DESIGN OF AN INFRARED PROJECTOR FPGA-BASED COMPUTER ARCHITECTURE

by

Robert Hampton Hill Haislip

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Masters of Science in Electrical and Computer Engineering

Spring 2012

© 2012 Robert Hampton Hill Haislip
All Rights Reserved

Approved for Public Release - Distribution is Unlimited

DESIGN OF AN INFRARED PROJECTOR FPGA-BASED COMPUTER ARCHITECTURE

by

Robert Hampton Hill Haislip

Approved: _____
Fouad E. Kiamilev, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: _____
Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

Approved for Public Release - Distribution is Unlimited

ACKNOWLEDGMENTS

First and foremost, I thank the Lord for blessing me with the opportunity to write a master's thesis, as the Word says, "Every good and perfect gift is from above" (James 1:17). I believe He used my advisor and dear friend Fouad Kiamilev to provide continual support and encouragement as I pushed through the difficulties I faced daily in engineering. I thank Fouad for being so understanding of my schedule and life outside of the lab - I could not have stayed on the engineering path, if it were not for this. Fouad makes sure the lab environment is family-oriented; meaning, the doctoral students take care of the master's students, who take care of the undergraduates, so that along the way, the students themselves become a parent or child to another student. Having this naturally growing lab environment makes working with other students both challenging and exciting, and I thank Fouad again for making me a part of the experience.

Nicholas Waite is one of the greatest assets in CVORG, and I have to especially thank him for his substantial help in this project. He is one of the master minds behind the design of this highly complex computer architecture, and I thank him for sitting by my side in the wee small hours of the morning, helping me fix the bugs found in my VHDL code.

Rodney McGee is one of the best engineers and friends I have ever had the privilege to work with. I thank him for not only significantly contributing in the develop of the SLEDs system, but also for his effective management skills to utilize the gifts of all the CVORG members and attack the SLEDs project most efficiently. Corey Lange is another highly motivated engineer that showed me not only the

Approved for Public Release - Distribution is Unlimited

basics of PCB, VLSI, and VHDL design, but how to use them effectively. He and Rodney helped me get on the right foot whenever I became disoriented from the frustrations of slow progress.

I would also like to thank the SLEDs team from the University of Iowa: Ed Koerperick, Dennis Norton, and especially Thomas Boggess, the principal investigator of the project, for his management and encouragement to make it through this phase of the project. In addition, John Lawler of ATEC for his knowledge of thermals in IC design. I thank everyone from CVORG who has worked on SLEDs over the years: Ryan Hoover, Joshua Marks, Robert Rehrig, Furkan Cayci, Nicole Wells, and David Koeplinger.

Last, but certainly not least, I would like to thank my Mom & Dad for their love and support in my life. I would have never been able to pursue the opportunities available at the University of Delaware if it were not for you. In fact, I would have never met my soon-to-be wife Yael Hernandez, the greatest blessing of my life.

This project is funded by the Test Resource Management Center (TRMC) Test and Evaluation/Science & Technology (T&E/S&T) Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) under Contract No. W91ZLK-06-C-0006.

Approved for Public Release - Distribution is Unlimited

TABLE OF CONTENTS

LIST OF FIGURES . . . . . **viii**
LIST OF TABLES . . . . . **x**
LIST OF LISTINGS . . . . . **xi**
ABSTRACT . . . . . **xii**

Chapter

1 INTRODUCTION **1**

1.1 Background 1
1.2 Motivation 1
1.3 Preliminary Test Setup 2

2 SERIAL INTERFACE ARCHITECTURE **6**

2.1 Purpose 6
2.2 UART 6

2.2.1 UART Decoder 8
2.2.2 UART Encoder 8
2.2.3 UART External/Internal Connections 10

3 INTERNAL BUS ARCHITECTURE **11**

3.1 Bus Overview 11
3.2 Front Side Bus 11

3.2.1 FSB Example 13

3.3 Back Side Bus 15

Approved for Public Release - Distribution is Unlimited

4 PYTHON SCRIPTS 17

4.1 Purpose 17
4.2 Overview 17
4.3 Foundational Functions 18

4.3.1 SLEDs Serial Interface 18
4.3.2 Basic Register Operations 19
4.3.3 BRAM Memory Test 20

4.4 Loading and Running a SLEDs Program 20

5 PIXEL WRITE MODULE 22

5.1 Purpose 22
5.2 Overview 22
5.3 Timer Core Module 24
5.4 SLEDs Addresser Module 25
5.5 DAC Driver Module 27
5.6 Same/Change Pixel Value 29

5.6.1 Implementation 30

5.7 Test of Pixel Write Module 31

6 CONCLUSIONS 34

6.1 SLEDs Testing at Eglin 34

6.1.1 Full Array Yield Test 36
6.1.2 Dysfunctional Corner 38
6.1.3 Row/Column Test 39
6.1.4 Diagonal Test 40
6.1.5 High-Density Coverage Test 41
6.1.6 Corner Arrow Test 42

7 FUTURE WORK 43

7.1 System Upgrade 43

Approved for Public Release - Distribution is Unlimited

7.2 New FPGA Evaluation Board 45

BIBLIOGRAPHY 46

Approved for Public Release - Distribution is Unlimited

LIST OF FIGURES

1.1 Cryogenic Dewar to Test SLEDs 2

1.2 PCB Carrier Mounted to Back of Dewar 3

1.3 SLEDs Full View 4

1.4 SLEDs Connections Overview 5

2.1 BRAM Bit Mapping 7

2.2 UART Encoder Diagram 9

2.3 Front Side Bus and Registers 10

3.1 Register-to-Bus Overview 12

3.2 Front Side Bus (FSB) Example 14

3.3 SLEDs Opcode Instructions 16

5.1 Pixel Write Block Diagram 23

5.2 Sequencing to Successful Pixel Lighting [1] 24

5.3 Logic Analyzer View of Timer Core Signals 26

5.4 SLEDs Addresser Routes FPGA Signals to 40-Pin Expansion Port [2] 27

5.5 DAC Evaluation Board [3] 28

5.6 Same/Change Pixel Value Algorithm 30

Approved for Public Release - Distribution is Unlimited

5.7 Writing 0xAAAA to SLEDs NW Corner 31

5.8 Same/Change DAC Loop Test 33

6.1 Control Electronics Setup for Eglin Testing 35

6.2 SLEDs Carrier Package in Dewar for Eglin Testing 35

6.3 Full Array Yield Test 36

6.4 Missing Column Explanation 37

6.5 Thermal of Gouge Mark 38

6.6 Gouge Mark on Surface 38

6.7 Row/Column Test 39

6.8 Diagonal Test 40

6.9 High-Density Coverage Test 41

6.10 Corner Arrow Test Pictures 42

7.1 Future SLEDs Hardware Setup 44

7.2 The Spartan-3A DSP FPGA Video Starter Kit [4] 45

Approved for Public Release - Distribution is Unlimited

LIST OF TABLES

4.1 Python Headers for Front Side Registers 17

4.2 Pixel Time Default Values 21

Approved for Public Release - Distribution is Unlimited

LIST OF LISTINGS

4.1 Pixel Write Time Registers 18

4.2 SLEDs Serial Interface Class 19

4.3 Write & Read From Registers 19

4.4 Simple Memory Test 20

4.5 Loading & Running a SLEDs Program from BRAM 21

5.1 Python Code for Same/Change DAC Loop Test 32

6.1 Python Code for Drawing Full Grid 36

6.2 Python Code for Writing R/C Lines 39

6.3 Python Code for Drawing Diagonal Line 40

6.4 Python Code for High-Density Coverage Test 41

6.5 Python Code for Drawing Corner Arrows 42

Approved for Public Release - Distribution is Unlimited

ABSTRACT

The military uses large Infrared (IR) projector arrays to test its detector arrays. The IR projector generates scenes of moving objects and standing landscapes for hardware in the loop (HWIL) sensor testing to simulate real world scenarios. The IR projector is a two-dimensional array of superlattice light-emitting diodes (SLEDs) with a custom driver IC and FPGA-based control system.

The purpose of this thesis is to document the custom Field Programmable Gate Array (FPGA) computer architecture that drives the SLEDs. This framework is able to convert standard PC simulations to the custom SLEDs protocol using precise timing to control addresses, luminosity, and framing of the SLEDs. The work described in this paper examines the computer architecture in detail and the related testing completed on the current 512 x 512 SLEDs system.

[1]

[1] Approved for Public Release - Distribution is Unlimited

Approved for Public Release - Distribution is Unlimited

Chapter 1

INTRODUCTION

1.1 Background

The military uses large Infrared (IR) projector arrays to test its detector arrays. The IR projector generates scenes of moving objects and standing landscapes for hardware in the loop (HWIL) sensor testing to simulate real world scenarios [5]. The HWIL system requires an array made up of 512 x 512 pixels refreshed at a high frame rate.

The IR projector is a two-dimensional array of superlattice light-emitting diodes (SLEDs) with a custom driver IC and FPGA-based control system. For a detailed explanation of the physical SLEDs design, see Corey Lange's paper entitled: "Design and Development of a 512 x 512 Infrared Emitter Array System" [6]. The purpose of this thesis is to document the custom Field Programmable Gate Array (FPGA) computer architecture that drives the SLEDs. This framework is able to convert standard PC simulations to the custom SLEDs protocol using precise timing to control addresses, luminosity, and framing of the SLEDs. The work described in this paper examines the computer architecture in detail and the related testing completed. In addition, future work for optimizing this generation of the SLEDs project is discussed.

1.2 Motivation

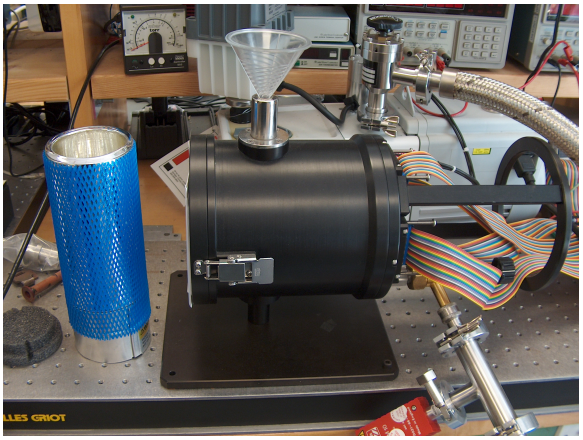
The VLSI design of the 512 x 512 SLEDs was near completion prior to the start of this thesis; however, the framework to electronically control the driver array

Approved for Public Release - Distribution is Unlimited

and associated electronics was lacking. During the months developing the VHSIC Hardware Description Language (VHDL) code, the SLEDs were bump-bonded to the IC LED driver and eventually wire-bonded to a custom PCB, which adapted the SLEDs microchip pads to standard headers and made a connection to the FPGA hardware possible. This status in the physical SLEDs development motivated quick progress in the design and implementation of the computer architecture needed to utilize them.

1.3 Preliminary Test Setup

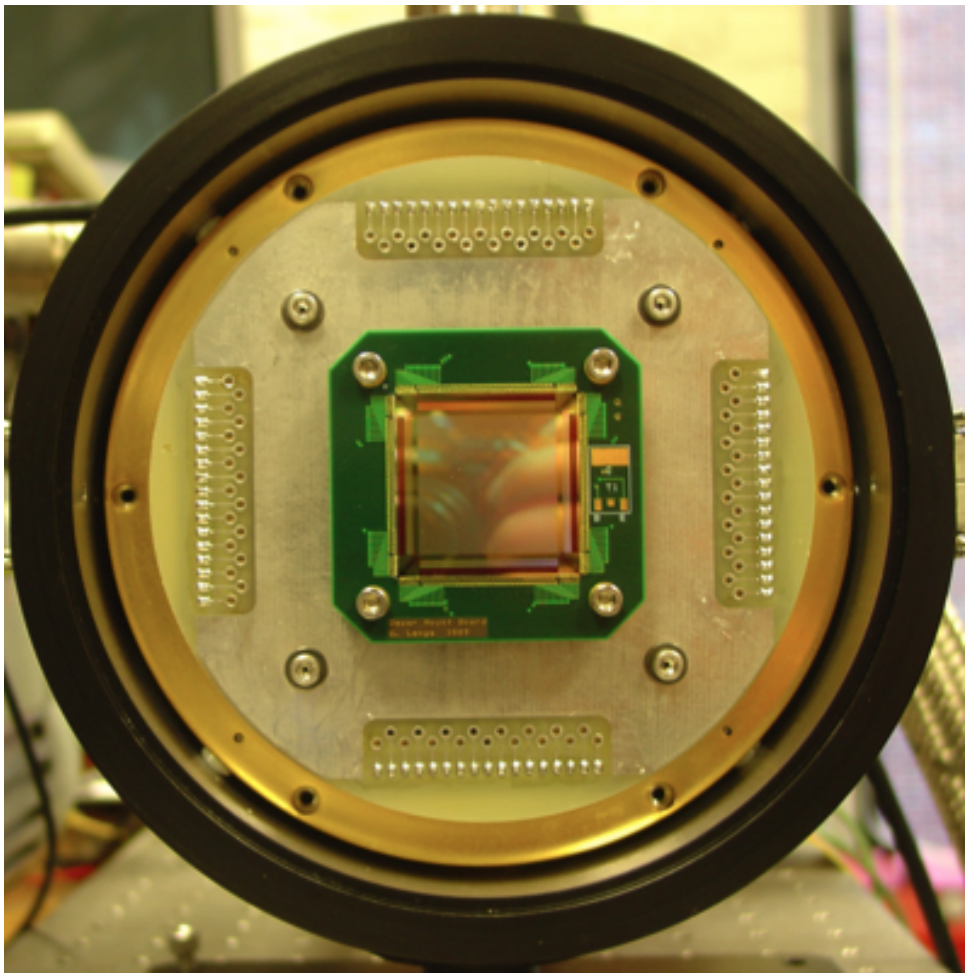
Figure 1.1: Cryogenic Dewar to Test SLEDs



To test functionality and yield, the SLEDs are mounted in a 4-inch diameter cryogenic dewar and wired to FPGA control signals. Using 512 x addresses and 512 y addresses would require too many pads and pins to fit inside the dewar and is impractical given the number of available GPIO on the FPGA. See Figure 1.1 for a picture of the test setup. Instead, a log-2 decoding scheme is implemented, requiring only 9 signals per side: 8 x (or 8 y) and 1 quadrant selection. As explained in Corey Lange's paper [7], the 512 x 512 SLEDs is made up of 4 smaller 256 x 256 chips that have been stitched together symmetrically. See Figure 1.3(a).

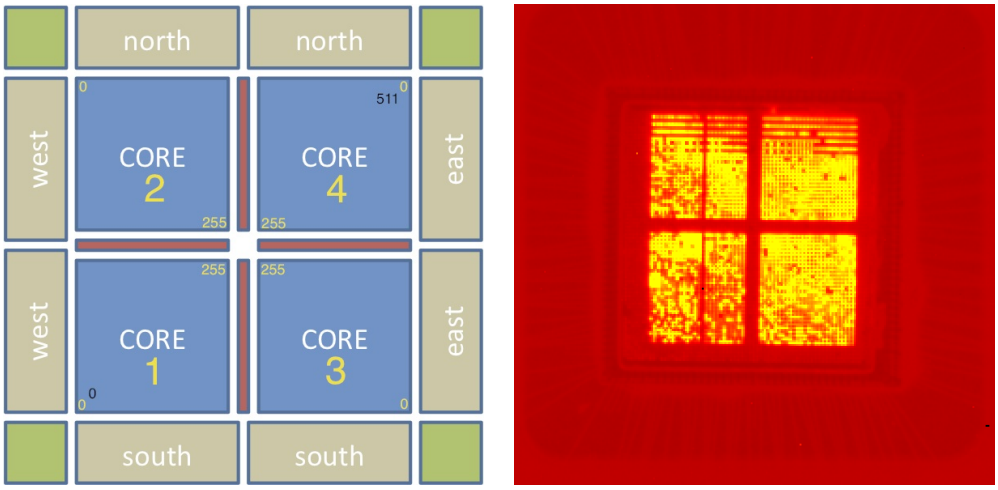
Approved for Public Release - Distribution is Unlimited

Figure 1.2: PCB Carrier Mounted to Back of Dewar



Approved for Public Release - Distribution is Unlimited

Figure 1.3: SLEDs Full View



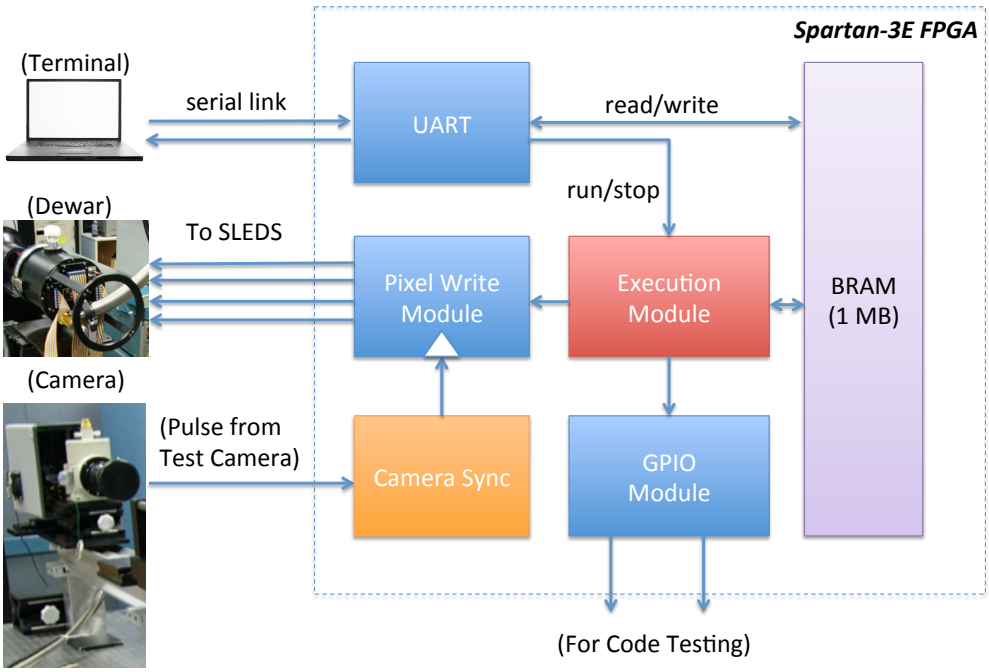
(a) Map of SLEDs Quadrants

(b) IR Image of SLEDs Quadrants

This decoding scheme is paramount in the VHDL design. Each quadrant needs to be intentionally enabled so that drawing an image will make the four corners seamless, as if there was one chip. To effectively move between quadrants while selecting pixels, the control electronics need to have a wide range of utilities. The Xilinx Spartan-3E evaluation board is used as the central control electronics, which includes a Spartan-3E FPGA with a 50MHz clock, 40-pin GPIO expansion port, 8 onboard LEDs for debugging, and 4 onboard switches. As shown in Figure 1.4, the SLEDs testing setup connects the Spartan-3E evaluation board to a dewar interface board, which routes signals to (x, y) addresses and other controls signals directly to the dewar.

Approved for Public Release - Distribution is Unlimited

Figure 1.4: SLEDs Connections Overview



Approved for Public Release - Distribution is Unlimited

Chapter 2

SERIAL INTERFACE ARCHITECTURE

2.1 Purpose

A unique serial interface is necessary to communicate between the workstation terminal and the FPGA that drives the control signals to the SLEDs. The purpose of this chapter is to describe the VHDL architecture of that interface.

2.2 UART

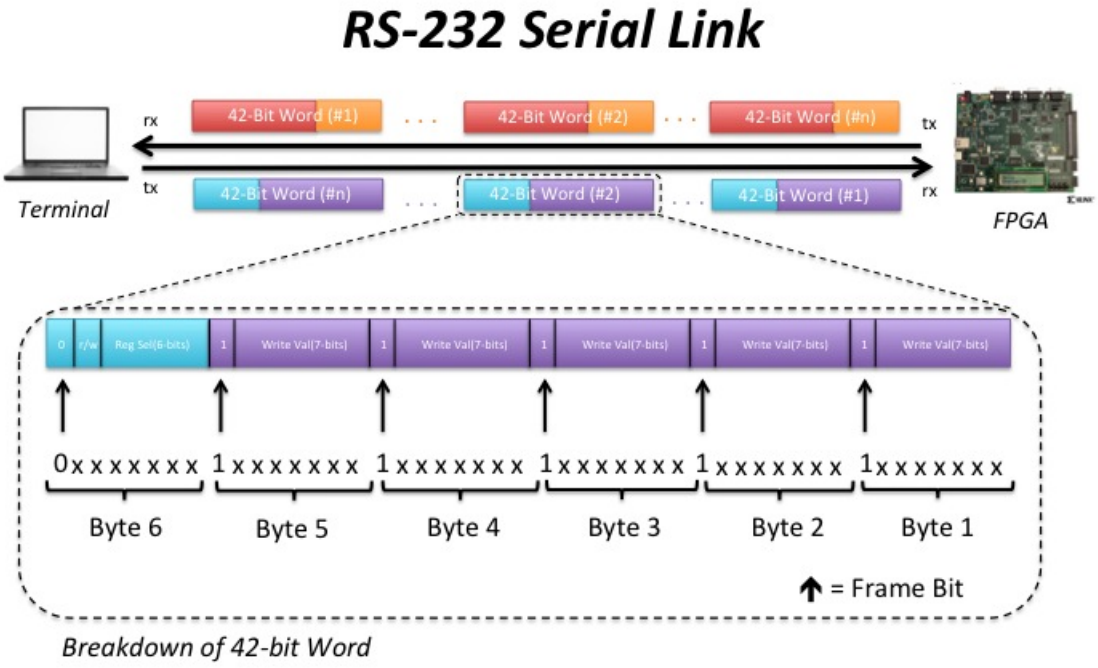
The Universal Asynchronous Receiver/Transmitter (UART) is a VHDL module that sends parallel data through a serial line. UARTs are commonly used in conjunction with EIA (Electronic Industries Alliance) RS-232 standard, which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment [8]. In the SLEDs architecture, the UART module needs not only parallel to serial conversion, but special decoding and encoding modules to process the data format. The basic framework for the UART was taken from one of Pong P. Chu's examples written in *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version* [8], but the decoding and encoding units were made custom.

To run tests on the SLEDs, Python-generated programs are loaded in real time to the FPGA framework using this UART design. More information on the Python programming process can be found in Chapter 4. The SLEDs data is fed from the terminal to the FPGA in the form of a 42-bit word that includes the following key pieces of information: front side read/write (1 bit), front side register

Approved for Public Release - Distribution is Unlimited

address (6 bits), and the write value (35 bits). As shown in Figure 2.1, the write value is the BRAM word with a width of 35 bits: 3 bits for opcode selection, 8 bits for x address, 8 bits for y address, and 16 bits for the DAC value. Then, 6 bits are needed to select the front side register address (selecting either the pixel write module, execution module, BRAM access module, or GPIO module). The final bit selects whether the user is writing to a front side register or reading from it. For more information on these modules, see Chapter 3.

Figure 2.1: BRAM Bit Mapping



Approved for Public Release - Distribution is Unlimited

2.2.1 UART Decoder

To decrypt these 42-bit words, a VHDL decoder module uses combinational logic to organize the incoming bytes from the UART. Since the arrival of bytes from the receiver must be in order to properly use them, the UART decoder first checks to make sure a defined sequence, or frame, is present as 6 bytes are passed into the FIFO buffer. To indicate the start of a frame, the first byte reserves its most significant bit (MSB) to be '0'. The other five bytes must reserve their MSBs to be '1'.

As shown in Figure 2.1, the 42-bit word is composed of 6 bytes (6 bytes x 8 bits/byte = 48 bits), using 6 bits (48 bits - 6 bits = 42-bit word) to identify each of the 6 bytes. The first byte is the address byte, which combines the front side read/write bit with the 6 front side register address bits, and the second through sixth are data bytes. The address byte reserves its MSB to be '0', and the data bytes reserve their MSB to be '1'.

Once a valid frame occurs (i.e. reading from right to left, the decoder receives: 0xxxxxxx, 1xxxxxxx, 1xxxxxxx, 1xxxxxxx, 1xxxxxxx, 1xxxxxxx) in the FIFO buffer, the decoder module will place the 42-bit word on the front side bus (FSB) and set the FSB busy flag to active. Depending on the front side read/write bit, the FSB will be set to either read or write mode. Also, since bit errors occur during transmission along the tx/rx wires, this frame validation doubles as a simple 1-bit checksum.

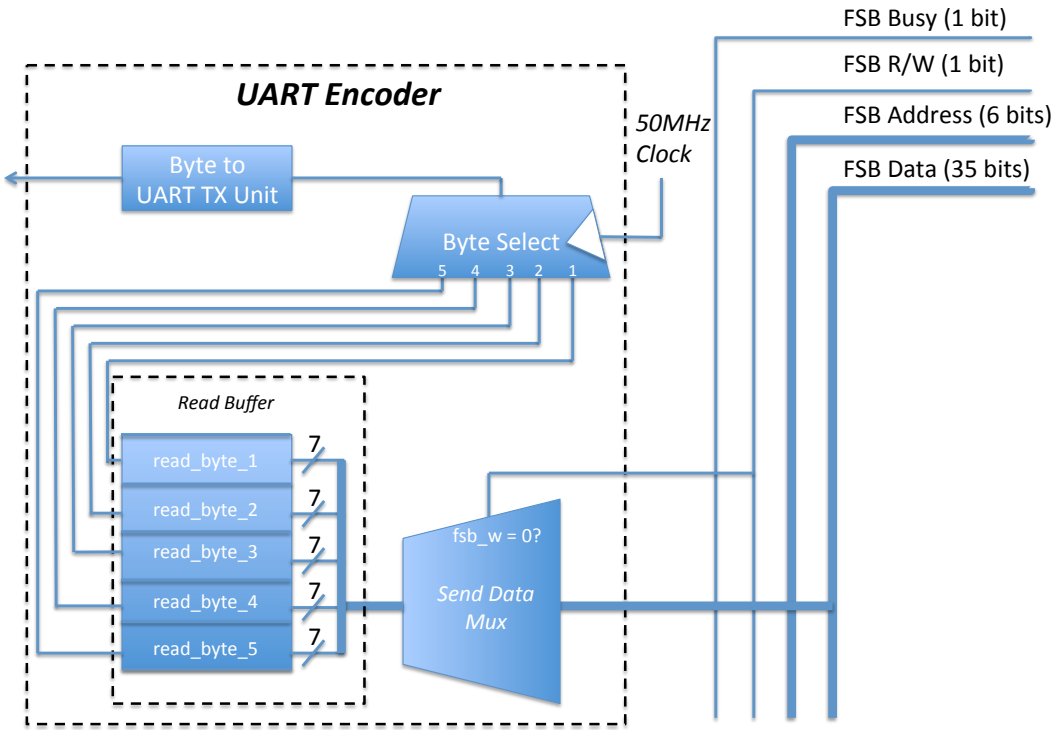
2.2.2 UART Encoder

The encoder unit is designed for the FPGA to format the 42-bit word requested from the terminal when a read command is sent. To send a read command, the terminal must send the first byte with the front side read/write bit set to '0' (i.e. 00xxxxxx 1xxxxxxx 1xxxxxxx ...). When the front side bus is set to read mode, the 42-bit word located at the 6-bit front side address is placed on the front side bus.

Approved for Public Release - Distribution is Unlimited

The encoder unit waits for these conditions to be met and then grabs the data from the bus and feeds it to a byte-converter finite state machine (FSM). See Figure 2.2 for a detailed diagram of the encoding process. The byte-converter FSM shifts the 42-bit word by 7 bits every clock cycle, tags a '1' to the MSB, and feeds it to the UART tx unit, which serializes the 8-bit parallel data for RS-232 communication to the terminal. Since the word is 42 bits wide, the entire encoding process takes (42 bits) / (7 bits/clock cycle) = 6 clock cycles, which with a 50MHz clock is (6 cycles)/ ($50x10^6$(cycle/sec)) = 12 microseconds.

Figure 2.2: UART Encoder Diagram

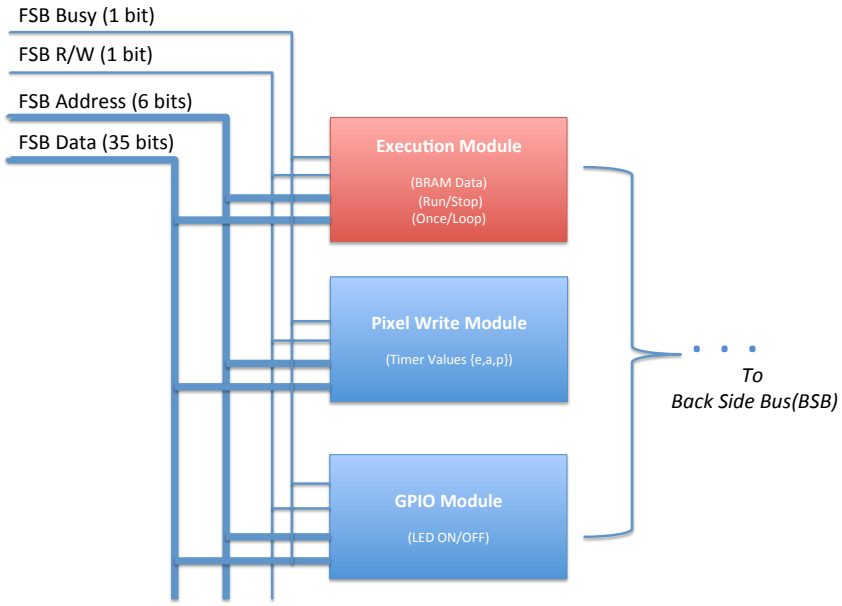


Approved for Public Release - Distribution is Unlimited

2.2.3 UART External/Internal Connections

Internally, the UART decoder and encoder units both connect to the front side bus (FSB), a communications interface with direct access to registers. After the UART data is decoded, the FSB becomes busy with the decoded data present on the wire. On the front side bus, registers are able to be written to and read from directly from a workstation terminal command at any stage in the SLEDs program. The program can be run once, looped, or stopped at any time. As shown in the "SLEDs Connections Overview" diagram from Chapter 1, the three main units in the SLEDs framework are the pixel write module, execution module, and GPIO module, each of which use the front side and back side bus to store, load, and run commands.

Figure 2.3: Front Side Bus and Registers



Approved for Public Release - Distribution is Unlimited

Chapter 3

INTERNAL BUS ARCHITECTURE

3.1 Bus Overview

The SLEDs FPGA registers use both front side bus (FSB) and back side bus (BSB) computer communication interfaces, each with different purposes. The FSB and BSB names and concepts originated from Intel Corporation, and were originally used for connecting an Intel processor to the rest of the computer system [9]. The SLEDs application of the FSB, as briefly mentioned in Chapter 2, is designed for immediate communication with the UART decoder and encoder. The BSB is designed as an interface between the BRAM and pixel write module. The following sections will go into detail of these bus system operations, but first it is important to understand the overarching process.

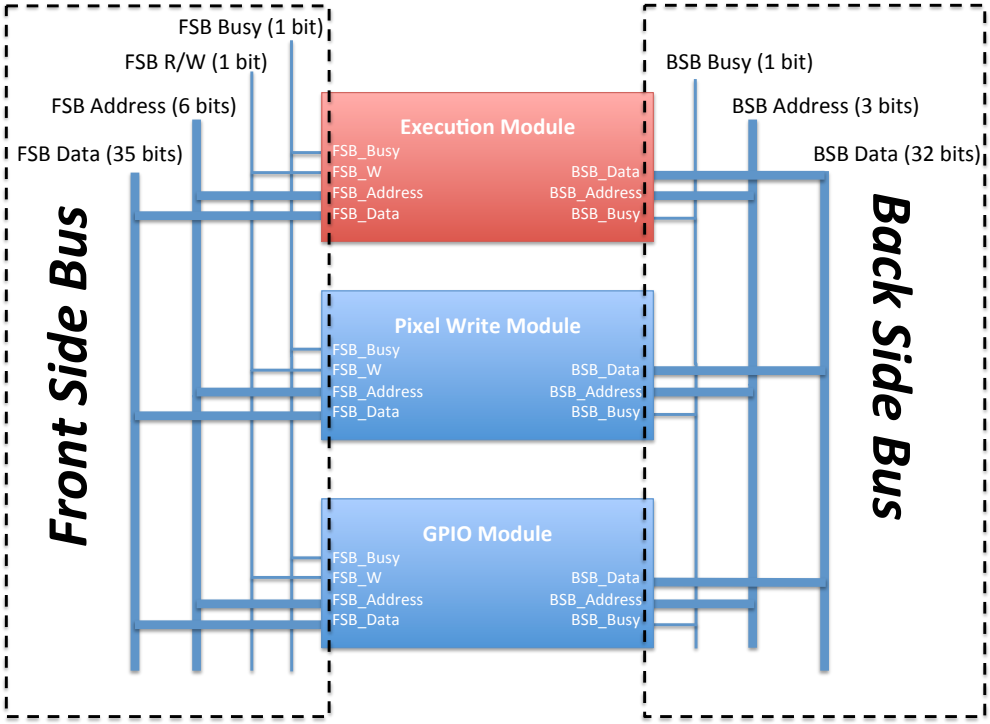
Figure 3.1 illustrates the framework of the register-to-bus interface. The FSB is designed for access to external ports, as the UART is both the FSB controller *and* the mediator to the workstation terminal. The UART decoder determines if the registers are in read or write mode, and also if new data is available on the line. The BSB, however, is designed for internal access only. In the BSB framework, the execution module (labeled in red) controls the line and determines which register's back side address will be selected and written to.

3.2 Front Side Bus

The front side bus is designed using tristate logic, allowing the data on the line to be set either Low - '0', High - '1', or High Impedance - 'Z'. This design

Approved for Public Release - Distribution is Unlimited

Figure 3.1: Register-to-Bus Overview



Approved for Public Release - Distribution is Unlimited

requires each of the front side registers that are inactive (i.e. its address is not selected) to send on the data bus a 'Z', so as not to pull down or pull up the desired data on the line.

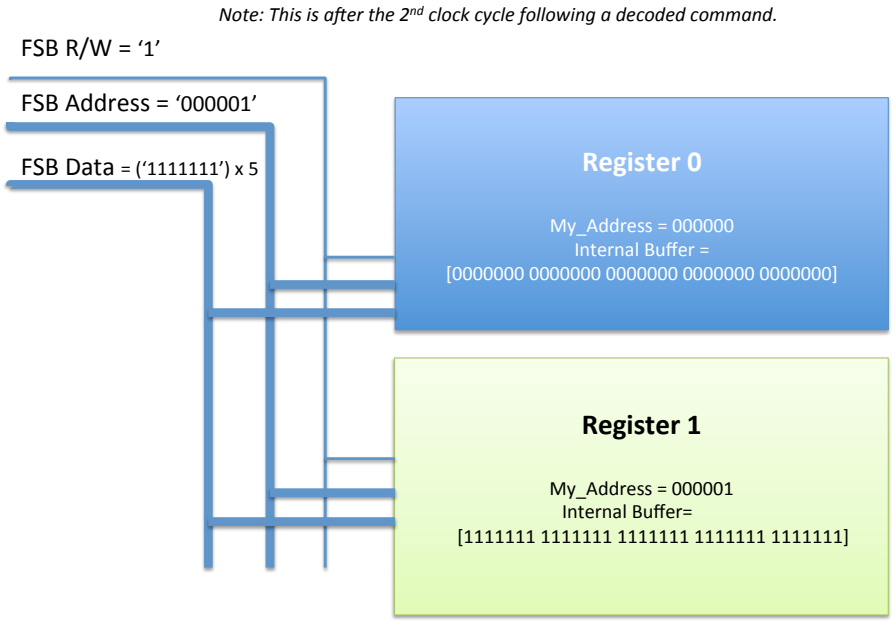
3.2.1 FSB Example

When a command is decoded from the UART decoder, the corresponding register address is selected and the given data values are written to that register. Consider the following example to illustrate this register selection and operation process (see Figure 3.2 for visual representation). Say, for instance, there are two registers that connect to the front side bus with addresses '000000' and '000001', respectively. If the workstation terminal user wants to write all 1's to register '000001', he will send a bit stream with address '000001' and data value '1111111 1111111 1111111 1111111 1111111'. Register '000001' will compare on the rising clock edge the value on the FSB address line. If that address matches its own address (which in this case, it does), the register will then proceed and check if the FSB read/write bit is set to high (write) or low (read). If the address does not match then no process within that module will be enabled.

Continuing with this example, assume the address matches and the FSB read/write bit is set to high; the FSB will then be set to busy and an internal buffer will grab the 35 1's in parallel and store them until the next clock cycle. On the following clock cycle, the particular function of the register will operate. This could be anything from simply relaying the internal buffer values to external ports, as is the case of the GPIO module, or starting a complex sequence of comparators, as found in the *Timer Core* registers inside the pixel write module (See Chapter 5).

Approved for Public Release - Distribution is Unlimited

Figure 3.2: Front Side Bus (FSB) Example



Approved for Public Release - Distribution is Unlimited

3.3 Back Side Bus

The back side bus is generally faster than the FSB, as it deals directly with the BRAM without any wait time lost due to external sources. Unlike the FSB, this computer architecture uses an opcode scheme to specify the operation to be completed. A program instruction holds the address of a particular opcode along with the operands to that instruction. Typical opcodes found in computer architectures are ADD, BRANCH, JUMP, and other common machine language instructions. In the SLEDs architecture, the first 3 bits of every word-data stored in BRAM is reserved to determine the opcode. As discussed in Chapter 2, the size of the word-data is 35 bits. With 3 bits now reserved for opcode selection, the remaining data size of the operands is 32 bits. Figure 3.3 shows the bit mapping of the eight SLEDs opcodes. Also, notice that there are four separate pixel write opcodes. Since the word size is limited to 35 bits, embedding the 9th bit in the opcode address allowed the rest of the 32 bits to be used for the (x, y) address and DAC value at full resolution.

Approved for Public Release - Distribution is Unlimited

Figure 3.3: SLEDs Opcode Instructions

System Control Module	000	*Don't Care (32 Bits)*
Time Count Module	001	*Time Value to Wait Until (32 Bits)*
Pin Output Mode	010	*Output Pin Values (32 Bits)*
Global Reset	011	*Don't Care (16 Bits) | DAC Reset Val (16 Bits)*
Pixel Write Q0	100	*SLEDs X (8 Bits) | SLEDs Y (8 Bits) | DAC Val (16 Bits)*
Pixel Write Q1	101	*SLEDs X (8 Bits) | SLEDs Y (8 Bits) | DAC Val (16 Bits)*
Pixel Write Q2	110	*SLEDs X (8 Bits) | SLEDs Y (8 Bits) | DAC Val (16 Bits)*
Pixel Write Q3	111	*SLEDs X (8 Bits) | SLEDs Y (8 Bits) | DAC Val (16 Bits)*

Approved for Public Release - Distribution is Unlimited

Chapter 4

PYTHON SCRIPTS

4.1 Purpose

In order to generate the binary files containing specific operation instructions for the FPGA computer architecture, Python scripts are necessary. The purpose of this chapter is to provide a functional analysis of the Python code in relation to the key operations of the SLEDs system.

4.2 Overview

As discussed in Chapter 3, the front side registers have immediate access to the UART decoder. These registers have specific 6-bit addresses for identification. Table 4.1 shows the Python global variables used, along with their assigned bit vector.

Table 4.1: Python Headers for Front Side Registers

Global Variable	BitVector	Corresponding Register Name
BRAM_ADDRESS	000100	BRAM Access
BRAM_DATA	000101	BRAM Data
PIXEL_ADDRTIME	001000	Pixel Write Address Match Time
PIXEL_PRECHTIME	001001	Pixel Write Evaluate Match Time
PIXEL_PIXELTIME	001010	Pixel Write Pixel Match Time
EXEC_ADDRESS	001100	Execution Unit
EXEC_FLAGS	001101	Execution Status
GPIO_OUTPUT	110000	GPIO Output

Approved for Public Release - Distribution is Unlimited

Below is an example of writing to the Pixel Write Time registers, setting the following parameters: address match time, evaluate match time, and pixel complete match time. See Figure 3.3 for the bit assignments in each of these modules.

Listing 4.1: Pixel Write Time Registers

```
def set_pixel_time_registers( self, A_time, E_time_short, E_time_long,
                              P_time_short, P_time_long):
        self.write_register( PIXEL_ADDRTIME,
                BitVector( size=14, intVal=US_TO_TIME(A_time))
                + BitVector( size=21))
        self.write_register( PIXEL_PRECHTIME,
                BitVector( size=14, intVal=US_TO_TIME(E_time_short))
                + BitVector( size=21, intVal=US_TO_TIME(E_time_long)))
        self.write_register( PIXEL_PIXELTIME,
                BitVector( size=14, intVal=US_TO_TIME(P_time_short))
                + BitVector( size=21, intVal=US_TO_TIME(P_time_long)))
```

4.3 Foundational Functions

4.3.1 SLEDs Serial Interface

To interface with the FPGA serial port, Python code must establish a connection over RS-232 communications. Below in Listing 4.2 is the Python code for initializing a connection with the SLEDs UART, running at a baud rate of 9,600. The serial_SLEDS_interface class includes definitions for writing and reading from the FPGA, which formats the 42-bit words to fit the UART decoder. As discussed in Chapter 2, the 42-bit word requires 6 extra bits to frame the incoming words, making a total of 48 bits transmitted.

Approved for Public Release - Distribution is Unlimited

Listing 4.2: SLEDs Serial Interface Class

```
class serial_SLEDS_interface(SLEDS_interface):
        def write(self, bvec ):
                outvec = pack("BBBBBB", int(bvec[0:8]), int(bvec[8:16]),
                                int(bvec[16:24]), int(bvec[24:32]),
                                int(bvec[32:40]), int(bvec[40:48]))
                self.port.flushInput()
                self.port.write( outvec )
                self.port.flush()

        def read(self):
                self.port.flush()       # wait for all writing to be done
                a = self.port.read(5)
                assert len(a) == 5, "did not get 5 bytes back"
                bytes = [ BitVector( size=8, intVal= x ) for x
                             in unpack('BBBBB', a) ]
                bvec = reduce( BitVector._add_,
                            [ x[1:] for x in bytes ] )
                return bvec

        def __init__(self, serial_port, baud_rate=9600):
                self.port = Serial(port=serial_port,baudrate=baud_rate,timeout=0.5)
```

4.3.2 Basic Register Operations

To read and write from the front side registers, the Python program must generate bit vectors that fit the 42-bit word format. Listing 4.3 shows the concatenation process for creating a write and read vector for a register on the FPGA. Also, see the assertions included in the read register function, as they are designed to help the user troubleshoot.

Listing 4.3: Write & Read From Registers

```
def write_register( self, register, datastr ):
        SPACER = BitVector(bitstring = '0')
        self.write(reduce(BitVector._add_,\
        [SPACER+datastr[x*7:(x+1)*7] for x in range(0, 5)],\
        BitVector(bitstring='11'+register)))
```


Approved for Public Release - Distribution is Unlimited

```
def read_register( self, register ):
        self.write( BitVector( bitstring='10'+ register) + BitVector(size=40))
        self.port.flush()
        try:
           x = self.read()
        except AssertionError:
           print '<read_error>'
           x = BitVector( size=35)
        self.port.flushInput()
        return x
```

4.3.3 BRAM Memory Test

Once a connection is established between the workstation terminal and the FPGA, a simple memory test is administered to ensure the expected program is stored in BRAM. The memory test shown in Listing 4.4 returns a 35-bit vector based on the address passed.

Listing 4.4: Simple Memory Test

```
def memtest_cmd(addr):
  return BitVector( size = 7, intVal=addr % 128 )
          + BitVector( size = 7, intVal=(addr+1)%128 )
          + BitVector( size = 7, intVal=(addr+2)%128 )
          + BitVector( size = 7, intVal=(addr+3)%128 )
          + BitVector( size = 7, intVal=(addr+4)%128 )
```

4.4 Loading and Running a SLEDs Program

After the system's operability is verified from the BRAM memory test, the actual SLEDs program is stored in BRAM. Below is the Python code to load and run a program in loop mode. The default settings for the pixel times are displayed in Table 4.2.

Approved for Public Release - Distribution is Unlimited

Listing 4.5: Loading & Running a SLEDs Program from BRAM

```
def load_and_run_loop(s, prog, runtime=0, **pixeltimes):
        pt = default_pixel_times #Set pixel time to default
        pt.update(pixeltimes) #Update pixel times
        s.set_gpio_outputs('1') #Turn LED on during program load
        s.set_pixel_time_registers( **pt ) #Set the pixel write time registers
        s.load_program( prog ) #Load Instructions
        s.set_gpio_outputs('0') #Turn LED off after program load completes
        print 'running...cntl-C_to_stop...',
        s.run_program_loop()  #Run program stored in BRAM in Loop
        try:
            if runtime == 0:
                while(True): sleep(0.5)
            else:
                sleep(runtime)
        except KeyboardInterrupt:
            pass
        s.stop_program()
```

Table 4.2: Pixel Time Default Values

Pixel Write Time Name	Default Value (Microseconds)
Address Match	0.5
Evaluation Match (Short)	1
Evaluation Match (Long)	80
Pixel Complete Match (Short)	1.5
Pixel Complete Match (Long)	81

Any SLEDs program can be run on the FPGA using this function. The only limitation is the size of the program, as it is restrained to the BRAM width. The Spartan-3E has approximately 360Kbits of storage available in BRAM, which given the 35-bit BRAM words, allows for near 10,533 pixel writes, (360Kbits x (1024bits/1Kbit))/(35bits/word) = 10,533 words, requiring 14 bits (2^{14} = 16,384 addresses) to be to be allocated for the BRAM address. To find information on the SLEDs programs administered, see Chapter 6.

Approved for Public Release - Distribution is Unlimited

Chapter 5

PIXEL WRITE MODULE

5.1 Purpose

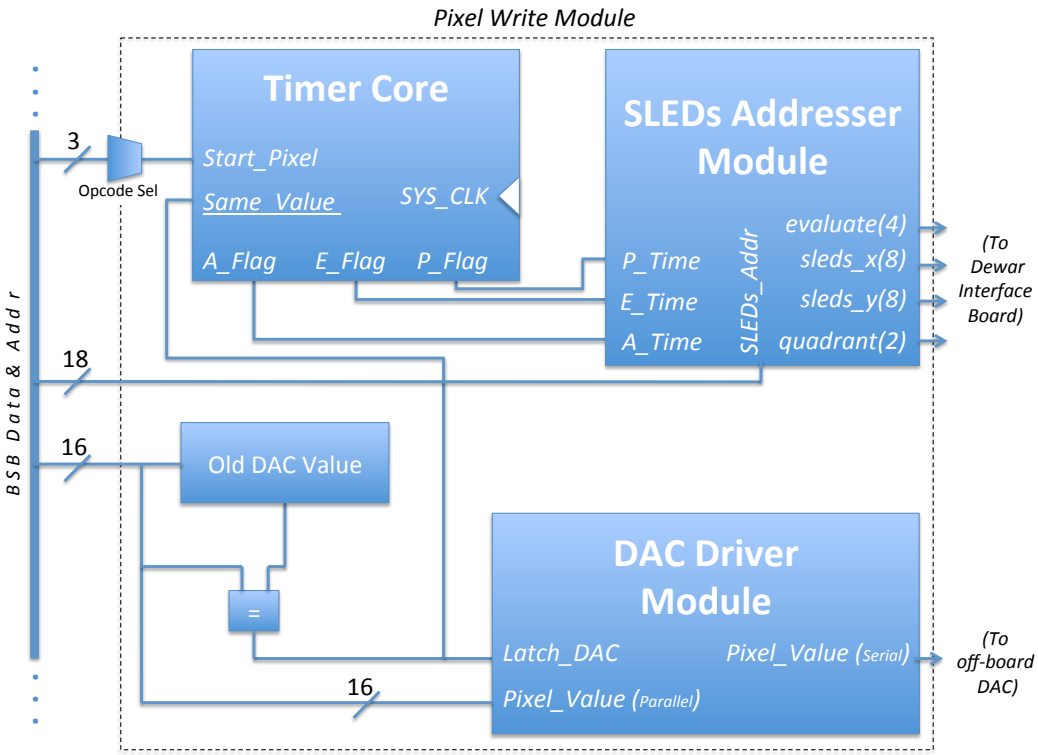
The pixel writing process is the central module of the SLEDs architecture. The driver chip of the LEDs requires 20 control signals (8 x, 8 y, 4 quadrant selection signals) and 1 analog signal (brightness of the LEDs) to operate properly, all which need to be driven by VHDL architecture. The purpose of the pixel write module is to provide those signals with appropriate timing to the driver chip.

5.2 Overview

As shown in Figure 5.1, the pixel write module connects to the BSB and is composed of three main units: *Timer Core*, *DAC Driver*, and *SLEDs Addresser*. The *Timer Core* generates the separate clocks needed to appropriately schedule addressing, evaluation, and completion of the pixel writing process. First, the *SLEDs Addresser* unit selects the desired pixel address from the 512 x 512 array at the instant the *Timer Core* generates the "address trigger." Then, the *DAC Driver* will latch the SLEDs analog value to the chip when the *Timer Core* generates the "evaluation trigger." Finally, the pixel will be turned off when the *Timer Core* generates the "pixel-complete trigger."

Approved for Public Release - Distribution is Unlimited

Figure 5.1: Pixel Write Block Diagram

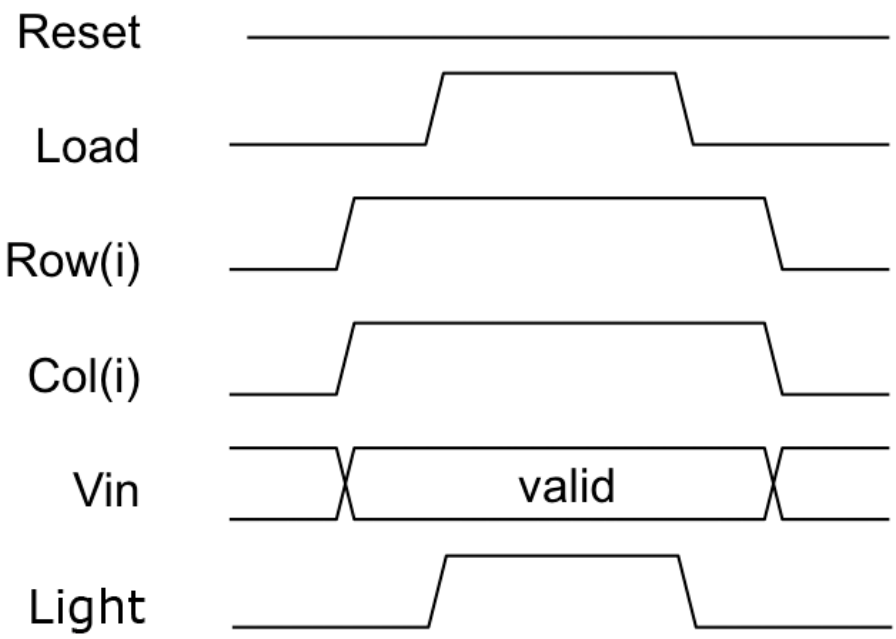


Approved for Public Release - Distribution is Unlimited

5.3 Timer Core Module

As previously mentioned, there are three components that factor into the timing of SLEDs: address time, evaluation time, and pixel complete time. As shown in Figure 5.2, the i^{th} row and column must be selected at the same time as the input LED drive voltage (Vin), which is the address time. After the addresses and drive voltage are ready, the SLEDs will wait to emit light until a "Load" signal is triggered. When the "Load" trigger fires, the pixel lights for a predetermined evaluation time. After the evaluation time saturates, the pixel turns off and remains off until a "Pixel Complete" flag triggers the start of the next pixel.

Figure 5.2: Sequencing to Successful Pixel Lighting [1]



Each of the timing components are matched on a user-defined 32-bit value. The address time takes approximately 20 microseconds to select the appropriate x

Approved for Public Release - Distribution is Unlimited

and y controls for any given pixel. To establish 20 microseconds of delay before the evaluation time begins, the *Timer Core* VHDL architecture must compare a counter register to a 32-bit value that would, in real time, take 20 microseconds. The counter register is always initialized to 0 at the beginning of every pixel write command. Given the 50 MHz clock from the Spartan-3E evaluation board, the comparison to the counter register will be made on the rising edge of every clock cycle for $(20\text{x}10^{-6})/(20\text{x}10^{-9}) = 1{,}000$ times. Therefore, the user-defined 32-bit value for the address register will be 1,000 in decimal, or '00000000000000000000001111101000' in 32-bit form.

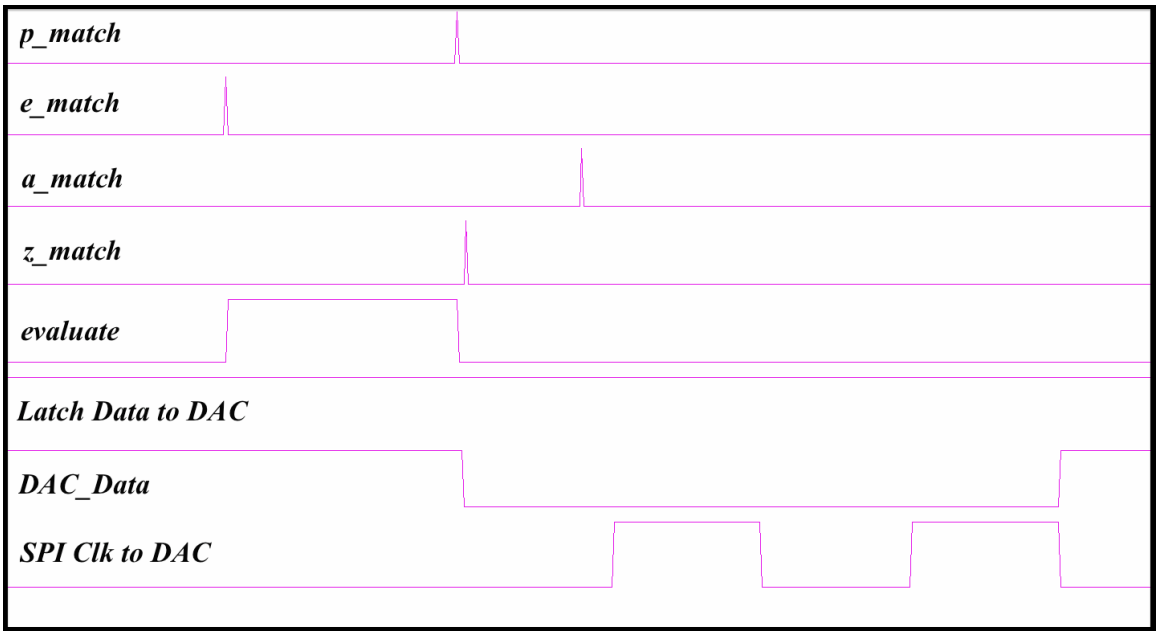
A similar calculation can be performed for the evaluation and pixel complete times. However, those values will need to be cumulative from the previous stage. For instance, the evaluation time begins 20 microseconds after the address time. Therefore, the user-defined evaluation time must be set to 20 microseconds (from beginning to address time) + 20 microseconds (from address time to evaluation time) = 40 microseconds total, in order to follow the correct sequence of firing a pixel. Figure 5.3 shows a screenshot from a logic analyzer of the *Timer Core* signals exiting the FPGA in this sequence. Notice at the bottom of the screenshot that the DAC signals from the FPGA for the next pixel (not shown) are generated the instant the pixel-complete signal from the current pixel occurs. This is designed to expedite the loading time.

5.4 SLEDs Addresser Module

The signals that physically interface with the SLEDs are generated from the *SLEDs Addresser*. To match this interface, the FPGA outputs 3.3V digital logic on a 40-pin expansion port, which is then translated into a 5V signal to compatibly control the digital logic on the SLEDs chip. To find more information on this hardware translation process, see Corey Lange's paper [6]. Figure 5.4 illustrates the routing from the FPGA to the expansion port.

Approved for Public Release - Distribution is Unlimited

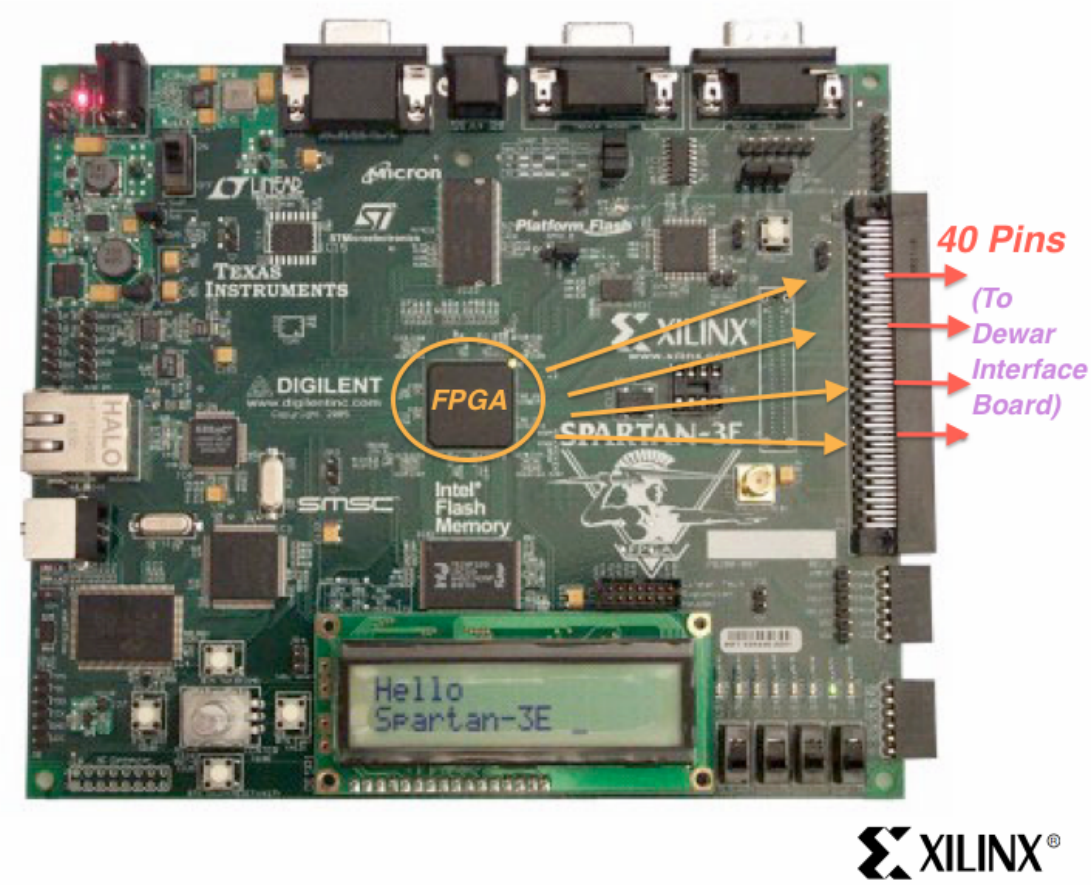
Figure 5.3: Logic Analyzer View of Timer Core Signals



The *SLEDs Addresser* also manages the quadrant selector. The 8 x and 8 y signals will select enough for a 256 x 256 array, but this would only make a quarter of the 512 x 512 chip selectable: 256 x 256 = 65,536 (x, y) addressable pixels and 512 x 512 = 263,680 total pixels, which when dividing the two areas 65,536/263,680 = 1/4 the number of (x, y) addressable pixels. To be able to select any pixel in the 512 x 512 lattice, each corner (256 x 256 area) has a quadrant selection bit (0 = disable quadrant, 1 = enable quadrant). This quadrant selection bit is embedded in the BSB opcode address, allowing the pixel write module to feed the desired pixel (x, y) location and quadrant selection simultaneously to the *SLEDs Addresser*.

Approved for Public Release - Distribution is Unlimited

Figure 5.4: SLEDs Addresser Routes FPGA Signals to 40-Pin Expansion Port [2]



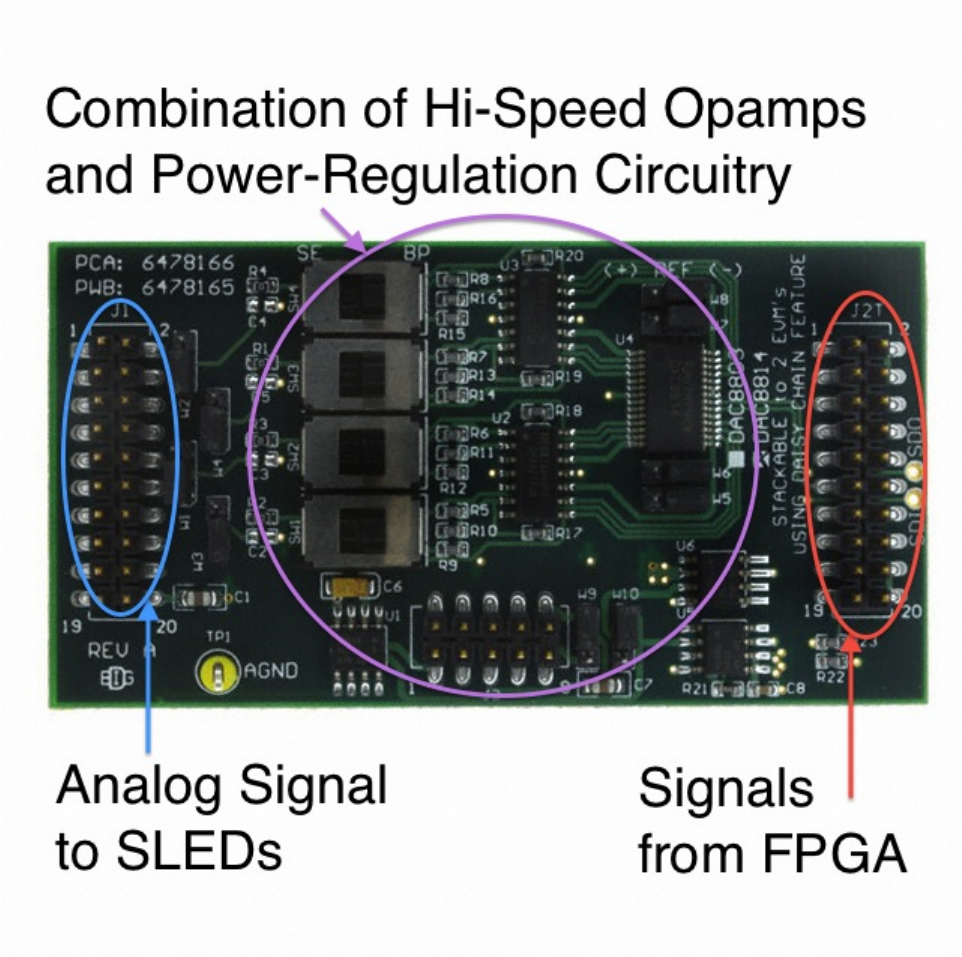
5.5 DAC Driver Module

The *DAC Driver* also connects directly to the 40-pin expansion port of the Spartan-3E evaluation board. The off-board DAC is the Texas Instruments (TI) DAC8803 evaluation board, requiring two critical signals from the FPGA: a 16-bit Pixel Value and a 1-bit Load-DAC trigger. In addition, the *DAC Driver* uses Serial Peripheral Interface (SPI) communications, requiring an extra SPI clock signal to synchronize the 16-bit serial Pixel Value. After storing the 16-bit serial data in its

Approved for Public Release - Distribution is Unlimited

local memory, the DAC remains idle until the 1-bit Load-DAC signal becomes logic low, at which point the analog output signal will be delivered to the SLEDs. See Figure 5.5 for a picture of TI's evaluation board. With the given PMOS transistor design of the SLEDs, an analog signal of 0V results in full intensity, while an analog voltage of 12V turns the pixel off.

Figure 5.5: DAC Evaluation Board [3]



Approved for Public Release - Distribution is Unlimited

5.6 Same/Change Pixel Value

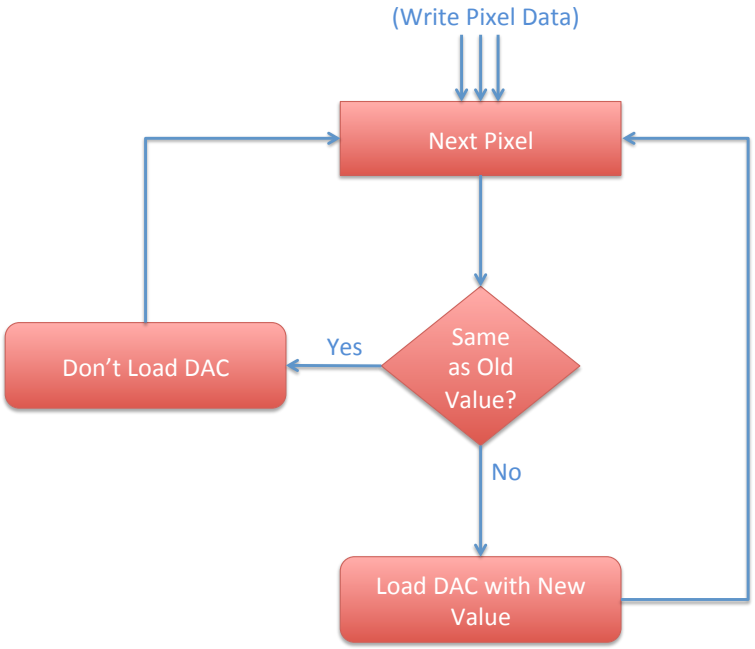
In effort to save bandwidth in the pixel writing speed, a same/change pixel value algorithm is implemented into the pixel write module. Figure 5.6 illustrates this process. The DAC requires extra setup time when a new 16-bit value is sent to its interface. This time is a combination of delivering 16 bits serially at the SPI baud rate, storing the new value in DAC local memory, and the switching speed of TI's opamps. When two or more pixel write commands are consecutively requested with the same DAC value, however, this extra setup time can be avoided. Since the DAC will maintain outputting its previous analog voltage (unless the device is set to disabled mode), the only real change necessary is the address selected, saving valuable time.

To demonstrate the value of saving this time, first examine the case without any algorithm. For instance, if every pixel were selected at the given frame refresh rate 60Hz, the number of pixel writes would be tremendous: 512 pixels x 512 pixels x 60 cycles/sec = 15,728,640 square pixel writes/sec! This would leave a 50MHz clock with just over 3 clock cycles to deliver the DAC data and select the SLEDs address, which Figure 5.3 demonstrates is not possible.

Using this same/change pixel value algorithm, there is no need to go through the writing of every pixel. Instead, only the pixels that are different in value need to be selected. Most of the SLEDs testing does not require more than a quarter of the pixels to have different values than the rest of them, cutting the frequency of pixel writes to: 512 pixel changes (assuming the pixel values alternate between pixels) x 60 cycles/sec = 30,720 pixel writes/sec. Now, the 50MHz clock has close to 1,600 cycles to deliver the DAC data and select the SLEDs address, which is much more preferable.

Approved for Public Release - Distribution is Unlimited

Figure 5.6: Same/Change Pixel Value Algorithm



5.6.1 Implementation

To implement this algorithm, a register called "Old_DAC_Value" inside the pixel write module holds the current pixel value. When the pixel write module is selected on the BSB, a vector comparator checks if the new value is the same or different. If the latter, an SC (Same/Change) flag is set to logic HIGH. The *Timer Core* uses this flag to adjust the timing required for setting up the DAC. The *SLEDs Addresser* knows not to send a "Load DAC" signal. If the two pixel values are the same, then the *Timer Core* will adjust to a faster clock generation.

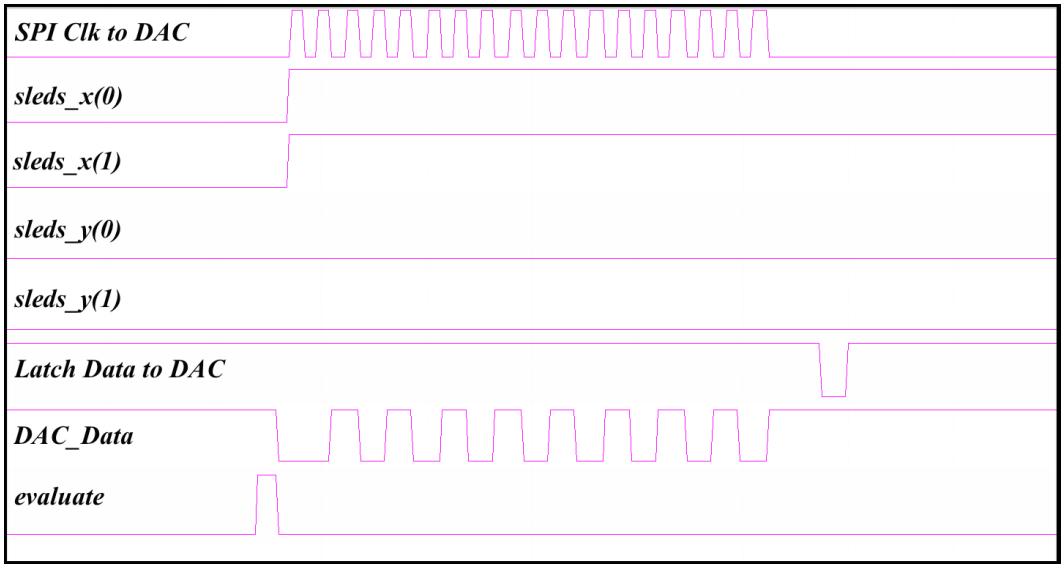
Approved for Public Release - Distribution is Unlimited

The "Old_DAC_Value" register is updated with the next pixel write value, and the process is repeated until the program is stopped.

5.7 Test of Pixel Write Module

To test the functionality of the Pixel Write Module, Python scripts were written to select different quadrants of the array without changing the (x, y) address, writing 0xAAAA as the 16-bit DAC value. The reason for sending a DAC value of 0xAAAA is the benefit of its binary appearance when viewed from the logic analyzer, clearly defining every bit in its alternating pattern: '1010101010101010'. As is evident in Figure 5.7, the Northwest (NW: X(0) = 1, Y(0) = 0, X(1) = 1, Y(1) = 0) corner is successfully selected, with the DAC SPI clock generation starting at the same time as the transmission of the 16-bit Pixel Value. The success of this test demonstrated the *Timer Core*, *SLEDs Addresser*, and *DAC Driver* all worked together as designed.

Figure 5.7: Writing 0xAAAA to SLEDs NW Corner



Approved for Public Release - Distribution is Unlimited

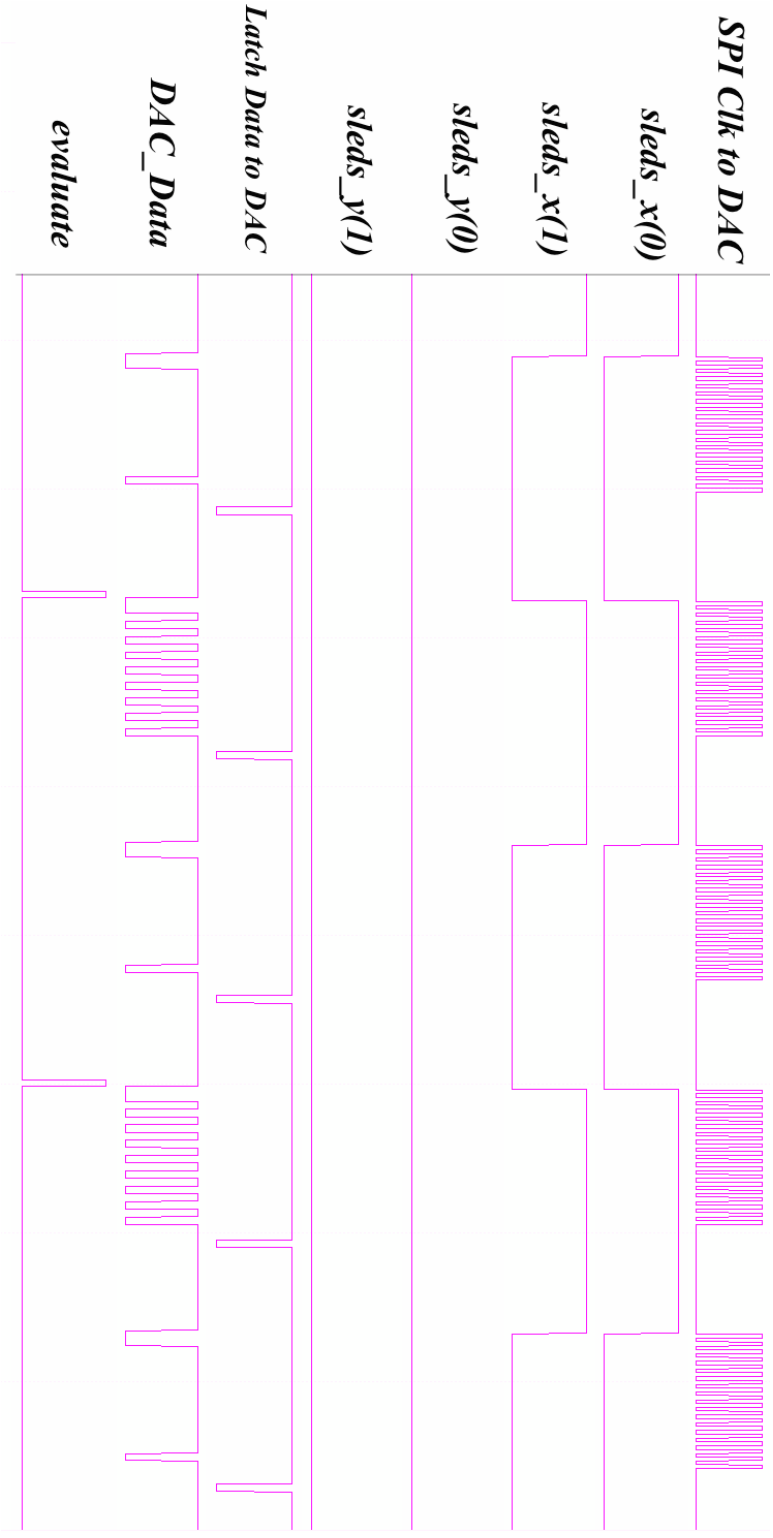
The final experiment before moving into the full system, was to test the same/change pixel value function. As mentioned in the previous section, only pixels with different DAC values from the previous pixel value will be sent a DAC value. The Python script (see Listing 5.1 below) generated binaries for a simple loop program in BRAM, which executed 4 pixel-write commands to two separate addresses (0, 0) and (1, 0) in the same quadrant (Q=0), writing the following DAC values in a loop: (0xAAAA, 0x0000, 0x0000, 0xAAAA). Figure 5.8 shows how every other pixel write command causes the 16-bit DAC value to be written to the external DAC. Also, when the pixel write command is a consecutive repeat (i.e. writing 0x0000 to 0x0000 or 0xAAAA to 0xAAAA), no 16-Bit value is sent to the external DAC.

Listing 5.1: Python Code for Same/Change DAC Loop Test

```
SC_DAC_Loop_Test =
                  [ write_pixel_cmd(0,1,0,0x0000),
                    write_pixel_cmd(0,1,0,0xAAAA),
                    write_pixel_cmd(0,0,0,0xAAAA),
                    write_pixel_cmd(0,0,0,0x0000)]
```


Approved for Public Release - Distribution is Unlimited

Figure 5.8: Same/Change DAC Loop Test



Approved for Public Release - Distribution is Unlimited

Chapter 6

CONCLUSIONS

6.1 SLEDs Testing at Eglin

During the week of 23 - 27 January 2012, MWIR camera testing of the SLEDs took place at Eglin Air Force Base, Florida. The following are the results from this testing. The pictures are provided by Eglin. A single 512 x 512 MWIR array was tested at room temperature and cryogenic temperatures. These results demonstrate the functionality of not only the FPGA-based computer architecture driving the array, but the SLEDs itself.

The setup is composed of five stages: workstation terminal, Spartan-3E evaluation board, dewar interface board, the SLEDs carrier package, and the cryogenic dewar. A picture of the full setup is shown in Figure 6.1. The workstation terminal attaches directly to the Spartan-3E evaluation board using RS-232 protocol. The workstation sends the SLEDs program to the FPGA. The digital signals from the Spartan-3E FPGA are level shifted from 3.3V to 5V logic. This stage is necessary to interface with the SLEDs 5V (x, y) address signals. The SLEDs carrier package is a gold plated PCB that attaches directly to the bottom surface of the RIIC and mounts to the front of the dewar (see Figure 6.2), connecting to the dewar interface board via ribbon cables.

Approved for Public Release - Distribution is Unlimited

Figure 6.1: Control Electronics Setup for Eglin Testing

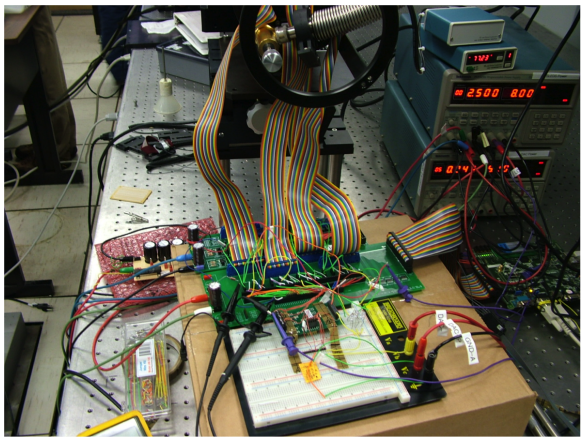
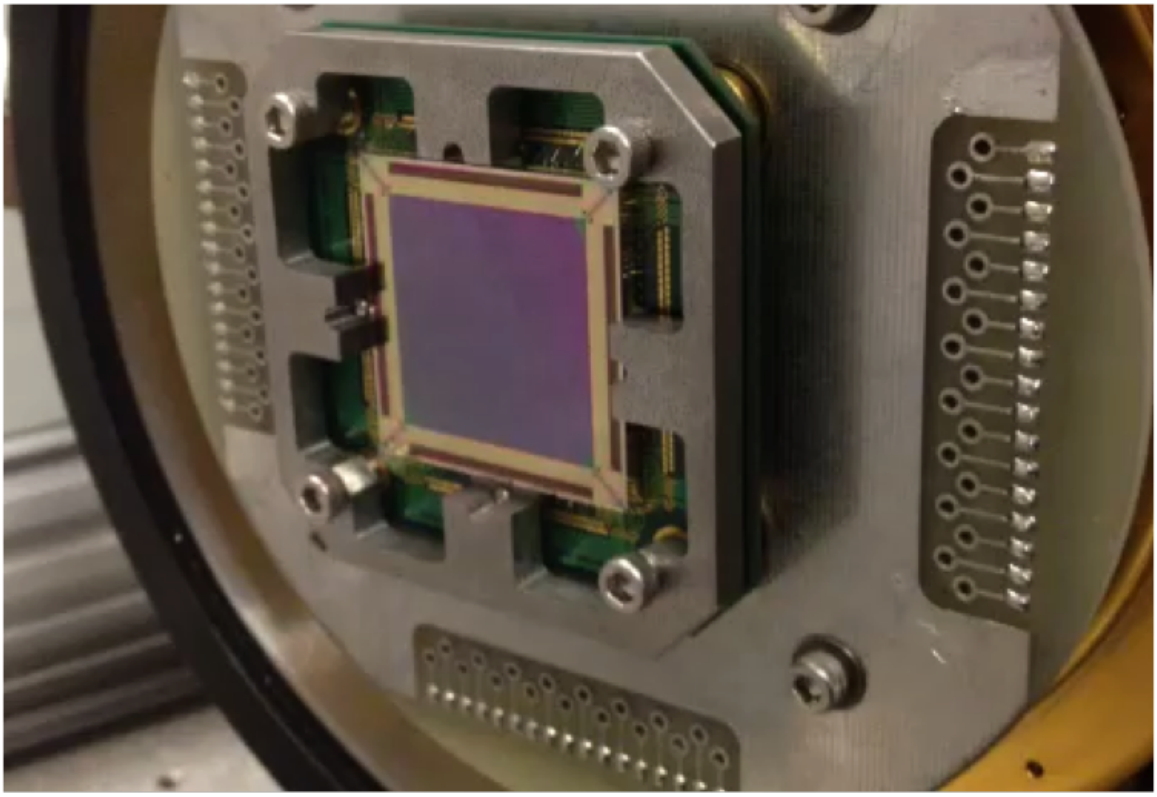


Figure 6.2: SLEDs Carrier Package in Dewar for Eglin Testing

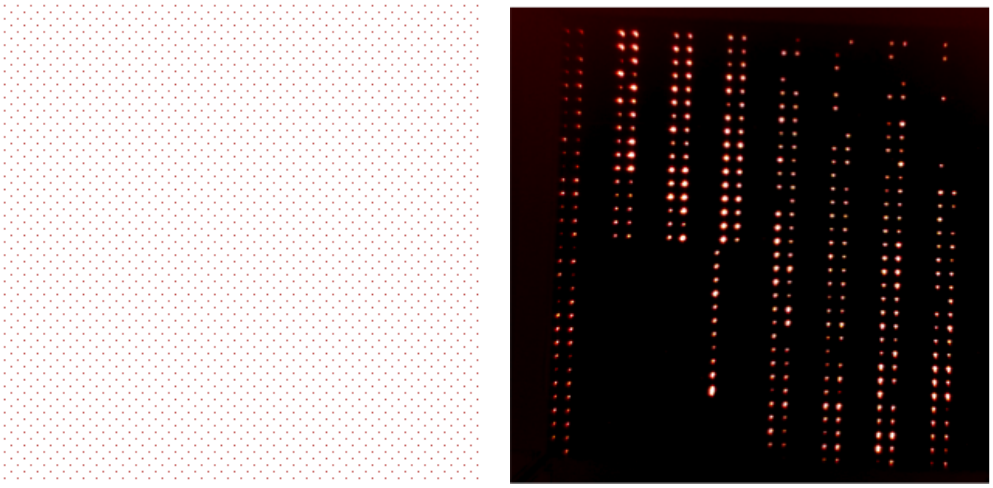


Approved for Public Release - Distribution is Unlimited

6.1.1 Full Array Yield Test

The first test completed was a grid selection of every 16th pixel in the array. In theory, this would select 512/16 pixels = 32 pixels per side - See Figure 6.4(a). The actual test, however, only selected every other column. As shown in Figure 6.4(b), there are 28 columns selected, with a dysfunctional southwest corner. See Section 6.1.2 for more detail on the dysfunctional corner.

Figure 6.3: Full Array Yield Test



(a) Expected Full Array Pixel Selection (b) Actual Full Array Pixel Selection

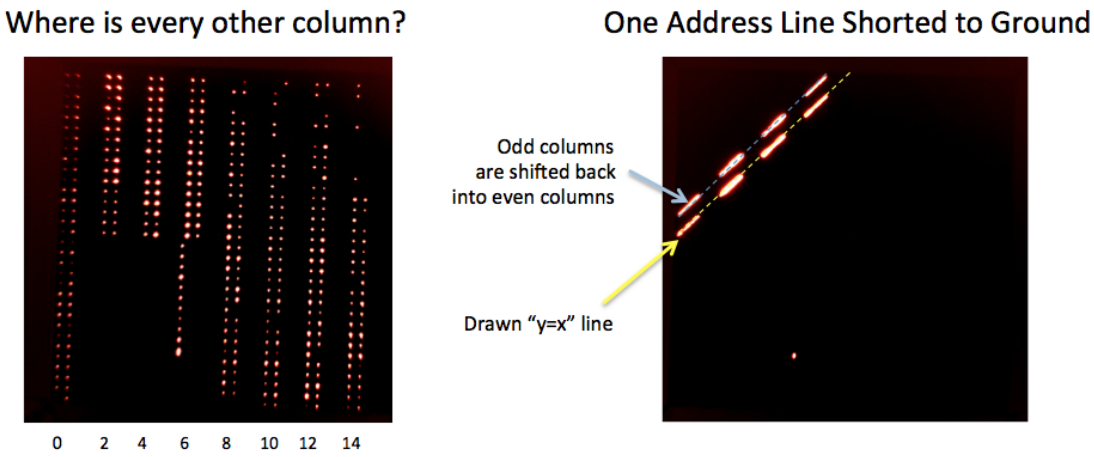
Listing 6.1: Python Code for Drawing Full Grid

```
def whole_chip_grid(d, xstep=16, ystep=16, xoffset=0, yoffset=0):
  return (
    quadr_grid_prog(dval=dval,Q=0,xstep=xstep, ystep=ystep, xoffset=xoffset, yoffset=yoffset) +
    quadr_grid_prog(dval=dval,Q=1,xstep=xstep, ystep=ystep, xoffset=xoffset, yoffset=yoffset) +
    quadr_grid_prog(dval=dval,Q=2,xstep=xstep, ystep=ystep, xoffset=xoffset, yoffset=yoffset) +
    quadr_grid_prog(dval=dval,Q=3,xstep=xstep, ystep=ystep, xoffset=xoffset, yoffset=yoffset))
```


Approved for Public Release - Distribution is Unlimited

To find an explanation for this unintentional column skip, a second test was administered on the SLEDs. Now, a straight line with a slope of 1 (i.e. the equation y = x) was selected in the northwest corner. As shown in Figure 6.4, a second line is still selected on the diagonal. This proves one bit of the x/y decoder control signals is shorted. Since one address bit is always zero, the pixel write commands wrap odd columns back into even ones. To effectively extrapolate the SLEDs device yield, data from every-other-column is analyzed.

Figure 6.4: Missing Column Explanation



Approved for Public Release - Distribution is Unlimited

6.1.2 Dysfunctional Corner

The dysfunctional southwest corner is a result of a gouge mark found on the surface of the hybrid. This unintentional gouge was from an accident that occurred during the wire bonding process. Fortunately, the other three quadrants of the SLEDs can still be effectively evaluated.

Figure 6.5: Thermal of Gouge Mark

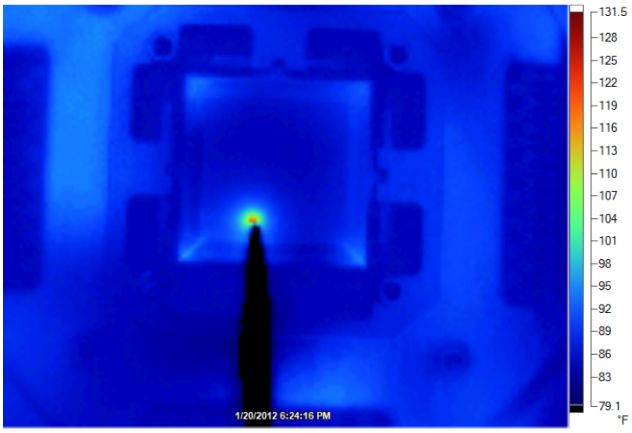
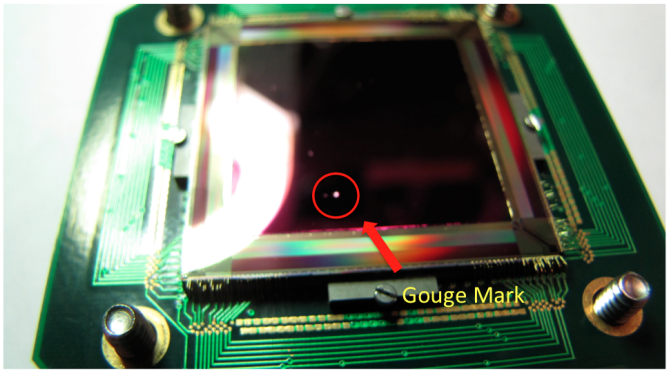


Figure 6.6: Gouge Mark on Surface

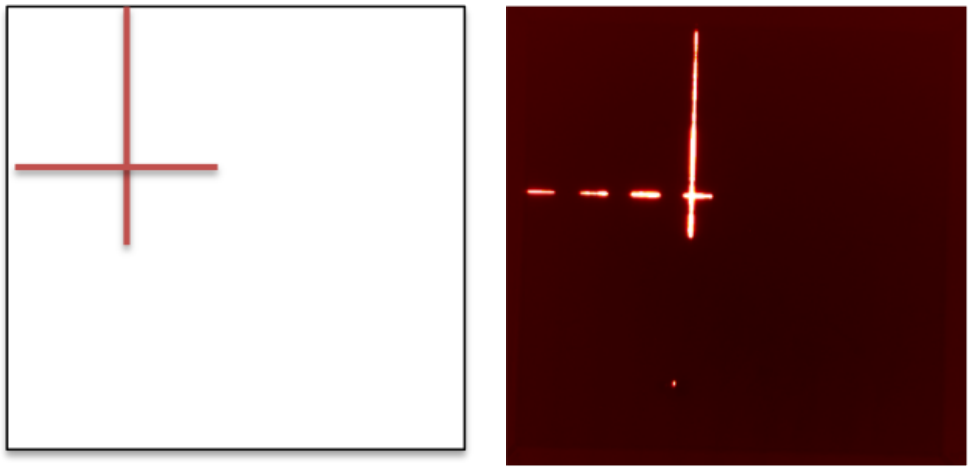


Approved for Public Release - Distribution is Unlimited

6.1.3 Row/Column Test

The next step was to test the row and column decoding on the SLEDs. As shown in Figure 6.8(b), the northwest corner was successfully selected in the expected location. To run this test, a Python script fed a binary file of 32 pixel write commands for each line to the FPGA: 32 consecutive pixels for the row and 32 consecutive pixels for the column. From the VHDL architecture, only these 64 pixels are selected. Every other (x, y) location in the 512 x 512 array is given the default background DAC value. The brightness of the LEDs shows how efficiently the framing and refresh rate are designed.

Figure 6.7: Row/Column Test



(a) Expected Row/Column Selection (b) Actual Row/Column Selection

Listing 6.2: Python Code for Writing R/C Lines

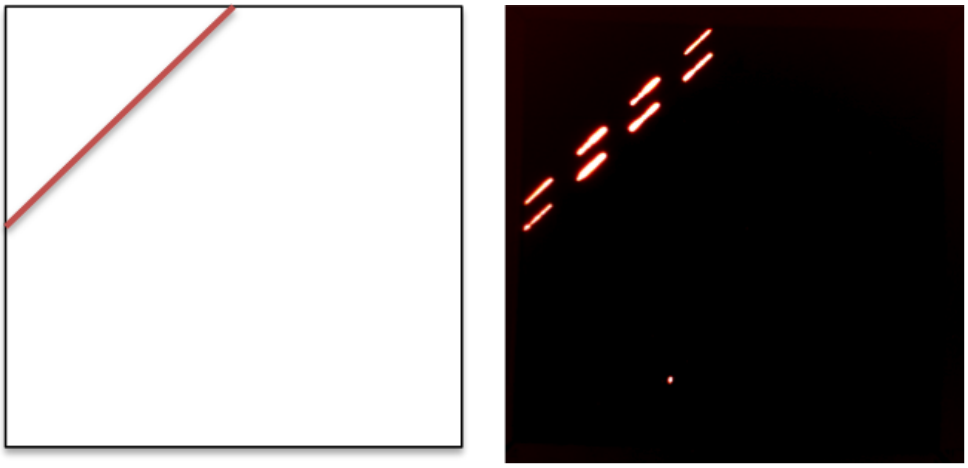
```
def hline(dacval,Q, y, xstart, xend):
  return [write_pixel_cmd(Q,y,x,dacval) for x in range(xstart,xend+1)]
def vline(dacval,Q, x, ystart, yend):
  return [write_pixel_cmd(Q,y,x,dacval) for y in range(ystart,yend+1)]
```


Approved for Public Release - Distribution is Unlimited

6.1.4 Diagonal Test

The diagonal test served two purposes: to prove one bit of the SLEDs x/y decoding is shorted (see Section 6.1.2) and to prove the Python scripts constructed the proper binary file to select non-orthagonal intersections of pixels. Listing 6.3 shows the scripting algorithm used to draw a diagonal line on the SLEDs.

Figure 6.8: Diagonal Test



(a) Expected Diagonal Selection (b) Actual Diagonal Selection

Listing 6.3: Python Code for Drawing Diagonal Line

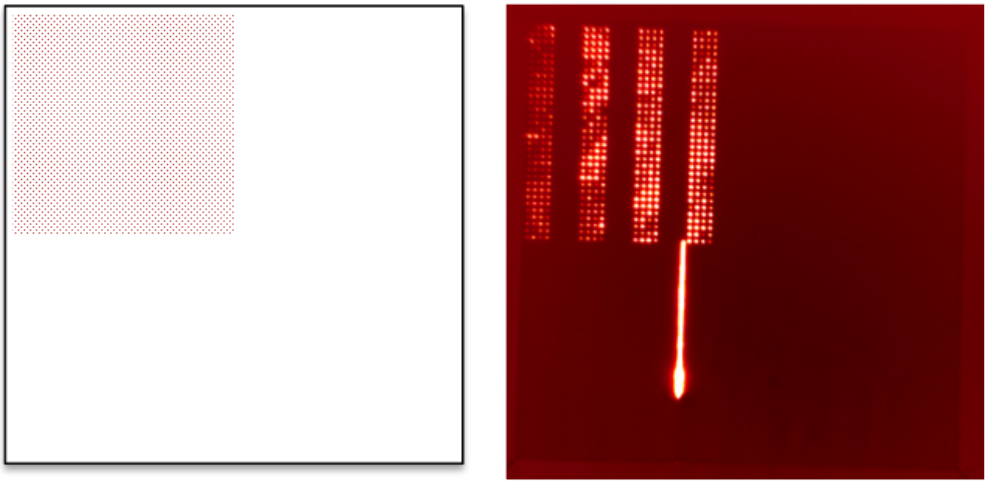
```
def diagLine(dacval, Q, xstart, ystart, xend, yend):
  slope = (yend - ystart)/(xend - xstart)
  return [write_pixel_cmd (Q, (ystart+ t*slope), (xstart+t), dacval)
          for t in range(0, (xend - xstart) +1)]
```


Approved for Public Release - Distribution is Unlimited

6.1.5 High-Density Coverage Test

The high-density coverage test is designed to test the SLEDs operability when pixels are selected in close proximity. As opposed to the full array yield test in Section 6.1.1 where every 16th pixel is selected, the high-density test selects every 4th pixel in the array. To limit the current density and potential thermal damage to the hybrid, only the northwest quadrant is enabled. Figure 6.10(b) shows four times as many pixels lit as Figure 6.4(b). Listing 6.4 shows the Python code to (x, y) address step in the northwest quadrant (Q = 0).

Figure 6.9: High-Density Coverage Test



(a) Expected High-Density Selection (b) Actual High-Density Selection

Listing 6.4: Python Code for High-Density Coverage Test

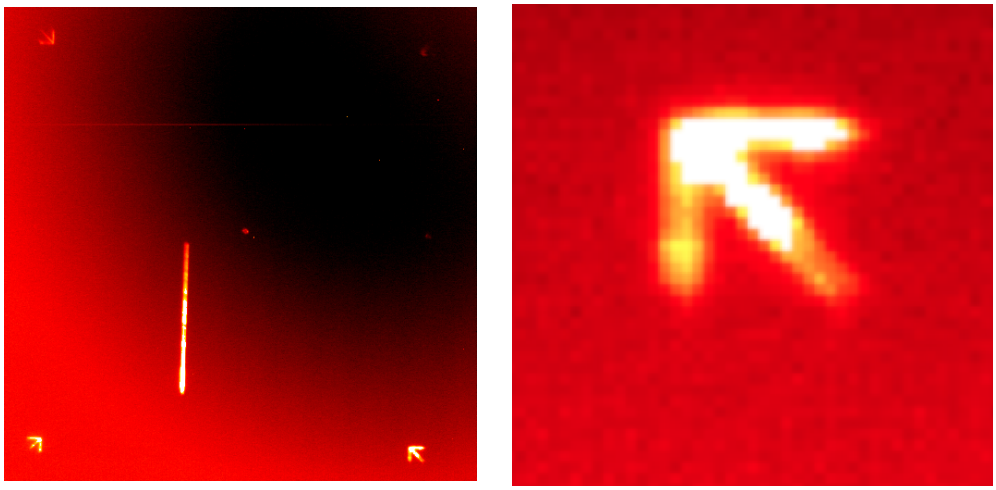
```
def sq_prog(dval,Q=0,xoffset=0,yoffset=0,xstep=64,ystep=64,xsize=4,ysize=4):
  out = []
  for x in range(0,xsize):
        for y in range(0,ysize):
          out += quadr_grid_prog(dval,Q,xoffset+x,yoffset+y,xstep,ystep)
  return out
```


Approved for Public Release - Distribution is Unlimited

6.1.6 Corner Arrow Test

The corner arrow test served to prove the SLEDs capablility to project characters other than lines. The FPGA is able to drive any combination of pixels, which is ultimately the goal of this project. Listing 6.5 shows the intricate combination of horizontal line (hline) and vertical line (vline) programs to create the arrow image.

Figure 6.10: Corner Arrow Test Pictures



(a) Full View (b) Zoom View

Listing 6.5: Python Code for Drawing Corner Arrows

```
def corner_arrows(dacval,NumPixels):
  return (hline(dacval,0,y=(255-NumPixels),xstart=0,xend=NumPixels)
   + vline(dacval,0,x=NumPixels,ystart=(255-NumPixels),yend=255)
   + hline(dacval,1,y=(255-NumPixels),xstart=(255-NumPixels),xend=255)
   + vline(dacval,1,x=(255-NumPixels),ystart=(255-NumPixels),yend=255)
   + hline(dacval,2,y=NumPixels,xstart=0,xend=NumPixels)
   + vline(dacval,2,x=NumPixels,ystart=0,yend=NumPixels)
   + hline(dacval,3,y=NumPixels,xstart=(255-NumPixels),xend=255)
   + vline(dacval,3,x=(255-NumPixels),ystart=0,yend=NumPixels))
```


Approved for Public Release - Distribution is Unlimited

Chapter 7

FUTURE WORK

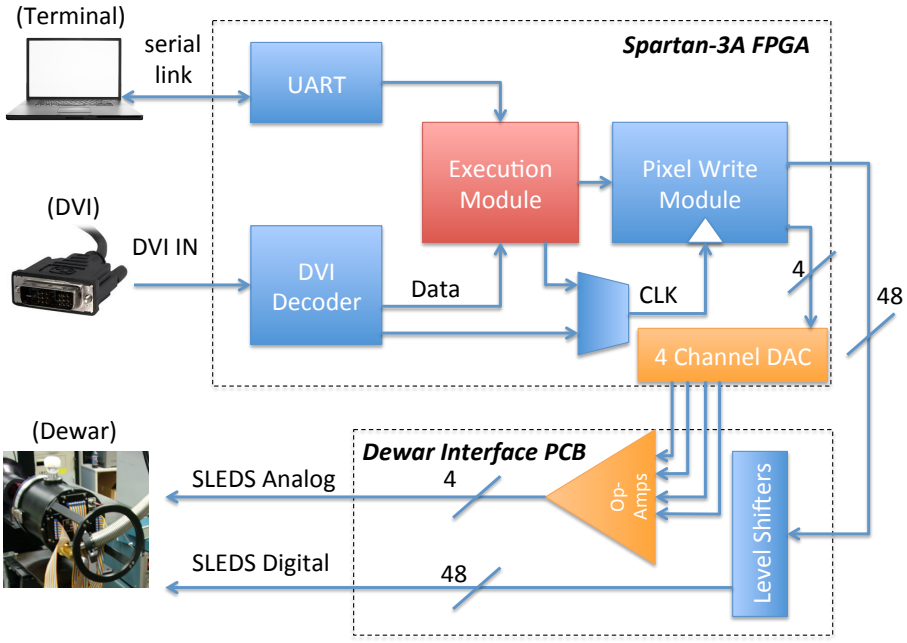
7.1 System Upgrade

The next step is upgrading the control electronics for the 512 x 512 SLEDs for optimal performance. The included features of the upgrade are DVI input, 4 channel DAC, and high speed opamps. The DVI signals from an external source will deliver the pixel write information, which new VHDL design will need to decode in order to work with the current FPGA-based computer architecture. The 4 channel DAC and high speed opamps will allow more pixels to be selected on the SLEDs at the same frame rate.

As shown in Figure 7.1, a similar connection scheme will be used in the next generation of the SLEDs control electronics. The dewar interface board will have high speed level shifters to avoid any auxiliary delays, while also delivering a SLEDs-compatible driver electronics system. The current FPGA-based computer architecture described in this thesis will be the foundation, but much time is anticipated necessary for implementing the DVI decoding logic, as this feature is completely unchartered in the current system.

Approved for Public Release - Distribution is Unlimited

Figure 7.1: Future SLEDs Hardware Setup

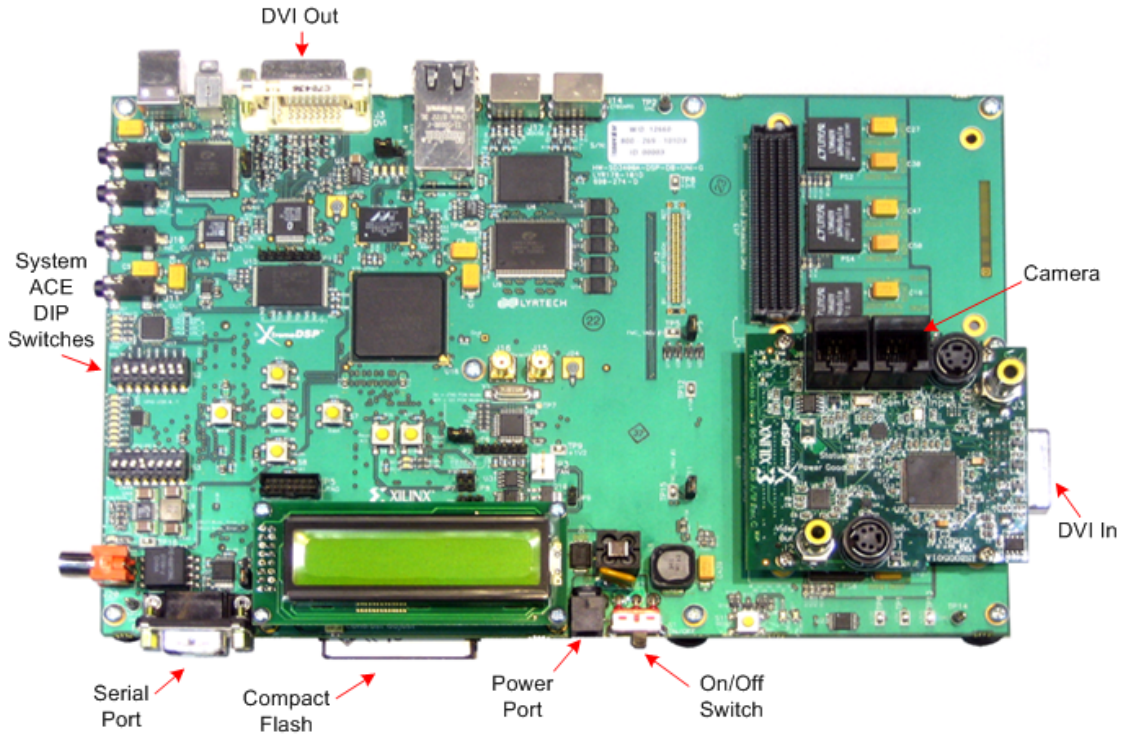


Approved for Public Release - Distribution is Unlimited

7.2 New FPGA Evaluation Board

The Spartan-3A DSP FPGA Video Starter Kit is a significant upgrade in FPGA high speed performance, as it is tuned for video digital signal processing. As shown in Figure 7.2, the daughter card attached to the starter kit takes DVI input, which will help as a starting ground for the development of the DVI decoding module.

Figure 7.2: The Spartan-3A DSP FPGA Video Starter Kit [4]



With these new features the future SLEDs system will be faster than ever, taking this FPGA-based computer architecture to an exciting new level.

Approved for Public Release - Distribution is Unlimited

BIBLIOGRAPHY

[1] Rodney Thomas Tyler McGee. Design, fabrication, iteration, and testing of the 68x68 superlattice led infrared projection system and associated electronics. Master's thesis, University of Delaware, 2010.

[2] Xilinx. *Spartan-3E FPGA Starter Kit Board User Guide*, 1.2 edition, January 2011.

[3] Texas Instruments. *DAC8803/14 EVM User's Guide*, May 2006.

[4] Xilinx. *Spartan-3A DSP FPGA Video Starter Kit: User Guide*, 2.1 edition, March 2010.

[5] Naresh C. Das. Light emitting diode arrays for hwil sensor testing. *Proc. SPIE*, 5785, 2005.

[6] Corey Wyatt Lange. Design and development of a 512 x 512 infrared emitter array system. Master's thesis, University of Delaware, 2011.

[7] Corey Lange, Rodney McGee, Nicholas Waite, Robert Haislip, and Fouad E. Kiamilev. System for driving 2d infrared emitter arrays at cryogenic temperatures. *Technologies for Synthetic Environments: Hardware-in-the-loop XVI*, 8015, April 2011.

[8] Pong P. Chu. *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version.* Wiley-Interscience, Hoboken, NJ, 2008.

[9] Todd Langley. Introduction to intel architecture: The basics. January 2009.

Project Funding & Disclaimer

This project is funded by the Test Resource Management Center (TRMC) Test and Evaluation/Science & Technology (T&E/S&T) Program through the US Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) under Contract No. W91ZLK-06-C-0006.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Test Resource Management Center (TRMC) Test and Evaluation/Science & Technology (T&E/S&T) Program and/or the US Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI).