Mid Sweden University
The Department of Information Technology and Media (ITM)
Author: Vivek Alaparthi, Alireza Davoudian
E-mail address: vial1001@student.miun.se , alda1300@student.miun.se
Study programme: MSc in Electronic Design, 120 ECTS
Examiner: Najeem Lawal, najeem.lawal@miun.se
Tutors: Amir Yousaf, amir.yousaf@miun.se
Date:2013-05-27

VHDL Project

Digital System Design With VHDL-6 Credits

Group 01 Subgroup 04

Student1- Alireza Davoudian

Student2- Vivek Alaparthi

Mittuniversitetet
MID SWEDEN UNIVERSITY

Table of Contents

Table of Figures

1 Introduction

This project uses two range sensors which are attached to Nexys2 FPGA board. The sensors used to find the distance of the object in this project are SRF05. The two sensors separately measures the distance of the object and send the values to a PC running Matlab. Each sensor measurement value is of 2 bytes.

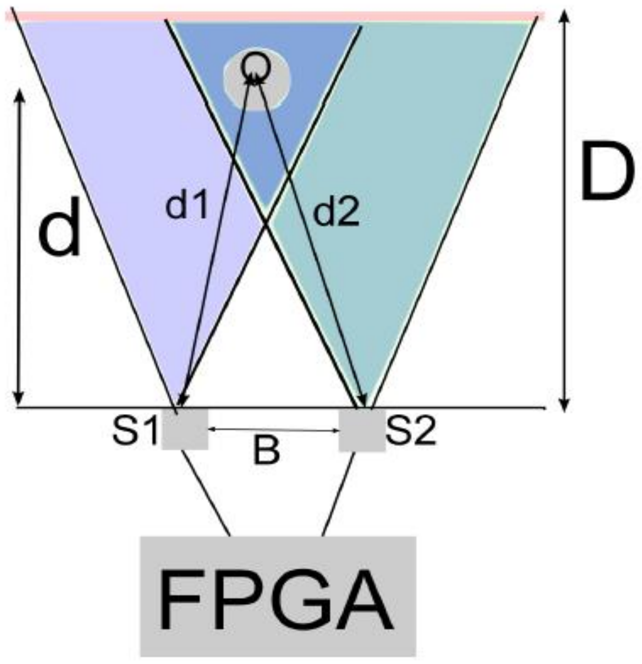


Figure 1: Project setup[1]

The above figure represents the project setup using two sensors which are connected to FPGA board. D is the maximum distance that can be read by the sensors which is equal to 400 cm according to datasheet. "d" is the distance of the object from the midpoint of the sensors and B is the baseline separation between sensors.

1.1 Overall aim

The aim of the project is to design a setup which runs with FPGA Nexys2 board that can measure the distance of the object using two range sensors. The values measured by the sensors should be converted to cm using some calculations available on SRF05 datasheet. Each range sensor value is of two bytes (16 bits) which is sent to PC running Matlab. The communication between the FPGA board and PC is carried out through RS232 cable (UART communication). PC again sends 4 bytes of data back to FPGA board. First two bytes represent the distance from the midpoint between two sensors and the last two bytes represent the angle of the object from the midpoint between two sensors. Both the range sensor readings and the data received from the PC should be displayed on 7-segment. A character should be displayed which represents the property followed by three digit value of the property.

1.2 Contributions

Student1 contribution of the project is range sensor reading. Student2 contribution of the project is seven segment display and UART transmission.

2 Theory

2.1 Nexys2 board[2]:

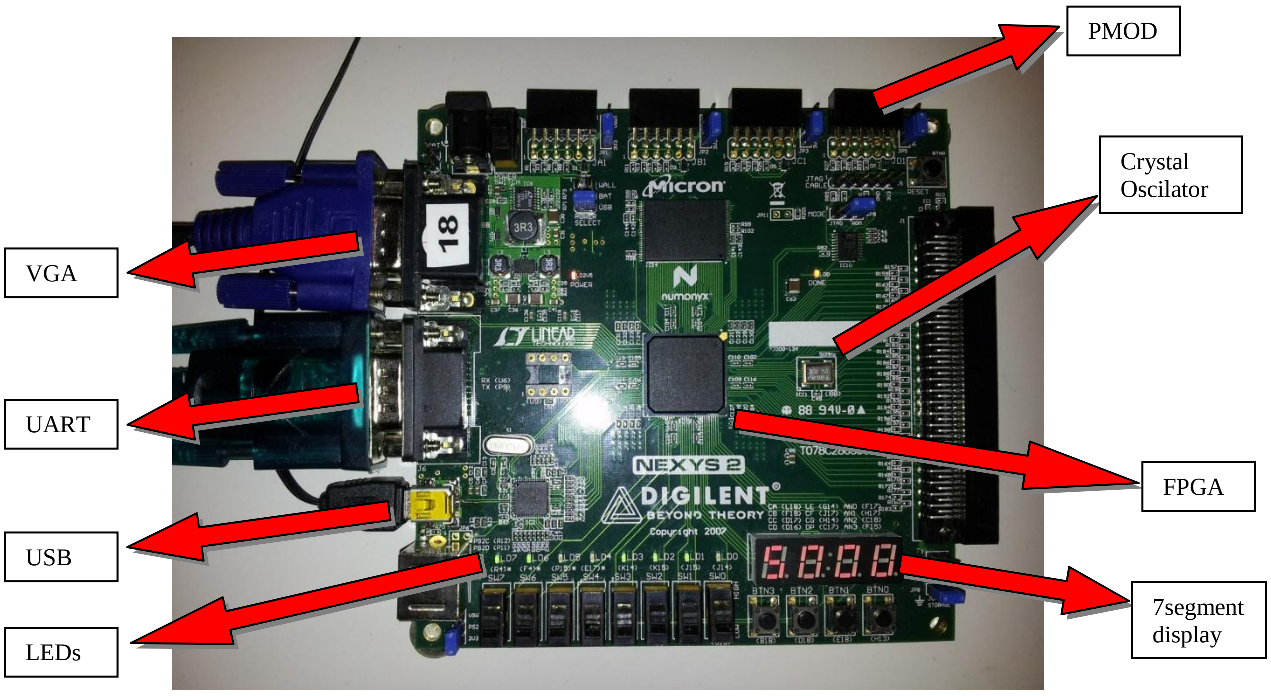


Figure 2: Image explains different peripherals of nexys2 board

The Nexys2 is a Xilinx Spartan 3E FPGA circuit development platform. Its onboard high-speed USB2 port, 16Mbytes of RAM and ROM, and several I/O devices and ports make it an ideal platform for digital systems of all kinds, including embedded processor systems based on Xilinx's MicroBlaze. The USB2 port provides board power and a programming interface, so the Nexys2 board can be used with a notebook computer to create a truly portable design station.

The Nexys2 brings leading technologies to a platform that anyone can use to gain digital design experience. It can host countless FPGA-based digital systems, and designs can easily grow beyond the board using any or all of the five expansion connectors. Four 12-pin Peripheral Module (Pmod) connectors can accommodate up to eight low-cost Pmods to add features like motor control, A/D and D/A conversion, audio circuits, and a host of sensor and actuator interfaces. All user accessible signals on the Nexys2 board are ESD and short-circuit protected, ensuring a long operating life in any environment.

The Nexys2 board is fully compatible with all versions of the Xilinx ISE tools, including the free WebPack. Now anyone can build real digital systems for less than the price of a textbook.

2.1.1 User I/O

The Nexys2 board includes several input devices, output devices, and data ports, allowing many designs to be implemented without the need for any other components.

2.1.2 Inputs: Slide Switches and Pushbuttons

Four pushbuttons and eight slide switches are provided for circuit inputs. Pushbutton inputs are normally low, and they are driven high only when the pushbutton is pressed. Slide switches generate constant high or low inputs depending on their position. Pushbutton and slide switch inputs use a series resistor for protection against short circuits (a short circuit would occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output).

2.1.3 Outputs: LEDs

Eight LEDs are provided for circuit outputs. LED anodes are driven from the FPGA via 390-ohm resistors, so a logic '1' output will illuminate them with 3-4ma of drive current. A ninth LED is provided as a power-on LED, and a tenth LED indicates FPGA programming status. Note that LEDs 4-7 have different pin assignments due to pinout differences between the -500 and the -1200 die.

2.1.4 Outputs: Seven-Segment Display

The Nexys2 board contains a four-digit common anode seven-segment LED display. Each of the four digits is composed of seven segments arranged in a "figure 8" pattern, with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark. Of these 128 possible patterns, the ten corresponding to the decimal digits are the most useful.
The anodes of the seven LEDs forming each digit are tied together into one "common anode" circuit node, but the LED cathodes remain separate. The common anode signals are available as four "digit enable" input signals to the 4-digit display. The cathodes of similar segments on all four displays are

connected into seven circuit nodes labeled CA through CG (so, for example, the four "D" cathodes from the four digits are grouped together into a single circuit node called "CD"). These seven cathode signals are available as inputs to the 4-digit display. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate The segments of the digit whose corresponding anode signal is asserted.

A scanning display controller circuit can be used to show a four-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession, at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-qUARTer of the time, but because the eye cannot perceive the darkening of a digit
Before it is illuminated again, the digit appears continuously illuminated. If the update or "refresh" rate is slowed to around 45 hertz, most people will begin to see the display flicker.

In order for each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms, for a refresh frequency of 1 KHz to 60Hz. For example, in a 60Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for ¼ of the refresh cycle, or 4ms. The controller must drive the cathodes with the correct pattern when the corresponding anode signal is driven. To illustrate the process, if AN0 is asserted while CB and CC are asserted, then a "1" will be displayed in digit position 1. Then, if AN1 is asserted while CA, CB And CC is asserted, and then a "7" will be displayed in digit position 2. If AN0 and CB, CC are driven for 4ms, and then A1 and CA, CB, CC are driven for 4ms in an endless succession, the display will show "17" in the first two digits. An example timing diagram for a four-digit controller is provided.

2.1.5 USB Port

The Nexys2 includes a high-speed USB2 port based on a Cypress CY7C68013A USB controller. The USB port can be used to program the on-board Xilinx devices, to perform user-data transfers at up to 38Mbytes/sec, and to provide power to the board. Programming is accomplished with Digilent's free Adept Suite Software. User data transfers can also be accomplished using the Adept software, or custom user software can be written using Digilent's public API's to access the Nexys2 USB connection. Information on using Adept and/or the public API's to transfer data can be found on the Digilent website.

The USB port can also provide power to the Nexys2 board if the power select jumper is set to "USB". The USB specification requires that attached devices draw no more than 100mA until they have requested more current, after which up to 500mA may be drawn. When first attached to a USB host, the Nexys2 board requests 500mA, and then activates a transistor switch to connect the USB cable voltage to the main input power bus. The Nexys2 board typically draws around 300mA from the USB cable, and care should be taken (especially when using peripheral boards) to ensure that no more than 500mA is drawn.

2.1.6 Serial Port

The Nexys2 contains a two-wire serial port based on an ST Microelectronics ST3232 voltage converter. The ST3232 converts the signal levels used by RS-232 communications (-12 to -3 for a logic ‘1’ and 12V to 3V for a logic ‘0’) to the 3.3V signals used by the FPGA. Since only two signals are connected (RXD and TXD), an FPGA-based serial port controller can only use software handshaking protocols (XON/XOFF).
The Nexys2 serial port is useful for many applications, and in particular for debugging and working with Xilinx’s MicroBlaze embedded processor.

The two devices connected to either end of a serial cable are known as the Data Terminal Equipment (DTE) and the Data Communications Equipment (DCE). The DCE was originally conceived to be a modem, but now many devices connect to a computer as a DCE. A DTE “source” device uses a male DB-9 connector, and a DCE “peripheral” device uses a female DB-9 connector. Two DTE devices can be connected via a serial cable only if lines two and three (RXD and TXD) are crossed; producing what is known as a null modem cable. A DTE and DCE device can be connected with a straight-through cable. The Nexys2 is configured as a DCE device, with the assumption it will most typically be connected to a DTE device like a computer.

3 Methodology

The implementation of the project is divided into separate modules Sensor reader, UART Transmitter controller, UART Receiver controller, seven segments display driver.

3.1 Seven segment display driver:

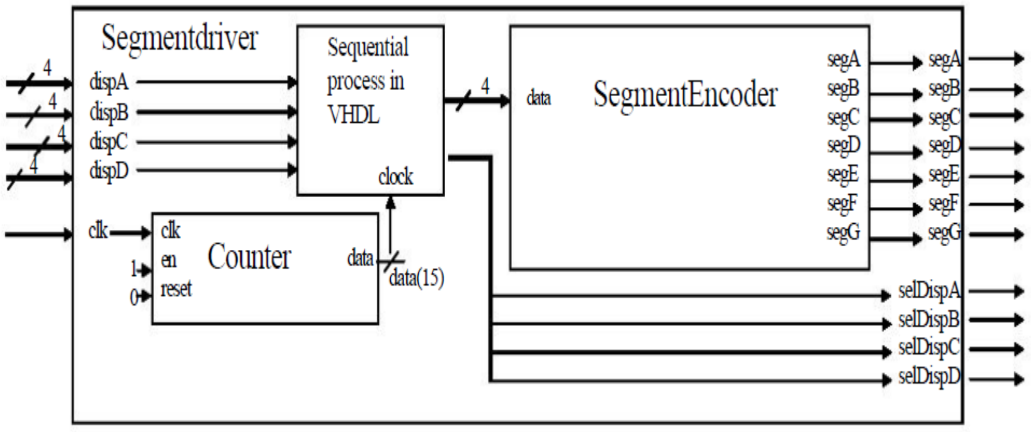


Figure 3: Block represents the seven segment display driver

In the above figure dispA to dispD represents input data to each one of the digits on the 4 digit display assembled on the nexys2 board. There is one clock input from the 50MHz crystal oscillator. The outputs segA to segF are driving the cathodes of each segment of all digits. The 4 digit display selDispA to selDispD are anode driven to a high voltage. The above block contains 3 different blocks Segment driver, Counter and Segment Encoder.

3.1.1 Segment driver:

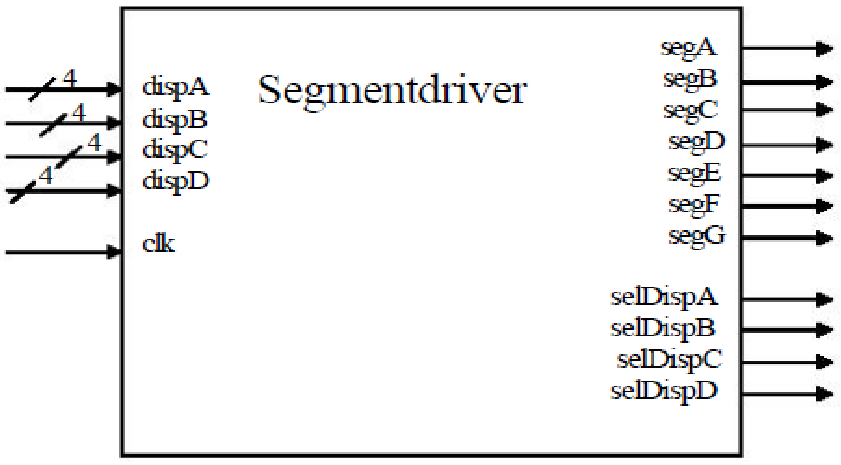


Figure 4: Block represents the segment driver

This block contains 4 inputs each contain 4 bits and a clock signal as input. The outputs are 4 digit display and seven segment of each digit display.

3.1.2 Counter

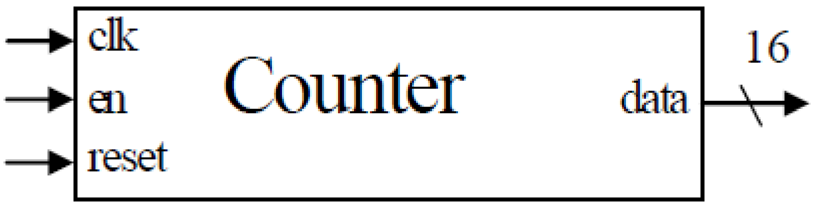


Figure 5: Block represents the counter

The counter block contains the input as clock, enable and reset signals. The clock signal receives the input from 50MHz crystal oscillator. The output is 16 bit data.

3.2 Range finder

3.2.1 Introduction

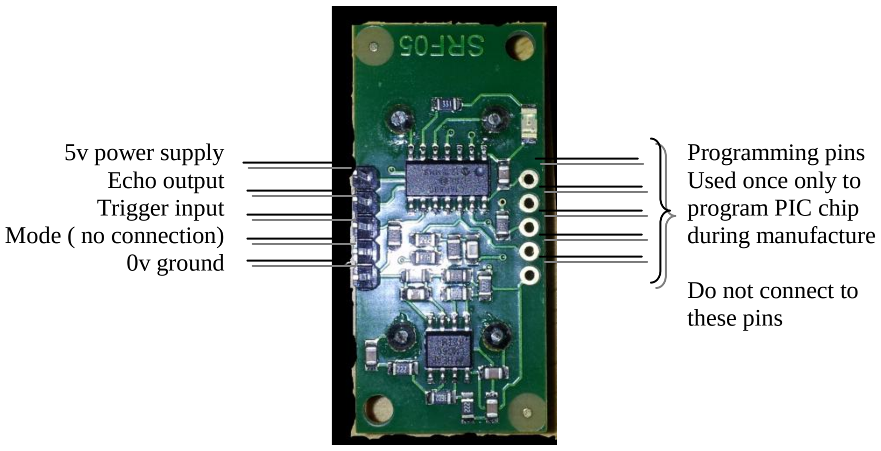


Figure 6: Image explains the different pins of SRF05 range sensor

The range sensor that was used in this project is SRF05 which is an ultrasonic distance meter. It can measure distances up to 4 meters. SRF05 works in two modes. One operating mod is to use a single pin for both trigger and echo, thereby saving valuable pins on your controller. When the mode pin is left unconnected, the SRF05 operates with separate trigger and echo pins.

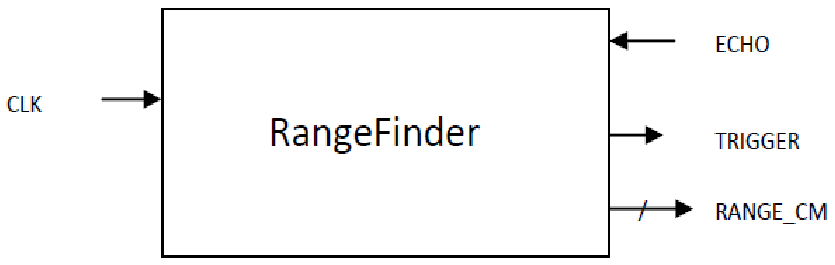


Figure 7: Image showing the block diagram of range finder

The range finder block contains CLK, ECHO as inputs and TRIGGER, RANGE_CM as outputs. This block used to find the distance of the object from sensors. The sensors are triggered by nexys2 board through trigger signal. According to the datasheet of SRF05 sensor, the sensor should be triggered minimum of 10usec. The sensor produces the echo automatically when it is triggered by the nexys board.

There are two modes for using the SRF05 range sensor.

Mode1: This mode uses separate echo and trigger pins. This is the simplest mode to use. To use this mode, the mode pin should leave as unconnected.

Mode2: This mode uses single pin for both echo and trigger signals. This mode is designed to save valuable pins on embedded controllers. To use this mode, the mode pin should be connected to 0V ground pin. The echo and trigger signals uses the same pin in this mode. The SRF05 will not raise the echo until immediately after the end of trigger signal. Ultrasonic burst is produced by the SRF05 after the end of trigger signal. So, the SRF05 takes 700usec to raise echo signal after the end of trigger signal.

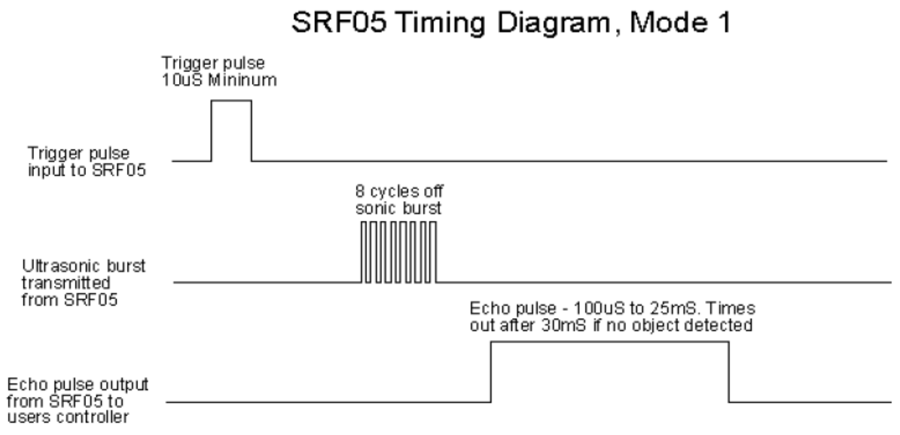


Figure 8: Image showing timing diagram SRF05 sensor[3]

3.2.2 Electrical Specifications (Typical)

The electrical specifications for SRF05 was described in [4] and can be found in the following table:

Power	5V DC (Variations from this may affect oscillator frequency and accuracy)
I (Supply)	1mA average (IC only. Each SRF05 takes about 4mA on average) Beam Spread @ 40kHz (-6dB)
Boot Up Time	About 500mS after power-on / reset.
Sample Delay	Depends on range. You must include a 50mS Pause between trigger requests.
Osc. Accuracy	1% Factory Calibrated (Microchip spec)
Range Accuracy	+ / - 1 inch or +/- 10 microseconds.
Serial Polarity IN	PICAXE T2400 . All others 2400 baud Inverted.
Serial Polarity OUT	As above.
Serial Line load	Both serial connections must use a 2K2 resistor in-line to limit current.

Table 1: Electrical specification for SRF05

3.3 UART Components[5]:

The UART contains two separate circuits packaged in one component. One circuit is for receiving serial information and another circuit is for transmitting serial information. The receiver takes one byte of serial data from RxD port and converts it into one byte of parallel data. The transmitter takes a byte of parallel information from DBIN port and it into a byte of serial information. This data is then transmitted to TxD port.

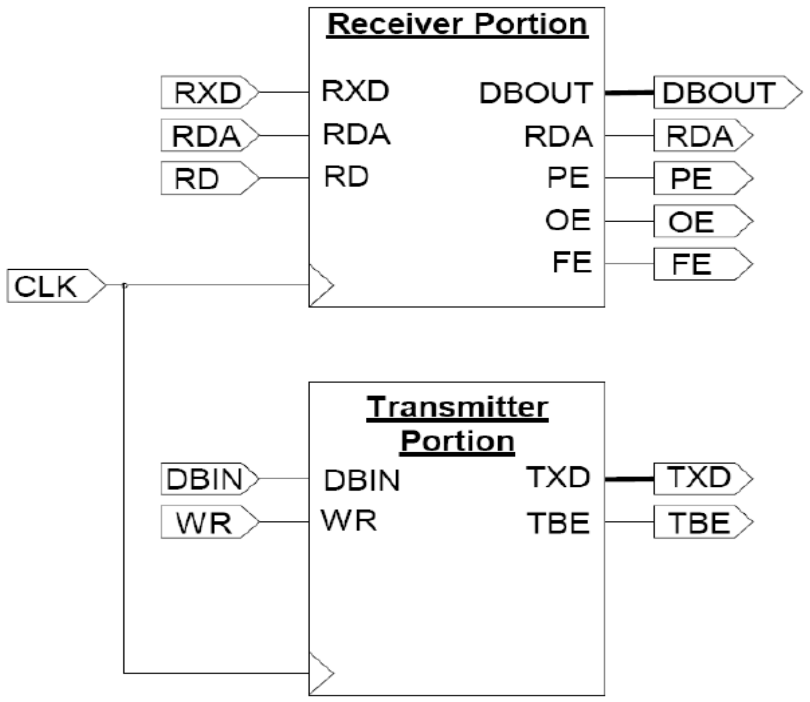


Figure 9: Image showing both receiver and transmitter circuits

3.3.1 Receiver circuit operation:

The receiver circuit of the UART accepts serial data and converts it into parallel data. The receiver portion contains serial data controller, two counters for synchronization, a shift register and error bit controller. The shift register is used to store incoming data from RxD port. The data transferred to RxD port with a specific baud rate. Serial data controller is used to synchronize data acquisition phase. Serial data controller uses a state machine and has two synchronization counters used in synchronization. The state machine is capable of reading the RxD port after each bit transmission.

In idle state, the serial data pin (RxD port) is held high. The idle state remains unchanged until the RxD port goes low. When the RxD goes low, the Eight Delay state is entered. This state is used to make sure that the receiver port read in between each bit transmission. The counter increments at the rate of 16 times faster than baud rate. In Eight Delay state, the counter is allowed to count up to 8.

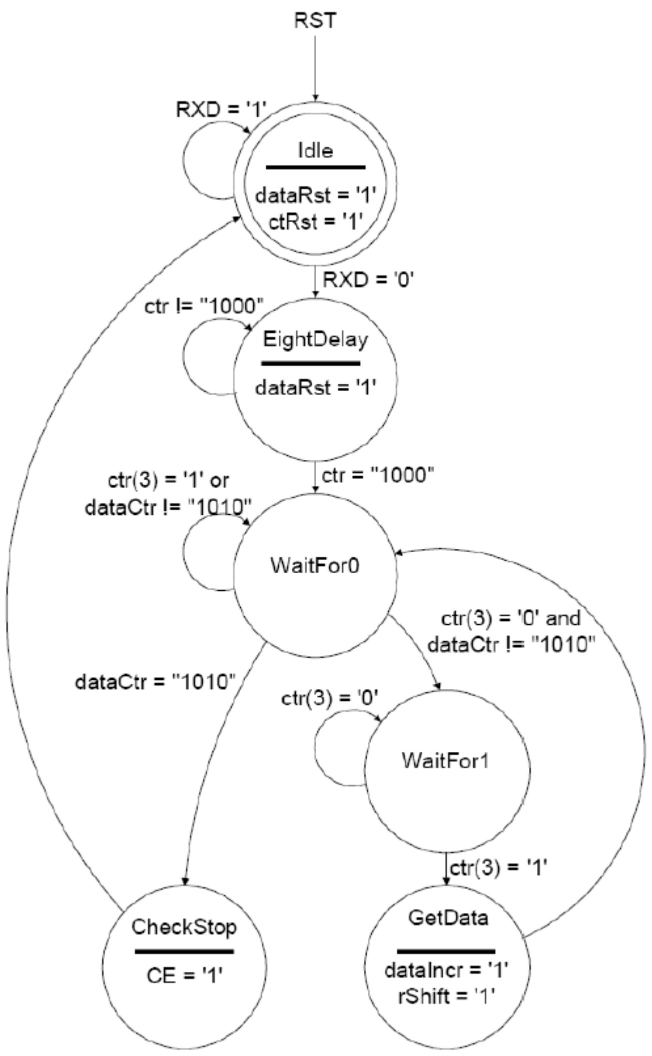


Figure 10: Image showing receiving state machine[5]

These two states depend on the most significant bit of the "ctr" counter for their next state. These states precede the GetData state, which shifts the value of RxD at the time that the state is entered. The GetData state increments the datactr counter, which is used to keep track of number of bits have been transferred into shift register. Once the counter reaches low check stop state is entered. This state also enables the error bit controller. The idle state is loaded immediately after the check stop stage. The timing diagram of the shifting procedure can be seen in Figure11.

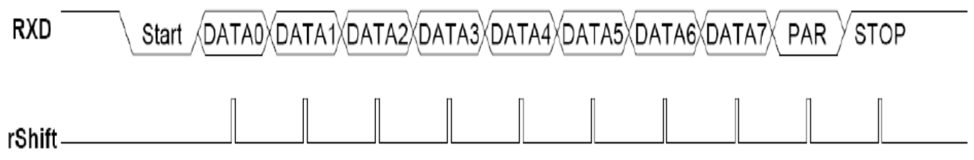


Figure 11: Image showing receiver timing diagram

The error bit controller analyzes the data received for three types of errors. They are parity error, frame error and over write error. The parity error is because of the disagreement of parity bit with the number of one’s in the data portion of the shift register. The UART component is set to odd parity by default.

A frame error is because that the UART is not reading the incoming data at right time. This error occurs if the stop bit is not set to one.

The over write error indicates that the data byte just received need to over write a previously received and never read data byte.

3.3.2 Transmitter Circuit Operation:

The transmitter portion of the UART accepts a byte of data from DBIN port and transmits it as serial data on TxD port. The baud rate for transmission and receiving portion of the UART are the same.

The transfer portion of the UART must contain a transfer controller; two counters for synchronization and a transfer shift register to transmit the byte stored in DBIN port. The transfer controller uses two counters for synchronization and a transfer shift register to transmit the byte stored in DBIN port. The transfer controller uses two counters, one for controlling the rate at which the data byte must be transmitted and the other to count the number of bits transmitted. The TxD port is set equal to the least significant bit of transfer shift register, allowing data to be transmitted by simply right shifting the register for each transmission. The state diagram for transfer controller is shown below:

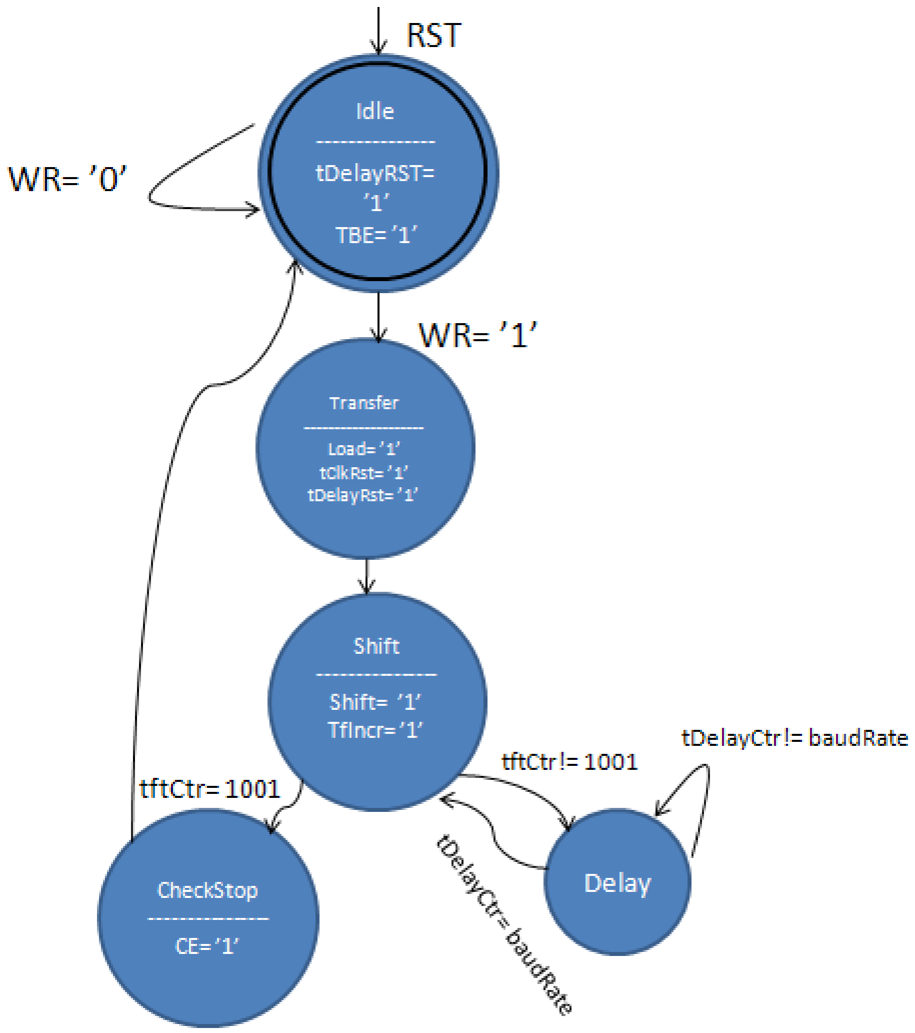


Figure 12: Image showing the transferring state machine

The transfer portion of the UART stays idle until the WR port goes high. When the WR port goes high, the UART write the data that is available in DBIN port.

The next state is transfer state which prepares the transfer shift register to transmit data. By setting load='1', the shift register loaded with start bit, DBIN , parity bit and stop bit. The two synchronization counters can be reset by making reset signals tClkRst and tDelayRst to '1'.

In shift state the shift signal is set to '1' in order to shift the shift register. The tflncr signal is set to '1' to increment the counter. If the data counter is not equal to nine, the data in the transfer shift register has not yet been shifted. If all the data from the transfer shift register has not yet been shifted, the Wait Write state is loaded.

The Delay state is to check the transmission process sends out the data at appropriate baud rate.

Transmission is complete when the WaitWrite state has been entered. This state is needed to assure that the WR port is held high.

4 Design / Implementation

4.1 Range finder:

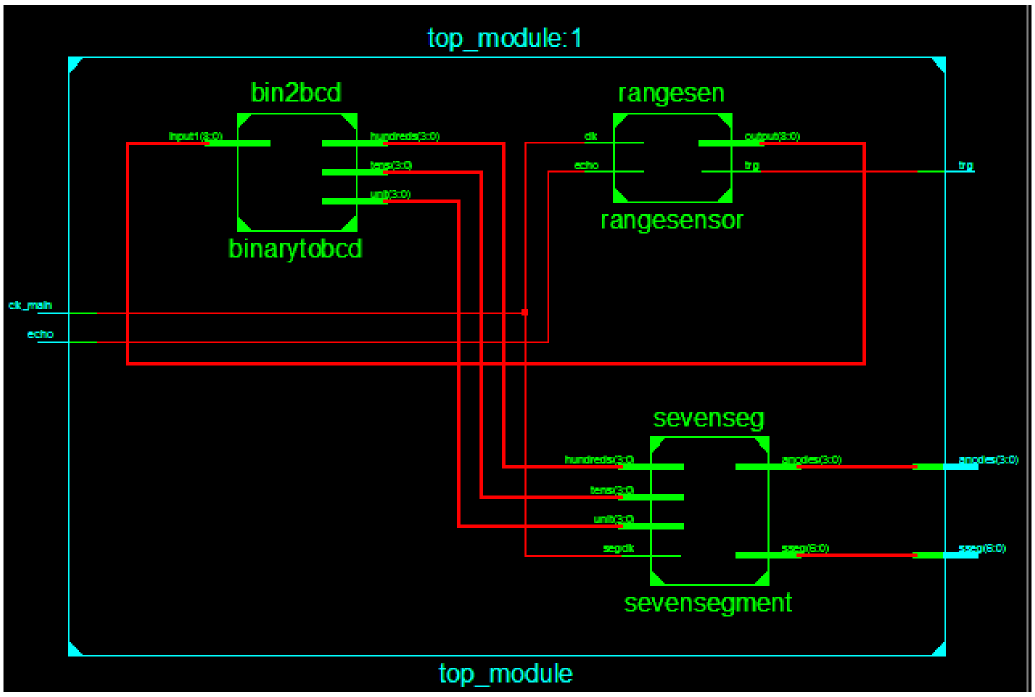


Figure 13: Image showing range sensor module connected to BCD converter and seven segment

It should be noted that the simulations are generated by Xilinx Isim [2].

In the above figure it is shown that the range sensor module is connected to binary to BCD converter and the BCD module is connected to seven segment display module. Each module is clearly explained below.

4.1.1 Range sensor module:

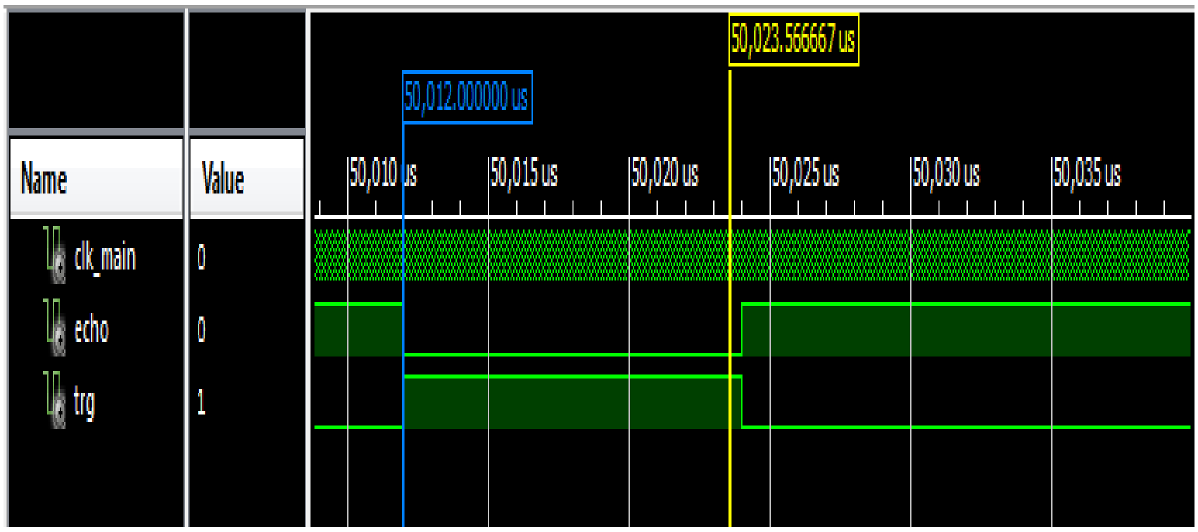


Figure 14: Image showing trigger signal generated in range sensor module

Range sensor module has two inputs clock and echo signal. The clock is generated by the crystal oscillator of 50 MHz from nexys2 board. The echo signal is generated by the range sensor SRF05. The range sensor module is having one output that is trigger signal. In the above figure it is clearly shown that the trigger signal is high for 12 microseconds. The trigger signal should be generated after 50 mili seconds, i.e. 20 times a second according to the datasheet of SRF05. This is to make sure that the ultra-sonic beep has faded away and should not cause false echo on next triggering. This waveform is generated by forcing the values of echo and trigger signals in the test bench of the Xilinx software.

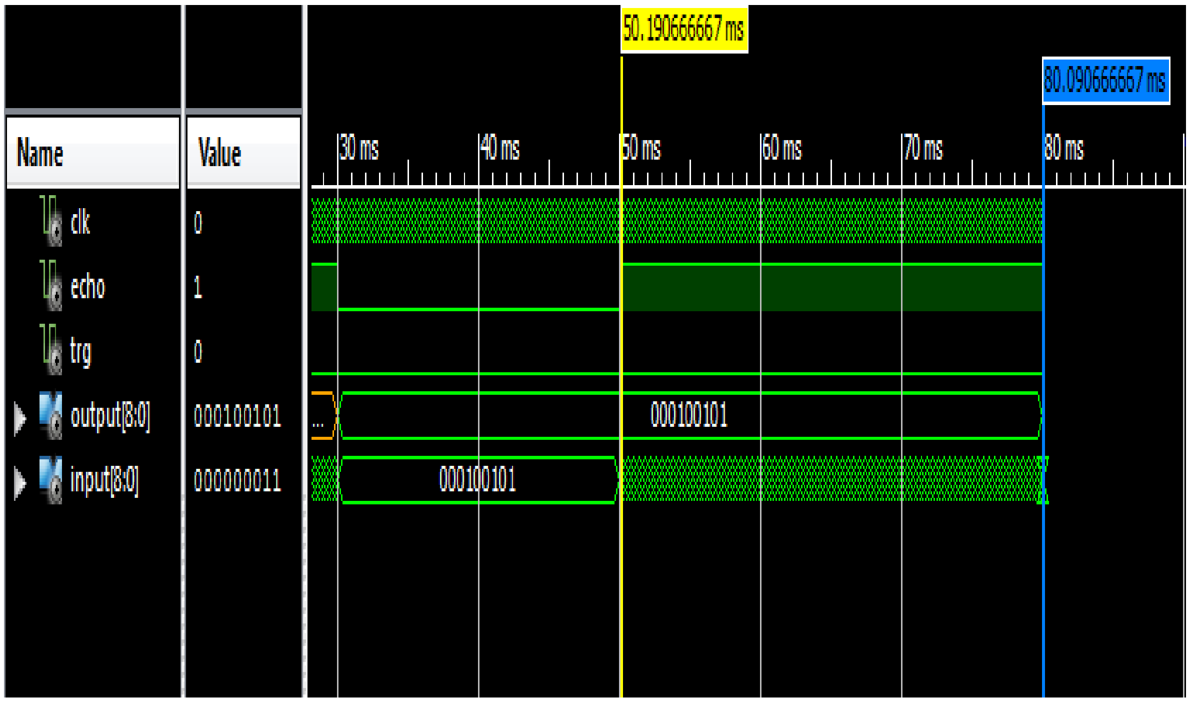


Figure 15: Image showing echo signal generated in range sensor module

The echo signal is high for 30 mili seconds. After that the echo signal IS low for 20 mili seconds. This is supposing that the object is detected after 30milli seconds after the echo is generated. Whenever the echo signal is high the counter starts incrementing along with the clock signal. If an object is detected automatically the echo signal goes low and the counter stops incrementing. The counter value is converted to centimeters and then into 9 bit binary value. The binary value is given to a signal output as shown in the image above. The calculations of converting the counter value into centimeters are clearly given in section 4.1.2.

4.1.2 Distance measurement:

In section 4.1.1 it is explained that the counter is responsible for measuring the width of the echo. According to the data sheet of SRF05, if the width of the echo is measured in microseconds then it must be divided by 58 to convert it to centimeters. In Xilinx a function should be written for division operation. Each clock cycle of FPGA clock represents 20 nano seconds. It can be converted into centimeters using the below function

$$(X * 20) / (1000*58) = X/2900 = X \text{ (in cm)}$$

The number of clock cycles is multiplied by 20 nano seconds, since each FPGA clock cycle represents 20 nano seconds. This result is divided by 1000 to convert it into microseconds. Now the width of the echo in micro seconds can be divided by 58 to convert it into centimeters.

Dividing a number X with a big number like 2900 uses lot of hardware resources. Instead of using division this number is multiplied by 3 and the result is divided by 8192 which is not perfect, because 2900 multiplied by 3 is equal to 8700.

$$(3 * X) / (3 * 2900) = (3 * X) / (8700) \approx (3 * X) / (8192)$$

To implement this I used a counter (count1) to calculate the width of the echo signal as explained above. The counter values are converted to integer values and then multiplied by 3 and the result is divided by 8192. This result is stored in another signal int2 which is converted to 9 bit binary value. This binary value is given to the signal input. When the object is detected the echo signal becomes ‘0’ and the final 9 bit binary value of the input signal is given to the output signal. The final output of range sensor module is as shown in the figure below.

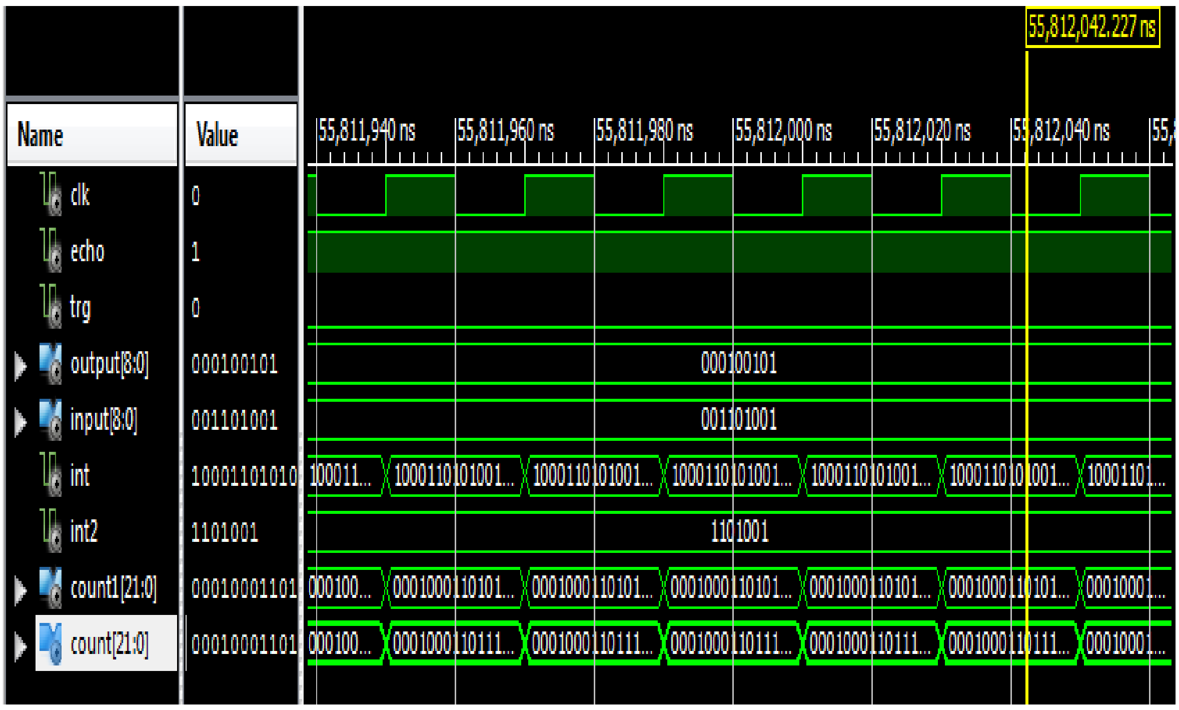


Figure 16: Image showing the width of the echo signal converted to centimeters

In the above image the 9 bit binary value of input signal is not same as output signal. This is because the output signal contains the previous binary final value of the input signal when the echo signal becomes ‘0’. This is clearly shown in the image below.

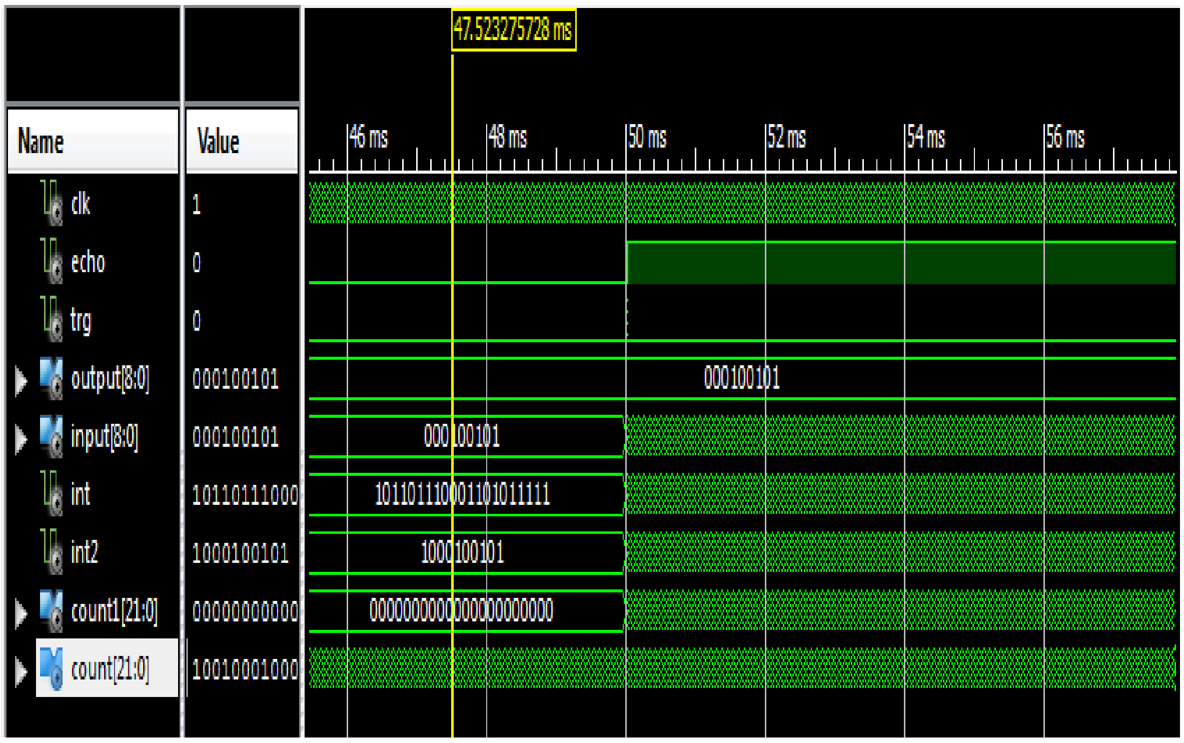


Figure 17: Image showing the output signal contains the previous final value of input signal

The width of the echo signal is converted to centimeters and the 9 bit value is given to UART module. In UART module depending on the dbOutSig signal range sensor 1 or range sensor 2 data is given to binary to BCD converter to convert the 9 bit binary value into decimal number.

4.2 Binary to BCD converter:

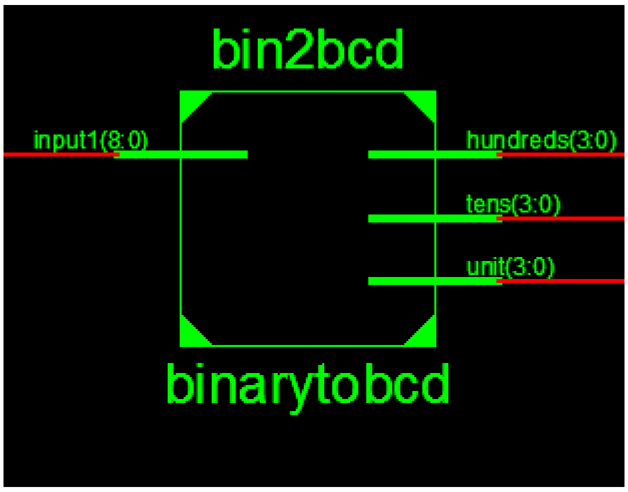
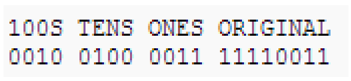


Figure 18: Binary to BCD converter module

This module converts the binary output of the range sensor output into decimal number. For converting a binary value to BCD double dabble method is used.

4.2.1 Double dabble method:

Double dabble algorithm[6] is used to convert a binary number to BCD. Space must be reserved to store both original number and BCD representation.



```
100S TENS ONES ORIGINAL
0010 0100 0011 11110011
```

Figure 19: BCD presentation of binary numbers

The algorithm uses the following steps to convert binary number to BCD:

Step1: The algorithm runs for n times (number of bit in binary number). Each time the number is left shifted one time.

Step2: Before the left shift operation, each 4 bit number greater than 4 should be incremented by 3.

Step3: Go to Step1

```
0000 0000 0000 11110011      Initialization
0000 0000 0001 11100110      Shift
0000 0000 0011 11001100      Shift
0000 0000 0111 10011000      Shift
0000 0000 1010 10011000      Add 3 to ONES, since it was 7
0000 0001 0101 00110000      Shift
0000 0001 1000 00110000      Add 3 to ONES, since it was 5
0000 0011 0000 01100000      Shift
0000 0110 0000 11000000      Shift
0000 1001 0000 11000000      Add 3 to TENS, since it was 6
0001 0010 0001 10000000      Shift
0010 0100 0011 00000000      Shift
```

Figure 20: Example of double dabble algorithm

In the above example the number 243_{10} is converted to BCD number using double dabble algorithm

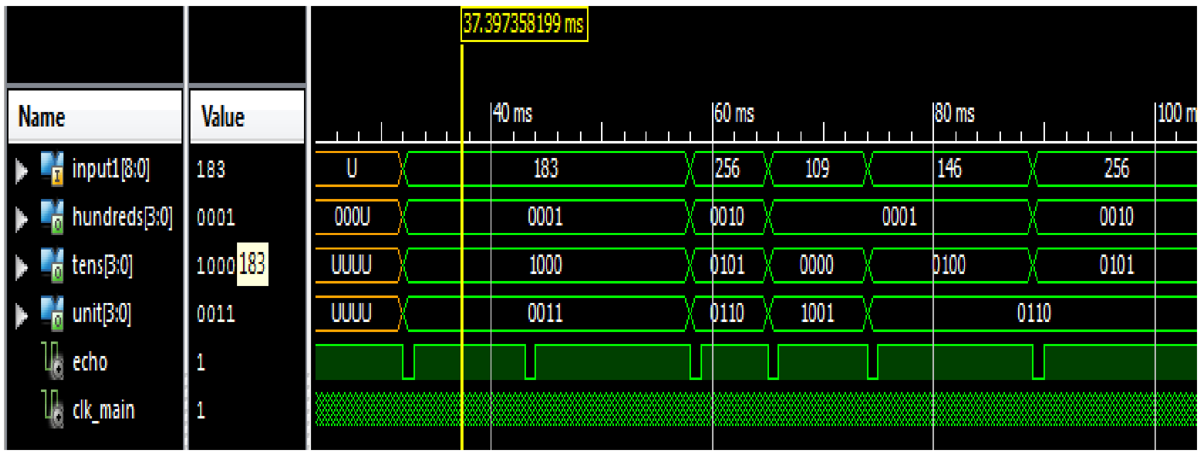


Figure 21: Image showing the example of converting binary number to BCD

The above simulation result clearly showing the conversion of width of echo signal to BCD number. The output of the range sensor is stored is given to the input signal (input1) of BCD converter module. In BCD converter module the number is converted to decimal number using double dabble algorithm and is stored in the variables like hundreds, tens and unit.

4.3 Seven segment display:

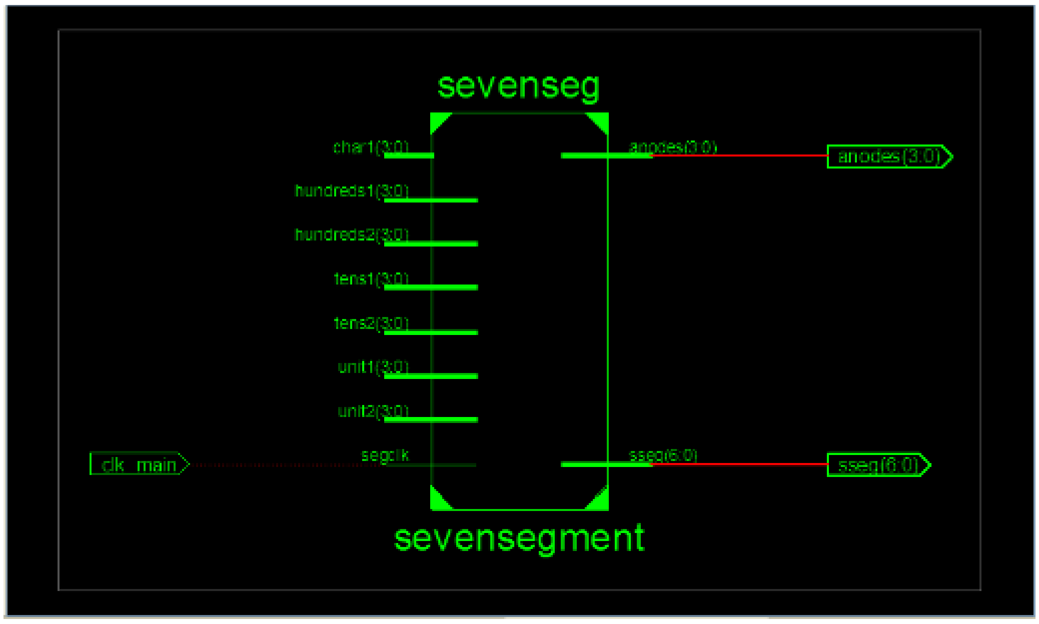


Figure 22: Seven segment module

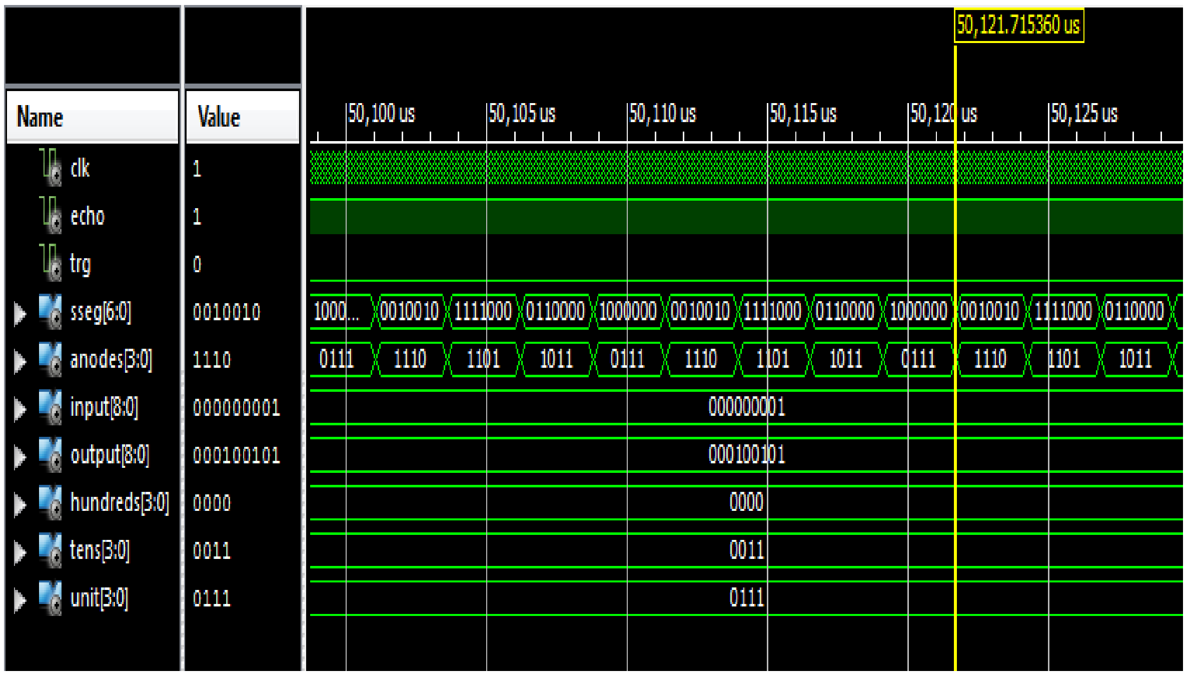


Figure 23: Image showing the output of seven segment display using simulation

The above image is taken when the range sensor output is given to BCD module and then to seven segment display. In actual project implementation output of range sensor is given to UART. In the above figure, it is shown an example of seven segment display output. In this example the range sensor output is “000100101”. This is given as input to BCD. The BCD output is “0000 0011 0111” which is equal to the decimal value of 37. This is taken as the first sensor data. So, this data is represented with character s.

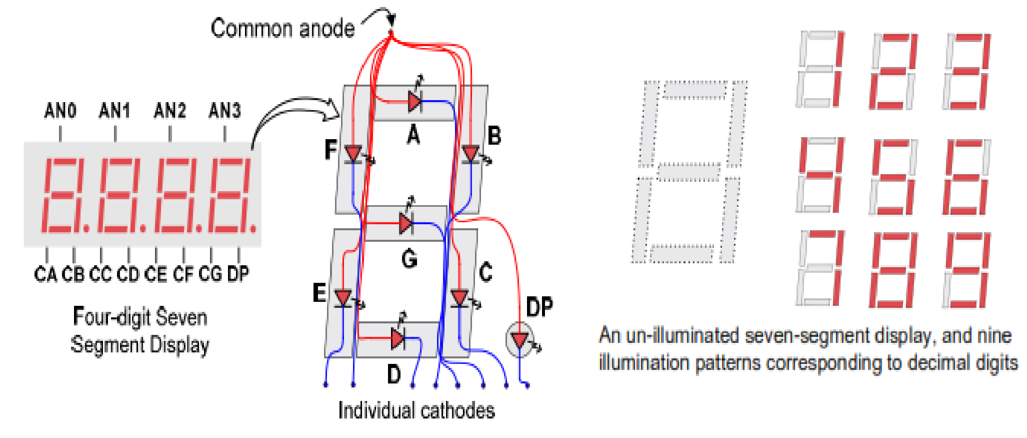

Common anode
AN0
AN1
AN2
AN3
CA CB CC CD CE CF CG DP
Four-digit Seven Segment Display
A
B
C
D
E
F
G
DP
Individual cathodes
An un-illuminated seven-segment display, and nine illumination patterns corresponding to decimal digits

Figure 24: Nexys2 seven segment display[7]

The LED’s of the seven segment are having common anodes and individual cathodes. The nexys 2 board contains 4 individual seven segment displays with common anodes ANO, AN1, AN2, AN3.

Code explanation: In the seven segment module all the anodes are enabling in circular sequence. The FPGA clock is having a very high frequency of 50MHz. According to the nexys2 reference manual all the four anodes should be driven once every 16ms. Because human eye cannot detect the numbers updating at a frequency of 50MHz. I am using a frequency of 200ms in my seven segment module. Various numbers on the seven segment display is shown by enabling the cathodes of the LEDS’s.

In the above example the numbers displaying by the seven segment display are:

“0110000” ----------- Represents the character‘s’

“1000000”------------ Represents the digit ’0’

]

“0010010”------------ Represents the digit ‘3’

“1111000”------------ Represent the digit ‘7’

4.4 UART Transmission[5]:

The transmitter portion of the UART accepts a byte of data from DBIN port and transmits it as serial data on TxD port. The baud rate for transmission and receiving portion of the UART are the same.

The transfer portion of the ART must contain a transfer controller; two counters for synchronization and a transfer shift register to transmit the byte stored in DBIN port. The transfer controller uses two counters for synchronization and a transfer shift register to transmit the byte stored in DBIN port. The transfer controller uses two counters, one for controlling the rate at which the data byte must be transmitted and the other to count the number of bits transmitted.

The transfer controller contains 3 states *Idle state, transfer state and shift state.*

Idle state: In idle state one byte of data is loaded into *dbInSig* signal which is mapped with *DBIN* (Data Bus in).

Transfer state: In transfer state load signal is made high which converts the data in *tfReg* (Transfer holding register) into a frame by adding start bit, parity bit and stop bit. In this state the data is moved from *tfReg* into *tfSReg* (Transfer shift register).

Shift state: In shift state the shift signal is made high and the date in the *tfSReg* is right shifted by adding ones on the left hand side. The data is transmitted bit by bit by right shifting the *tfS* register and then send to *TXD* (transmitter) port.

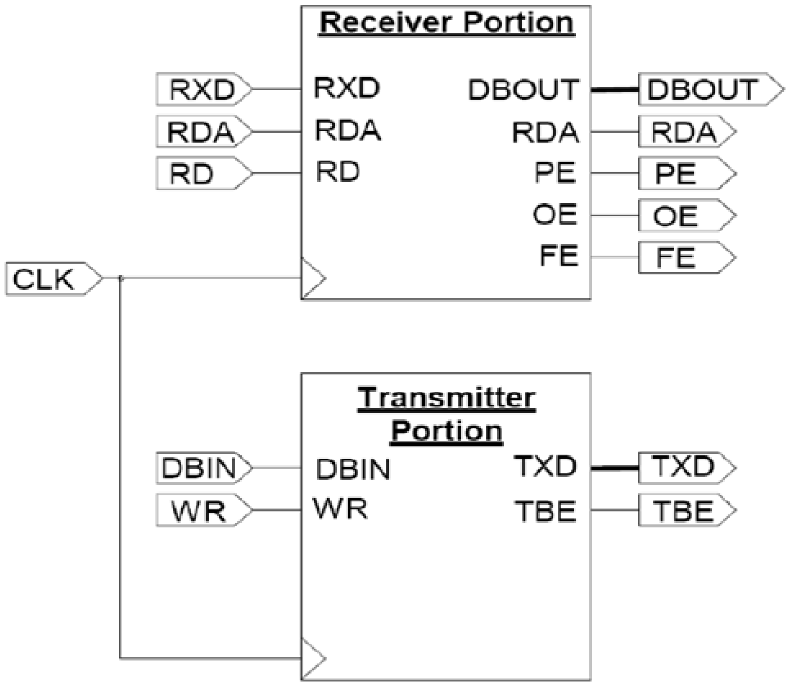
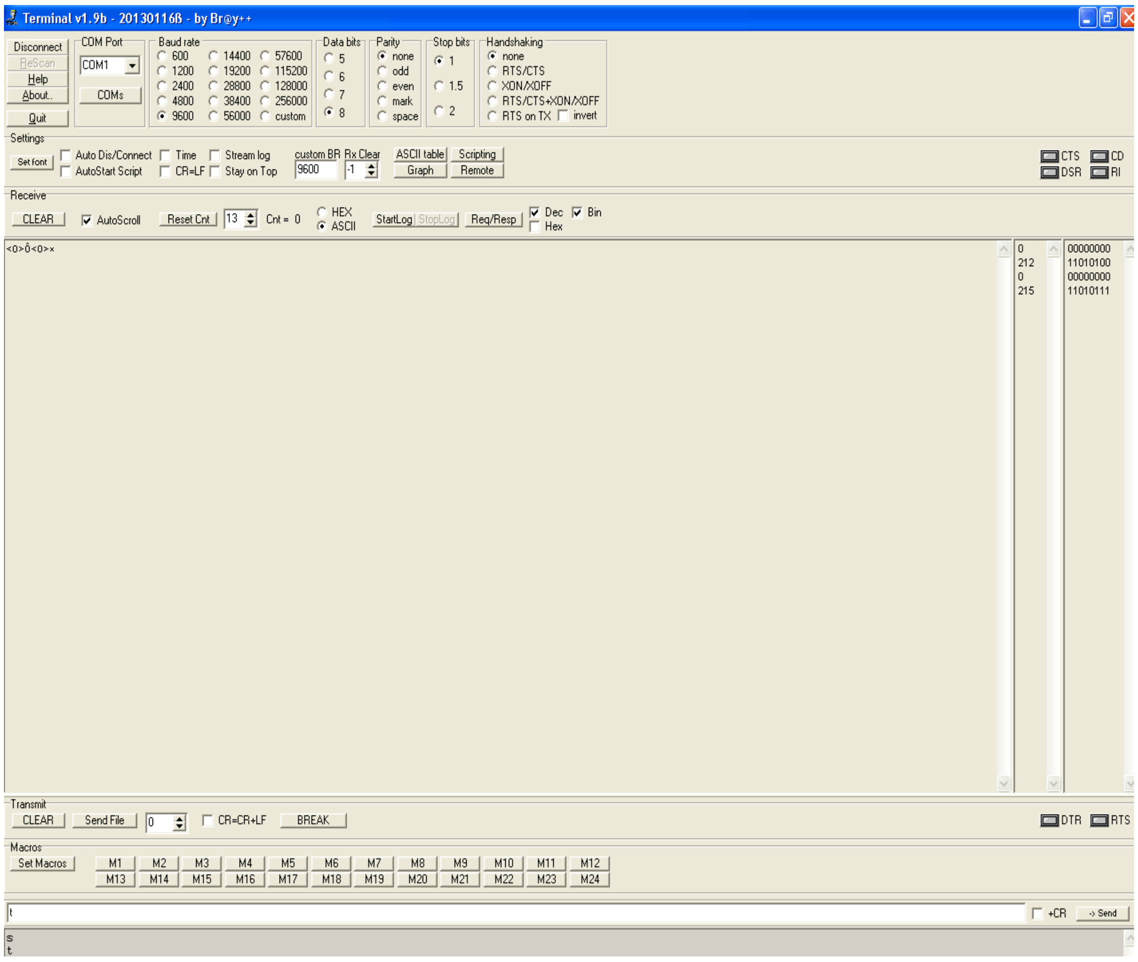


Figure 25: Image showing both transmitter and receiver circuits[5]



```
COM Port: COM1
Baud rate: 9600
Data bits: 8
Parity: none
Stop bits: 1
Handshaking: none
custom BR: 9600   Rx Clear: -1
Receive: AutoScroll [✓]   13   Cnt = 0   ASCII   Dec [✓]   Bin [✓]

<0>Ô<0>×

0      00000000
212    11010100
0      00000000
215    11010111

Transmit: 0
Input: t

s
t
```

Figure 26: Image showing the output of transmission on terminal

5 Results

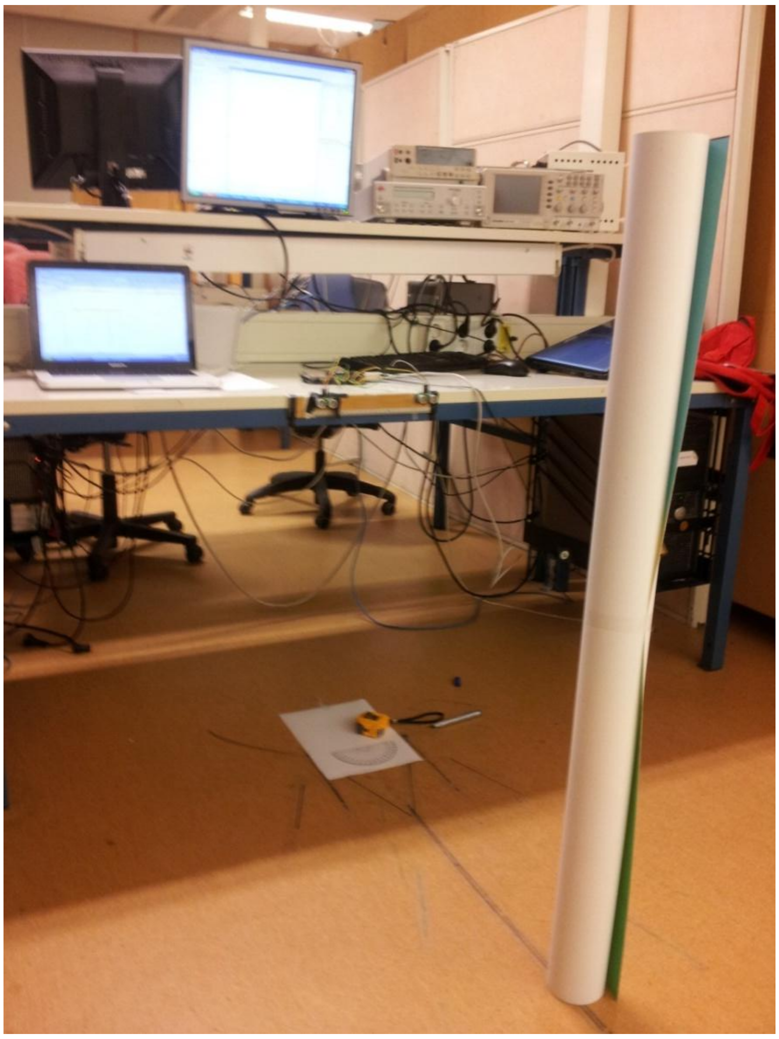


Figure 27:Experimental setup scene

In the above figure an object is placed infront of the sensors. The sensors are triggered using Matlab script and the distance of the object and the angle are calculated using Matlab script.

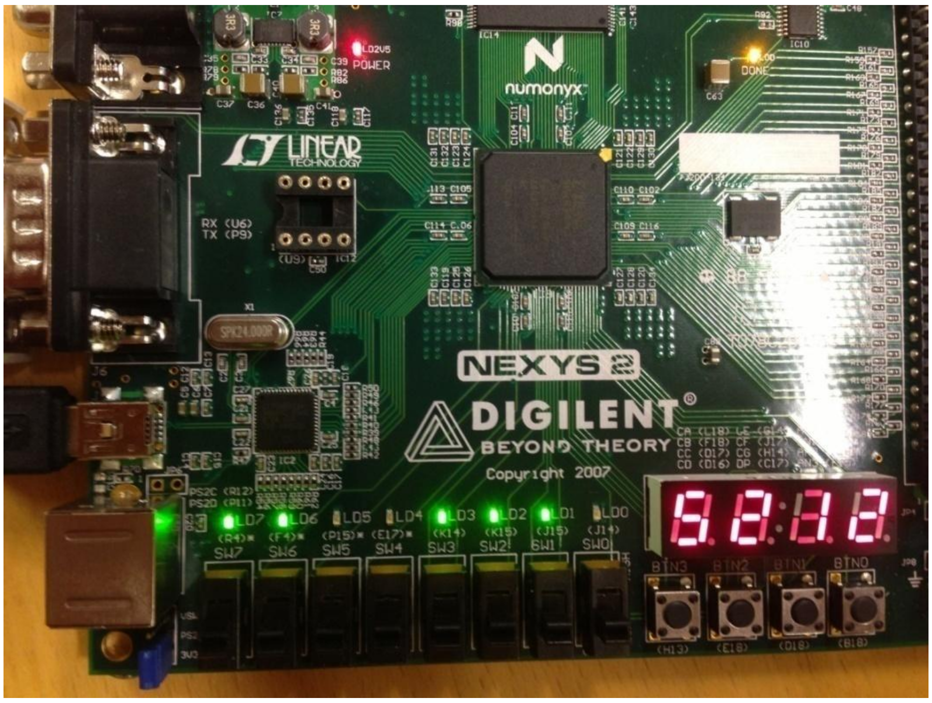


Figure 28: Image showing the first sensor reading with character s on nexys2 board

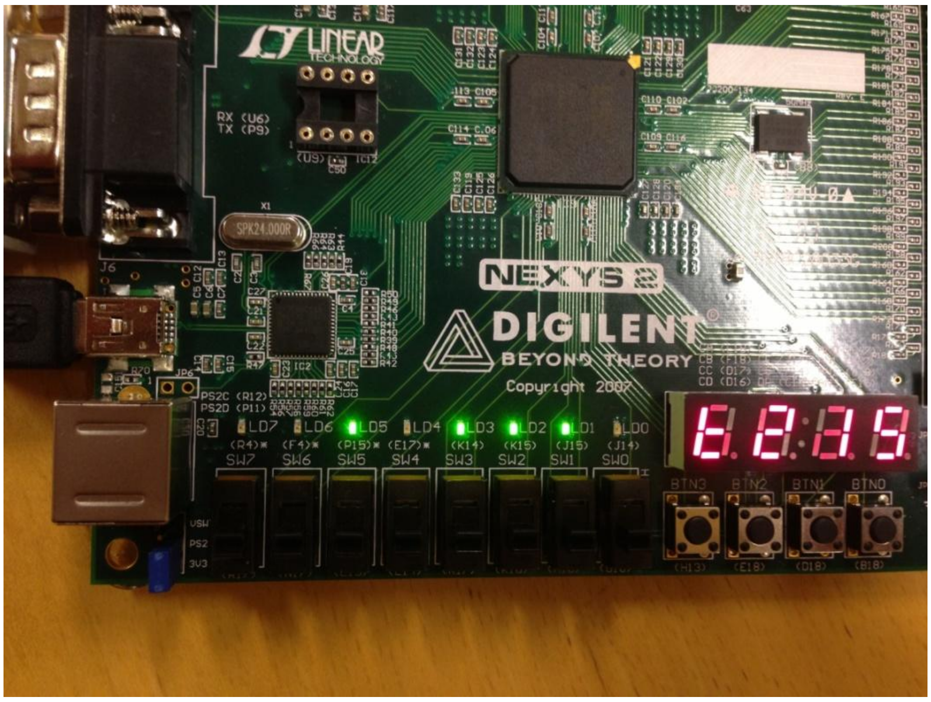


Figure 29: Image showing the second sensor reading with character t on nexys2 board

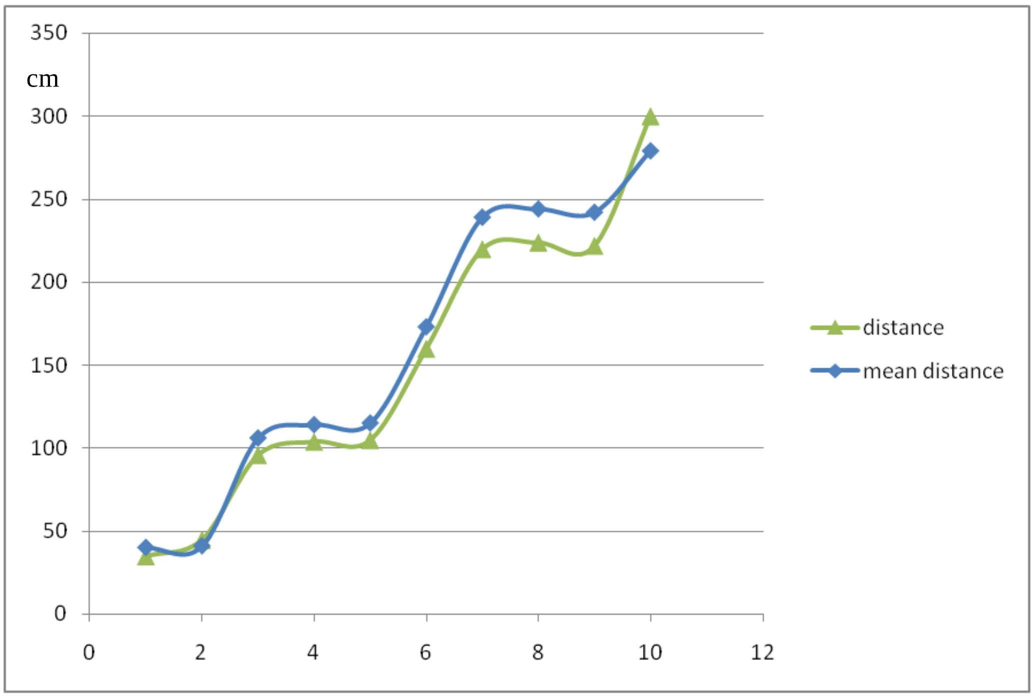


Figure 30: difference between mean distance values taken from Matlab and real distance values

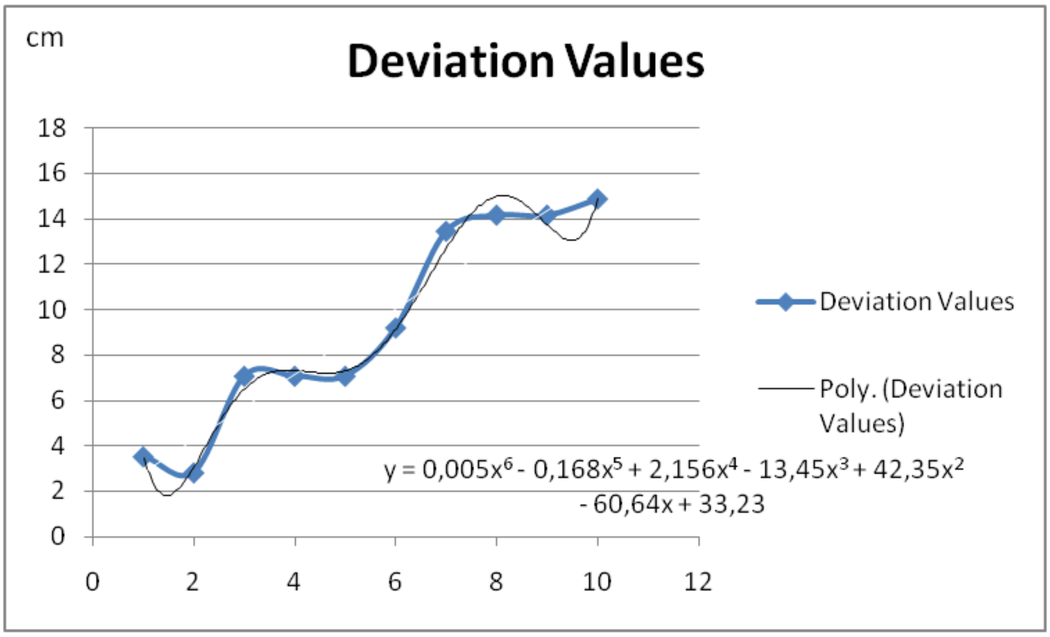


Figure 31: deviation values graph and the modeled function

As can be seen in Figure 30, valued achieved by the sensors are mostly, except the last experiment, lower than the real distance of the test object to the sensors.

Experiments deviation graph is shown in Figure 31, and as it is nonlinear and near to the plotted level 6 polynomial equation, the error distance is not uniform for all rounds and therefore this setup is not reliable for sensitive measurements.

6 Conclusions

In this project the SRF05 range sensor is tested using project setup. The project setup includes two sensors and based on the data given by the sensors distance from the mid-point between the sensors to the object and angle is calculated. By comparing the values of the distance obtained by sensors and by measuring with tape the values obtained from sensors are not precise. A graph is plotted between the distance values obtained by sensors and in reality. The error of the distance is increasing as the distance of the object from the sensors is increasing. The values of the distance obtained by sensors and measured by tape are tabulated and as per the error of the distance from the mid-point of the object at various distance is calculated by the equation. As the experiment room for all tests was the same, it excludes the environment factors for error existence. This error may occur while converting the length of the echo into centimeters, because instead of dividing 8700, here it is divided by 8192 to reduce the usage of hardware resources. But this is very minute. Although it is mentioned in SRF05 datasheet that there may be ±1 inch radius distance error in measurement, but as the experiments were shown deviations much more than expected and, there may be other factors that influence the sensor readings. Therefore it can be concluded there is a percentage error in the values that obtained in the project setup by using SRF05 sensors.

7 References

1. Yousaf, A. 2013; Available from: http://apachepersonal.miun.se/~amiyou/vhdl/project.pdf.
2. *ISim User Guide*. 2011; Available from: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/plugin_ism.pdf.
3. *SRFO5-Ultra-Sonic Ranger* Available from: http://www.robot-electronics.co.uk/htm/srf05tech.htm.
4. *MUSRF05 Interface I.C.*; Available from: http://www.fgcvme.co.uk/MUSRF05%20Data%20Sheet.pdf
5. *RS232 Reference Component*. 2008.
6. Hussain, W. (2012) *Double-Dabble Binary-to-BCD Conversion Algorithm*.
7. *Diligent Nexys2 Board Reference Manual*. 2011 [cited 2013; Available from: http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf

Appendix A: top_module.vhd (port mapping)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity top_module is
port(
clk_main : in  STD_LOGIC;
echo     : in  STD_LOGIC;
echo2    : in  STD_LOGIC;
trg      : out  STD_LOGIC;
trg2     : out  STD_LOGIC;
sseg     : out std_logic_vector(6 downto 0);
anodes   : out std_logic_vector(3 downto 0);
TXD      : out std_logic := '1';
RXD      : in std_logic := '1';
RST      : in std_logic := '0';
LEDS     : out std_logic_vector(7 downto 0)

    );
end top_module;

architecture Behavioral of top_module is

component rangesen
port(
     clk     : in  STD_LOGIC;
     echo    : in  STD_LOGIC;
     trg     : out  STD_LOGIC;
     echo2   : in  STD_LOGIC;
     trg2    : out  STD_LOGIC;
     output : out std_logic_vector(8 downto 0);
     output2: out std_logic_vector(8 downto 0);
     check1  : in  std_logic_vector(7 downto 0)

     );
end component;

component bin2BCD
port(  input1        : in STD_LOGIC_VECTOR(8 downto 0);
       input2        : in STD_LOGIC_VECTOR(8 downto 0);
       hundreds      : out  STD_LOGIC_VECTOR(3 downto 0);
       tens          : out  STD_LOGIC_VECTOR(3 downto 0);
       unit          : out  STD_LOGIC_VECTOR(3 downto 0);
       hundreds_BCD : out  STD_LOGIC_VECTOR(3 downto 0);
       tens_BCD      : out  STD_LOGIC_VECTOR(3 downto 0);
       unit_BCD      : out  STD_LOGIC_VECTOR(3 downto 0)
         );
end component;

component sevenseg
port(
      segclk    :   in std_logic;
      unit1     :   in  STD_LOGIC_VECTOR(3 downto 0);
      tens1     :   in  STD_LOGIC_VECTOR(3 downto 0);
      hundreds1:in  STD_LOGIC_VECTOR(3 downto 0);
      unit2     :   in  STD_LOGIC_VECTOR(3 downto 0);
      tens2     :   in  STD_LOGIC_VECTOR(3 downto 0);
      hundreds2:in  STD_LOGIC_VECTOR(3 downto 0);
      sseg      : out STD_LOGIC_VECTOR(6 downto 0);
      anodes    : out std_logic_vector( 3 downto 0)
   );
end component;

component UART1
port(
TXD        : out std_logic := '1';
RXD        : in std_logic := '1';
CLK        : in std_logic;
LEDS       : out std_logic_vector(7 downto 0);
RST        : in std_logic:= '0';
Tclk       :  inout std_logic;
trig1      :  out std_logic_vector(7 downto 0) ;
segment    :  out std_logic_vector(8 downto 0);
segment1 :  out std_logic_vector(8 downto 0);
cm         :  in std_logic_vector(8 downto 0);
cm2        :  in std_logic_vector(8 downto 0)
 );
end component;


      signal h :   STD_LOGIC_VECTOR(3 downto 0);
      signal t :   STD_LOGIC_VECTOR(3 downto 0);
      signal u :   STD_LOGIC_VECTOR(3 downto 0);

      signal h1      :   STD_LOGIC_VECTOR(3 downto 0);
      signal t1      :   STD_LOGIC_VECTOR(3 downto 0);
      signal u1      :   STD_LOGIC_VECTOR(3 downto 0);
      signal trigger1:std_logic_vector(7 downto 0) ;
      signal dist1   : STD_LOGIC_VECTOR(8 downto 0);
      signal dist2   : STD_LOGIC_VECTOR(8 downto 0);
      signal segment_r1: std_logic_vector(8 downto 0);
      signal segment_r2: std_logic_vector(8 downto 0);
begin
rangesensor: rangesen port map (  clk => clk_main,
                                  echo => echo,
                                  echo2 => echo2,
                                  trg => trg,
                                  trg2 => trg2,
                                  check1=>trigger1,
                                  output => dist1,
                                  output2 => dist2
                                 );


 binarytoBCD: bin2BCD  port map (  input1    => segment_r1,
                                   input2    => segment_r2,
                                   hundreds => h,
                                   tens      => t,
                                   unit      => u,
                                   hundreds_BCD =>h1,
                                   tens_BCD     =>  t1,
                                   unit_BCD     =>   u1
                                  );
sevensegment: sevenseg port map ( segclk   => clk_main,
                                  hundreds1 => h,
                                  tens1     => t,
                                  unit1     => u,
                                  hundreds2 => h1,
                                  tens2     => t1,
                                  unit2     => u1,
                                  sseg     => sseg,
                                  anodes   => anodes
                                  );

DataCntrl: UARTl port map          ( TXD   =>   TXD,
                                    RXD=>   RXD,
                                    CLK=>   clk_main,
                                    segment => segment_r1,
                                    segment1 => segment_r2,
                                    RST=>   RST,
                                    trig1    =>   trigger1,
                                    cm       =>   dist1,
                                    cm2      =>   dist2,
                                    LEDS     =>   LEDS
                                    );
  end Behavioral;
```

RS

Appendix B: rangesensor.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std;
entity rangesen is
    Port (
           clk : in  STD_LOGIC;
           echo: in  STD_LOGIC;
           trg : out  STD_LOGIC;
           echo2: in  STD_LOGIC;
           trg2 : out  STD_LOGIC;
           output : out std_logic_vector(8 downto 0);
           output2 : out std_logic_vector(8 downto 0);
           check1 : in std_logic_vector(7 downto 0)
);
end rangesen;



architecture Behavioral of rangesen is
signal count:std_logic_vector(21 downto 0):=
"0000000000000000000000";
signal count1: std_logic_vector(21 downto 0);
signal count2:std_logic_vector(21downto
0):="0000000000000000000000";
signal count3: std_logic_vector(21 downto 0);
signal input : STD_LOGIC_VECTOR(8 downto 0);
signal input2 : STD_LOGIC_VECTOR(8 downto 0);
signal int : integer range 0 to 2500000;
signal int2 : integer range 0 to 916;
signal int3:integer range 0 to 2500000;
signal int4:integer range 0 to 916;
begin

process(clk)
begin
if(clk'event and clk = '1') then
  if (check1 = X"73") then --check dbInsig contains ascii value
of s or not
    if(count <= "0000000000001001011000"  ) then -- 600 clock
cycles or 12 us
    trg <= '1';
    count <=count+"0000000000000000000001" ;
    else
    trg <= '0';
    count <=count+"0000000000000000000001" ;

if (count ="1001100010010110100000") then--2500000 clock cycles
or 50ms
    count<="0000000000000000000000";
end if;
end if;
end if;
end if;
end process;

process(clk)
begin
  if(clk'event and clk = '1') then
    if (check1 = X"74")  then  --check dbInsig contains ascii
value of t or not
    if(count2 <= "0000000000001001011000"  ) then
    trg2 <= '1';
    count2 <=count2+"0000000000000000000001" ;
    else
    trg2 <= '0';
    count2 <=count2+"0000000000000000000001" ;
    if (count2 ="1001100010010110100000") then
    count2<="0000000000000000000000";

  end if;
  end if;
  end if;
 end if;
 end process;

process(clk, echo)
begin
if clk'event and clk='1' then
      if echo = '1' then
      int <= conv_integer(count1(21 downto 0));
      int2 <= ((int)*3)/8192;             -- convert to
centimeters
      input<= conv_std_logic_vector(int2,9);
      count1 <=count1+"0000000000000000000001";
      else
      output <=input;
      count1 <="0000000000000000000000";
     end if;
     end if;
end process;

process(clk, echo2)
begin

if clk'event and clk='1' then
      if echo2 = '1' then
      int3 <= conv_integer(count3(21 downto 0));
      int4 <= ((int3)*3)/8192;
      input2<= conv_std_logic_vector(int4,9);
      count3 <=count3+"0000000000000000000001";
      else
      output2 <=input2;
      count3 <="0000000000000000000000";
     end if;
 end if;

end process;
end Behavioral;
```

Appendix C: binarytoBCD.vhd

```
----------------------------------------------------------------
------------------
-- Company: MIUN
----------------------------------------------------------------
------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity bin2BCD is
       port(input1      : in STD_LOGIC_VECTOR(8 downto 0);
               input2      : in STD_LOGIC_VECTOR(8 downto 0);
               hundreds : out  STD_LOGIC_VECTOR(3 downto 0);
               tens          : out  STD_LOGIC_VECTOR(3 downto
0);
               unit          : out  STD_LOGIC_VECTOR(3 downto
0);
               hundreds_BCD : out  STD_LOGIC_VECTOR(3 downto 0);
               tens_BCD          : out  STD_LOGIC_VECTOR(3
downto 0);
               unit_BCD          : out  STD_LOGIC_VECTOR(3
downto 0));

end bin2BCD;

architecture Behavioral of bin2BCD is
begin

process(input1)
variable i : integer:=0;
variable BCD : std_logic_vector(20 downto 0);
begin
BCD := (others => '0');
BCD(8 downto 0) := input1;
for i in 0 to 8 loop
--double dabble algorithm for binary to BCD
BCD(19 downto 0) := BCD(18 downto 0) & '0';
if(i < 8 and BCD(12 downto 9) > "0100") then
BCD(12 downto 9) := BCD(12 downto 9) + "0011";
end if;
if(i < 8 and BCD(16 downto 13) > "0100") then
BCD(16 downto 13) := BCD(16 downto 13) + "0011";
end if;
if(i < 8 and BCD(20 downto 17) > "0100") then
BCD(20 downto 17) := BCD(20 downto 17) + "0011";
end if;
end loop;
--BCD output of sensor1
hundreds <= BCD(20 downto 17);
tens <= BCD(16 downto 13);
unit <= BCD(12 downto 9);

end process;
```

```
process(input2)
variable j : integer:=0;
variable BCD2 : std_logic_vector(20 downto 0);
begin
BCD2 := (others => '0');
BCD2(8 downto 0) := input2;

for j in 0 to 8 loop

BCD2(19 downto 0) := BCD2(18 downto 0) & '0';
if(j < 8 and BCD2(12 downto 9) > "0100") then
BCD2(12 downto 9) := BCD2(12 downto 9) + "0011";
end if;
if(j < 8 and BCD2(16 downto 13) > "0100") then
BCD2(16 downto 13) := BCD2(16 downto 13) + "0011";
end if;
if(j < 8 and BCD2(20 downto 17) > "0100") then
BCD2(20 downto 17) := BCD2(20 downto 17) + "0011";
end if;
end loop;

--BCD output of sensor2

hundreds_BCD<= BCD2(20 downto 17);
tens_BCD <= BCD2(16 downto 13);
unit_BCD <= BCD2(12 downto 9);

end process;

end Behavioral;
```

Appendix D: sevensegment.vhd

```
-----------------------------------------------------------------
------------------
-- Company: MIUN
-----------------------------------------------------------------
------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sevenseg is
 port(segclk: in std_logic;
      unit1:   in  STD_LOGIC_VECTOR(3 downto 0);
      tens1:   in  STD_LOGIC_VECTOR(3 downto 0);
      hundreds1:in  STD_LOGIC_VECTOR(3 downto 0);
      unit2:   in  STD_LOGIC_VECTOR(3 downto 0);
      tens2:   in  STD_LOGIC_VECTOR(3 downto 0);
      hundreds2:in  STD_LOGIC_VECTOR(3 downto 0);
      char1 : in std_logic_vector(3 downto 0);
      sseg: out STD_LOGIC_VECTOR(6 downto 0);
      anodes : out std_logic_vector(3 downto 0));

end sevenseg;

architecture Behavioral of sevenseg is
--Initialising arrays to store s,t,d,A values
               signal r_anodes: std_logic_vector(3 downto 0);
       type dbuff is array (0 to 3) of std_logic_vector(3 downto
0);
               signal x:dbuff;
      type dbuff1 is array (0 to 3) of std_logic_vector(3 downto
0);
               signal y:dbuff1;
      type dbuff2 is array (0 to 3) of std_logic_vector(3 downto
0);
               signal z:dbuff2;
      type dbuff3 is array (0 to 3) of std_logic_vector(3 downto
0);
               signal a:dbuff3;
      type dbuff4 is array (0 to 3) of std_logic_vector(3 downto
0);
               signal b:dbuff4;


begin
           --assigning sensor values to arrays
           x(0)<=unit1;
           x(1)<=tens1;
           x(2)<=hundreds1 ;
           x(3) <="0000" ;
           y(0)<=unit2;
           y(1)<=tens2;
           y(2)<=hundreds2 ;
y(3) <="0001" ;
a(0)<="0000";
```

```
a(1)<="0000";
a(2)<="0000" ;
a(3) <="0010" ;
b(0)<="0000";
b(1)<="0000";
b(2)<="0000" ;
b(3) <="0011" ;


proess(segclk,x(0),x(1),x(2),x(3),y(0),y(1),y(2),y(3),a(0),a(1),
a(2),a(3),b(0),b(1),b(2),b(3),count1)
begin
-- creating 2 seconds delay for displaying s,t,d,A values
if rising_edge(segclk) then
if count1 < "00000101111101011110000100000000" then
z(0)<=x(0);
z(1)<=x(1);
z(2)<=x(2) ;
z(3) <=x(3) ;
count1 <= count1 + 1;
elsif
 count1 < "00001011111010111100001000000000" then
z(0)<=y(0);
z(1)<=y(1);
z(2)<=y(2) ;
z(3) <=y(3) ;
count1 <= count1 + 1;
elsif
count1 < "00010001111000011010001100000000" then
z(0)<=a(0);
z(1)<=a(1);
z(2)<=a(2) ;
z(3) <=a(3) ;
 count1 <= count1 + 1;
 elsif
count1 < "00010111110101111000010000000000" then
z(0)<=b(0);
z(1)<=b(1);
z(2)<=b(2) ;
z(3) <=b(3) ;
 count1 <= count1 + 1;
else
count1 <= (others=>'0');
end if;
end if;

end process;


-- lower the clock frequency to 10hz(200ms)
--This makes seven segment data visible to human eyes
begincountClock: process(segclk, counter,count)
    begin
       if rising_edge(segclk) then
       if count < "10000000" then
       count <= count + 1;
 else
counter<=counter+1;
count<=(others=>'0');
end if;
end if;
end process;

anodes <= r_anodes;
-- incrementing the anodes with 200ms delay
process(counter,unit1,tens1,hundreds1,char1)
begin

case counter(1 downto 0) is
            when "00" => r_anodes <="1110" ;
            when "01" => r_anodes <= "1101";
           when "10" => r_anodes <= "1011";
            when "11" => r_anodes <= "0111";
            when others => r_anodes <= "1111";
end case;
    -- displays the s,t,d,A values on seven segments
      case r_anodes is
                when "1110"  =>

                  if z(0) = "0000"  then
                  sseg <= "1000000"; -- 0
                  elsif z(0) = "0001" then
                  sseg <= "1111001"; -- 1
                  elsif z(0) = "0010" then
                  sseg <= "0100100"; -- 2
                  elsif z(0) = "0011" then
                  sseg <= "0110000"; -- 3
                  elsif z(0) <= "0100" then
                  sseg <= "0011001"; -- 4
                  elsif z(0) = "0101" then
                  sseg <= "0010010"; -- 5
                  elsif z(0) = "0110" then
                  sseg <= "0000010"; -- 6
                  elsif z(0) = "0111" then
                  sseg <= "1111000"; -- 7
                 elsif z(0) = "1000" then
                  sseg <= "0000000"; -- 8
                        elsif z(0) = "1001" then
                  sseg <= "0010000"; -- 9
                  elsif z(0) = "1010" then
                  sseg <= "0001000"; -- A
                  elsif z(0) = "1011" then
                  sseg <= "0000011"; -- B
                  elsif z(0) = "1100" then
                  sseg <= "1000110"; -- C
                  elsif z(0) = "1101" then
                  sseg <= "0100001"; -- D
                  elsif z(0) = "1110" then
                  sseg <= "0000110"; -- E
                        else
                        sseg <= "0000000";
                        end if;


             when "1101"  =>

 if z(1) = "0000"  then
                  sseg <= "1000000"; -- 0
                  elsif z(1)  = "0001" then
                  sseg <= "1111001"; -- 1
                  elsif  z(1)  = "0010" then
                  sseg <= "0100100"; -- 2
                  elsif z(1)  = "0011" then
                  sseg <= "0110000"; -- 3
                  elsif  z(1)  = "0100" then
                  sseg <= "0011001"; -- 4
                  elsif  z(1)  = "0101" then
                  sseg <= "0010010"; --5
                  elsif  z(1)  = "0110" then
                  sseg <= "0000010"; -- 6
                  elsif  z(1)  = "0111" then
                  sseg <= "1111000"; -- 7
                  elsif  z(1)  = "1000" then
                  sseg <= "0000000"; -- 8
                  elsif  z(1)  = "1001" then
                  sseg <= "0010000"; -- 9
                  elsif  z(1)  = "1010" then
                  sseg <= "0001000"; -- A
                  elsif  z(1)  = "1011" then
                  sseg <= "0000011"; -- B
                  elsif  z(1)   = "1100" then
                  sseg <= "1000110"; -- C
                  elsif  z(1)   = "1101" then
                  sseg <= "0100001"; -- D
                        elsif  z(1)   = "1110" then
                  sseg <= "0000110"; -- E
                    else                 sseg <= "0000000";
                        end if;


              when "1011"  =>

                  if z(2) = "0000"  then
                  sseg <= "1000000"; -- 0
                  elsif z(2)  = "0001" then
                  sseg <= "1111001"; -- 1
                  elsif z(2)  = "0010" then
                  sseg <= "0100100"; -- 2
                  elsif z(2)  = "0011" then
                  sseg <= "0110000"; -- 3
                  elsif z(2)  = "0100" then
                  sseg <= "0011001"; -- 4
                  elsif z(2)  = "0101" then
                  sseg <= "0010010"; --5
                  elsif z(2)  = "0110" then
                  sseg <= "0000010"; -- 6
                  elsif z(2)  = "0111" then
                  sseg <= "1111000"; -- 7
                        elsif z(2)  = "1000" then
                  sseg <= "0000000"; -- 8
                  elsif z(2)  = "1001" then
                  sseg <= "0010000"; -- 9
                  elsif z(2)  = "1010" then
                  sseg <= "0001000"; -- A
                  elsif z(2)  = "1011" then
                  sseg <= "0000011"; -- B
                  elsif z(2)  = "1100" then
                  sseg <= "1000110"; -- C
                 elsif z(2)  = "1101" then
                  sseg <= "0100001"; -- D
```

```
                    elsif z(2)  = "1110" then
                    sseg <= "0000110"; -- E
                          else
                          sseg <= "0000000";
                          end if;


when "0111" =>
                    if (z(3) = "0000")  then
                    sseg <= "0010010"; --s
                    elsif (z(3) = "0001") then
                    sseg <= "0000111"; -- t
                          elsif (z(3) = "0010") then
                    sseg <= "0100001"; -- d
                    elsif (z(3) = "0011") then
                    sseg <= "0001000"; -- A
                    else
                    sseg <= "1111111"; -- nothing
                          end if;


when others =>
 sseg <= "1111111";

end case;
end process;

end Behavioral;
```

Appendix E: DataCntrl.vhd

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

-----------------------------------------------------------------
---------
--
--Title:    Main entity
--

-----------------------------------------------------------------
---------
entity UARTl is
Port (
TXD : out std_logic := '1';
RXD : in std_logic := '1';
CLK : in std_logic;
LEDS     : out std_logic_vector(7 downto 0) ;
RST : in std_logic:= '0';
trig1 : out std_logic_vector(7 downto 0) ;
tclk:inout std_logic;
segment: out std_logic_vector(8 downto 0);
segment1: out std_logic_vector(8 downto 0);
char : out std_logic_vector(3 downto 0);
cm:in std_logic_vector(8 downto 0);
cm2:in std_logic_vector(8 downto 0)
);
end UARTl;

architecture Behavioral of UARTl is

-----------------------------------------------------------------
---------
-- Local Component, Type, and Signal declarations.
-----------------------------------------------------------------
---------

-----------------------------------------------------------------
---------
--
--Title:    Component Declarations
--

-----------------------------------------------------------------
---------
component RS232RefComp
   Port (
TXD     : out std_logic:= '1';
RXD     : in std_logic;
CLK     : in std_logic;
DBIN    : in std_logic_vector (7 downto 0);
DBOUT   : out std_logic_vector (7 downto 0);
RDA : inout std_logic;
```

```
TBE : inout std_logic := '1';
RD  : in std_logic;
WR  : in std_logic;
PE  : out std_logic;
FE  : out std_logic;
OE  : out std_logic;
RST : in std_logic:= '0';
tclk:inout std_logic);
end component;
-----------------------------------------------------------------
---------
--
--Title:    Type Declarations
--

--
-----------------------------------------------------------------
---------
--different states for transmitting data
    type mainState is (
    stReceive,
    state1,
    state2,
    state3,
    state4
     );

-----------------------------------------------------------------
signal dbInSig  :   std_logic_vector(7 downto 0);
signal dbOutSig :   std_logic_vector(7 downto 0);
signal rdaSig   :   std_logic;
signal tbeSig   :   std_logic;
signal rdSig    :   std_logic;
signal wrSig    :   std_logic;
signal peSig    :   std_logic;
signal feSig    :   std_logic;
signal oeSig    :   std_logic;
signal stCur    :   mainState := stReceive;
signal stnext   :   mainState;

-----------------------------------------------------------------
--------
-- Module Implementation
-----------------------------------------------------------------
--------

begin

UART: RS232RefComp port map (
TXD     => TXD,
RXD     => RXD,
CLK     => CLK,
DBIN    => dbInSig,
DBOUT   => dbOutSig,
RDA => rdaSig,
TBE => tbeSig,
RD  => rdSig,
WR  => wrSig,
PE  => peSig,
FE  => feSig,
OE  => oeSig,
RST     => RST,
tclk=>tclk);
                                    CLK     => CLK,
                                    DBIN    => dbInSig,
                                    DBOUT   => dbOutSig,
                                    tclk=>tclk);
-----------------------------------------------------------------
---------

--Title: Main State Machine controller

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


process (tclk, RST)
--process for changing the states and runs with tclk (transmis--
--sion clock)
begin
if (tclk = '1' and tclk'Event) then

if RST = '1' then
stCur <= stReceive;
else
stCur <= stnext;

end if;
end if;
end process;


-----------------------------------------------------------------
---------
process (tclk,stCur, rdaSig, dboutsig)

begin
if (tclk'Event and tclk = '1') then


case stCur is
-----------------------------------------------------------------
---------

--Title: stReceive state

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

 --IDLE state
 when stReceive =>
 rdSig <= '0';
 wrSig <= '0';

 if rdaSig = '1' then

if (dbOutSig = X"73") then  --check dbOutSig contains ascii ----
--value of s
trig1 <=dbOutSig;                   trig1 <=dbOutSig;
```

```
dbInSig <= (15 downto 9=> '1')& cm2(8);--loading first byte of -
--data from sensor1
segment<= cm(8 downto 0);-- sending sensor1 data to seven seg-
ment                                  segment<= cm(8 downto 0);--
sending sensor1 data to seven segment

stnext <= state1;
elsif (dbOutSig = X"74") then ----check dbOutSig contains ascii
--value of t
trig1 <=dbOutSig;

                               trig1 <=dbOutSig;

dbInsig <=  (15 downto 9=> '1')& cm(8);--loading first byte of
data from sensor2
segment1<= cm2(8 downto 0);-- sending sensor2 data to seven
segment


stnext <= state1;
else
stnext <= stReceive;

 end if;
 end if ;

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

--Title: stSend state

-------------------------------------------------------------------
---------
--sending the data through UART
when state1 =>
rdSig <= '0';
wrSig <= '1';
stnext <= state2 ;
when state2 =>
  -- back to idle state to load second byte of data
rdSig <= '0';
wrSig <= '0';
if (dbOutSig = X"74") then
dbInsig <= cm2( 7 downto 0) ;--loading second byte of data from
--sensor1
end if;
if (dbOutSig = X"73") then
dbInsig <= cm( 7 downto 0) ;--loading second byte of data from -
--sensor2
end if;
stnext <= state3 ;
                         when state3 =>
   --sending the data through UART
rdSig <= '0';
wrSig <= '1';
stnext <= state4 ;
                         when state4 =>
    -- stops sending the data through UART and making RDA port -
--low
 rdSig <= '1';
 wrSig <= '0';
 stnext <= stReceive ;
 end case;
 end if ;
 end process;
end Behavioral;
```

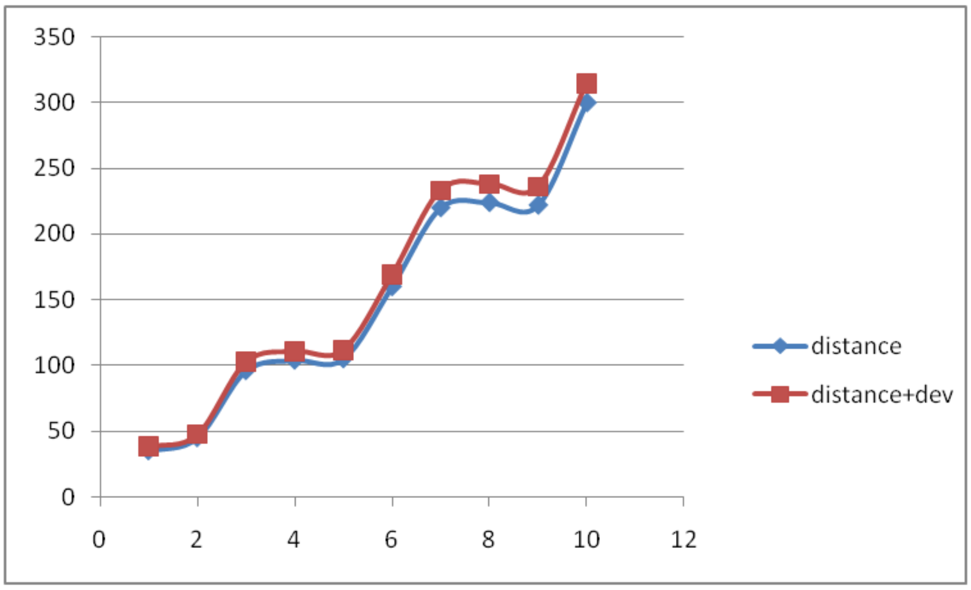
Appendix F: matlab.m

```
clear all;
clc;
nexys = serial('COM1', 'BaudRate', 9600, 'Parity', 'odd',
'Terminator', '', 'Timeout', 1);
fopen(nexys);

 for i=1:2
     fprintf(nexys, 's');
     scan_s1 = fscanf(nexys,'%s')-0;
     fprintf(nexys, 't');
     scan_s2 = fscanf(nexys,'%s')-0;
 end
 if  (scan_s1(1,1)==254)
  scan_s1(1,1)= 0;
  sensor1 = scan_s1(1,2);

 else
     if  (scan_s1(1,1)==255)
     scan_s1(1,1) = 256 ;
     sensor1 = scan_s1(1,2)+scan_s1(1,1);
 end ;
 end ;

if  (scan_s2(1,1)== 254)

  scan_s2(1,1)= 0;
  sensor2 = scan_s2(1,2);
else
     if  (scan_s2(1,1)==255)
     scan_s2(1,1) = 256 ;
     sensor2 = scan_s2(1,2)+scan_s2(1,1);
 end ;
 end ;




% Alpha is the opposite angle of the side a
  a=sensor1;
  b=sensor2;
  c=22; % Distance between two Range Sensors

%  numerator=(b.^2)+(c.^2)-(a.^2); %Täljare
 numerator=round((b.^2)+(c.^2)-(a.^2)); %Täljare
 demoninator=2*b*c;  %Nämnare

 cos_alpha=numerator/demoninator;

 alpha=acosd(cos_alpha); %The angle alpha in degrees

 l=sind(alpha)*b; %The distance from the object linear to be-
tween the sensors
```

```
d=cosd(alpha)*b; %The distance from one corner from the "l"

x=(c/2)-d; %The distance from "l" to the middle of the baseline

dist_float=sqrt(l.^2+x.^2) %The distance from the middle of the
baseline to the object


angle_float=asind(l/dist_float) %The angle from the middle
baseline to the object

distance = (uint16(dist_float))
angle = uint16(angle_float)

fwrite(nexys,d,'uchar');
fwrite(nexys,distance,'uint16');
fwrite(nexys,angle,'uchar');



    fclose(nexys);
    delete(nexys);
    clear nexys;
```

Appendix G: Experiment calculations

	Sensing Values				
	Matlab Value				Real Value
	angle	mean distance	distance from S1	distance from S2	distance
1	85	40			35
2	76	41	45	40	45
3	87	106	106	107	96
4	71	114	111	118	104
5	74	115	118	112	105
6	87	173	173	174	160
7	85	239	238	240	220
8	85	244	245	243	224
9	80	242	240	244	222
10	69	279	275	283	300


distance
distance+dev