

INTRODUCTION.....1

HARDWARE SETUP1

SEVEN SEGMENT DISPLAY1

SERVO SUBSYSTEM1

JOYSTICK CONTROL SUBSYSTEM1

ACCELEROMETER SUBSYSTEM.....4

CAMERA COMMAND CONTROL5

VGA DISPLAY SYSTEM6

SUMMARY AND CONCLUSIONS7

INTRODUCTION

IN THIS SECTION:

- What Can it do?
- How was it designed?
- What tools were used?

What Can it do?

The system built by and for computer engineers allows for the closest bare metal experience on the market. Having direct access to memory and controlling exactly what components fill your system, the possibilities for invention is endless. Developed with ease of use and quick application the system comes packed with various components from servo control hardware to I²C communications.

How was it Designed?

Using the pre-installed hardware, the system can do multiple operations such as control, movement, and calculations. The mounted servos allow the user to see their project come to life and with multiple ways of control it comes with the possibilities are endless. Some control is but not limited to accelerometer, joystick, and switches.

What tools Were Used?

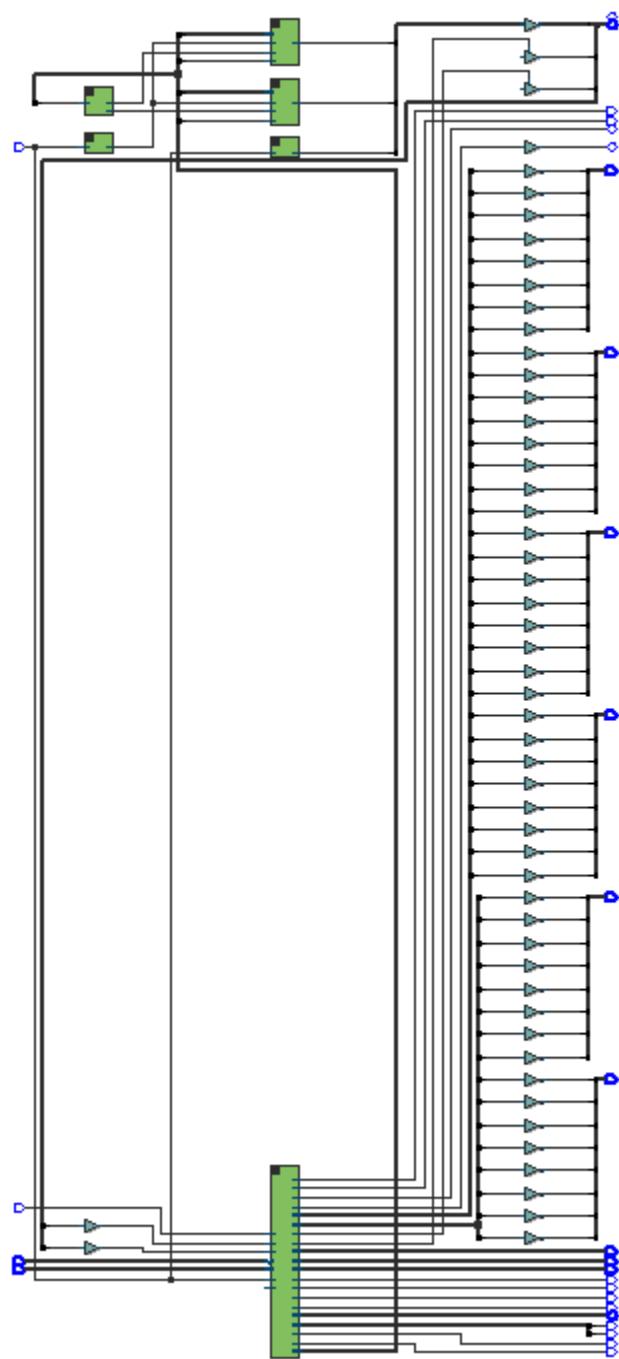
Using the Quartus project program, burning designs down to the board is a breeze. Allowing for free-flowing ideas and customizable systems. Nios2 programing environment powered by Eclipse allowed for the much beloved C language to flow through the hardware. Need a component you don't have? Using Qsys in the Quartus program allows for those programs to be integrated right into your system along with your other crucial components.

HARDWARE SETUP

IN THIS SECTION:

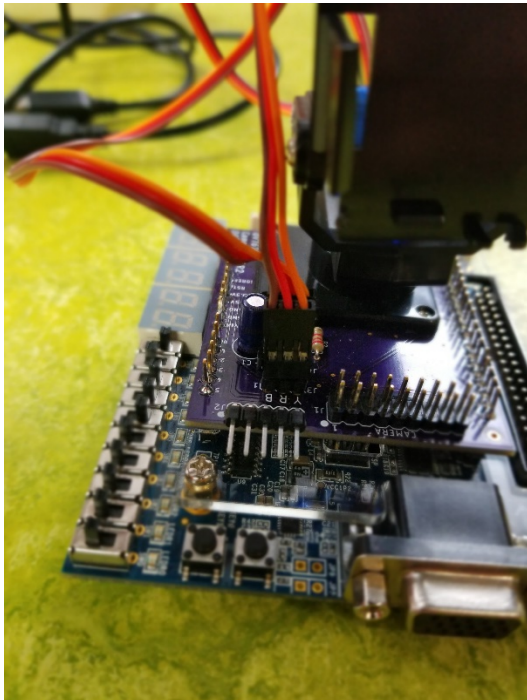
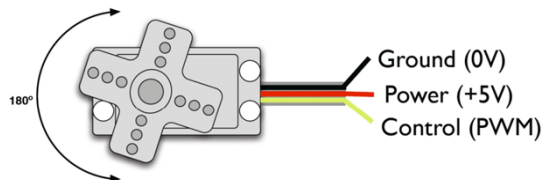
- BLOCK DIAGRAM
- WIRING
- PIN CONFIGURATIONS

BLOCK DIAGRAM



WIRING

The diagram above shows the entire computer system. The internal processor with the external components around it, are all wired up with the inputs and output of the board. The servo component's outputs are connected to Pin J3 and J4 to allow passage to the servo on the board mount. The rest of the Arduino IO's are mapped to the Arduino shield and can be used as inputs and outputs to connect multiple peripherals.



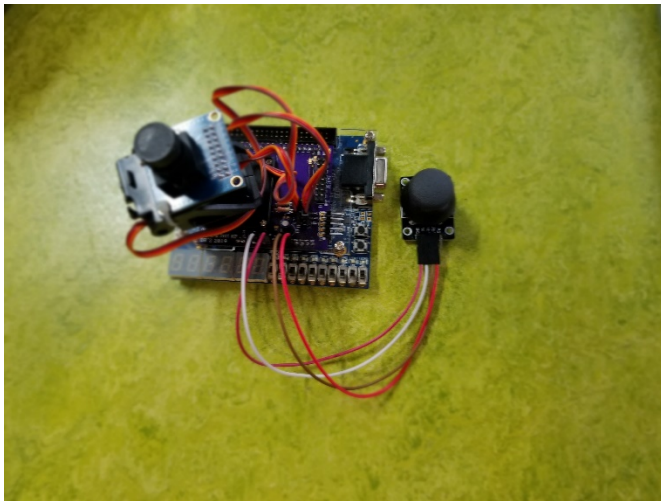
The Red wire supplies power to the servo. The black wire is the ground and the yellow wire is

the control line through which a pulse is supplied to the control chip inside the servo. These wires should be connected to DE-10 lite in such a way that is showed below in the picture.

We created a 1:2 demultiplexer for choosing between servo 1 and servo 2. If the output of the demux is 0 then servo 1 will work and if the output of the demux is 1 then servo 2 will work.

This enable select signal is bit 8. To power the servo we used an active high reset signal. The servo is powered up only if this signal is high which our bit 9 and the remaining bits 0-7 is our actual data. This was added to our DE_10 computer system as shown below.

vRx of the joystick is connected to channel 1 analog pin 1 and vRy of the connected to the channel 2 analog pin 2 as shown in the picture below. Ground and Vcc of the joystick are connected the ground and Vcc of our board.



The camera must be connected accordingly in order to communicate with the i2c peripheral in the de10 Lite board. To scope the clock and data lines, the analog discovery kit's logic 0 and logic 1 should be connected to the SCL and SDA pins on the PCB, which is connected to Arduino pin 14 and Arduino pin 15 on the board.

PIN CONFIGURATIONS

Table 3-7 Show all Pin Assignment of Expansion Headers

Signal Name	FPGA Pin No.	Description	I/O Standard
GPIO_0	PIN_V10	GPIO Connection [0]	3.3-V LVTTL
GPIO_1	PIN_W10	GPIO Connection [1]	3.3-V LVTTL
GPIO_2	PIN_V9	GPIO Connection [2]	3.3-V LVTTL
GPIO_3	PIN_W9	GPIO Connection [3]	3.3-V LVTTL
GPIO_4	PIN_V8	GPIO Connection [4]	3.3-V LVTTL
GPIO_5	PIN_W8	GPIO Connection [5]	3.3-V LVTTL
GPIO_6	PIN_V7	GPIO Connection [6]	3.3-V LVTTL
GPIO_7	PIN_W7	GPIO Connection [7]	3.3-V LVTTL
GPIO_8	PIN_W6	GPIO Connection [8]	3.3-V LVTTL
GPIO_9	PIN_V5	GPIO Connection [9]	3.3-V LVTTL
GPIO_10	PIN_W5	GPIO Connection [10]	3.3-V LVTTL
GPIO_11	PIN_AA15	GPIO Connection [11]	3.3-V LVTTL
GPIO_12	PIN_AA14	GPIO Connection [12]	3.3-V LVTTL
GPIO_13	PIN_W13	GPIO Connection [13]	3.3-V LVTTL
GPIO_14	PIN_W12	GPIO Connection [14]	3.3-V LVTTL
GPIO_15	PIN_AB13	GPIO Connection [15]	3.3-V LVTTL
GPIO_16	PIN_AB12	GPIO Connection [16]	3.3-V LVTTL
GPIO_17	PIN_Y11	GPIO Connection [17]	3.3-V LVTTL
GPIO_18	PIN_AB11	GPIO Connection [18]	3.3-V LVTTL
GPIO_19	PIN_W11	GPIO Connection [19]	3.3-V LVTTL
GPIO_20	PIN_AB10	GPIO Connection [20]	3.3-V LVTTL
GPIO_21	PIN_AA10	GPIO Connection [21]	3.3-V LVTTL
GPIO_22	PIN_AA9	GPIO Connection [22]	3.3-V LVTTL
GPIO_23	PIN_Y8	GPIO Connection [23]	3.3-V LVTTL
GPIO_24	PIN_AA8	GPIO Connection [24]	3.3-V LVTTL
GPIO_25	PIN_Y7	GPIO Connection [25]	3.3-V LVTTL
GPIO_26	PIN_AA7	GPIO Connection [26]	3.3-V LVTTL
GPIO_27	PIN_Y6	GPIO Connection [27]	3.3-V LVTTL
GPIO_28	PIN_AA6	GPIO Connection [28]	3.3-V LVTTL
GPIO_29	PIN_Y5	GPIO Connection [29]	3.3-V LVTTL
GPIO_30	PIN_AA5	GPIO Connection [30]	3.3-V LVTTL
GPIO_31	PIN_Y4	GPIO Connection [31]	3.3-V LVTTL
GPIO_32	PIN_AB3	GPIO Connection [32]	3.3-V LVTTL
GPIO_33	PIN_Y3	GPIO Connection [33]	3.3-V LVTTL
GPIO_34	PIN_AB2	GPIO Connection [34]	3.3-V LVTTL
GPIO_35	PIN_AA2	GPIO Connection [35]	3.3-V LVTTL

The Arduino shield that is attached to the board has been connected as input/output multipurpose pins. These pins allow for data to be written to the peripherals and written to the board. Also, the read commands would also work using the pins mentioned. Certain pins have been given special tasks such as Arduino pin 15 that provides a modified system clock to the camera and pin's 17 and 19 control the pulse width modulation that move and control the servos mounted on top of the Arduino shield. These pins can be utilized by the user via the main Quartus VHDL code that is allocated as top level.

SEGMENT SEVEN DISPLAY SYSTEM

IN THIS SECTION:

- OVERVIEW OF THE PERIPHERAL
- MEMORY MAP AND REGISTER USAGE
- THEORY OF OPERATION
- HARDWARE CONNECTIONS
- TESTING AND USAGE

OVERVIEW OF THE PERIPHERAL

The DE 10 Lite Board has a seven-segment display that can be used to display letters, numbers and symbols. It has six seven segment displays where each segment has an assigned base address and by writing to it can allow us to display letters, numbers and symbols. Each segment is a led and it should be high so that we can display.

MEMORY MAP AND REGISTER USAGE

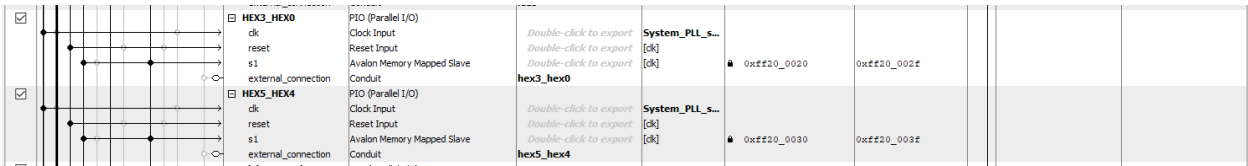


Figure 1 QSYS CAPTURE OF SEGMENT SEVEN DISPLAY

The above picture shows that HEX zero to HEX three is assigned to one base address and HEX four and HEX five is assigned to one base address because each HEX has seven segments and each segment should be high to turn on the led to display and this can be done by writing a ‘1’ to turn the led high and ‘0’ to turn the led off. Based on the number, symbol and letter, the base address can be modified accordingly.

THEORY OF OPERATION

The seven segments can be turned on or off by making each segment to a high and low to turn off. The image below shows, the position of each segment that needs to be high to turn on the led and display something. The segment seven needs an 8-bit input and gives out a 8 bit output. Each bit specifies a position on the actual Led segment as shown in the picture below. Our board has six seven segment displays and the mapping of each segment is like the picture shown below. HEX 0 to HEX 3 has a 32-bit base address assigned to it and HEX 4 and HEX 5 has a 32-bit address assigned to but in which only 16 bits are used.

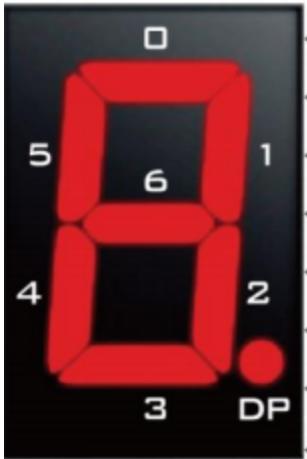


Figure 2 Position of each bit on the Display

HARDWARE CONNECTIONS

The Segment seven display was inbuilt and no external hardware connections were made to work on the segment seven display.

TESTING AND USAGE

The picture below shows the bit patten to display each number. Based on the local we want to display these numbers. the bit patterns below can be written on each address.

```
#define seg7_zero  0b00111111
#define seg7_one   0b00000110
#define seg7_two   0b01011011
#define seg7_three 0b01001111
#define seg7_four  0b01100110
#define seg7_five  0b01101101
#define seg7_six    0b01111101
#define seg7_seven  0b00000111
#define seg7_eight 0b01111111
#define seg7_nine  0b01101111
```

The code below explains how to turn on all the leds on all the segments

```
//write seg 0 to 3
*(seg7disp03) |= (0xFFFFFFFF);
//write seg 4 and 5
*(seg7disp45) |= (0xFFFF);
```

The code below shows how to clear the segment seven display

```
//clear
*(seg7disp03) &= ~(0xFFFFFFFF);
```

SERVO SUB-SYSTEM

IN THIS SECTION:

- OVERVIEW OF THE PERIPHERAL
- MEMORY MAP AND REGISTER USAGE
- THEORY OF OPERATION
- HARDWARE CONNECTIONS
- TESTING AND USAGE

Overview of the peripheral

A servo motor is a device that is widely used in radio-controlled model cars and airplanes. It is basically a DC motor that rotates which is very accurate and fast in responding to command signals. It consists of Gears, 3potentiometer, DC motor and a circuit. For a servo motor to function, it requires a pulse width generated periodically. The time of each pulse determines the position of the servo. Our servo has a range of pulse width 1ms, 1.5ms, 2ms which represents - 45°, 0°, 45° respectively. The time period required for every pulse width is 20ms. We are going to use this peripheral to move our camera in a specified direction.

Theory of operation

A servo motor is a DC motor that works on pulse width generated periodically as shown in the picture below. The time of each pulse determines the position of the servo based on the range of frequencies. Using the processor clock on our DE10-lite would require a very large counter. To overcome this obstacle a clock divider was used to reduce the clock speed to a 200 KHz speed. This was done by toggling an output signal every 125 rising edges. After that the we constructed a servo component that used a counter to create a 20 ms period wave. The counter had to count up to 4000 rising edges then reset to 0. A conditional statement was used to control the PWM signal to the Physical servo. In order to achieve a 1ms, 1.5ms, and 2ms long high signal, the amount of time the signal was set to High was 200 plus the input from the switches or register data input.

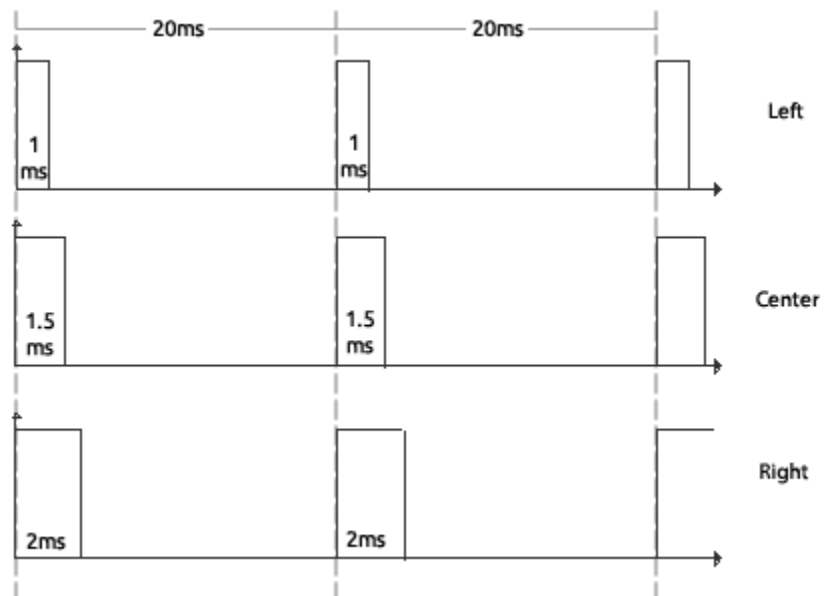


Figure 1: Pulse width Modulation of the servo motor

Memory map and register usage

The Servo_control register was created using the Qsis program, and memory mapped to location 0xff20_0070. Using the NIOS II eclipse programing Environment, writing to the register was possible. The outputs of the register are used to provide input signal for the servo control units. Servo_control is a 10-bit register, with bits 7 down to 0 providing data for the pulse width, bit 8 used for Enable Select, which decides which servo is active, bit 9 resets both servo units at once.


<input checked="" type="checkbox"/>		Servo_control clk Clock Input reset Reset Input s1 Avalon Memory Mapped Slave external_connection Conduit	PIO (Parallel I/O) Double-click to export Double-click to export Double-click to export	System_PLL... [clk] [clk]	0xff20_0070
-------------------------------------	---	--	---	--	-------------

Figure 2: Adding Servo Component in Qsys

Hardware connections

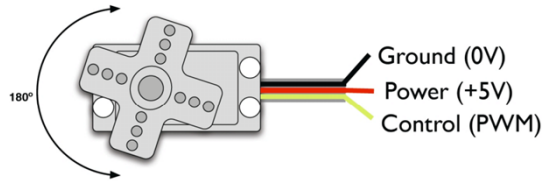


Figure 3: Servo Motor Connections

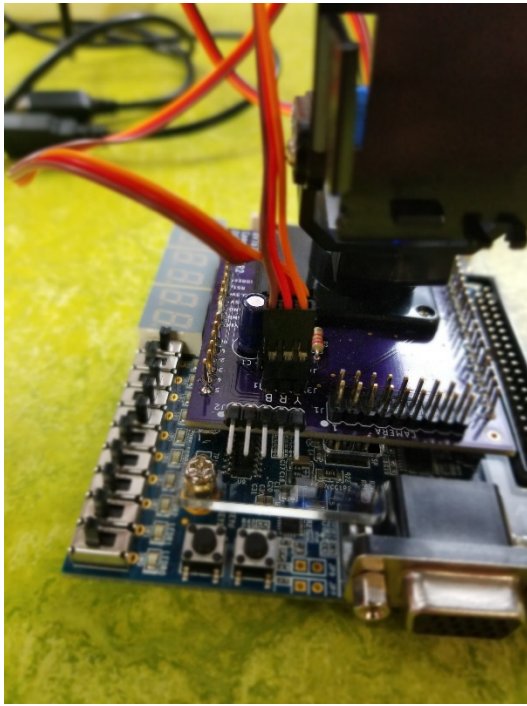


Figure 4: Servo Motor Connection to the Board's J3 and J4

The Red wire supplies power to the servo. The black wire is the ground and the yellow wire is the control line through which a pulse is supplied to the control chip inside the servo. These wires should be connected to DE-10 lite in such a way that is showed below in the picture.

We created a 1:2 demultiplexer for choosing between servo 1 and servo 2. If the output of the demux is 0 then servo 1 will work and if the output of the demux is 1 then servo 2 will work.

This enable select signal is bit 8. To power the servo we used an active high reset signal. The servo is powered up only if this signal is high which our bit 9 and the remaining bits 0-7 is our

actual data. This was added to our DE_10 computer system as shown below.

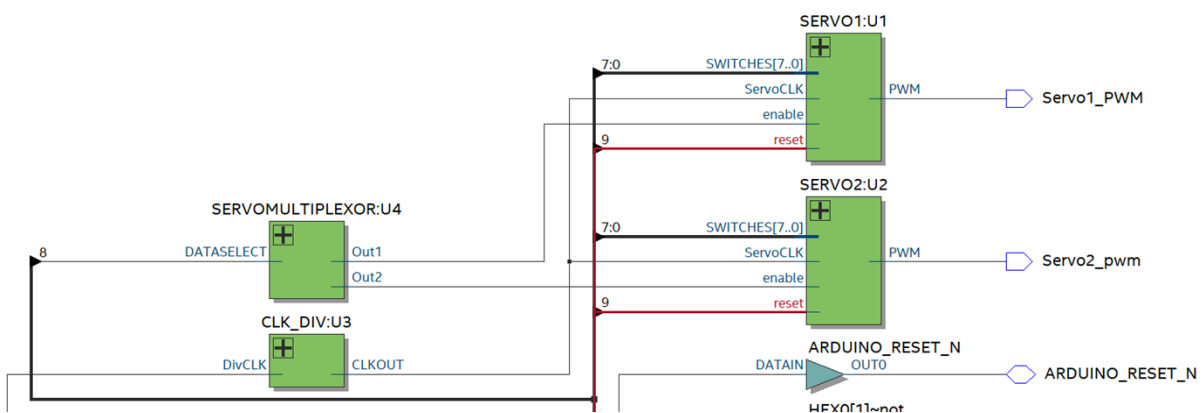
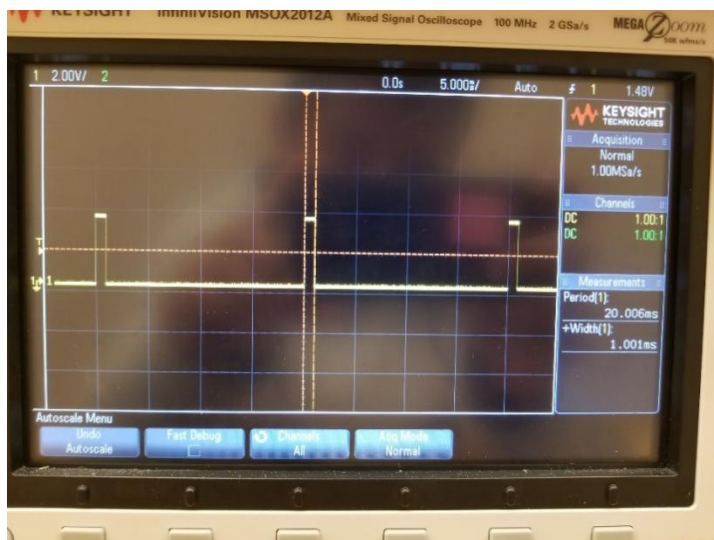


Figure 5: RTL viewer for the internal connections made for servo functionality

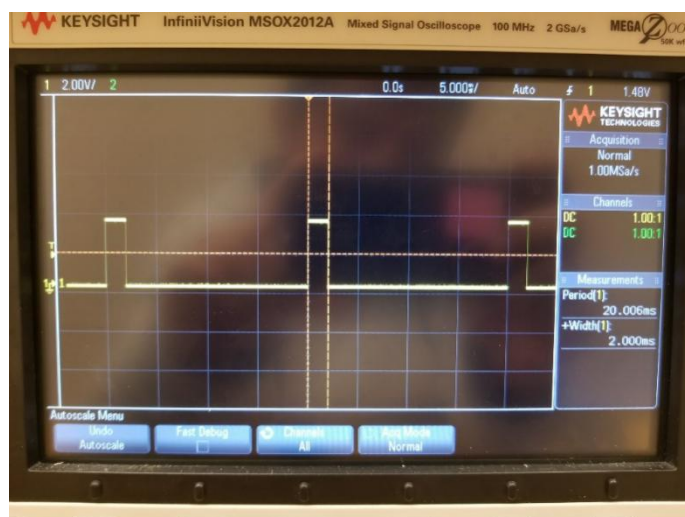
Testing and usage

To confirm our servo's functionality, tested by sending data to the servo through the switches and confirmed the functionality based on the oscilloscope results show below at position 0, 100, 200. According to the servo components manual, for position 0, a 1ms pulse should be generated, for position 100, 1.5ms pulse should be generated and for position 200, 2ms pulse should be generated. The results matched with the actual waveforms in the component's datasheet.

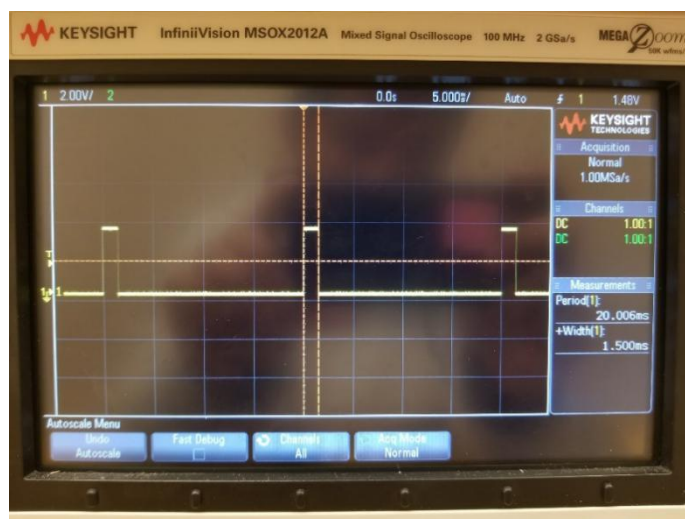
For 0:



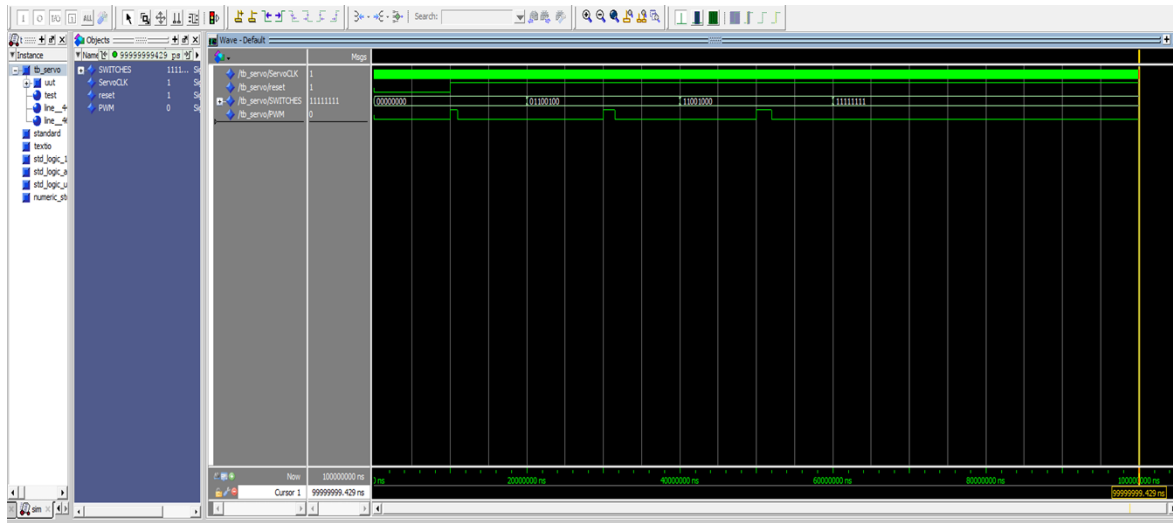
For 200:



For 100:



Testbench Results for testing the servo's functionality with our clock divider



After creating a block of memory for the servo, we initialized a pointer to the servo's memory address. We then wrote binary 0, 100, 200 to verify the servo's functionality.

Code Sample for moving the Servo in the left direction:

```
#define Servo (volatile alt_u32*) SERVO_CONTROL_BASE//9: reset 8: Servo Select: 7..0:
Count/ Position
//
```

```
#define Servo_Data 0
#define Servo_EnableSelect 8// 0: Servo2, 1: Servo1
//decides whether the servo is on or ff
#define Servo_Reset 9//Active low
```

```
void allLeft(){
//clear address
*(Servo) &= ~(0x3FF);
//turn off the servo
*(Servo) |= (0<<Servo_Reset);
//enable servo 1
*(Servo) |= (1<< Servo_EnableSelect);
//write 200 to the address to move left
*(Servo)|= (0xC8<<Servo_Data);
//turn on the servo
*(Servo) |= (1<<Servo_Reset);
usleep(200000);
//turn off the servo
*(Servo) |= (0<<Servo_Reset);
```

}

JOYSTICK CONTROL SUBSYSTEM

IN THIS SECTION:

- OVERVIEW OF THE PERIPHERAL
- MEMORY MAP AND REGISTER USAGE
- THEORY OF OPERATION
- HARDWARE CONNECTIONS
- TESTING AND USAGE

Introduction:

Our project needs a joystick to rotate the camera in the direction specified by user. It works with a pair of analog inputs. The joystick consists of two potentiometers placed at 90 degrees and are connected to a short stick. Each potentiometer represents a specific axis. It has five pins, ground, vcc of 5v, vRx, vRy, Sw. vRx is the horizontal analog signal, vRy is the vertical analog signal and SW is the clicking switch.



Figure 1 Joystick Peripheral

Theory of Operation:

The joystick produces an output voltage varying from 0v to 5v based on its position Rx produces the voltage value when the joy stick is moved in x-direction and Ry produces the voltage value when the joystick is moved in y direction. When the joystick is moved, there should be a change in values depending on the position.

Memory Map:

The ADC_PLL and joystick_ADC was mapped to the memory address as shown below .

joystick_ADC.sequencer_csr	0xff20_0210 - 0xff20_0217	0xff20_0210 - 0xff20_0217	
joystick_ADC.sample_store_csr	0xff20_0400 - 0xff20_05ff	0xff20_0400 - 0xff20_05ff	

Figure 2 Memory address of PLL and the Joystick_ADC

Control Register Layout:

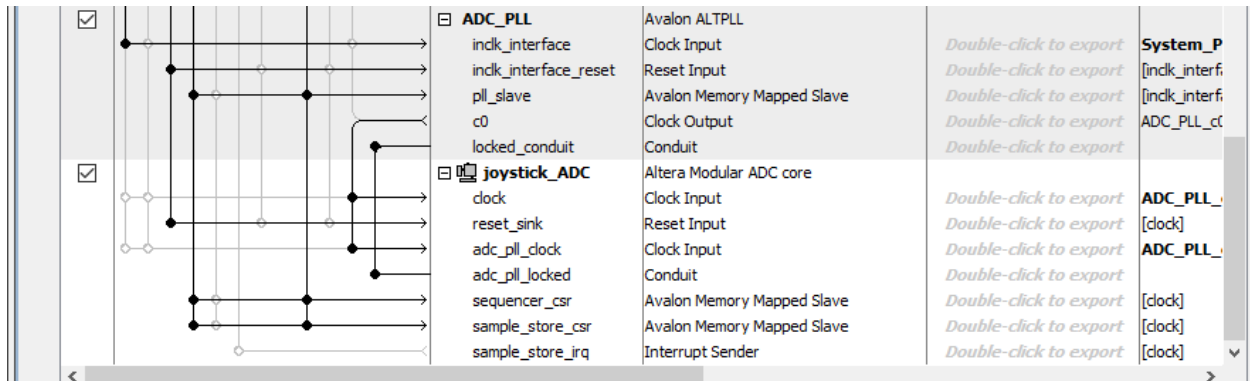


Figure 3: Qsys Connections

Two new registers were added to the Computer System. These consist of The ADC_PLL register and the JoyStick_ADC register. The ADC_PLL provides the ADC with the proper clock frequency. The ADC_PLL provided a 10 MHz clock from the initial 50MHz internal clock to the Analog to digital converter. The JoyStick_ADC was the Analog to digital converter used to convert the analog signals from a joystick to digital signals the Servos could use. This was done by using the Sequencer register and the Sample register. The Sequencer allows the option of continuous conversions or single conversions. The Sample register hold holds the values of the converted analog signal and keeps updating after every conversion.

Wiring:

vRx of the joystick is connected to channel 1 analog pin 1 and vRy of the connected to the channel 2 analog pin 2 as shown in the picture below. Ground and Vcc of the joystick are connected the ground and Vcc of our board.

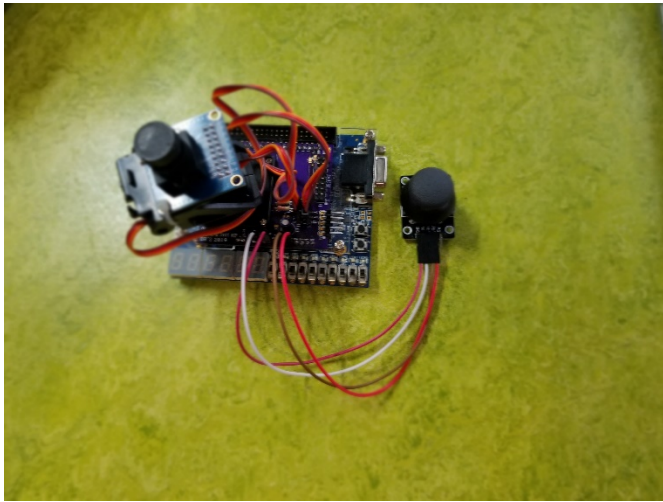


Figure 4: Hardware Connections

API Documented:

The joy stick using the sequencer core registers and the sample storage core register. The command register in the sequencer core register must be used to turn the run the sequencer to get the values. The bit 0 of the register should be a 1 to run the sequencer and 0 to stop the sequencer. The ADC sample register is a 32 bit address but the adc sample contained in it is 12 bit. It has 64 slots, channel 1 is slot 0 and an offset of 4 added to it is channel 2. All the

initializations are shown below.

```
#include <stdio.h>
#include <system.h>
#include <unistd.h>
#include "alt_types.h"

#ifndef JOYSTICKAPI_H_
#define JOYSTICKAPI_H_

//addresses
#define JoyStick_Sampler_Register (volatile alt_u32*) JOYSTICK_ADC_SAMPLE_STORE_CSR_BASE
#define JoyStick_Sampler_Channel_1 (volatile alt_u32*) 0xff200400
#define JoyStick_Sampler_Channel_2 (volatile alt_u32*) 0xff200404
#define JoyStick_Sequencer (volatile alt_u32*) JOYSTICK_ADC_SEQUENCER_CSR_BASE

#define PLL (volatile alt_u32*) ADC_PLL_BASE

void JoyStick_init();
int getY();
int getX();

#endif
```

X direction is read from Channel and Y direction is read from channel 2 as shown below. The read values are converted by dividing it by 20 and sent to the servo controller register to rotate the servo accordingly.

```
#include <stdio.h>
#include "System.h"
#include "alt_types.h"
#include "joy_stick.h"
#include "servo.h"
//initialize the joystick and turning it on
void JoyStick_init() {
    *(JoyStick_Sequencer) &= ~(0xF);
    *(JoyStick_Sequencer) |= (0b1<<0);
}
//Gets the values when the Joystick is moved in Y direction
int getY() {
    return ((int) (*JoyStick_Sampler_Channel_1));
}
//Gets the values when the Joystick is moved in X direction
int getX() {
    return ((int) (*JoyStick_Sampler_Channel_2));
}
```

Validation Process and Results:

Test 1: Reading the Values from the ADC register

We were able to read the values from Channel 1 and Channel 2. We got a range of values from 0 to 4096 while we moved the joystick and 2048 for the rest position.

Test 2: Converting the read value to determine the position of the servo

We converted the read value by dividing it over 20.48 to get the value to determine the position. For example, we needed the servo to move right, so we needed a binary value of 200 written to the servo register, so we divided the read voltage value over 200 and got 20.48. we used this number as the denominator for any voltage value to convert it the value required to move the servo.

ACCELEROMETER SUB-SYSTEM

IN THIS SECTION:

- OVERVIEW OF THE PERIPHERAL
- MEMORY MAP AND REGISTER USAGE
- THEORY OF OPERATION
- HARDWARE CONNECTIONS
- TESTING AND USAGE

INTRODUCTION:

DE 10 lite board has an accelerometer peripheral that measures acceleration in three directions (X-axis, Y-axis, Z-axis). It consists of an ADXL345 chip. The ADXL345 is a small, thin and ultralow power accelerometer that can be controlled using either a SPI or a I2C interface. For this project, we used the Accelerometer SPI mode component from the university program Library to get the x,y and z axis values using which the servo is controlled.

ARCHITECTURE/ THEORY OF OPERATION:

Accelerometer can be controlled by both I2C and SPI communications, where I2C or SPI is the Master and ADXL345 chip is the slave. The CS or the chip select is set to High to enable and I2C and low to enable SPI. For this project, we have used the SPI communication interface. The maximum SPI clock speed is 5MHz. if the power is applied to the ADXL345 chip before the clock polarity and phase of the host processor are configured, the CS pin should be brought to high. Operation at an output data rate at 800Hz is recommended and anything higher than that may cause undesirable effects on the acceleration data.

For this project, the accelerometer SPI mode component from the university program library was used. The accelerometer core can be instantiated in Qsys. The UP-accelerometer Core can be controlled using NIOS II tools through the Hardware Abstraction Layer. This can be done by reading and writing to the memory and directly accessing the registers assigned to the Accelerometer SPI. The following sections will explain how to use the registers and the drivers to communicate with Accelerometer using the Hardware Abstraction Layer (HAL).

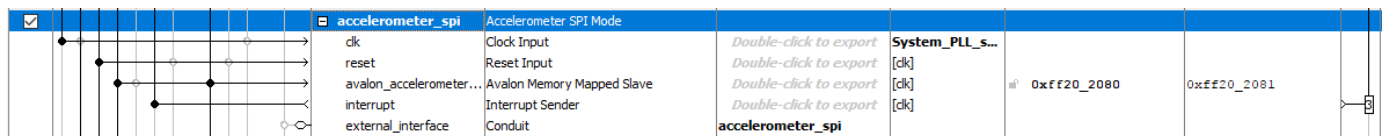
UP Accelerometer Core is packaged with C language Libraries that can be accessed through the Hardware abstraction layer. The functions used in the API will enable the user to access the accelerometer and get values for X-axis, Y-axis and Z-axis. These Function used are summarized in the table below. The output of these functions is used to move the servo in a respective direction. In order to move the servos using the accelerometer this system uses small algorithmic function to allow the data from the accelerometer to be used by the servo component. The values of the accelerometer chip span from -250 to 250. In order to reduce this number to span between 0 and 200, a constant value of 250 was added to the value of the accelerometer. Further dividing the value by 2.5 resulted in a proportionally correct number to move the servo.

Header File Used: `#include "altera_up_avalon_accelerometer_spi.h"`

Function	Description
<code>alt_up_accelerometer_spi_open_dev(Const char *name)</code>	It opens the accelerometer_spi device if found and if not found it returns NULL
<code>alt_up_accelerometer_spi_read_x_axis(alt_up_accelerometer_spi_dev *accel_spi, alt_32 *x_axis)</code>	It reads the x-axis values from both the registers, DATA0 and DATA1 and converts the value to a signed integer

alt_up_acceleromter_spi_read_y_axis(alt_up_acceleromter_spi_dev *accel_spi, alt_32 *y_axis)	It reads the y axis values from both the registers, DATAY0 and DATA Y1 and converts the value to a signed integer
alt_up_acceleromter_spi_read_z_axis(alt_up_acceleromter_spi_dev *accel_spi, alt_32 *z_axis)	It reads the z axis values from both the registers, DATAZ0 and DATAZ1 and converts the value to a signed integer

Table 1: Functions Used to control the Accelerometer

MEMORY MAP:


<input checked="" type="checkbox"/>	accelerometer_spi	Accelerometer SPI Mode				
	clk	Clock Input	Double-click to export	System_PLL_s...		
	reset	Reset Input	Double-click to export	[clk]		
	avalon_accelerometer...	Avalon Memory Mapped Slave	Double-click to export	[clk]	0xff20_2080	0xff20_2081
	interrupt	Interrupt Sender	Double-click to export	[clk]		
	external_interface	Conduit				

The SPI controlled accelerometer component was placed into the computer system using the QSYS program. Mapping this component into the computer goes as follows: the clock connects to the system clock of the “System_PLL” component, reset is connected to the reset source of the same component, the slave component connects to the Nios2 Data master and the JTAG’s Master, and the interrupt was connected to the “irq” register of the Nios2 processor. The conduit is then connected through the projects main VHDL project.

Slider_Switches.s1	0xff20_0040 - 0xff20_004f	0xff20_0040 - 0xff20_004f
SysID.control_slave	0xff20_2040 - 0xff20_2047	0xff20_2040 - 0xff20_2047
accelerometer_spi.avalon_acceleromet...	0xff20_2080 - 0xff20_2081	0xff20_2080 - 0xff20_2081
j2c_0.csr	0xff20_2100 - 0xff20_213f	0xff20_2100 - 0xff20_213f

The address of the accelerometer for both the Nios2 processor and the JTAG was mapped to the address above. Theoretically this value can be changed to any open address of one byte long.

CONTROL REGISTER LAYOUT:

Offset in bytes	Register Name	Read/Write	7...6	5...0
0	address	W	0	Addr
1	data	R/W	Data	

Figure1: Accelerometer Core Registers

Address Register: The address register is used to transmit the address of a register in the Accelerometer.

Data Register: The data register is used to read and write data to the registers in the accelerometer.

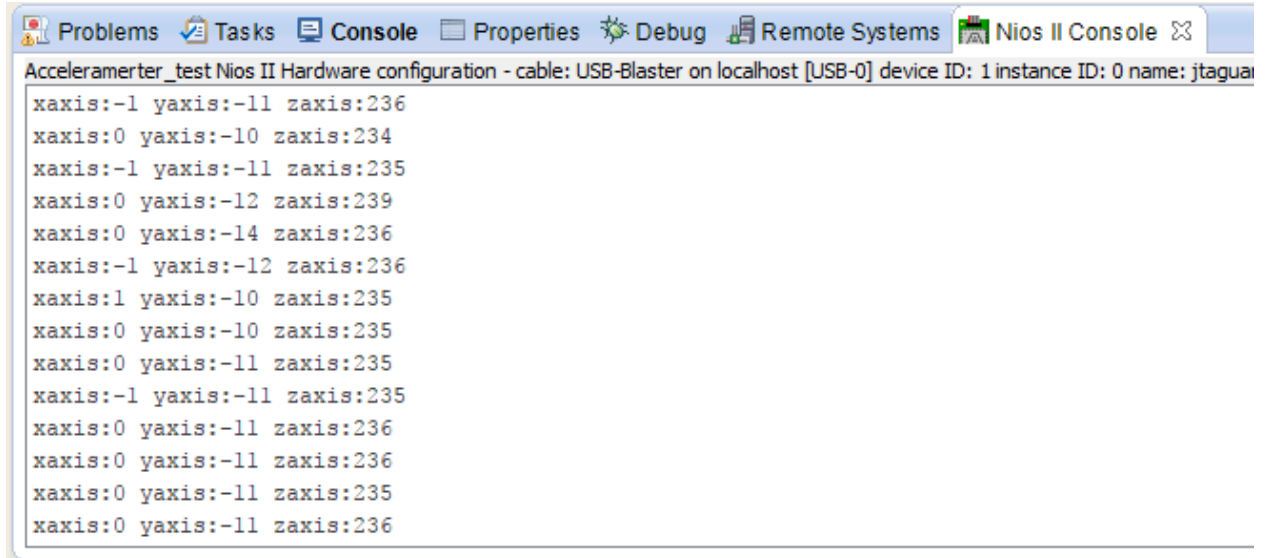
Control Register: This device has many control registers. There are eight bits long and are used to define various operations. They can be modified by a write operation and read by a read operation. It has three data registers that hold the values of X-axis, Y-axis and Z-axis.

DATA0, DATA1	Contains X axis Values
DATAY0, DATAY1	Contains Y axis Values
DATAZ0, DATAZ1	Contains Z axis Values

Table 3: Data registers that contains the axis values

RESULTS:

An example of the data from the accelerometer is shown below as a simple print statement runs the functions to call the data.



The screenshot shows the Nios II Console window with the title bar 'Accelerameter_test Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtagui'. The console output displays 15 lines of accelerometer data, each containing three values: xaxis, yaxis, and zaxis. The xaxis values range from -1 to 1, yaxis values range from -14 to -11, and zaxis values range from 234 to 236.

```

xaxis:-1 yaxis:-11 zaxis:236
xaxis:0 yaxis:-10 zaxis:234
xaxis:-1 yaxis:-11 zaxis:235
xaxis:0 yaxis:-12 zaxis:239
xaxis:0 yaxis:-14 zaxis:236
xaxis:-1 yaxis:-12 zaxis:236
xaxis:1 yaxis:-10 zaxis:235
xaxis:0 yaxis:-10 zaxis:235
xaxis:0 yaxis:-11 zaxis:235
xaxis:-1 yaxis:-11 zaxis:235
xaxis:0 yaxis:-11 zaxis:236
xaxis:0 yaxis:-11 zaxis:236
xaxis:0 yaxis:-11 zaxis:235
xaxis:0 yaxis:-11 zaxis:236

```

The accelerometer will display 3 types of data: x-axis, y-axis, and z-axis. These can be used to determine the movement of the board and control the servo component using dedicated functions.

CAMERA COMMAND CONTROL

IN THIS SECTION:

- OVERVIEW OF THE PERIPHERAL
- MEMORY MAP AND REGISTER USAGE
- THEORY OF OPERATION
- HARDWARE CONNECTIONS
- TESTING AND USAGE

Overview:

The Inter – Integrated Circuit protocol is a serial communication protocol that allows multiple slave components to communicate to one Master. It is used for short distance communications within a single device. It requires a clock signal (SCLK) and data signal (SDA) for communication. The Clock Signal is generated by the master. The I2C bus is open drain meaning they can pull the corresponding signal line low but cannot drive it high. Each signal line has Pull up resistor on it, to resistor the signal to high when no device is asserting low. The figure below shows that it is open drain with pull resistors.

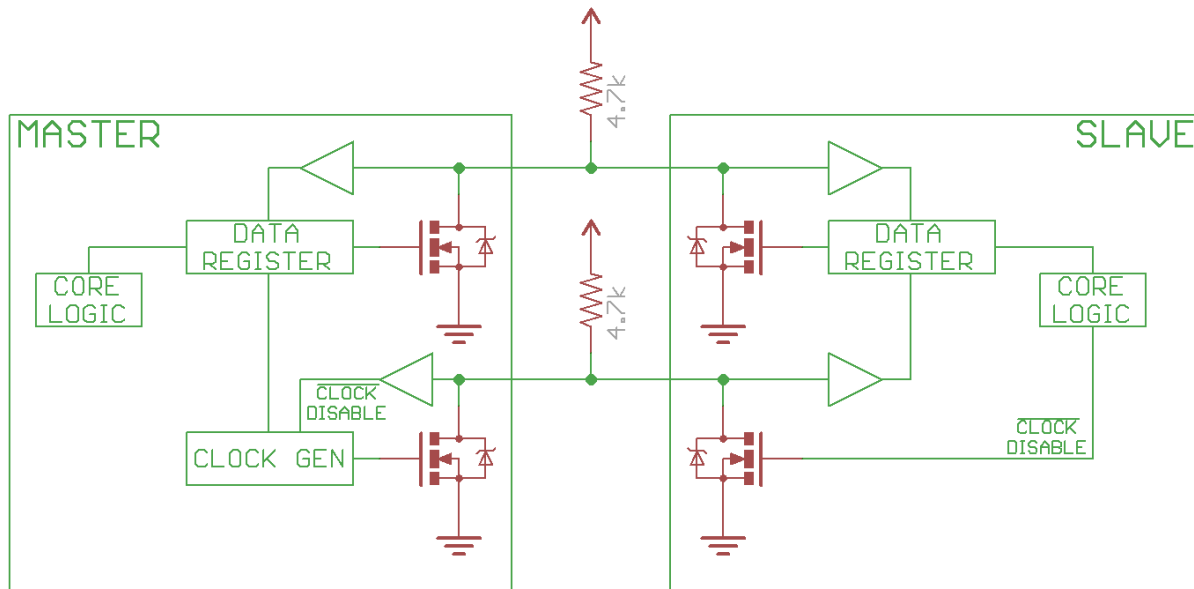
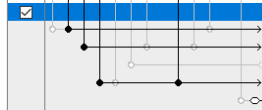


Figure 3 I2C Open Drain with pull up resistor (Fun, n.d.)

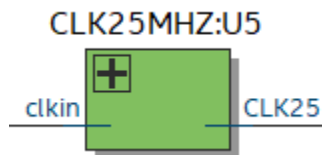
Memory Map and Register Usage:

<input checked="" type="checkbox"/>		i2c_0	Altera Avalon I2C (Master)					
		clock	Clock Input	<i>Double-click to export</i>	System_PLL_s...			
		reset_sink	Reset Input	<i>Double-click to export</i>	[dock]			
		interrupt_sender	Interrupt Sender	<i>Double-click to export</i>	[dock]			
		csr	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[dock]			
		i2c_serial	Conduit	i2c		0xff20_2100	0xff20_213f	

The I²C component was placed into the computer system using the QSYS program. Mapping this component into the computer goes as follows: the clock connects to the SDRAM clock of the “System_PLL” component, reset sink is connected to the reset source of the same component, and the interrupt was left unconnected to the system, the control slave register is connected to the master component of the Nios2 and JTAG component. The conduit is then connected through the project's main VHDL project.

SysID.control_slave	0xff20_2040 - 0xff20_2047	0xff20_2040 - 0xff20_2047
accelerometer_spi.avalon_acceleromet...	0xff20_2080 - 0xff20_2081	0xff20_2080 - 0xff20_2081
i2c_0.csr	0xff20_2100 - 0xff20_213f	0xff20_2100 - 0xff20_213f

The address of the I²C for both the Nios2 processor and the JTAG was mapped to the address above. Theoretically this value can be changed to any open address of 319 byte long.



The I²C component needs an acknowledgment from the peripheral it is communicating with, however, the camera peripheral needs a slower clock than the 50 MHz clock of the system. The component above reduces the clock speed to half the original clock speed, 25 MHz.

Theory of operation:

I2C protocol is more complex than UART or SPI. It is synchronous serial communications that is half duplex. It can have two master components that can communicate with more than two slaves as shown in Figure 2. For I2C, the Data must be valid while the clock is high. The DATA frame of I2C contains a START condition, address frame, data frame and Stop condition as show in Figure 3. Start condition is used to initiate the address frame by making the SCL high and pulling the SDA low. The address frame consists of 7bits followed by a R/W bit where 1 is a read and 0 is a write operation. The 9th bit is NACK/ACK bit. After the address frame is transmitted, the data frame will transmit. The master will leave the SCL high and depending the R/W bit the master or slave will place the data in the frame. The stop condition is when the SDA line is pulled from low to high.

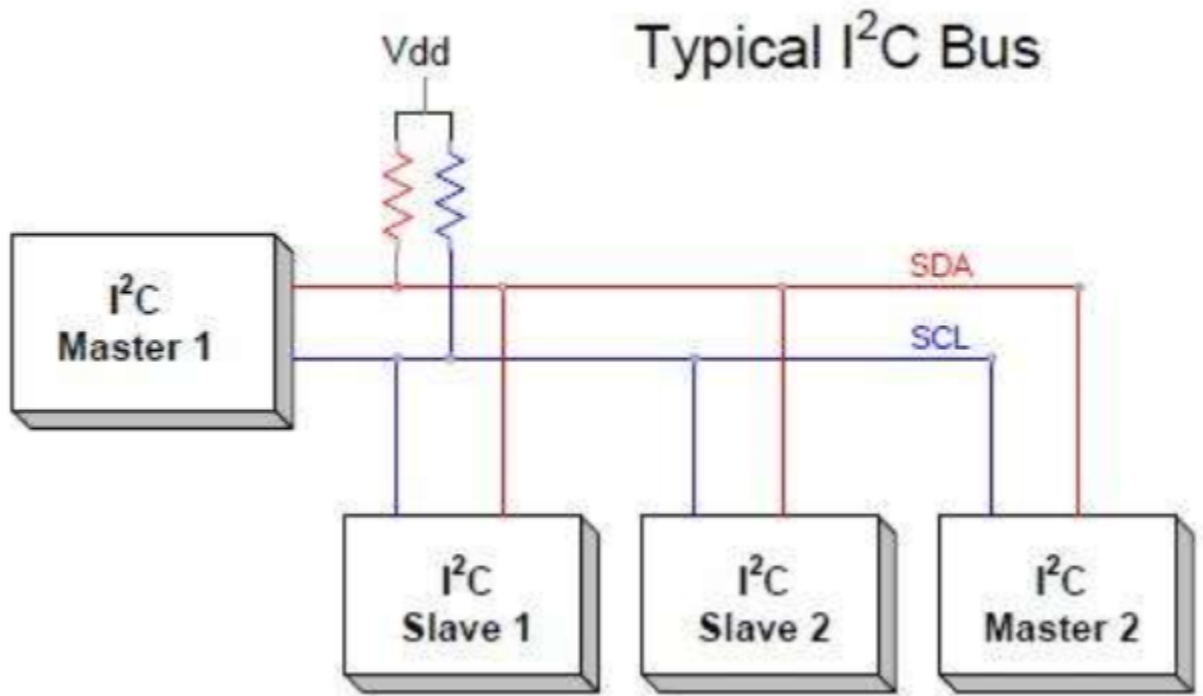


Figure 4 I2C with two masters and two Slaves

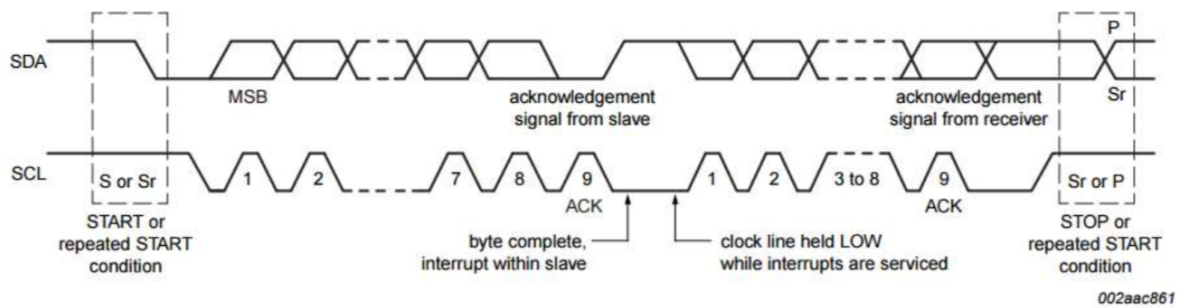


Figure 5 DATA TRANSFER ON I2C BUS

For our project, we needed I2C to communicate with camera. By adding the Avalon i2c master to our SOC we can make the i2c in DE-10 Lite board to communicate with the camera. The camera needs a 24 MHz clock, but the system clock is 50Mhz, so a component called slow clock was added to the computer system in order to communicate with the camera. PWDN signal and the RESET signal must also be connected. With i2c, we can read and write to the camera. To write “a 3-phase transmission cycle” is used and to read a “a two-phase write transmission” and “a two-phase read transmission” as shown in the figure 4 and figure 5 below. The ID address would be 7-bit 0x21 followed by bit 0 low for write and bit 1 for read. Writing data to a address and reading back from the same address should verify the functionality of I2C with the camera and also the ACK should be received from the camera.

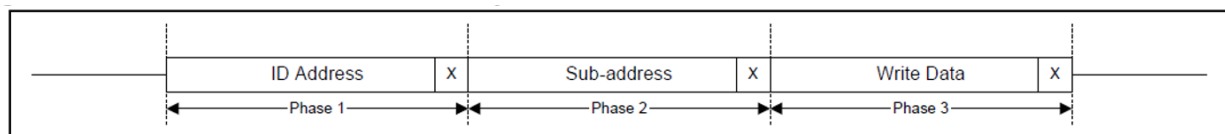
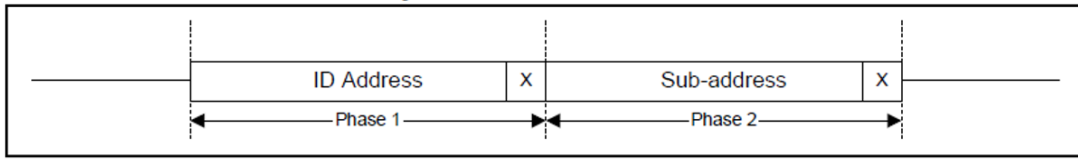


Figure 6 3-Phase Write Transmission For Write Cycle

2-Phase Write Transmission Cycle



2-Phase Read Transmission Cycle

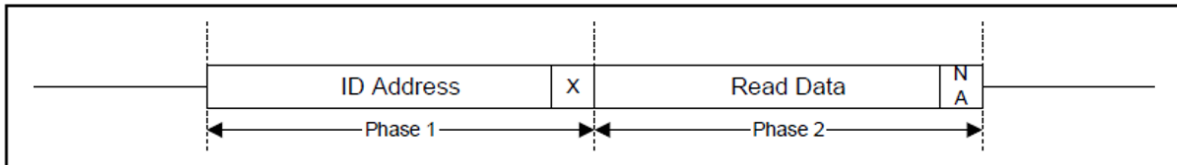
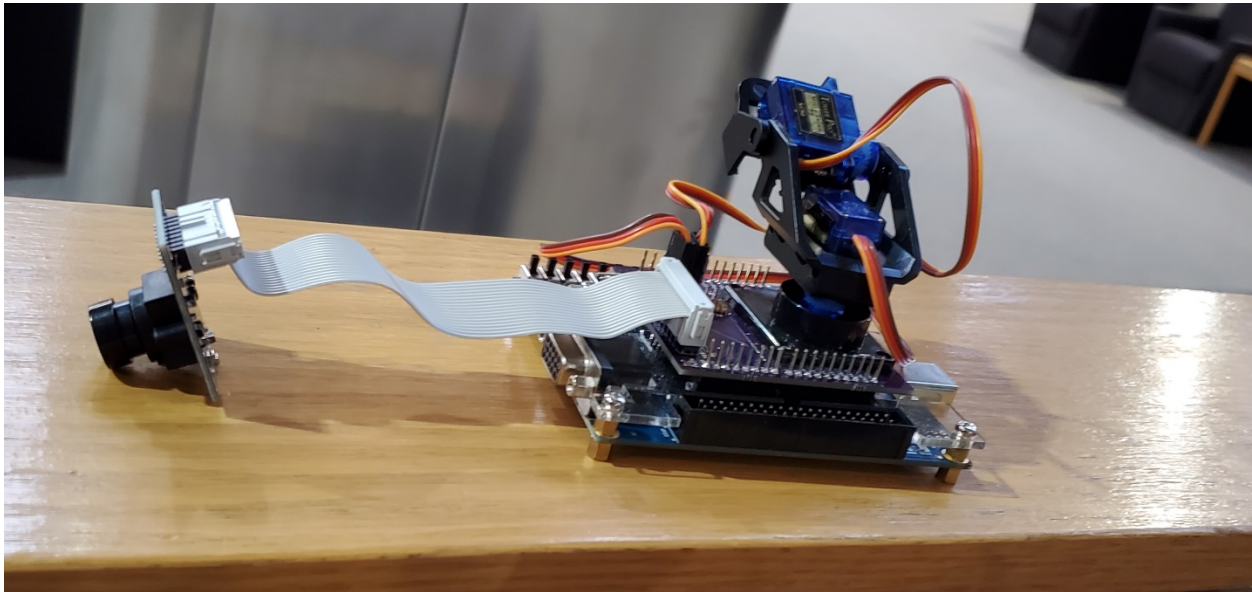


Figure 7 2 Phase write and read transmission for Read Cycle

Hardware connections:



The camera must be connected accordingly in order to communicate with the i2c peripheral in the de10 Lite board. To scope the clock and data lines, the analog discovery kit's logic 0 and logic 1 should be connected to the SCL and SDA pins on the PCB, which is connected to Arduino pin 14 and Arduino pin 15 on the board.

Testing and usage:

Write Test:

//gets the status of transmitted buffer, 0 if transmitted and 1 if failed

```
status = alt_avalon_i2c_master_tx(i2cDev, txBuffer, 2,
ALT_AVALON_I2C_NO_INTERRUPTS);

    if (status != ALT_AVALON_I2C_SUCCESS) {

        printf("Has failed");

        status=1;

    }

    else {

        printf("Success");

        status=0;

    }
```

Read Cycle Test:

//performs two phase write operation, if status 0 success and if status is 1 then failed

```
status = alt_avalon_i2c_master_tx(i2cDev, &address, 1,
ALT_AVALON_I2C_NO_INTERRUPTS);

    if (status != ALT_AVALON_I2C_SUCCESS) {

        printf("Has failed");

        status=1;

    }

    else {

        printf("Success");

    }
```

```

        status=0;
    }

    //performs two phase read operation, if status 0 success else it is failure
    status = alt_avalon_i2c_master_rx(i2cDev, 0, 1,
ALT_AVALON_I2C_NO_INTERRUPTS);

    if (status != ALT_AVALON_I2C_SUCCESS) {

        printf("Has failed");

        status=2;

    }

    else {

        printf("Success");

        status=0;

    }

```

The code above verifies the testing of the camera communicating with the board, if the read and write functions in the API will return Status with 0 if the device is able to communicate with the camera. The output verifying the result is in the figure below.

```

status for write=0
status for read=0
status for write=0
status for read=0
status for write=0
status for read=0
status for write=0
status for read=0

```

Figure 8 Verifying Read and write operations

The waveforms below explain the functionality of camera with read and write operations, as the figure 7 shows both read and write operations. Write operation is a 3 phase operation as the figure 8 shows a much zoomed in version with the ACK bit from the camera. In figure 8, the image shows the address command which is 0x21 followed by the address which is 0xFF and followed by the data return to the address, which is 0x53 with start, stop and ACK bit. In figure 9, the image shows the read operation and verifies that it read 0x53 from the address 0xFF by a doing a 2 phase write to find the address and 2 phase read to read the data written to the specific address. All the screenshots verifies the testing of the i2c communication with the camera, as it shows writing some data to any address and reading from the same address gives the previously written data.

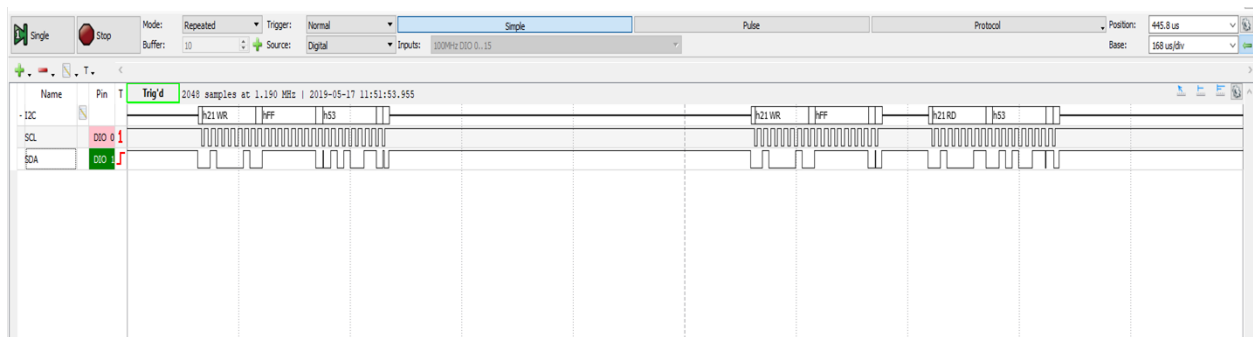


Figure 9 Read and write operations

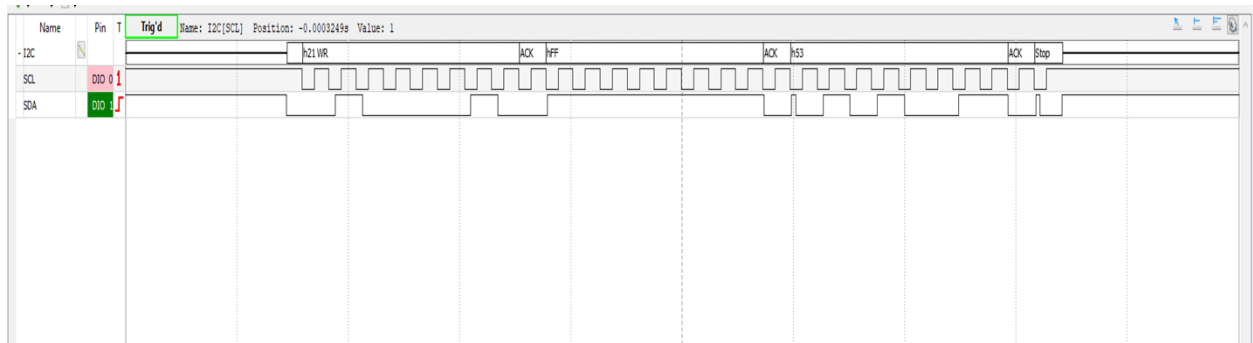


Figure 10 3-phase write operation, addr 0xFF and data 0x53

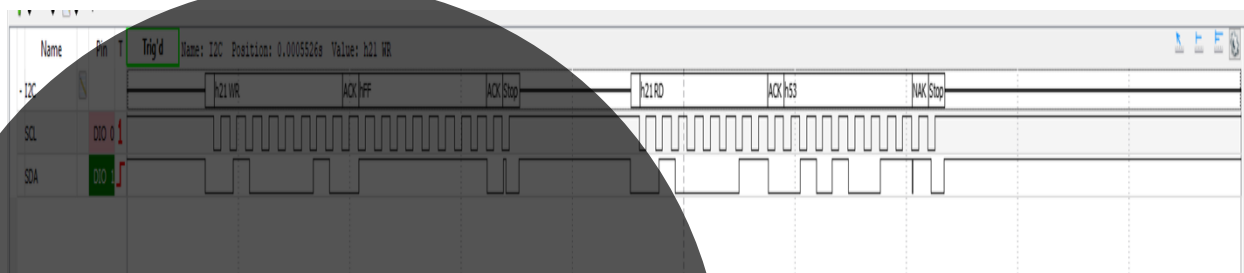


Figure 14 Read Operation, 2 phase write addr 0xFF, 2 phase read showing Data 0x53

IN THIS SECTION:

- OVERVIEW OF THE PERIPHERAL
- MEMORY MAP AND REGISTER USAGE
- THEORY OF OPERATION
- HARDWARE CONNECTIONS
- TESTING AND USAGE

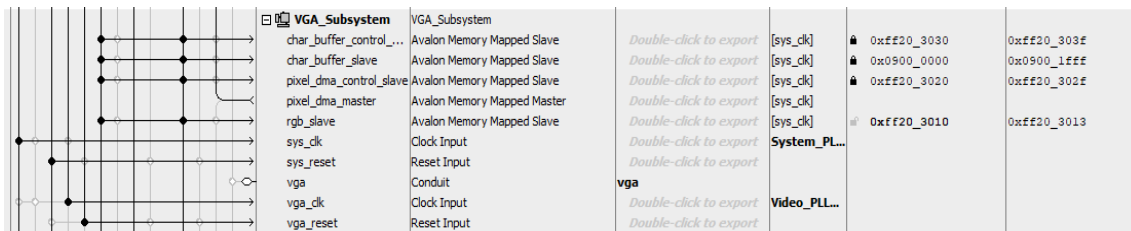
Overview of the peripheral

DE-10 Lite can provide VGA pattern function that allows the users to output color pattern to LCD monitor. A Pixel Frame Buffer is used to generate an image. This image will then be sent out to a VGA monitor with the VGA controller. The VGA system uses the Avalon streaming interface. An active low pulse of specific duration is duration is applied to the horizontal synchronization input of the monitor, which signifies the end of one row of data and the start of the new data.

Memory map and register usage

Figure 1 below shows, the registers and the memory location that is used by the VGA subsystem.

All the registers shown below was used in the Hardware abstraction layer.



Component	Register	Value	Address
char_buffer_control...	Avalon Memory Mapped Slave	[sys_clk]	0xff20_3030
char_buffer_slave	Avalon Memory Mapped Slave	[sys_clk]	0x0900_0000
pixel_dma_control_slave	Avalon Memory Mapped Slave	[sys_clk]	0xff20_3020
pixel_dma_master	Avalon Memory Mapped Master	[sys_clk]	0xff20_3010
rgb_slave	Avalon Memory Mapped Slave	[sys_clk]	0xff20_3013
sys_clk	Clock Input	System_PL...	
sys_reset	Reset Input	Video_PLL...	
vga	Conduit		
vga_clk	Clock Input		
vga_reset	Reset Input		

Figure 1 Qsys Capture showing memory address and registers

Theory of operation

To display an image or a video to a LCD or a monitor, the VGA subsystem is required. DE 10 Lite board has a 15 pin D-SUB connector for VGA output. The VGA signals are provided directly from MAX 10 FPGA and a 4-bit DAC that red green blue analog signals. The RGB output to the monitor must be off for a time period. During the data display interval, the RGB data drives each pixel in turn across the row being displayed. There is a time period called the

front porch where the RGB signals must again be off before the next hsync pulse occurs. The table shows different resolutions and durations of time period for horizontal and vertical timing.

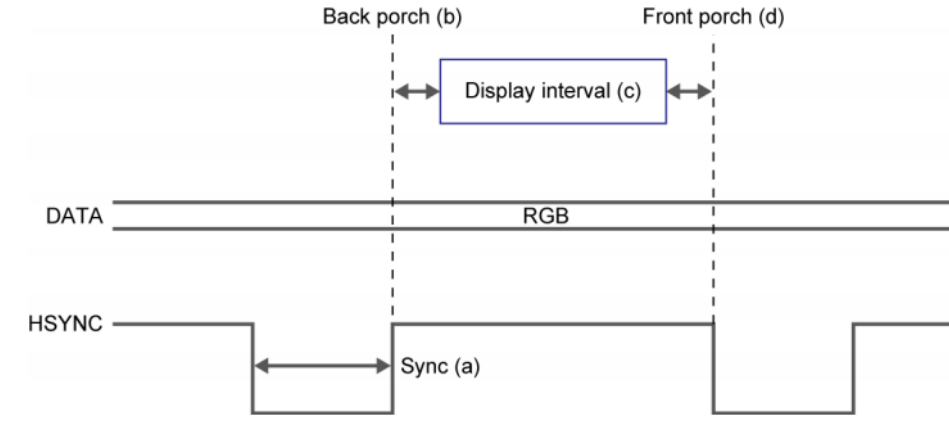


Figure 2 VGA HSYNC TIMING DIAGRAM

Table 3-9 VGA Horizontal Timing Specification

VGA mode		Horizontal Timing Spec				
Configuration	Resolution(HxV)	a(pixel clock cycle)	b(pixel clock cycle)	c(pixel clock cycle)	d(pixel clock cycle)	Pixel clock(MHz)
VGA(60Hz)	640x480	96	48	640	16	25

Table 3-10 VGA Vertical Timing Specification

VGA mode		Vertical Timing Spec				
Configuration	Resolution(HxV)	a(lines)	b(lines)	c(lines)	d(lines)	Pixel clock(MHz)
VGA(60Hz)	640x480	2	33	480	10	25

Figure 3 Table that shows Horizontal and vertical timing specifications

Hardware connections

The DE 10 lite boards output should be connected as input to the monitor's vga input, Figure 4 below Show's the hardware connections.

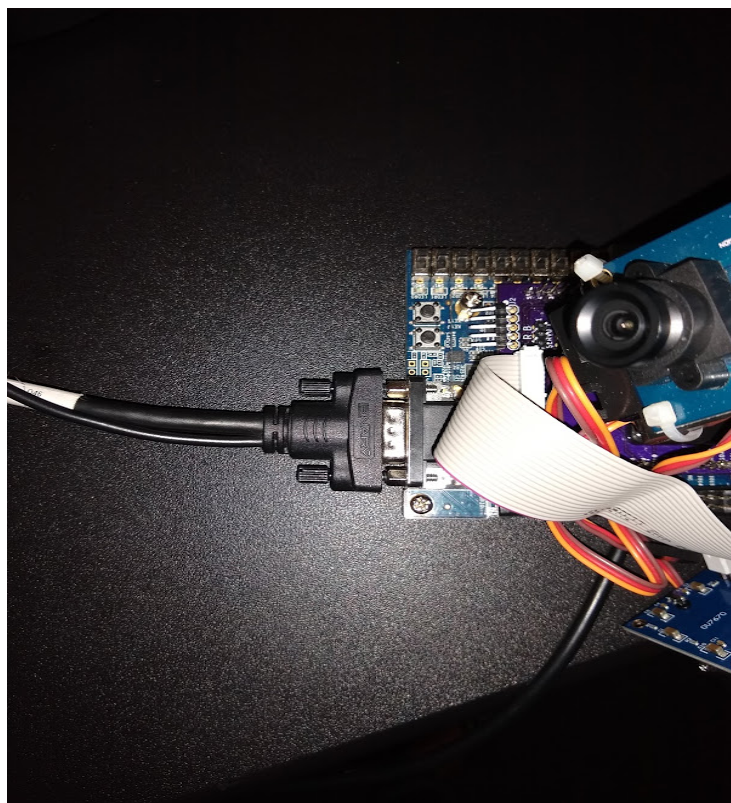


Figure 4 Hardware connections

Testing and usage

The picture below verifies that a pixel can be drawn, and the screen of the color can be changed.

The screen fill function in the Hardware abstraction layer library can be used to change the color, writing different values can change the screen color and the draw pixel function can be used to draw the library.

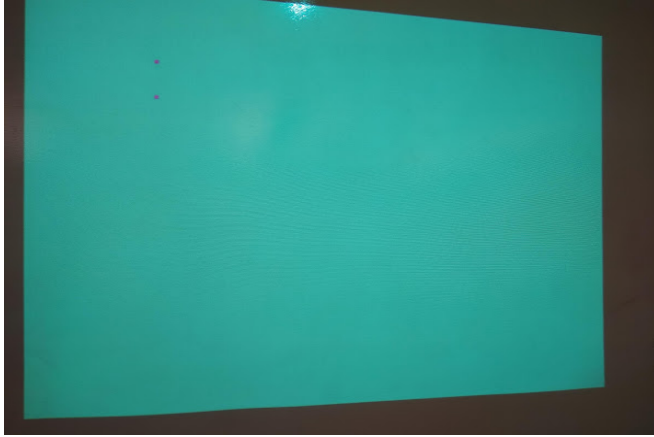


Figure 5 Change in Screen color with two pixels drawn

The picture below verifies the functionality to draw a horizontal and a vertical line. The library functions in the hardware abstraction layer can be used to draw the lines.

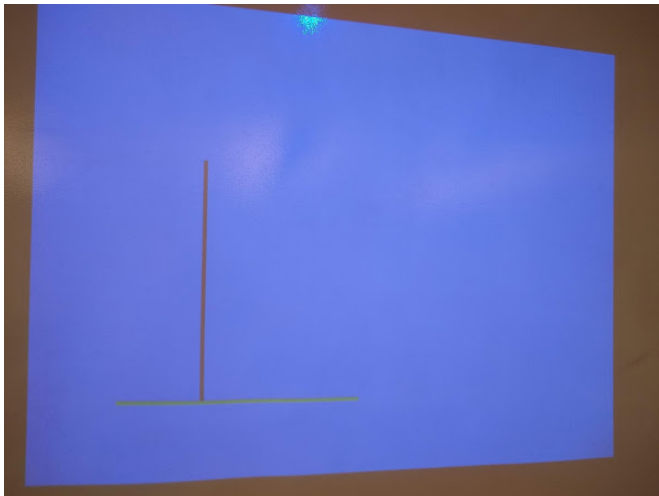


Figure 6 Vertical and horizontal line display

API documentation and code example:

```
void vga_draw_rectangle(int color_value, int x0, int y0, int x1,int y1){
    //parameters all four points, dev file pointer, color_value, back buffer
    alt_up_video_dma_draw_rectangle(pixel_buf_dma_dev,color_value, x0, y0, x1, y1,0);
}
```

The above example calls the library function in the Hardware abstraction layer to draw the rectangle.

```
void vga_clear_screen(int color_value){  
    //parameters to the library function devfile, color_value, back buffer  
    alt_up_video_dma_screen_fill(pixel_buf_dma_dev,color_value, 0);  
}
```

The above example calls the screen fill function in the Hardware abstraction layer to change the color of the monitor screen.

Problems encountered:

Segment seven:

- Writing one's in the wrong direction

Servo Motors:

- swapped the wires, we had connected brown, red and yellow instead of yellow, red brown.

- We did not clear the address before writing values to it, so it kept on adding up and nothing worked. We later identified the issue, so we cleared the address every time we wanted to write something to it.

Joystick:

- We thought Channel 1 and channel 2 were analog pin 1 and analog pin 2 on the board but it was pin 0 and pin 1 on the board.

- We didn't add an offset of 4 to read the value from channel 2

Accelerometer:

- had difficulties opening the dev file

- had vhdl errors

I2c command:

- connected the camera in the wrong direction

- was reading and writing to a wrong address

VGA system:

- qsys integration errors

- understanding the HAL functions, the parameters were so confusing

SUMMARY AND CONCLUSIONS:

Based on our knowledge in the field on computer engineering, we have completed the project that we wanted to build. we have completed all the milestones and got each component to work to its full potential. The camera module was not completed due to shortage of time. We have the I2C peripheral implemented, this peripheral can communicate with the camera. We have our VGA system for image processing. We have our servo, joystick and the accelerometer implanted to rotate the camera accordingly. We have all the sub-systems working to its full potential so that it makes it easier to communicate with camera. Throughout the quarter we have put lot of efforts and have gotten this far. Although we have faced lot of difficulties while working on each milestone as they are listed in the problems encountered section.

APPENDIX I

7seg.h

```

#include <stdio.h>
#include <system.h>
#include <unistd.h>
#include "alt_types.h"

#ifndef seg7display_H_
#define seg7display_H_

// Address
#define Seg07loc03 (volatile alt_u32*)HEX3_HEX0_BASE
#define Seg07loc45 (volatile alt_u32*)HEX5_HEX4_BASE
#define button_ptr (volatile alt_u8*)PUSHBUTTONS_BASE
#define slider_switchptr (volatile alt_u16*)SLIDER_SWITCHES_BASE
//Bit values for a number on the 7 segmented display
#define number0 0b00111111
#define number1 0b00000110
#define number2 0b01011011
#define number3 0b01001111
#define number4 0b01100110
#define number5 0b01101101
#define number6 0b01111101
#define number7 0b00000111
#define number8 0b01111111
#define number9 0b01101111
/*
 * Checks which number to write to 7 segmented display
 * parameter: int - address
 *           int - position
 * returns: void
 */
void segment7_pattern(int, int);
/*
 * Writes to the 7 segmented display
 * parameter: none
 * returns: void
 */
void display();
/*
 * Will read value from sliders and write to
 * 7 segmented display
 * parameter: none
 * return: void
 */
void number_display();

#endif

```


7seg.c

```

#include <stdio.h>
#include "system.h"
#include "alt_types.h"
#include <unistd.h>
#include "7seg.h"
//global variable for the slider value
static alt_u16 slider_value;
* Will read value from sliders and write to
void number_display() {
    //clears
    *(Seg071cc03 ) &= ~(0xFFFFFFFF);
    int temp;//temporary value
    //slider value
    slider_value = *(slider_switchptr );
    for (int i = 0; i < 4; i++) { //checks one's place then removes
        temp = slider_value % 10;
        slider_value = slider_value / 10;
        segment7_pattern(temp, i);
    }
}
/*
* Checks which number to write to 7 segmented display
* parameter: int - address
*             int - position
* returns: void
*/
void segment7_pattern(int digit, int position) {
    switch (digit) {
        case 0:
            display(number0, position);
            break;
        case 1:
            display(number1, position);
            break;
        case 2:
            display(number2, position);
            break;
        case 3:
            display(number3, position);
            break;
        case 4:
            display(number4, position);
            break;
        case 5:
            display(number5, position);
            break;
        case 6:
            display(number6, position);
            break;
        case 7:
            display(number7, position);
            break;
        case 8:
            display(number8, position);

```

Page 1

```
7seg.c

    break;
case 9:
    display(number9, position);
    break;
}
}
/*
 * Writes to the 7 segmented display
 * parameter: none
 * returns: void
 */
void display(int number, int position) {
    if (position == 0)
        *(Seg07loc03) |= (number << position * 8);
    if (position == 1)
        *(Seg07loc03) |= (number << position * 8);
    if (position == 2)
        *(Seg07loc03) |= (number << position * 8);
    if (position == 3)
        *(Seg07loc03) |= (number << position * 8);
}
```

ServoApi.h

```

#include <stdio.h>
#include <system.h>
#include <unistd.h>
//#include "alt_types.h"

#ifndef ServoAPI_H_
#define ServoAPI_H_

//addresses

#define Servo (volatile alt_u32*) SERVO_CONTROL_BASE//9: reset 8: Servo Select:
7..0: Count/ Position
#define Servo_Data 0
#define Servo_EnableSelect 8// 0: Servo2, 1: Servo1
#define Servo_Reset 9//Active low

//Intializes both servo position to the intial position.
void init();
void allLeft();
void allRight();
void allUp();
void allDown();
void movLeft();
void movRight();
void span();

#endif

```

```

ServoApi.c

#include <stdio.h>
#include "System.h"
#include "alt_types.h"
#include "ServoApi.h"

void init() {
    *(Servo) &= ~(0x3FFF);
}

void allLeft() {
    *(Servo) &= ~(0x3FFF);
    *(Servo) |= (0<<Servo_Reset);
    *(Servo) |= (1<< Servo_EnableSelect);
    *(Servo) |= (0x00<<Servo_Data);
    *(Servo) |= (1<<Servo_Reset);
    usleep(200000);
    *(Servo) |= (0<<Servo_Reset);
}

void allRight() {
    *(Servo) &= ~(0x3FFF);
    *(Servo) |= (0<<Servo_Reset);
    *(Servo) |= (1<< Servo_EnableSelect);
    *(Servo) |= (0b00000000<<Servo_Data);
    *(Servo) |= (1<<Servo_Reset);
    usleep(200000);
    *(Servo) |= (0<<Servo_Reset);
}

void allUp() {
    *(Servo) &= ~(0x3FFF);
    *(Servo) |= (0<<Servo_Reset);
    *(Servo) |= (0<< Servo_EnableSelect);
    *(Servo) |= (0b00000000<<Servo_Data);
    *(Servo) |= (1<<Servo_Reset);
    usleep(200000);
    *(Servo) |= (0<<Servo_Reset);
}

void allDown() {
    *(Servo) &= ~(0x3FFF);
    *(Servo) |= (0<<Servo_Reset);
    *(Servo) |= (0<< Servo_EnableSelect);
    *(Servo) |= (0x00<<Servo_Data);
    *(Servo) |= (1<<Servo_Reset);
    usleep(200000);
    *(Servo) |= (0<<Servo_Reset);
}

void span() {
    while(1) {
        allLeft();
        allRight();
        allDown();
        allUp();
    }
}

```

Page 1

```

ServoApi.c

}

}
void movLeft() {
}
void movRight() {
}

```

Page 2

hello_world.c

```

* "Hello World" example.

#include <stdio.h>
#include "system.h"
#include "ServoApi.h"
#include "JoyStickAPI.h"
#include "AccelAPI.h"
#include <altera_up_avalon_accelerometer_spi.h>

#include <alt_types.h>

int main(){
    printf("Hello from Nios II!\n");

    alt_up_accelerometer_spi_dev * acc_dev;
    acc_dev = alt_up_accelerometer_spi_open_dev ("/dev/accelerometer_spi");
    alt_u32 xAccel=0;
    alt_u32 yAccel=0;
    alt_u32 zAccel=0;

    while(1){
        alt_up_accelerometer_spi_read_x_axis(acc_dev,&xAccel);
        alt_up_accelerometer_spi_read_y_axis(acc_dev,&yAccel);
        alt_up_accelerometer_spi_read_z_axis(acc_dev,&zAccel);
        servo_accel_y(yAccel);
        servo_accel_x(xAccel);

        printf("xaxis:%li yaxis:%li zaxis:%li\n",xAccel,yAccel,zAccel);
        usleep(10000);
    }
}

```


AccelAPI.c

```
#include "AccelAPI.h"
#include <altera_up_avalon_accelerometer_spi.h>
#include <system.h>
#include <unistd.h>
#include <alt_types.h>

long Accel(alt_u32 xAccel, alt_u32 yAccel, alt_u32 zAccel, int select){
    alt_up_accelerometer_spi_dev * acc_dev;
    acc_dev = alt_up_accelerometer_spi_open_dev ("/dev/accelerometer_spi");
    xAccel=0;
    yAccel=0;
    zAccel=0;
    if(select == 0){
        return alt_up_accelerometer_spi_read_x_axis(acc_dev, &xAccel);
    }
    else if(select == 1){
        return alt_up_accelerometer_spi_read_y_axis(acc_dev, &yAccel);
    }
    else{
        return alt_up_accelerometer_spi_read_z_axis(acc_dev, &zAccel);
    }

    usleep(1000);
}
}
```


AccelAPI.h

```
#include <system.h>
#include <unistd.h>
#include <alt_types.h>

//#include "alt_types.h"

#ifndef AccelAPI_H_
#define AccelAPI_H_

//addresses

//Initializes both servo position to the initial position.
long Accel(alt_u32 xAccel, alt_u32 yAccel, alt_u32 zAccel, int select);

#endif
```

APPENDIX II

```

                                hello_world.c

    * "Hello World" example.

#include <stdio.h>
//include have the hardware
#include "7seg.h"
#include "JoyStickAPI.h"
#include "ServoAPI.h"
#include <altera_avalon_pio_regs.h>
#include <altera_avalon_i2c.h>
#include <altera_up_avalon_accelerometer_spi.h>

int buttonStatus();

int main() {
    //reset camera and leave rst_n at 1
    IOWR_ALTERA_AVALON_PIO_DATA(ARDUINO_RESET_N_BASE, 0x00);
    usleep(100000);
    IOWR_ALTERA_AVALON_PIO_DATA(ARDUINO_RESET_N_BASE, 0x01);
    usleep(100000);

    //intializes the hardware for use
    Servo_init();
    JoyStick_init();
    i2c_init();
    alt_up_accelerometer_spi_dev * acc_dev;
    acc_dev = alt_up_accelerometer_spi_open_dev("/dev/accelerometer_spi");
    alt_u32 xAccel = 0;
    alt_u32 yAccel = 0;
    alt_u32 zAccel = 0;
    //variables used to select which component controls axis
    char xAxis;
    char yAxis;
    //interface for user to input their desicions
    printf("Hello\nChoose what will control the x-axis of the Servo:");
    printf("\n1:Switches"
           "\n2:JoyStick"
           "\n3:Accelerometer");
    printf("\nInput: ");
    scanf("%c", &xAxis);

    printf("\nChoose what will control the y-axis");
    printf("\n1:Switches"
           "\n2:JoyStick"
           "\n3:Accelerometer");
    scanf("%c", &yAxis);
    //logic will set the controls
    while (1) {

        printf("status for write=%d\n", write_addr(0xFF, 0x53)); // Will test
the functionality
        // Return 1 for Nack and 0 for ACK
        printf("status for read=%d\n", read_addr(0xFF));    // print statement

        if (xAxis == '1') {

```

```

                                hello_world.c

                                movX(buttonStatus());
    } else if (xAxis == '2') {
                                movX(getX());
    } else {

axis      alt_up_accelerometer_spi_read_x_axis(acc_dev, &xAccel); // reads x
                                servo_accel_x(xAccel); // sends to servo
    }

    if (yAxis == '1') {
                                movY(buttonStatus());
    } else if (yAxis == '2') {
                                movY(getY());
    } else {
                                alt_up_accelerometer_spi_read_y_axis(acc_dev, &yAccel);
                                servo_accel_y(yAccel);
    }
}

    return 0;
}

int buttonStatus() {
    return *(button_ptr) & 0xF;
}

```

REFERENCES:

DE-10 LITE MANUAL

GOOGLE IMAGES

ONE NOTE

SPARK FUN DOCUMENT ON I2C