t^i \times (\vec{\omega}_t^i \times \vec{r})) + (\vec{\alpha}_t^i \times \vec{r})) \tag{5.8}$$

TABLE 5.1

DESCRIPTION FOR EACH SYMBOL IN EQUATION 5.8

Symbol	Description
$\vec{m}_t^t$	current measured acceleration in current coordinate
${}^tR^i$	transform matrix from initial coordinate to current coordinate
$\vec{a}_t^i$	current acceleration in initial coordinate
$\vec{g}^i$	gravity in initial coordinate
$\vec{\omega}_t^i$	current angular velocity in initial coordinate
$\vec{r}$	distance between camera center of mass and the accelerometer
$\vec{\alpha}_t^i$	current angular acceleration in initial coordinate

To recover relative camera translation, it is necessary to calculate the acceleration for each time point t in the initial coordinate system $\vec{a}_t^i$. Since camera rotation is not considered in our work, set

$$ {}^tR^i = I, \vec{\omega}_t^i = 0, \vec{\alpha}_t^i = 0 \tag{5.9} $$

Then Equation 5.8 becomes:

$$ \vec{m}_t^t = \vec{a}_t^i + \vec{g}^i \tag{5.10} $$

$$ \vec{a}_t^i = \vec{m}_t^t - \vec{g}^i \tag{5.11} $$

Now we can compute the camera's relative position by double integration, i.e., the integration step is performed once to obtain velocity and then repeated to obtain position. In order to reduce numerical integration error, trapezoid approximation is

adopted instead of rectangle approximation [35]. Thus, the camera's velocity and position are given as below.

$$\vec{v}_t^i = \vec{v}_{t-1}^i + (\vec{a}_{t-1}^i + \frac{\vec{a}_t^i - \vec{a}_{t-1}^i}{2}) * \Delta t \tag{5.12}$$

$$\vec{x}_t^i = \vec{x}_{t-1}^i + (\vec{v}_{t-1}^i + \frac{\vec{v}_t^i - \vec{v}_{t-1}^i}{2}) * \Delta t \tag{5.13}$$

where Δt is the interval between time points $t-1$ and t (or time interval between acceleration samples). As we are concerned with relative position, the initial position is set to zero and the initial velocity is also assumed to be zero:

$$\vec{x}_0^i = \vec{v}_0^i = \vec{0} \tag{5.14}$$

If the acceleration measurements are noise-free, then the camera translation obtained above is sufficient for deblurring. However, in practice, the sensor noise introduces significant errors. The x-axis noise characteristics for the sensor ADXL345 are shown in Figure 5.3.

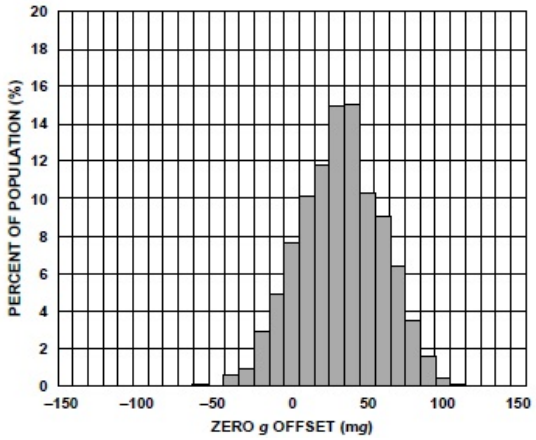


Figure 5.3. X-axis $0g$ Offset at $25°C$ [8]

The noise of the sensor almost follows a Gaussian distribution. We have measured the standard deviation of the accelerometer noise which is on the order of $0.1m/s^2$ by using samples when the accelerometer is held stationary. The necessity to integrate twice will cause larger changes in camera position estimation. Joshi et.al [23] proposed using an energy minimization framework to search for the optimal end point within a 1mm radius of the initially computed end point, subject to the constraints that the acceleration along that recovered path matches the measured accelerations best in the least-squares sense. However, this method is limited to exposure time. If the exposure time is too long, the end-point drift is more than a few centimeters and this method will take much more time. Instead, much simpler way of reducing acceleration noise is suggested. The previous work shows that conventional Butterworth [21] and Kalman filters [17] both work well at reducing Gaussian noise. Support for these filters rests in their ability to reduce high-frequency content, which is assumed to be noise-related. We have tried butterworth filters with several parameters varying, but they were generally found to reduce the signal's dynamic range too much. Kalman filter was also attempted without success. In the end, a 4^{th} order Butterworth filter with 3-dB frequency 200Hz[1] was found to reduce stationary noise without over-smoothing data.

5.4 Non-blind Deconvolution Algorithms

Once the blur kernel K is obtained, the latent image I can be estimated by using non-blind deconvolution algorithms. In this thesis, the maximum a posteriori method with Gaussian prior and sparse prior [25] are adopted to estimate the latent image I.

First, we reformulate the image blur model by replacing the convolution kernel K with a linear operator H.

$$B = HI + N \tag{5.15}$$

[1]The ODR of the accelerometer is set to 400Hz, so the spectrum bandwidth is 200Hz.

where the operator H is an $N \times N$ matrix and the images B and I are written as $N \times 1$ vectors. Then the maximum a posteriori explanation for I given B is:

$$I = \underset{I}{\operatorname{argmax}} P(I|B,H) \propto P(B|I,H)P(I) \tag{5.16}$$

Assuming the noise is an i.i.d Gaussian noise with variance σ^2, we can express the likelihood as:

$$P(B|I,H) \propto e^{-\frac{1}{2\sigma^2}\|B-HI\|^2} \tag{5.17}$$

To make the estimated image I more smooth, the latent image prior $P(I)$ is defined by using a set of filters G_k (in linear operator form).

$$P(I) = e^{-\alpha \sum_k \rho(G_k I)} \tag{5.18}$$

The simplest choice of filters G_k is the one-order horizontal and vertical derivative filters. It can also be useful to include second order derivatives. The function ρ is selected to be a sparse or heavy tailed function $\rho(z) = \|z\|^{0.8}$, or a Gaussian function $\rho(z) = \|z\|^2$. Taking the log of Equation 5.16, 5.17 and 5.18, the maximum a posteriori explanation for I reduces to minimizing:

$$\|B - HI\|^2 + \lambda \sum_k \rho(G_k I) \tag{5.19}$$

By minimizing Equation 5.19, we search for the I minimizing the reconstruction error $\|B - HI\|^2$ with the prior preferring I to be as smooth as possible.

Now we start with the simpler case, in which the image prior is Gaussian $\rho(z) = \|z\|^2$. By differentiating this equation to I and setting the derivative to zero, we can conclude that the optimal solution can be found by solving a sparse set of linear

equations: $Ax = b$ for

$$A = H^T H + \lambda \sum_k G_k^T G_k, b = H^T B \tag{5.20}$$

For solving this set of equations, explicitly inverting the matrix A is impractical. Because A is a square, symmetric and positive-definite matrix, the Conjugate Gradient method (CG) is selected to solve this problem. CG is the modification of the Steepest Descent method, which iteratively searches for the solution along the direction of steepest descent. The details about CG method is omitted here and only the final algorithm is given.

Algorithm 1 Conjugate Gradient Algorithm

Take $\forall x_0 \in \mathbb{R}^N$, calculate $r_0 = b - Ax_0$, let $p_0 = r_0$, set threshold $\epsilon > 0$, maximum number of iterations Maxit
for k = 0 to Maxit **do**
  $\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$, $x_{k+1} = x_k + \alpha_k p_k$, $r_{k+1} = r_k - \alpha_k A p_k$
  if $\|r_{k+1}\| < \epsilon$ **then**
    stop the loop
    $x = x_{k+1}$
  else
    $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$, $p_{k+1} = r_{k+1} + \beta_k p_k$
    $k = k + 1$
  end if
end for
$x = x_k$

If we use a sparse prior $\rho(z) = \|z\|^{0.8}$ instead of the Gaussian prior, the problem becomes minimizing:

$$\|B - HI\|^2 + \lambda \sum_k \|(G_k I)\|^{0.8} \tag{5.21}$$

which is no long a quadratic function. The minimization is performed using iteratively

reweighted least squares (IRLS) method [38]. Equation 5.21 can be reformulated as:

$$\|B - HI\|^2 + \lambda \sum_{k,x} w_x^k |(G_k I)_x|^2 \tag{5.22}$$

$$w_x^k = |(G_k I)_x|^{0.8-2} \tag{5.23}$$

where the image prior is calculated on each pixel location x. With this reformulation, the algorithm steps of IRLS is given below.

Algorithm 2 Iteratively Reweighted Least Squares Algorithm

Let $w_x^k = 1$ for all x, k
Set I_0 by minimizing Equation 5.22 with CG method, maximum number of iterations M
for iter = 1 to M **do**
  Update w_x^k using Equation 5.23 and I_0
  Update I_n by minimizing Equation 5.22 with CG method
  $I_0 = I_n$
end for
return I_0

From the algorithm above, it can be seen that w_x^k will come out higher for image locations in which the derivative of the current guess is too small, and lower when the derivative of the current guess is too high. Thus, the effect of the re-weighting process is to pull toward zero derivatives in image locations in which the current guess is smooth, and to pay the derivative penalty along the edges in the current guess [25].

5.5 Conclusion

This chapter gives the steps to estimate the blur kernel based on the measured acceleration data and describes how to recover the latent image from this estimated blur kernel and the observed blurred image. Next chapter will display the experimental results by using the methods in this chapter and the data from the hardware platform.

CHAPTER 6

EXPERIMENTAL RESULTS

This chapter displays some preliminary experimental results demonstrating how the raw image data and the acceleration data collected from the hardware platform can be used for image deblurring. Our result is also compared with those of Richardson-Lucy method, Fergus's method [16] and Cho's method [12].

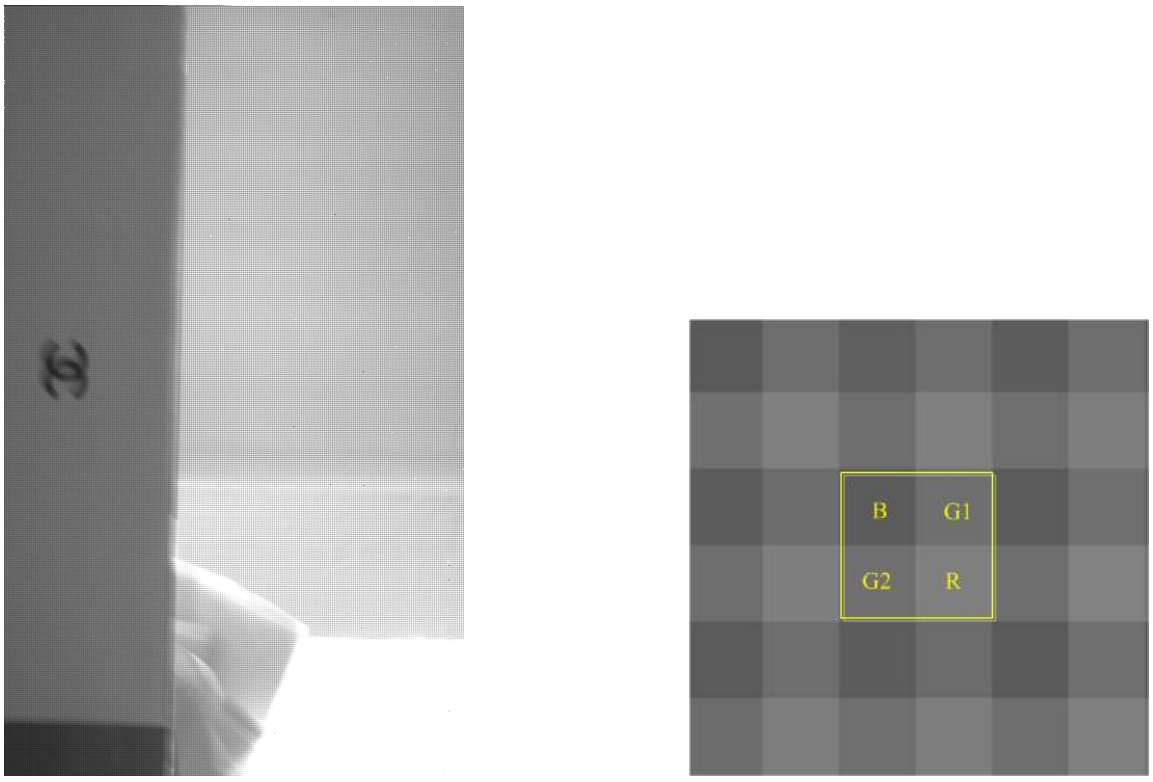
6.1 Data Collection

After downloading the files .sof and .elf into FPGA, we trigger one image by pressing *KEY[3]* on the VEEK-MT board. From the rising edge of the *TRIGGER* signal to the falling edge of the *FVAL* signal, the board is moved to produce a blurred raw image. Meanwhile, the accelerometer measures the 3-axis accelerations. According to the register configurations in the I^2C Sensor Configuration Module, the frame time t_{Frame}[1] is about 0.1879s and the exposure time t_{Exp}[2] is about 0.14195s. About 80 acceleration samples with sample interval 0.0025s are collected during t_{Frame}. However, we adopt ERS mode (i.e. rolling shutter mode) to take images, in which different rows are exposed at slightly different time points by $t_{Row} = 94.08us$. This means different rows correspond to different acceleration samples. Here we only consider the spatially-invariant blur kernel, so let the focused object in the image be constrained

[1] t_{Frame} is actually the interval between the rising edge of the *STROBE* signal and the falling edge of the *FVAL* signal.

[2] t_{Exp} is the interval between the rising edge and the falling edge of the *STROBE* signal referred to Subsection 3.2.6.

to a small region during the experiment. In this case, the blur kernels for each pixel of this object are almost the same. Figure 6.1(a) displays the raw image we take and the focused object is the Chanel logo. When we zoom in the image, we can clearly see that the pixels are arranged in *B, G1, G2, R* order, as shown in Figure 6.1(b).



(a) (b)

Figure 6.1. (a) Raw Image (b) Zoomed-in Pixels

6.2 PSF Estimation

The acceleration samples directly obtained from the accelerometer are not noise free so they must be filtered. As discussed in the previous chapter, a 4^{th} order Butterworth filter with 3-dB frequency 200Hz is selected to smooth the data. However, even with this filter, some data is still erroneous, so a threshold of discrimination between "valid data" and "invalid data" for the no movement condition must be implemented to remove mechanical noise. In the Main Function of the software frame, 100 acceleration samples are saved under stationary condition before the raw image is taken. We can calculate the standard deviation for each axis based on these samples. If one sample of some axis is smaller than the corresponding standard deviation, then it is regarded as a "valid data" for the no movement condition and set to 0. Figure 6.2 shows the acceleration data before and after these filtering schemes.

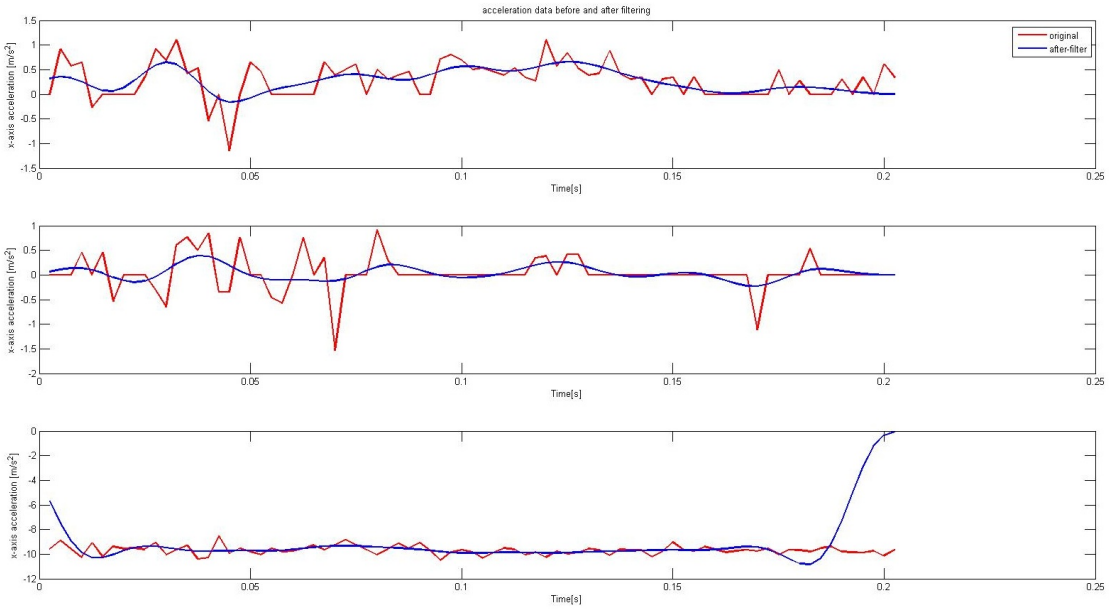


Figure 6.2. Acceleration Data Before and After Filtering

Note that during the experiment, z-axis is aligned with the gravity direction. As mentioned before, we assume the translation along z-axis is minimal. Experimentally this is found to be a safe assumption. In the third plot of Figure 6.2, the acceleration data of z-axis almost keeps constant $-9.81m/s^2$. In this case, we can directly ignore the data samples of z-axis and the gravity don't need to be subtracted from the acceleration data of x-axis and y-axis. In order to recover the Chanel logo accurately, only the first 65 samples of all 80 samples are used because the left samples are out of the exposure time of the Chanel logo. Now Equation 5.12 and Equation 5.13 are utilized to calculate the camera position at each time point t. The result is given in Figure 6.3.

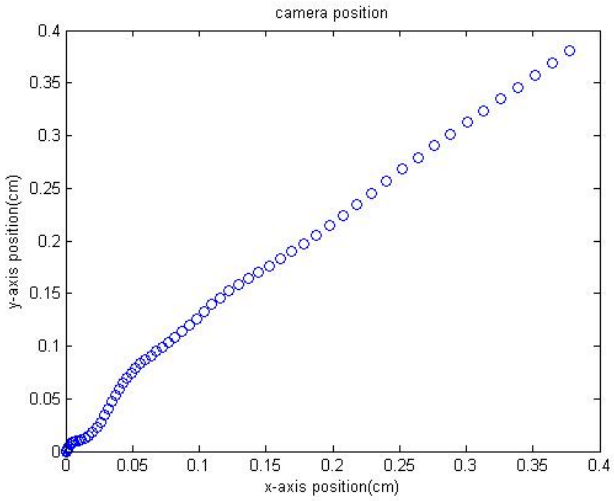


Figure 6.3. Camera Position at Each Time Point

Next the magnification parameter M need to be measured so that we can use Equation 5.7 to calculate the motion curve $C(t)$ on the image sensor. The CMOS

image sensor can be seen as a pinhole camera model, as shown in Figure 6.4. *H, h, D, d* are object size, image size, object distance and image distance respectively.

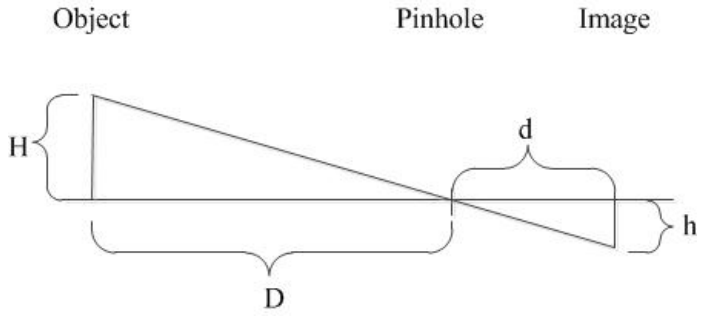


Figure 6.4. Pinhole Model of the Image Sensor

According to the principle of similar triangles,

$$\frac{H}{D} = \frac{h}{d} \tag{6.1}$$

The magnification M actually is d/D. To estimate d, we take several photos of a white board with a black square in the center at different distances D (Figure 6.5). The real size of the black square can be measured and its size on images can be read out as well. Thus, we can obtain a sequence of points with H/D as x and h as y and the polynomial fitting function in Matlab is adopted to estimate the slope d. The red circles in Figure 6.6 are original measured points and the blue line is the fitted line. The slope d is finally estimated to be 1639.8 pixels and the D for the Chanel raw image in Figure 6.1(a) is 21 inches. Then M is about 30.7424 pixel/cm.

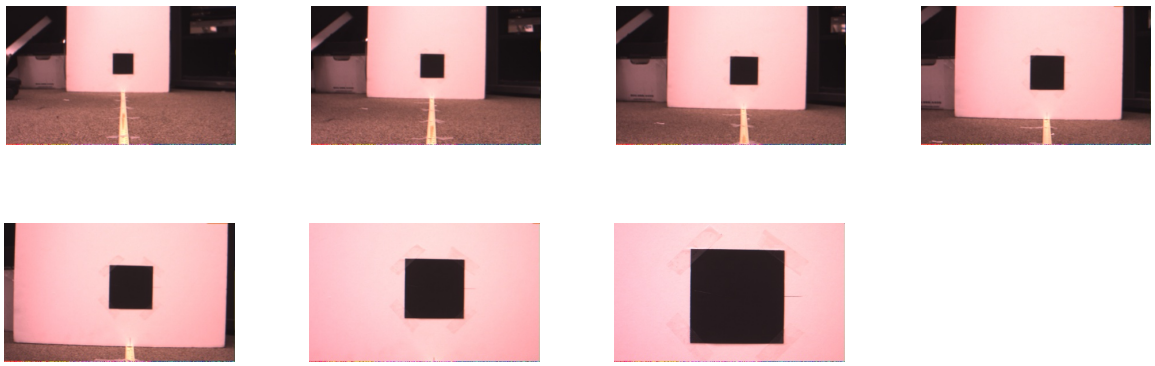


Figure 6.5. Images Taken for Measuring Magnification M

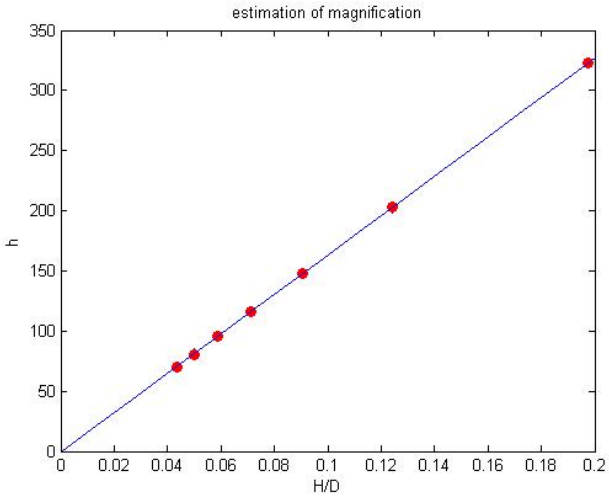


Figure 6.6. Estimate the Magnification M

Usually the points on the obtained motion curve $C(t)$ are very close to each oth-

er and most of them don't have integer coordinates, which causes us a problem to integrate the lens blur over the curve. So we up-scale all the points by 100 and interpolate 100 points between two original points. In this case, rounding the coordinates of these points can lead to less overlap. The final motion curve $C(t)$ is displayed in Figure 6.7.

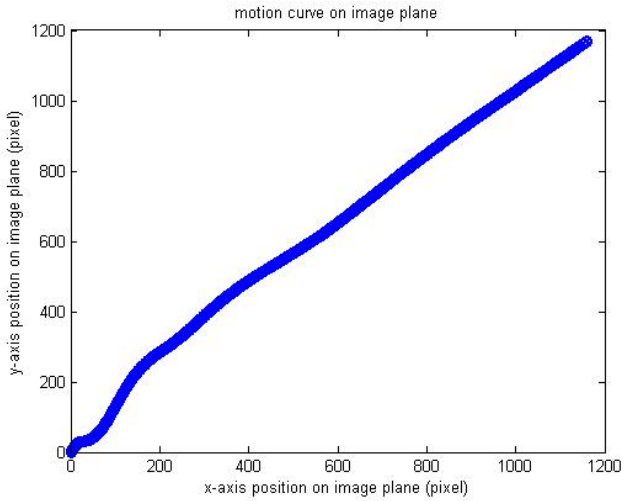


Figure 6.7. Motion Curve on Image Plane

For simplicity, s 7×7 Gaussian blur kernel with standard deviation 2 is found to be a good estimate of the lens blur, as shown in Figure 6.8(a). After integrating this Gaussian kernel over the motion curve $C(t)$, we still need to down-scale the estimated PSF to recover the right size. The down-scaling factor should be 1/100, but in fact some factors close to it can produce better results. This is because there always exists some errors in acceleration filtering, estimated magnification, object distance and so on. Note that the sum of the pixel values of PSF should equal to 1.

(a) (b)

Figure 6.8. (a) Gaussian Blur Kernel (b) Estimated PSF

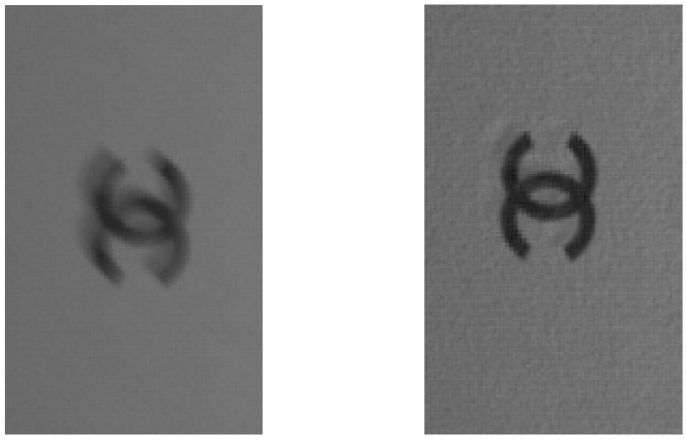
6.3 Image Deblurring

To test the validity of the PSF estimation and the captured data, we deblurr this captured raw image by first converting it to grayscale and applying the debblurring algorithm with the estimated PSF[3]. Figure 6.9(a) is the grayscale blurred image and (b) is the deblurred image using MAP with Gaussian prior. The algorithm of Liu et al. [28] is adopted to reduce the boundary artifacts in deconvolution process. The zoomed-in Chanel logo is shown in Figure 6.10.

[3]Converting to grayscale is performed using Matlab functions demosaic and rgb2gray.

(a) (b)

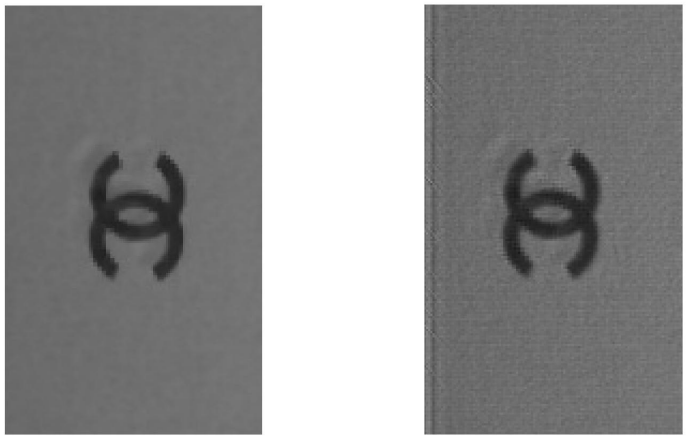
Figure 6.9. (a) Grayscale Blurred Image (b) Deblurred Image by Gaussian Prior



(a) (b)

Figure 6.10. (a) Zoomed-in blurred logo (b) Zoomed-in deblurred logo

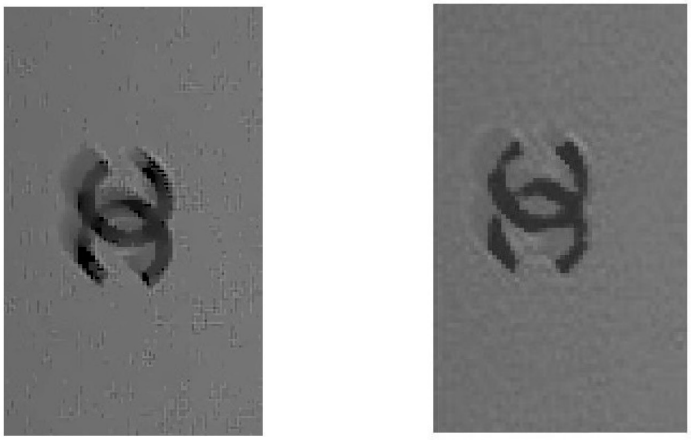
The deblurred results using sparse prior and Richardson-Lucy method are also given as comparisons.



(a) (b)

Figure 6.11. (a) Deblurred Image by Sparse Prior (b) Deblurred Image by Richardson-Lucy

To show the advantages of our method, we apply the blind deblurring algorithms of Cho et al. [12] and Fergus et al. [16] on the grayscale blurred image.



(a) (b)

Figure 6.12. (a) Deblurred Image by Cho et al. (b) Deblurred Image by Fergus et al.

From Figure 6.9 and Figure 6.10, it is obvious that our deblurred method almost removes the motion blur of the logo though there is still some residually ringing that is unavoidable due to high frequency loss during blurring. In addition, comparing Figure 6.10(b) and Figure 6.11(a) with Figure 6.11(b), we can see that the deconvolution algorithms with image prior can generate more natural and smooth images than the Richardson-Lucy method. Moreover, the comparison with the results in Figure 6.12 proves that utilizing the acceleration data to deblur the raw image data can perform better than the leading blind deblurring algorithms.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, we began by reviewing the existing image deblurring algorithms and analyzing the advantages and problems of each algorithm. For camera motion blur problem, the joint hardware and software approach, especially utilizing inexpensive inertial sensors for PSF estimation, generally can yield good results. Based on this approach, the raw image was considered to replace those common format images to enhance the deblurred result. This is because the raw image records the minimally processed data from the image sensor so that both PSF estimation from inertial sensor data and non-blind deconvolution algorithms can perform better.

To test this idea, an imaging system was built to access raw image data and acceleration data. This system was constructed on the Altera FPGA board VEEK-MT, in which the TRDB-D5M modeled the CMOS image sensor, the LCD screen displayed the taken image and the accelerometer ADXL345 recorded the 3-axis acceleration data. Based on the development flow, we described the design and implementation of each module in hardware frame and software frame respectively.

The motion blur process was typically modeled as a convolution of a sharp image and a blur kernel. The acceleration data obtained from the imaging system was used to form the camera motion curve and thus estimate the blur kernel (PSF). Note that several assumptions were given to simplify the PSF estimation process. The estimated PSF was then used in a maximum a posteriori restoration algorithm to recover the deblurred image.

Finally, the data collected from the built imaging system was adopted as an

example to show that the method proposed in this thesis can produce better results than that of the advanced blind deblurring algorithms under some assumptions.

In future, we would like to extend our current work in four aspects. First, better filter schemes or optimization framework should be selected to reduce drift caused by acceleration data noise. Second, we could utilize the timing correspondence between the acceleration samples and exposure time of each row to estimate spatially-varying PSF. Third, joint deblurring and demosaicing of the raw image can be considered. Finally, more sensors, such as gyroscopes, kinect and light sensor, can be added to the current hardware platform to provide more information for PSF estimation.

BIBLIOGRAPHY

1. JamesE Adams Jr, Aaron Deever, John F Hamilton Jr, Mrityunjay Kumar, Russell Palum, and Bruce H Pillman. Single capture image fusion with motion consideration. *Computational Photography: Methods and Applications*, 2:63, 2010.

2. Altera Corporation, San Jose, CA. *Introduction to Quartus II*, v.4.1 rev.1 edition, 06 2004.

3. Altera Corporation, San Jose, CA. *RAM-based shift register megafunction user guide*, v.10.1 edition, 11 2010.

4. Altera Corporation, San Jose, CA. *Embedded peripherals IP user guide*, v.11.0 edition, 06 2011.

5. Altera Corporation, San Jose, CA. *HAL API Reference*, v.11.0.0 edition, 05 2011.

6. Altera Corporation, San Jose, CA. *Design debugging using the SignalTap II logic Analyzer*, v.12.0.0 edition, 06 2012.

7. Altera Corporation, San Jose, CA. *SCFIFO and DCFIFO megafunctions user guide*, v.7.0 edition, 02 2012.

8. Analog Devices, Inc., Norwood, MA. *Digital accelerometer ADXL345*, rev.a edition, 04 2010.

9. Bob Atkins. Raw, jpeg and tiff. `http://photo.net/learn/raw`, 06 2008.

10. Hyeoungho Bae, Charless C Fowlkes, and Pai H Chou. Accurate motion deblurring using camera motion tracking and scene depth.

11. Leah Bar, Nahum Kiryati, and Nir Sochen. Image deblurring in the presence of impulsive noise. *International Journal of Computer Vision*, 70(3):279–298, 2006.

12. Sunghyun Cho and Seuongyong Lee. Fast motion deblurring. *ACM Transactions on Graphics (SIGGRAPH ASIA 2009)*, 28(5):article no. 145, 2009.

13. Sunghyun Cho, Jue Wang, and Seungyong Lee. Handling outliers in non-blind image deconvolution. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 495–502. IEEE, 2011.

14. Pong P Chu. *Embedded SoPC Design with Nios II Processor and VHDL Examples*. Wiley Online Library, 2011.

15. Altera Corporation. Interface protocols. ftp://ftp.altera.com/up/pub/Altera_Material/12.0/Laboratory_Exercises/Embedded_Systems/DE2/embed_lab9.pdf, 09 2011.

16. Rob Fergus, Barun Singh, Aaron Hertzmann, Sam T Roweis, and William T Freeman. Removing camera shake from a single photograph. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 787–794. ACM, 2006.

17. Antonio Filieri and Rossella Melchiotti. Position recovery from accelerometric sensors: algorithms analysis and implementation issues. http://home.dei.polimi.it/bellasi/lib/exe/fetch.php?media=students:filieri-melchiotti_projectreport.pdf, 05 2012.

18. Christopher J. Fisher, Tomoaki Tsuzuki, and James Lee. Utilization of the first in, first out (fifo) buffer in analog devices, inc. digital accelerometer. Technical report.

19. Jinwei Gu, Yasunobu Hitomi, Tomoo Mitsunaga, and Shree Nayar. Coded rolling shutter photography: Flexible space-time sampling. In *Computational Photography (ICCP), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

20. Ankit Gupta, Neel Joshi, C Lawrence Zitnick, Michael Cohen, and Brian Curless. Single image deblurring using motion density functions. *Computer Vision–ECCV 2010*, pages 171–184, 2010.

21. Roarke Horstmeyer. Camera motion tracking for deblurring and identification. http://web.media.mit.edu/~roarkeh/cameramotion_RWH.pdf, 05 2010.

22. Integrated Silicon Solution, Inc., San Jose, CA. *64M×8, 32M×16 synchronous dram*, rev.d edition, 11 2009.

23. Neel Joshi, Sing Bing Kang, C Lawrence Zitnick, and Richard Szeliski. Image deblurring using inertial measurement sensors. *ACM Transactions on Graphics (TOG)*, 29(4):30, 2010.

24. Neel Joshi, Richard Szeliski, and David J Kriegman. Psf estimation using sharp edge prediction. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

25. Anat Levin, Rob Fergus, Frédo Durand, and William T Freeman. Deconvolution using natural image priors. *Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory*, 2007.

26. Anat Levin, Rob Fergus, Frédo Durand, and William T Freeman. Image and depth from a conventional camera with a coded aperture. *ACM Transactions on Graphics (TOG)*, 26(3):70, 2007.

27. Anat Levin, Yair Weiss, Fredo Durand, and William T Freeman. Understanding and evaluating blind deconvolution algorithms. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1964–1971. IEEE, 2009.

28. Renting Liu and Jiaya Jia. Reducing boundary artifacts in image deconvolution. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pages 505–508. IEEE, 2008.

29. LB Lucy. An iterative technique for the rectification of observed distributions. *The astronomical journal*, 79:745, 1974.

30. Multek Display. *7" LCD capacitive touch screen*, rev.1 edition, 03 2011.

31. SK Nayar and M Ben-Ezra. Motion-based motion deblurring. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(6):689–698, 2004.

32. Michael Reichmann. Understanding raw files. `http://www.luminous-landscape.com/tutorials/understanding-series/u-raw-files.shtml`, 05 2004.

33. William Hadley Richardson. Bayesian-based iterative method of image restoration. *JOSA*, 62(1):55–59, 1972.

34. SanDisk Co., Sunnyvale, CA. *SanDisk secure digital card product manual*, v.1.9 edition, 12 2003.

35. Kurt Seifert and Oscar Camacho. Implementing positioning algorithms using accelerometers. *Freescale Semiconductor*, 2007.

36. Qi Shan, Jiaya Jia, and Aseem Agarwala. High-quality motion deblurring from a single image. In *ACM Transactions on Graphics (TOG)*, volume 27, page 73. ACM, 2008.

37. Ondřej Šindelář and Filip Šroubek. Image deblurring in smartphone devices using built-in inertial measurement sensors. *Journal of Electronic Imaging*, 22(1):011003–011003, 2013.

38. Charles V Stewart. Robust parameter estimation in computer vision. *Siam Review*, 41(3):513–537, 1999.

39. Terasic Technologies Inc., Dover, DE. *Terasic TRDB-D5M hardware specification*, v0.2 edition, 06 2009.

40. Terasic Technologies Inc., Dover, DE. *Video and embedded evaluation kit–Multitouch user manual*, v1.0 edition, 2011.

41. Terasic Technologies Inc., Dover, DE. *DE2-115 user manual*, v1.04 edition, 2012.

42. Tomoaki Tsuzuki. Adxl345 quick start guide. Technical report.

43. Oliver Whyte, Josef Sivic, Andrew Zisserman, and Jean Ponce. Non-uniform deblurring for shaken images. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 491–498. IEEE, 2010.

44. Jiagu Wu, Huajun Feng, Zhihai Xu, Qi Li, and Zhongliang Fu. Method to detect and calculate motion blur kernel. In *5th International Symposium on Advanced Optical Manufacturing and Testing Technologies*, pages 76566I–76566I. International Society for Optics and Photonics, 2010.

45. Mengfeng Yu, Ming Yang, and Xu Wu. Design of ultra-high-resolution photograph system based on fpga. *Journal of Ningbo Univeristy (NSEE)*, 24(1):30–33, 2011.

46. Lu Yuan, Jian Sun, Long Quan, and Heung-Yeung Shum. Image deblurring with blurred/noisy image pairs. In *ACM Transactions on Graphics (TOG)*, volume 26, page 1. ACM, 2007.

This document was prepared & typeset with pdfLaTeX*, and formatted with* NDdiss2$_\varepsilon$ *classfile (v3.2013β[2013/01/31]) provided by Sameer Vijay and updated by Megan Patnott.*