SMART LIBRARY LOCKER SYSTEM

LIM JIA XIANG (161615), LIEW MING HENG (161439)

School of Electrical & Electronic Engineering
Universiti Sains Malaysia

Submitted: 23 January 2025

ABSTRACT

The project called Smart Library Locker offers a solution for storing personal belongings when students enter the library. The smart library locker was enhanced with several features by utilizing the FPGA board as the main processor to replace the conventional library locker. Our smart library locker offers an RFID scanning system to scan a student's identity card with RFID to unlock the library locker. On the other hand, some scenarios happen when students forget to bring or lose their student's identity card, the system also offers to use a password system to unlock the library locker. A monitor display is used as the control panel for user interface features through the VGA connection with the FPGA board. Moreover, notification and locker history will be sent to the student's phone to implement the security system. A call for assistance is ready if the library locker malfunction no matter in terms of hardware or software. As a result the smart library locker integrates RFID and password-based authentication, a monitoring system for usage history.

Author Correspondence, e-mail: *mingheng0223@student.usm.my*
jiaxianglim57@student.usm.my

1. INTRODUCTION

1.1 Problem Statement

Libraries are important facilities in academic institutions by providing students and lecturers a conducive environment for learning and research. However, one persistent issue of traditional libraries is the lack of a secure and efficient system for managing personal belongings. Generally, most of the traditional lockers rely on outdated mechanisms such as keys or simple locks, which are prone to a variety of issues like key misplacement, unauthorized access, and insufficient security. These challenges compromise both the convenience and security of locker systems.

1.2 Objectives

- To enhance security and accessibility for library locker systems
- To develop a user-friendly interface with emergency support

1.3 Research Strategy

This study aims to address the limitations of conventional library lockers by designing and implementing a ***Smart Library Locker*** system. This research employs an Altera FPGA (DE1-SoC Board) as the central processing unit to enable advanced functionalities. The proposed solutions integrate ***RFID authentication*** and a ***password-based system*** to provide dual authentication options. Additionally, a ***VGA-connected monitor display*** serves as a user-friendly interface to allow library users to use the library locker in a convenient way while incorporating a power-efficient solution that leverages an ***ultrasonic sensor*** and a low-pass filter algorithm. This approach ensures the monitor operates only when a user is detected.

To further improve the functionality of the system, it also includes notifications and usage history tracking using a ***bluetooth module*** for enhanced security and convenience. An emergency button is also integrated to provide users with immediate access to technical assistance in case of malfunctions.

By combining the hardware and software innovations, this research aims to address existing limitations and provide a technologically advanced locker system tailored to the needs of the library users. This study contributes to the enhancement of campus life while ensuring convenience and robust security for library user's personal belongings.

2.0 Literature review

FPGA Implementation of a VGA Controller

The article 'FPGA Implementation of a VGA Controller' introduces an efficient hardware architecture for VGA monitor controllers based on FPGA technology. The design is compatible with the PLB bus and can be used in Xilinx FPGA-based systems. The architecture supports multiple display resolutions up to WXGA 1280 × 800 and has a customizable internal FIFO, making it suitable for several FPGA devices. Several key concepts are discussed in the article such as pixel data handling for VGA. The controller reads pixel data and drives colour and synchronisation signals to display images on the screen. Pixels are scanned in raster order at a pixel frequency, which depends on the display resolution and refresh rate. Besides the limited memory on FPGA devices, images are typically stored in external memory (e.g., SDRAM) and transferred to the VGA unit in data blocks. This requires an internal FIFO memory to temporarily store these data blocks. The paper also discusses enabling text mode through a software library, converting standard fonts into bitmap formats. This is done via modifying the memory space corresponding to the character position on the screen. This approach is preferred because it is more flexible and requires less logic resources than implementing a character generator in hardware.

Design and development of smart lock system based QRCode for library's locker at Faculty of Engineering, Universitas Riau

This article discusses the design and development of a smart lock system for library lockers at the Faculty of Engineering, Universitas Riau, using QR codes. The system aims to improve security and convenience compared to traditional physical key systems. There are several proposed solutions in the article for smart library locker systems such as customized mobile application, QR code locking system, Email notification etc. Firstly, users register via a mobile app and are verified by a library officer using their student ID to generate a QR code. Thus, the user scans the QR code using the mobile application to unlock the locker. The smart lock system offers several advantages over conventional key systems including: no risk of losing keys, user verification, personal data verification, and theft threat notification. In conclusion, the article presents a working prototype of a smart locker system based on QR codes, highlighting its advantages over traditional systems and identifying areas for future improvement such as the network connection.

3.0 Methodology

3.1 Functional Design of the Solution

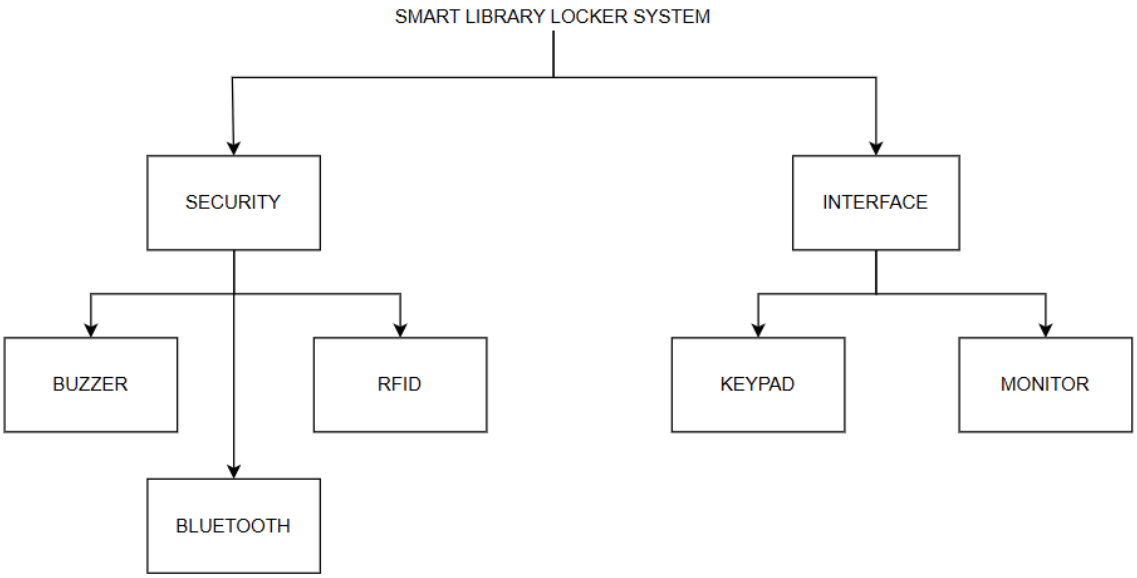


Figure 3.1.1: *Functional Design of Smart Library Locker System*

The system comprises hardware and software components that interact to facilitate secure and increase accessibility to library lockers. Key components include an FPGA board, an RC522 RFID module, HC06 Bluetooth module, HC SR-04 Ultrasonic sensor and a monitoring system. The design proposed a solution for the problem statement.

COMPONENT	***FUNCTIONS***
DE1-SOC FPGA	*Act as main processing unit to handle the preset program flow based on different conditions*
VGA (Video Graphics Array)	*Provides an interface from the DE1-SoC FPGA to output diagrams and text to the monitor screen*
Button	*Act as emergency button*
Buzzer	*Producing "bizz" sound to calling technical services when emergency button pressed*

Bluetooth Module (HC-06)	*Act as data transmission medium to track usage history and share it to the locker owner*
RFID Module (RC522)	*Act as one of authorisation access for the library locker*
4X4 Keypad	*Act as alternative authorisation access method for the library locker in case the card owner loses their card*
Ultrasonic Sensor (HC-SR04)	*Only activate the screen monitor when user approaching to save power during idle state*
LED	*Show the status of the locker*

Table 2.1: *Components used for the Designed Circuit*

3.2 Introduction to the Hardware and Components

DE1-SoC FPGA

Figure 3.2.1: *DE1-SoC FPGA*

The DE1-SoC FPGA is a development board equipped with a ***powerful Cyclone V System-on-Chip (SoC) device***. It integrates an ***ARM Cortex-A9 processor*** and ***FPGA fabric*** onto a single chip, enabling it to handle both general-purpose computing tasks and custom hardware implementations. As a development board, the DE1-SoC FPGA includes a wide

array of pre-built modules and interfaces, making it an adaptable and robust solution for embedded system development.

In this project, the DE1-SoC FPGA acts as the central processing unit, monitoring the operation of all connected peripherals as well as implementing preset control logic. It processes input signals from sensors, modules and some necessary on-board modules for the implemented solutions.

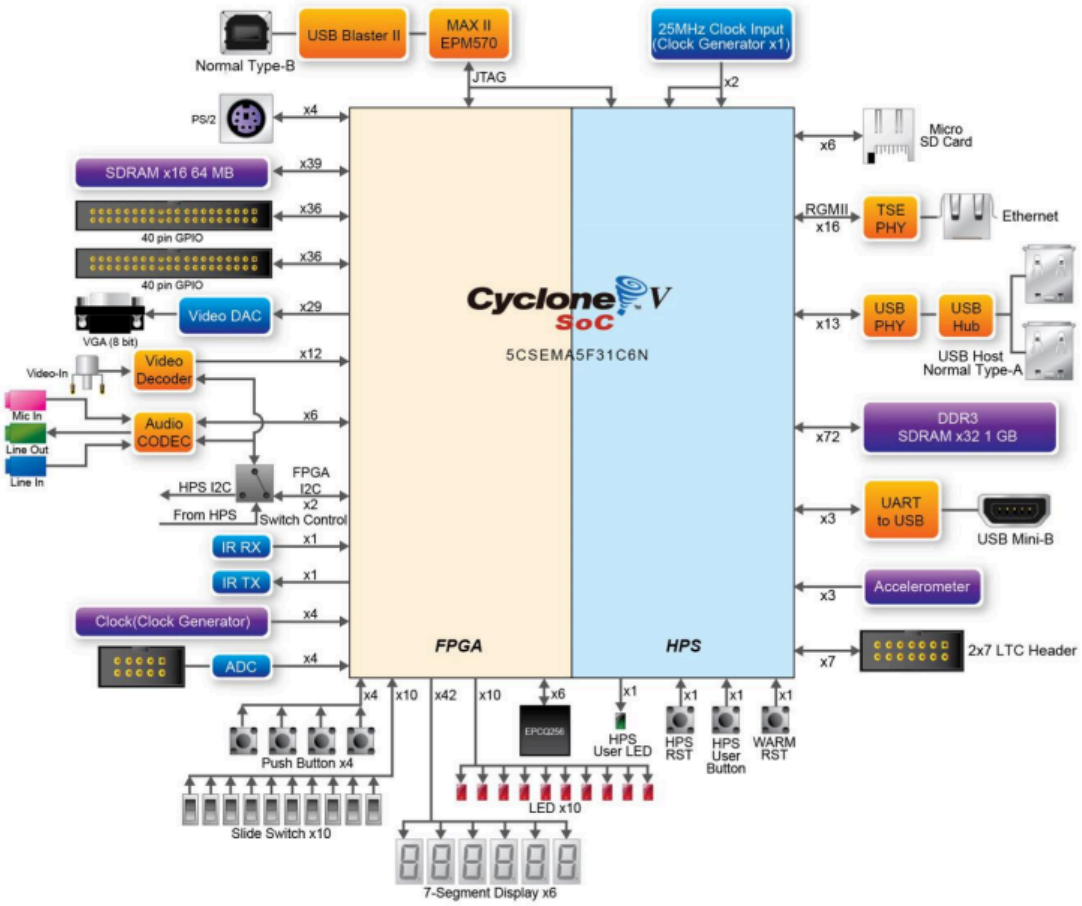


Figure 2-3 Block diagram of DE1-SoC

Figure 3.2.2: *Block Diagram of DE1-SoC FPGA*

VGA (Video Graphics Array)

Figure 3.2.3: *VGA (Video Graphics Array)*

VGA (Video Graphics Array) is a widely used display interface standard introduced by IBM in 1987 and is considered as the predecessor of modern digital interfaces like ***HDMI***. It defines an analog connection used for transmitting video signals from a device (FPGA or PC) to a display monitor. Due to its simplicity and support for a wide range of resolutions, it remains a popular choice for applications in embedded systems and prototyping.

For the working principles of VGA, it transmits video signals using five primary connections:

Connections	***Functions***
Red(R), Green(G), and Blue(B) Signals	These analog signals control the intensity of the primary colors to form the final color displayed on the screen.
Horizontal Sync (HSYNC)	It synchronizes the horizontal scanning of the monitor, determining when a new line starts.
Vertical Sync (VSYNC)	It synchronizes the vertical scanning of the monitor, determining when a new frame starts.

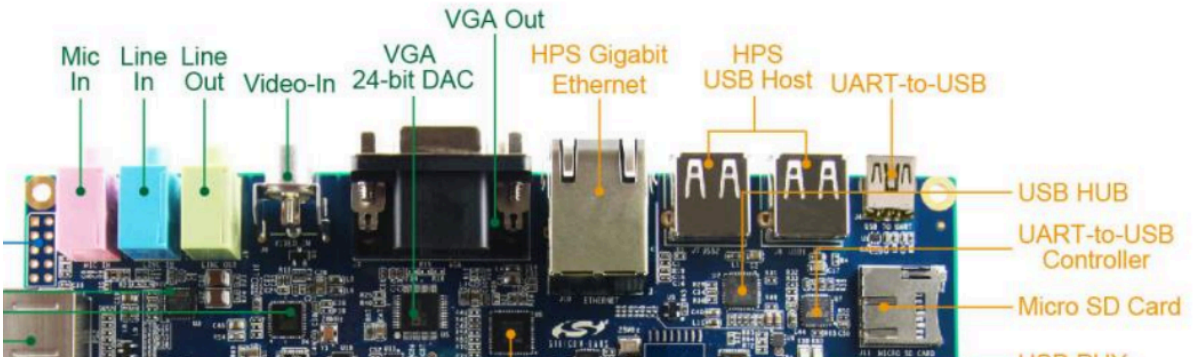


Figure 3.2.4: *VGA (Video Graphics Array) on DE1-SoC FPGA Board*

In this project, the VGA interface is utilized to display characters and visual information on a monitor, providing a user-friendly interface for the smart locker system. It enables real-time updates and allows users to interact with the system through visual feedback.

Button

Figure 3.2.5: *Push Button*

Push buttons are widely used input devices that function as momentary switches. In this project, the push button is implemented as an emergency button that activates a connected buzzer to seek for technical assistance in case of system malfunctions.

Buzzer

Figure 3.2.6: *Buzzer*

Buzzer is an electroacoustic device designed to emit sound when connected to power. It is commonly used for signaling or alert purposes in electronic systems as well as embedded systems. In this project, the buzzer serves as an emergency alert system. When the push button is pressed, the DE1-SoC FPGA processes the input signal through interrupt and activates the buzzer to notify technical assistance.

Bluetooth Module (HC-06)

Figure 3.2.7: *Bluetooth Module (HC-06)*

Bluetooth Module HC-06 is a wireless communication device widely used in embedded systems for short-range data transmission. This module operates on ***Bluetooth 2.0 standards*** and uses a ***serial communication protocol (UART)*** to interface with microcontrollers or processors. Due to its simplicity and reliable performance, the HC-06 is a popular choice for enabling wireless connectivity in various applications.

In this project, the HC-06 module functions as a transmission medium to wirelessly transmit usage history data from the smart locker system to the locker user. This feature enables the user to receive real-time notifications, providing immediate alerts in case of unauthorized access or attempted theft, thereby enhancing the overall security of the system.

RFID Module (RC522)

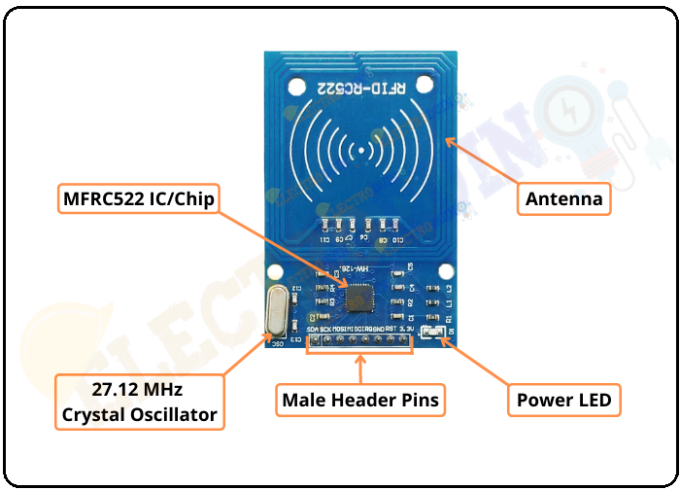


Figure 3.2.8: *RFID Module (RC522)*

RFID Module (RC522) is a low-cost, contactless communication module based on the ***MFRC522*** chip produced by NXP. It operates at a frequency of 13.56MHz and supports various RFID protocols such as ***ISO/IEC 14443 A/MIFARE***. The module is widely used in embedded systems for secure identification and authentication applications due to its ability to read and write RFID tags and cards. The RC522 communicates with microcontrollers or processors through an external peripheral communication interface (***SPI, I2C or UART interfaces***), providing flexibility for integration into different embedded systems.

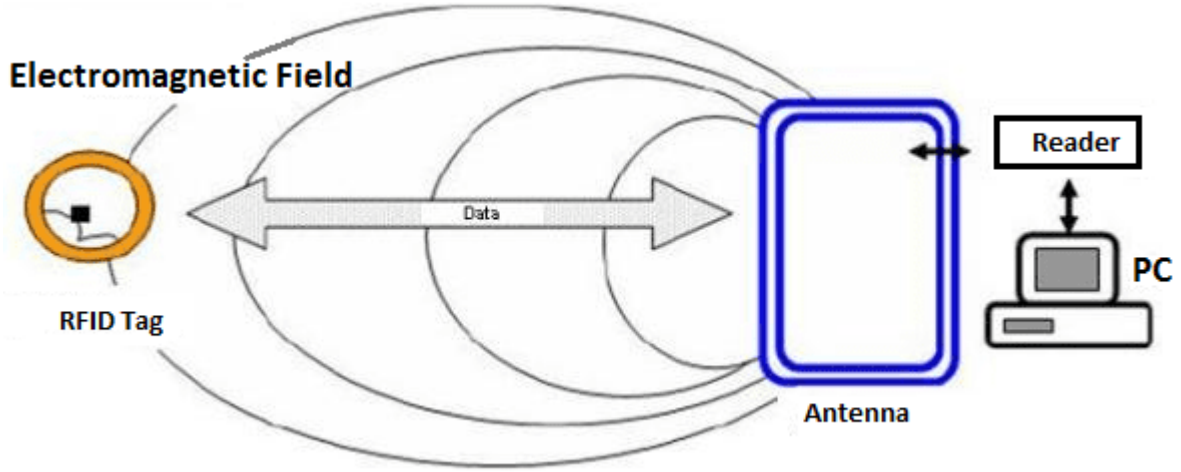


Figure 3.2.9: *Working Principle of RFID Module (RC522)*

For the working principle of the RFID module, the RFID reader module operates by generating a high-frequency electromagnetic field using its control unit and antenna coil. When an RFID tag enters the detection range of the reader module, the electromagnetic field induces a voltage in the antenna coil of the tag through mutual induction, powering the tag's microchip. Once powered, the tag begins transmitting data serially to the reader, which then captures the tag's information.

In this project, the RC522 module serves as one of the primary authentication mechanisms for the smart locker system. Users can unlock their assigned lockers by scanning their RFID cards or tags (Student Cards), which contain unique identification information. This method ensures a secure, efficient and contactless way to access the lockers. Besides that, the integration of the RC522 enhances user convenience and minimizes the need for physical keys, as well as reducing the risk of unauthorized access.

4X4 Keypad

Figure 3.2.10: *RFID Module (RC522)*

The 4X4 Keypad is a compact input device used for various embedded systems applications to enable users to key in the password. It consists of a matrix of 16 buttons arranged in 4 rows and 4 columns. Each button connects a specific row and column when pressed, allowing the system to identify which key has been activated by scanning the rows and columns. This setup minimizes the number of required GPIO pins compared to using individual buttons.

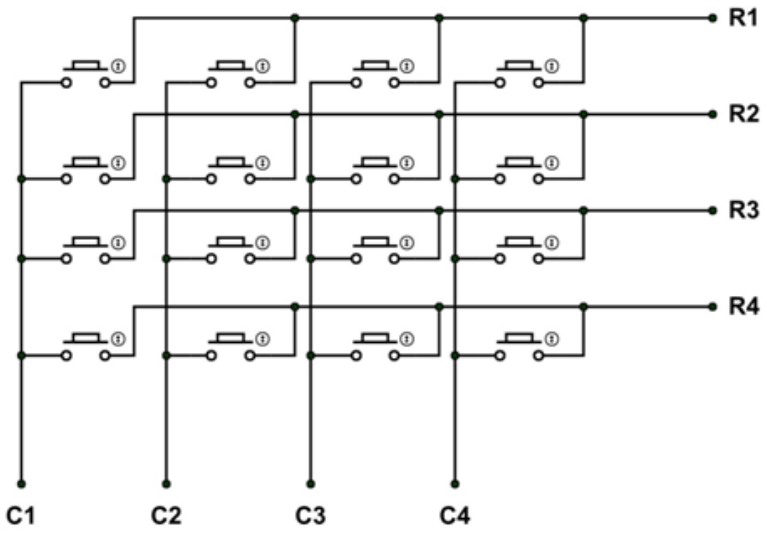


Figure 3.2.11: *Internal Structure of 4X4 Keypad Matrix*

In the smart locker system, the 4X4 keypad acts as an alternative authentication

method. In cases where the RFID card is unavailable, users can unlock the locker by entering a predefined password through the keypad. This approach ensures uninterrupted access, even if the user loses their RFID card, by providing a secure and reliable way to manually input credentials.

Ultrasonic Sensor (HC-SR04)

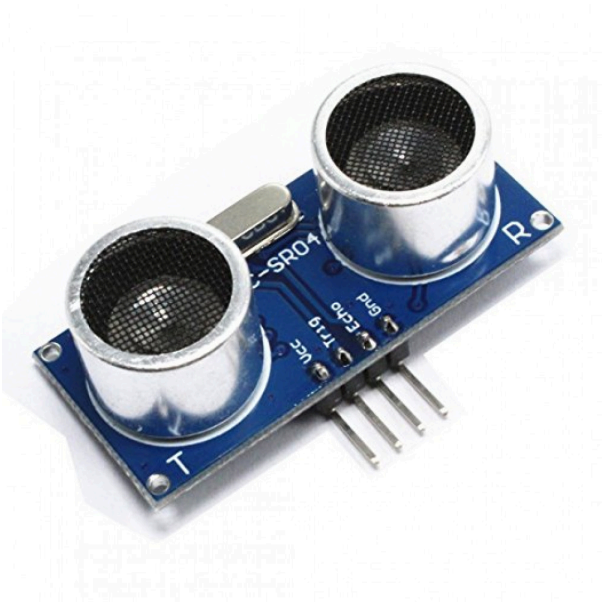


Figure 3.2.12: *Ultrasonic Sensor (HC-SR04)*

Ultrasonic Sensor (HC-SR04) is a widely used sensor module designed for precise distance measurement. It operates by emitting ultrasonic sound waves from its transmitter, which reflect off an object and return to the receiver. The time taken for the echo to return is used to calculate the distance to the object based on the speed of sound. This sensor is reliable, affordable, and commonly used in robotics, obstacle detection, and automation systems.

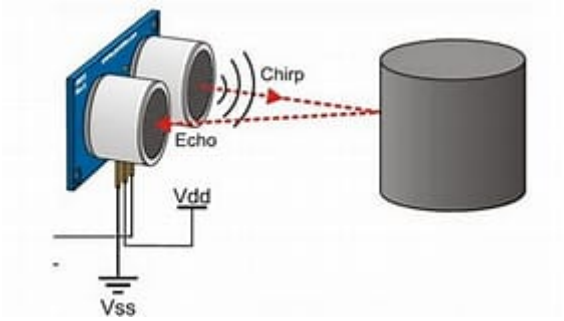


Figure 3.2.13: *Working Principle of Ultrasonic Sensor (HC-SR04)*

The ultrasonic sensor operates by measuring the time it takes for an ultrasonic sensor

wave sent by a transmitter to reflect off an object and return to the receiver, enabling precise distance calculations. The operated ultrasonic frequencies are above 20kHz, which are inaudible to humans, capable of detecting various types of materials, including solids, liquids, granules, and powders, as well as transparent, shiny, or color-changing objects.

In the smart locker system, the HC-SR04 ultrasonic sensor is integrated to enhance power efficiency. By detecting user presence near the locker, it enables the system to activate only which helps conserve power and ensure the locker system operates efficiently.

LED (Light Emitting Diode)

Figure 3.2.14: *LED*

An LED (Light Emitting Diode) is a semiconductor device that emits light when an electric current passes through it. It works on the principle of electroluminescence, where electrons recombine with holes within the semiconductor material, releasing energy in the form of light. LEDs are widely used for their energy efficiency, long lifespan, and ability to emit light in various colors.

In this smart locker system, a green LED is used to show the status of the locker status. When the locker is opened through authentication, the green LED will be lit up.

3.3 Software Components

The project not only requires hardware components, several software components are used to do the hardware configuration and also to upload code into the FPGA board.

Quartus II

Quartus II (***previous version of Quartus Prime***) is a comprehensive development platform used for designing, simulating and programming FPGAs, CPLDs, and SoC (System-on-Chip) devices. It provides tools for hardware design, synthesis, simulation, and programming. In our project, Quartus II used to set up hardware configuration in the FPGA fabric using Verilog. In addition, Quartus II also helps to integrate with the Qsys system that we set up in the Nios II. The software links together the Qsys design with the hardware components on the board.

Qsys

Qsys is a system integration tool within the Quartus II software suite. It is used to build complex FPGA systems by connecting intellectual property (IP) cores and custom modules in a graphical interface. Qsys offers multiple prebuilt IP cores for peripherals like SPI, UART, and GPIO, simplifying system design and allowing users to configure communication interfaces, such as memory-mapped or streaming interfaces, for modules. Importantly, Qsys generates HDL files and a *.sopcinfo* file to describe the design, which can be linked with Quartus II to do further integration.

Nios II Software Build Tools for Eclipse

Nios II Eclipse, part of the Nios II Software Build Tools (SBT), is an IDE used for developing software applications for the Nios II processor on an FPGA. It integrates software development with hardware design for embedded systems. Nios II Eclipse generates the *Board Support Package (BSP) files*[1], which includes HAL libraries and drivers for the hardware system described in the *.sopcinfo* file. This software allows us to write our C programming on it and generates executable files which is the *.elf* file to program and run on the FPGA board..

3.4 Hardware Circuit Setup

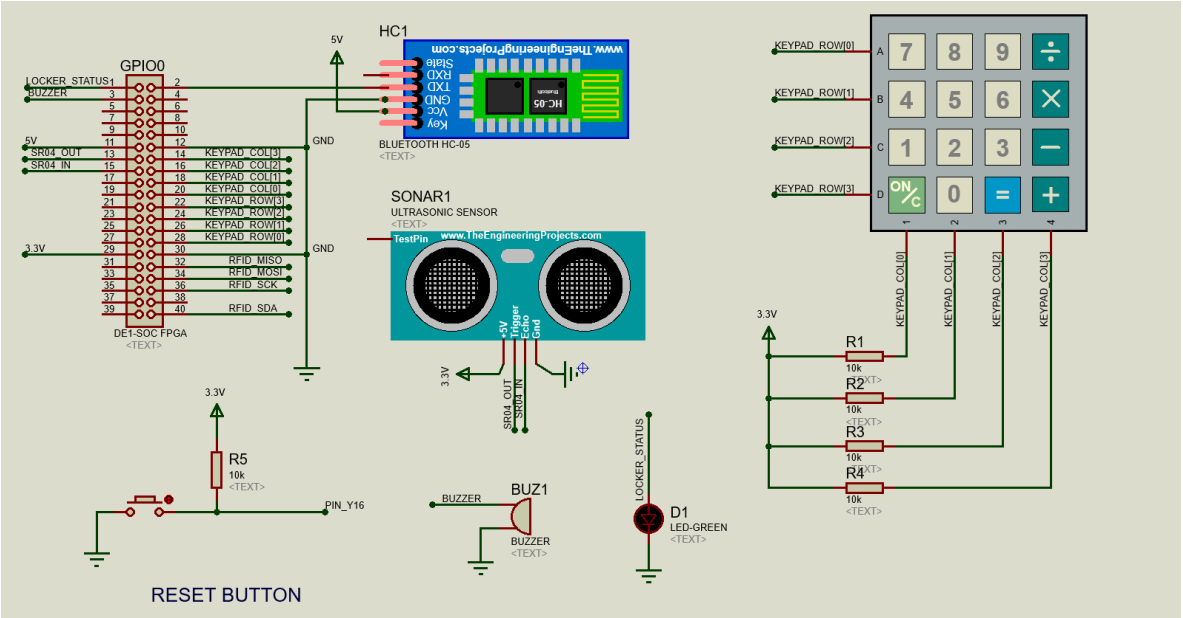


Figure 3.4.1: *Schematic Diagram of the Designed Circuit*

[1] ***Board Support Package (BSP) files:*** *Collection of software components and configuration files that provide essential support for bridging the gap between the hardware and the software stack.*

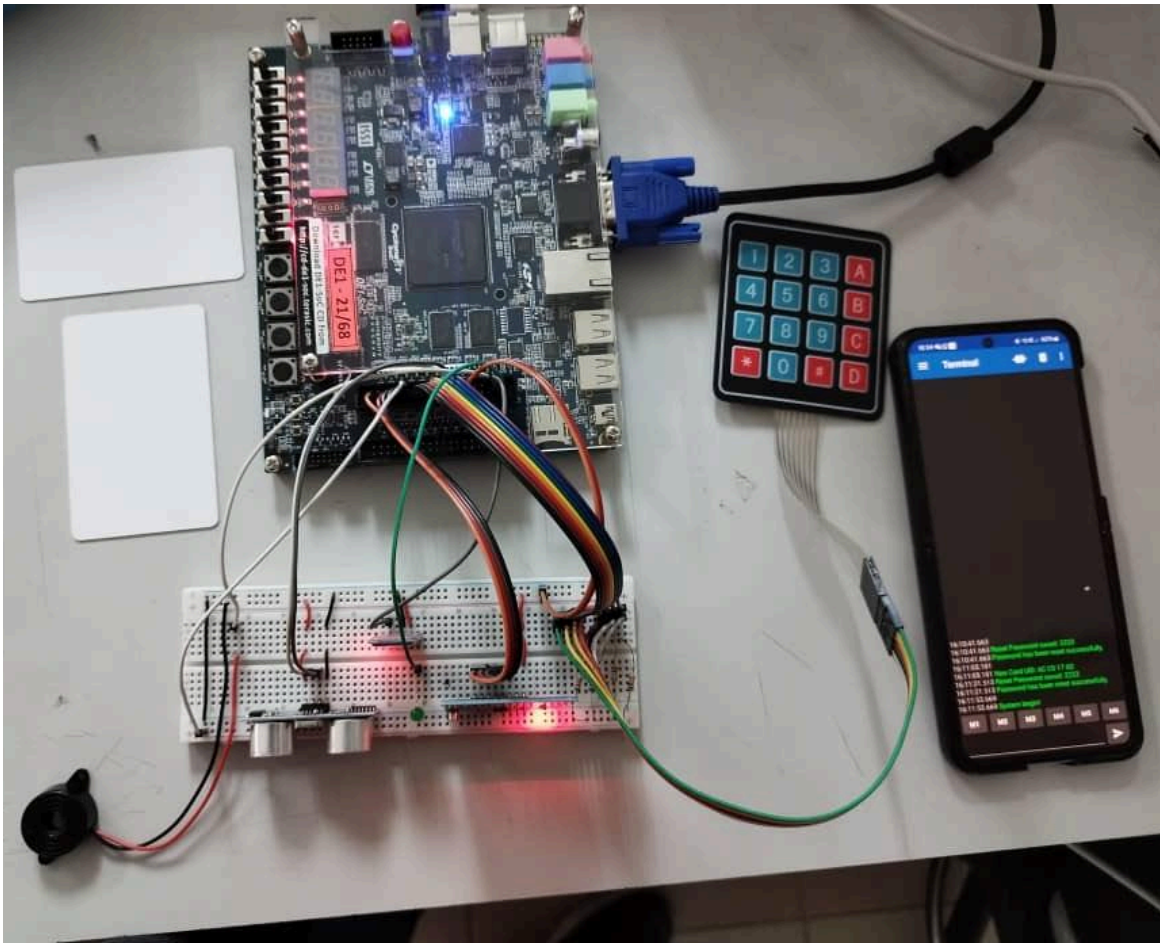


Figure 3.4.2: *Hardware Setup of the Designed Circuit*

Circuit above shows the connection for the whole project that connects all hardware components. The connection of the system as shown in the table below for better understanding.

Devices Pin	***Pin Allocation (FPGA)***
Locker_status	*GPIO_0[0]*
FPGA TX(Bluetooth)	*GPIO_0[3]*
Buzzer	*GPIO_0[4]*
Ultrasonic Trig	*GPIO_0[10]*
Ultrasonic Echo	*GPIO_0[12]*
keypad_col[3]	*GPIO_0[11]*
keypad_col[2]	*GPIO_0[13]*

keypad_col[1]	*GPIO_0[15]*
keypad_col[0]	*GPIO_0[17]*
keypad_row[3]	*GPIO_0[19]*
keypad_row[2]	*GPIO_0[21]*
keypad_row[1]	*GPIO_0[23]*
keypad_row[0]	*GPIO_0[25]*
SPI_MISO	*GPIO_0 [27]*
SPI_MOSI	*GPIO_0 [29]*
SPI_SCLK	*GPIO_0 [31]*
SPI_SS_n	*GPIO_0 [35]*
call_button	*Key[0]*
Reset	*KEY[3]*
25MHz clk - PLL (Status)	*LEDR[0]*

Table 3.4.1: *Pin Allocation*

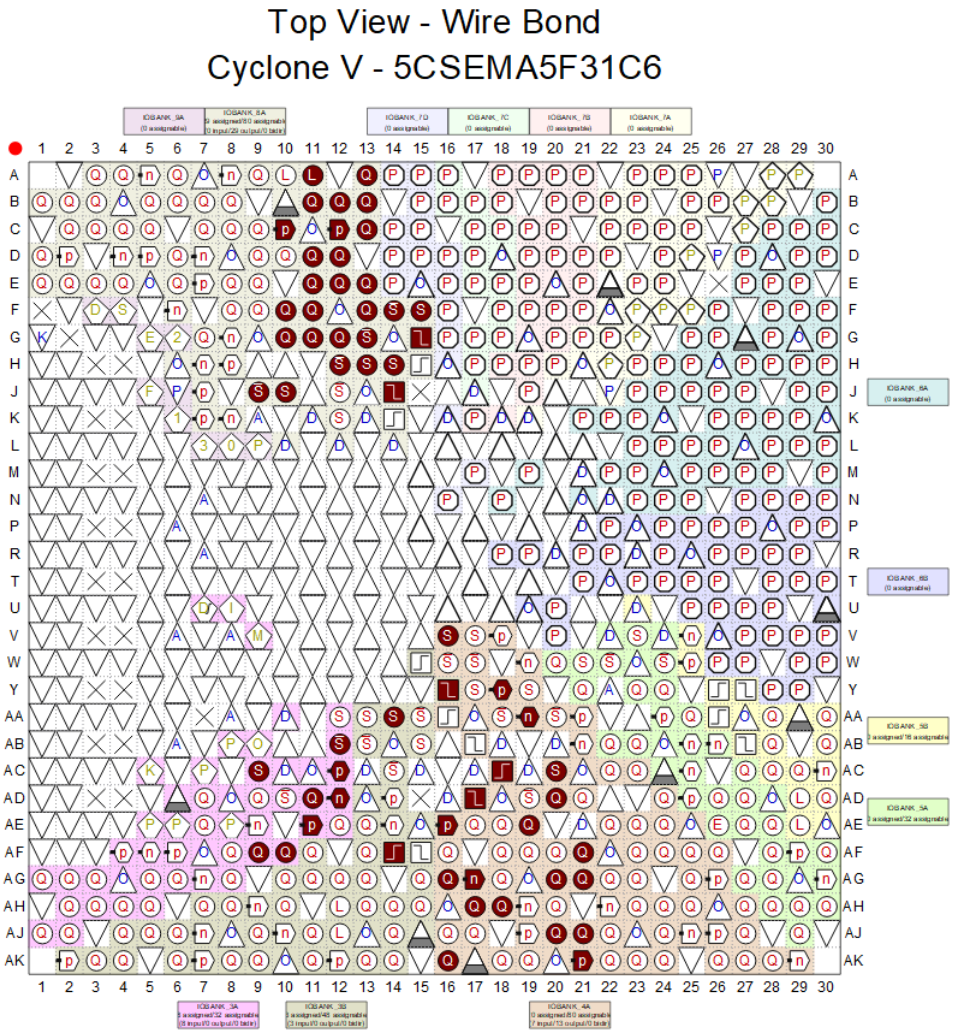
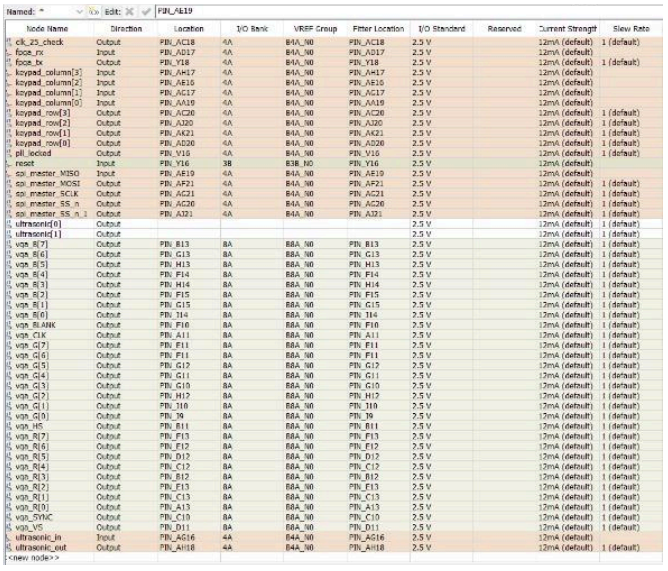


Figure 3.4.3: *Pin Planner in Quartus Prime*



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate
clk_25_check	Output	PIN_AC18	4A	B4A_N0	PIN_AC18	2.5 V		12mA (default)	1 (default)
fpga_rx	Input	PIN_AD17	4A	B4A_N0	PIN_AD17	2.5 V		12mA (default)	
fpga_tx	Output	PIN_Y18	4A	B4A_N0	PIN_Y18	2.5 V		12mA (default)	1 (default)
keypad_column[3]	Input	PIN_AH17	4A	B4A_N0	PIN_AH17	2.5 V		12mA (default)	
keypad_column[2]	Input	PIN_AE16	4A	B4A_N0	PIN_AE16	2.5 V		12mA (default)	
keypad_column[1]	Input	PIN_AG17	4A	B4A_N0	PIN_AG17	2.5 V		12mA (default)	
keypad_column[0]	Input	PIN_AA16	4A	B4A_N0	PIN_AA16	2.5 V		12mA (default)	
keypad_row[3]	Output	PIN_AC20	4A	B4A_N0	PIN_AC20	2.5 V		12mA (default)	1 (default)
keypad_row[2]	Output	PIN_AJ20	4A	B4A_N0	PIN_AJ20	2.5 V		12mA (default)	1 (default)
keypad_row[1]	Output	PIN_AK21	4A	B4A_N0	PIN_AK21	2.5 V		12mA (default)	1 (default)
keypad_row[0]	Output	PIN_AD20	4A	B4A_N0	PIN_AD20	2.5 V		12mA (default)	1 (default)
pll_locked	Output	PIN_V16	4A	B4A_N0	PIN_V16	2.5 V		12mA (default)	1 (default)
reset	Input	PIN_Y16	3B	B3B_N0	PIN_Y16	2.5 V		12mA (default)	
spi_master_MISO	Input	PIN_AE19	4A	B4A_N0	PIN_AE19	2.5 V		12mA (default)	
spi_master_MOSI	Output	PIN_AF21	4A	B4A_N0	PIN_AF21	2.5 V		12mA (default)	1 (default)
spi_master_SCLK	Output	PIN_AG21	4A	B4A_N0	PIN_AG21	2.5 V		12mA (default)	1 (default)
spi_master_SS_n	Output	PIN_AG20	4A	B4A_N0	PIN_AG20	2.5 V		12mA (default)	1 (default)
spi_master_SS_n_1	Output	PIN_AJ21	4A	B4A_N0	PIN_AJ21	2.5 V		12mA (default)	1 (default)
ultrasonic[0]	Output					2.5 V		12mA (default)	1 (default)
ultrasonic[1]	Output					2.5 V		12mA (default)	1 (default)
vga_B[7]	Output	PIN_B13	8A	B8A_N0	PIN_B13	2.5 V		12mA (default)	1 (default)
vga_B[6]	Output	PIN_G13	8A	B8A_N0	PIN_G13	2.5 V		12mA (default)	1 (default)
vga_B[5]	Output	PIN_H13	8A	B8A_N0	PIN_H13	2.5 V		12mA (default)	1 (default)
vga_B[4]	Output	PIN_F14	8A	B8A_N0	PIN_F14	2.5 V		12mA (default)	1 (default)
vga_B[3]	Output	PIN_H14	8A	B8A_N0	PIN_H14	2.5 V		12mA (default)	1 (default)
vga_B[2]	Output	PIN_F15	8A	B8A_N0	PIN_F15	2.5 V		12mA (default)	1 (default)
vga_B[1]	Output	PIN_G15	8A	B8A_N0	PIN_G15	2.5 V		12mA (default)	1 (default)
vga_B[0]	Output	PIN_J14	8A	B8A_N0	PIN_J14	2.5 V		12mA (default)	1 (default)
vga_BLANK	Output	PIN_F10	8A	B8A_N0	PIN_F10	2.5 V		12mA (default)	1 (default)
vga_CLK	Output	PIN_A11	8A	B8A_N0	PIN_A11	2.5 V		12mA (default)	1 (default)
vga_G[7]	Output	PIN_F11	8A	B8A_N0	PIN_F11	2.5 V		12mA (default)	1 (default)
vga_G[6]	Output	PIN_F11	8A	B8A_N0	PIN_F11	2.5 V		12mA (default)	1 (default)
vga_G[5]	Output	PIN_G12	8A	B8A_N0	PIN_G12	2.5 V		12mA (default)	1 (default)
vga_G[4]	Output	PIN_G11	8A	B8A_N0	PIN_G11	2.5 V		12mA (default)	1 (default)
vga_G[3]	Output	PIN_G10	8A	B8A_N0	PIN_G10	2.5 V		12mA (default)	1 (default)
vga_G[2]	Output	PIN_H12	8A	B8A_N0	PIN_H12	2.5 V		12mA (default)	1 (default)
vga_G[1]	Output	PIN_J10	8A	B8A_N0	PIN_J10	2.5 V		12mA (default)	1 (default)
vga_G[0]	Output	PIN_J9	8A	B8A_N0	PIN_J9	2.5 V		12mA (default)	1 (default)
vga_HS	Output	PIN_B11	8A	B8A_N0	PIN_B11	2.5 V		12mA (default)	1 (default)
vga_R[7]	Output	PIN_F13	8A	B8A_N0	PIN_F13	2.5 V		12mA (default)	1 (default)
vga_R[6]	Output	PIN_F12	8A	B8A_N0	PIN_F12	2.5 V		12mA (default)	1 (default)
vga_R[5]	Output	PIN_D12	8A	B8A_N0	PIN_D12	2.5 V		12mA (default)	1 (default)
vga_R[4]	Output	PIN_C12	8A	B8A_N0	PIN_C12	2.5 V		12mA (default)	1 (default)
vga_R[3]	Output	PIN_B12	8A	B8A_N0	PIN_B12	2.5 V		12mA (default)	1 (default)
vga_R[2]	Output	PIN_E13	8A	B8A_N0	PIN_E13	2.5 V		12mA (default)	1 (default)
vga_R[1]	Output	PIN_C13	8A	B8A_N0	PIN_C13	2.5 V		12mA (default)	1 (default)
vga_R[0]	Output	PIN_A13	8A	B8A_N0	PIN_A13	2.5 V		12mA (default)	1 (default)
vga_SYNC	Output	PIN_C10	8A	B8A_N0	PIN_C10	2.5 V		12mA (default)	1 (default)
vga_VS	Output	PIN_D11	8A	B8A_N0	PIN_D11	2.5 V		12mA (default)	1 (default)
ultrasonic_in	Input	PIN_AG16	4A	B4A_N0	PIN_AG16	2.5 V		12mA (default)	
ultrasonic_out	Output	PIN_AH18	4A	B4A_N0	PIN_AH18	2.5 V		12mA (default)	1 (default)
<<new node>>									

Figure 3.4.4: *Pin Assign List*

3.5 Software Setup

After the hardware circuit setup, the project requires several software setup in order to program the board. The step by step setup of the software is shown below:

3.5.1 Qsys Setup

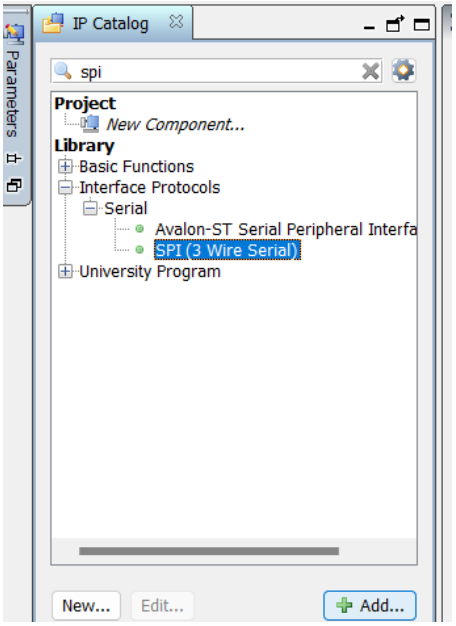


Figure 3.5.1.1: *Pin Assign List*

In the Qsys setup, the required IP cores are added into the system in the IP core library panel. For example, a SPI communication IP core is added into the Qsys system as demonstrated as shown in figure above. By using the same method, all IP cores that needed in the project such as system clock, ram, nios processor, UART JTAG, SPI communication, VGA, PLL, UART (RS232 Serial Port), GPIO pins and interrupt pin are added into the Qsys system. Table below shown the purpose of adding the IP cores or the usage of the IP cores:

IP Cores	***Usage / Function***
System Clock	*As the clock input for the system*
RAM	*To store memory*
Nios Processor	*The main processor of the system*
UART JTAG	*Used to upload program into the board*
SPI Communication	*Interface with RFID module*
VGA	*Interface with Monitor*

PLL	*Used to produce 25MHz that needed by VGA (640 X 480)*
UART (RS232 Serial Port)	*Connected to Bluetooth Module*
GPIO pins	*Connected to several sensor (keypad, Ultrasonic module, LED, Buzzer)*
Interrupt pin	*Used for call for assistance input button*

Table 3.5.1.1: *IP Cores used*

After all the required IP cores are added, the connection is made in the Qsys as the internal connection inside the Nios processor. All the Qsys connection of the project as shown in the figures below:

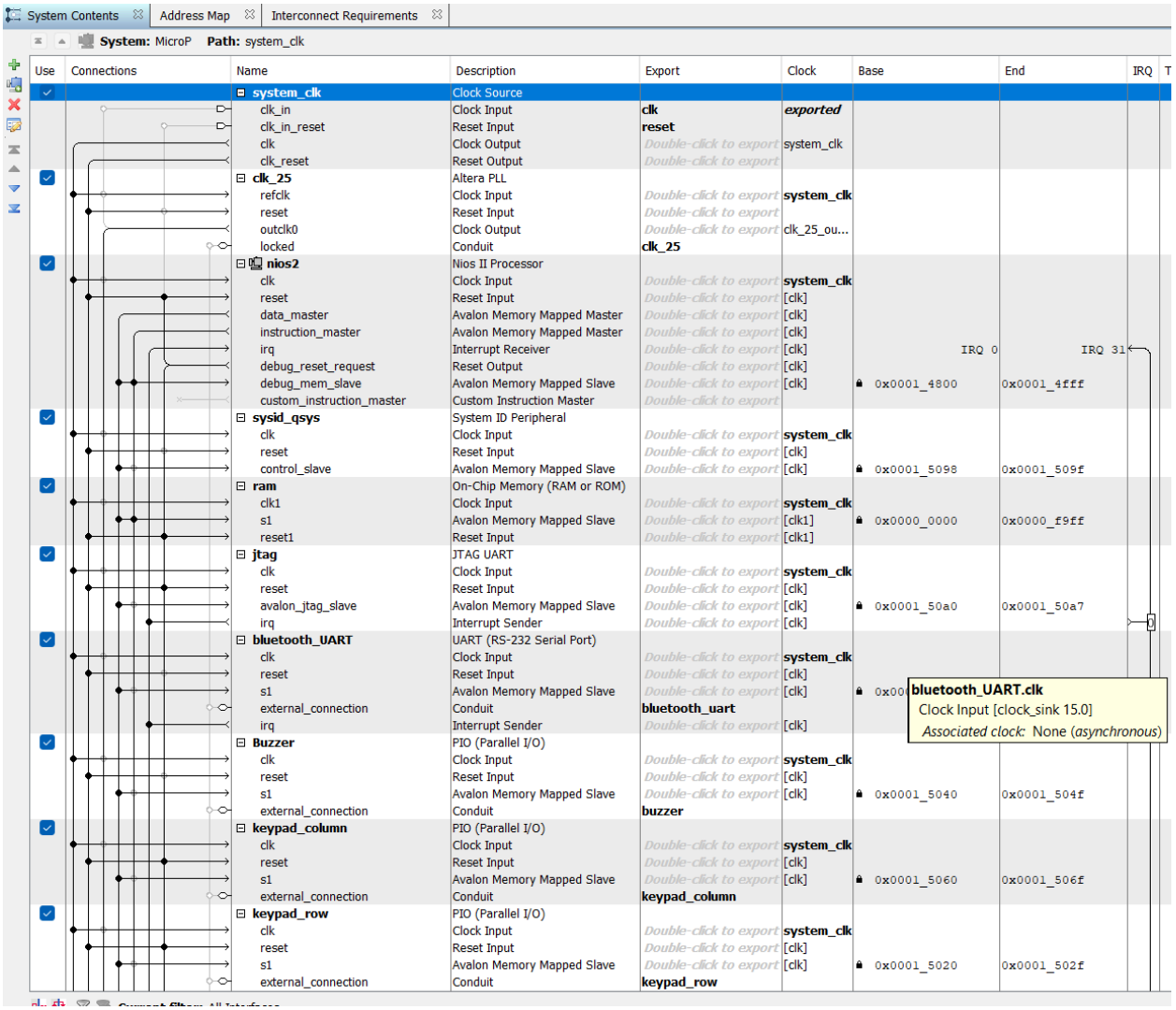


Figure 3.5.1.2: *Qsys Connection 1*

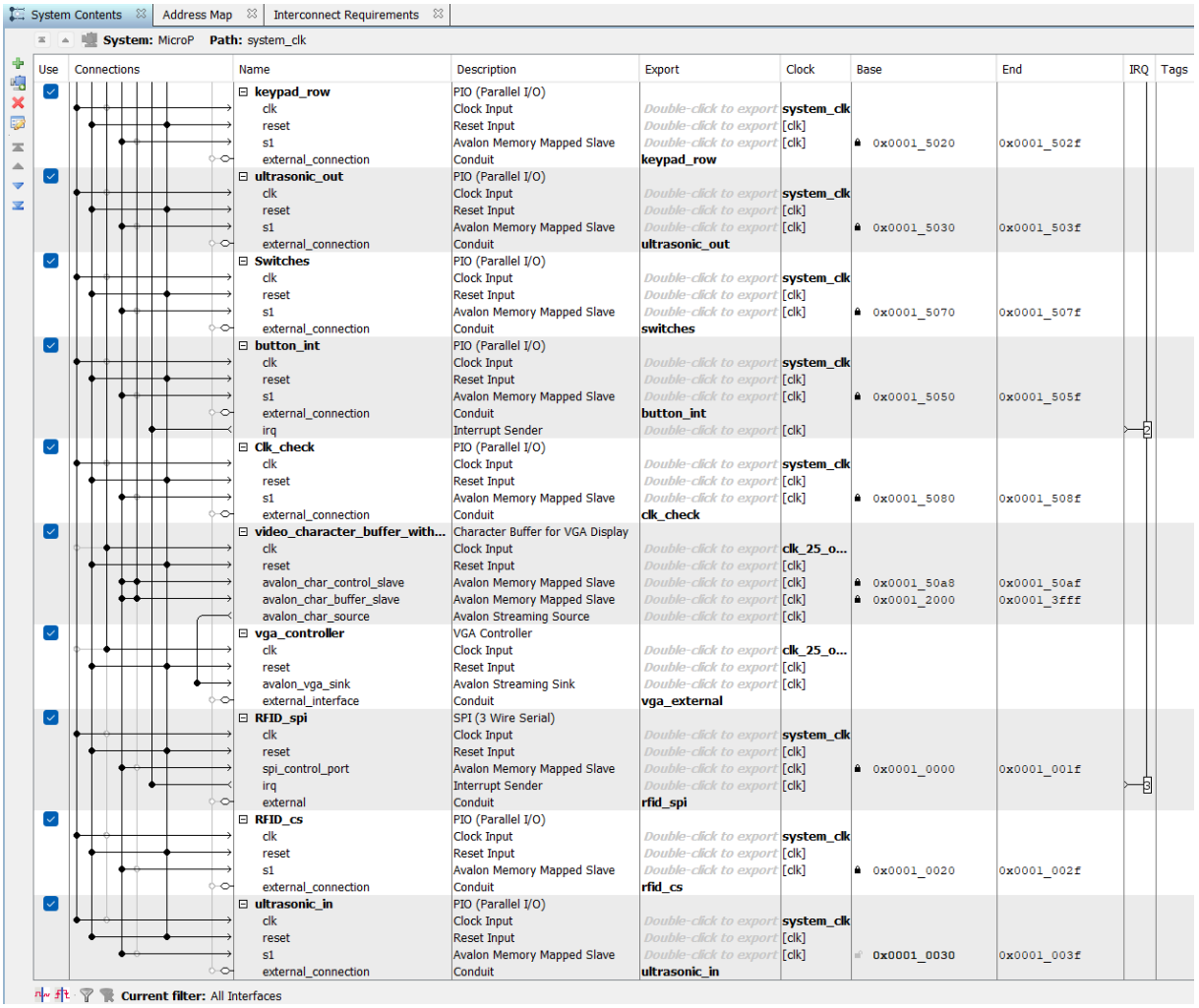


Figure 3.5.1.3: *Qsys Connection 2*

The base address is then assigned to the IP cores,the Qsys file is saved and generates the HDL design file with the HDL example used to link the Nios processor while in the Quartus software.

3.5.2 Quartus Setup

Quartus is set up after Qsys is done. A new project is opened and a new Verilog HDl file is added for Verilog coding. The project must link to the Qsys file that just set up by clicking the project option in the above option panel and choose add/remove file in project option to include the Qsys file.

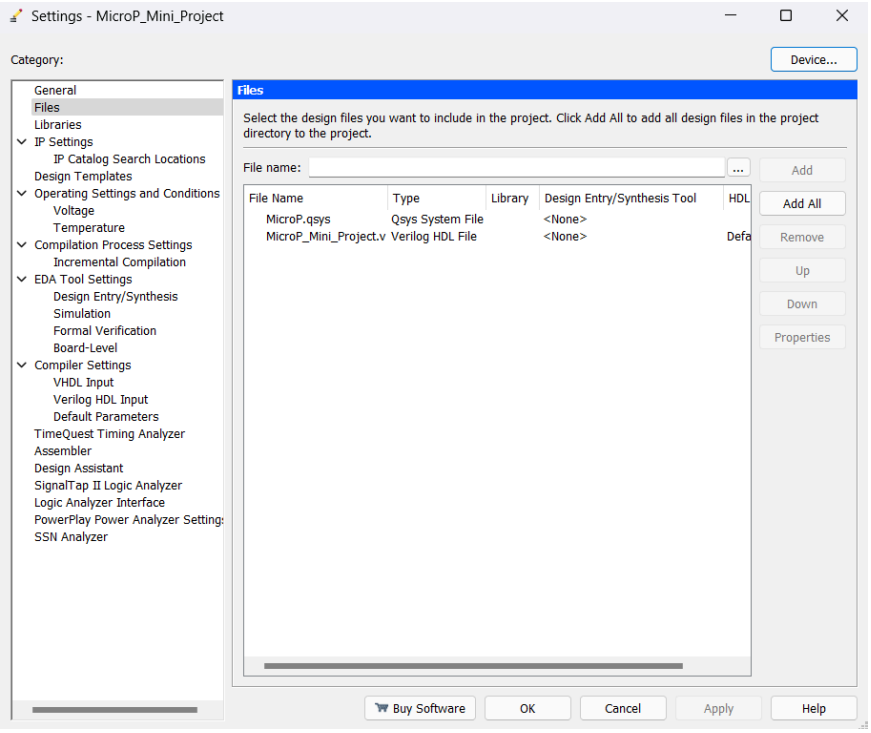


Figure 3.5.2.1: *Qsys file is added*

Make sure that the Qsys file is successfully added with a Verilog HDL file as the figure above. This setting links the Qsys system and the Verilog HDL file together in the same Quartus project.

```
module MicroP_Mini_Project(
input CLOCK_50,
input reset,
input [7:0]SW,
input [3:0]keypad_column,
input call_button,
output clk_25_check,
output [3:0]keypad_row,
output vga_CLK,
output vga_HS,
output vga_VS,
output vga_BLANK,
output vga_SYNC,
output [7:0]vga_R,
output [7:0]vga_G,
output [7:0]vga_B,
output pll_locked,
input fpga_rx,
output fpga_tx,
output buzzer,
input  ultrasonic_in,
output ultrasonic_out,
input  spi_master_MISO,
output spi_master_MOSI,
output spi_master_SCLK,
output spi_master_SS_n,
output spi_master_SS_n_1
);
```

Figure 3.5.2.2: *Verilog code 1*

```
MicroP u0 (
    .clk_clk                       (CLOCK_50),         //     clk.clk
    .reset_reset_n                 (reset),            //     reset.reset_n
    .switches_export               (SW),               //     switches.export
    .clk_check_export              (clk_25_check),     //
    .vga_external_CLK              (vga_CLK),          //        vga_external.CLK
    .vga_external_HS               (vga_HS),           //                    .HS
    .vga_external_VS               (vga_VS),           //                    .VS
    .vga_external_BLANK            (vga_BLANK),        //                    .BLANK
    .vga_external_SYNC             (vga_SYNC),         //                    .SYNC
    .vga_external_R                (vga_R),            //                    .R
    .vga_external_G                (vga_G),            //                    .G
    .vga_external_B                (vga_B),            //                    .B
    .clk_25_export                 (pll_locked),       //
    .bluetooth_uart_rxd            (fpga_rx),          //    bluetooth_uart.rxd
    .bluetooth_uart_txd            (fpga_tx),          //                  .txd
    .keypad_column_export          (keypad_column),    //  keypad_column.export
    .keypad_row_export             (keypad_row),       //     keypad_row.export
    .button_int_export             (call_button),      //     button_int.export
    .buzzer_export                 (buzzer),           //         buzzer.export
    .ultrasonic_in_export          (ultrasonic_in),    //  ultrasonic_in.export
    .ultrasonic_out_export         (ultrasonic_out),   // ultrasonic_out.export
    .rfid_spi_MISO                 (spi_master_MISO),        //       rfid_spi.MISO
    .rfid_spi_MOSI                 (spi_master_MOSI),        //                .MOSI
    .rfid_spi_SCLK                 (spi_master_SCLK),        //                .SCLK
    .rfid_spi_SS_n                 (spi_master_SS_n),        //                .SS_n
    .rfid_cs_export                (spi_master_SS_n_1)        //        rfid_cs.export
);

endmodule
```

Figure 3.5.2.3: *Verilog code 2*

Above Verilog code is written to configure the hardware connection that is outside the nios processor. Respective pins in the Qsys (Nios processor) are connected to their respective pins (Input/Output) for their functionality. The pins are then assigned by using the pin planner in the Quartus software as shown in Figure 3.4.3.

3.5.3 Nios II Eclipse

Nios II Eclipse, part of the Nios II Software Build Tools (SBT), is an IDE used to write C programming. Firstly, Nios II Eclipse is accessed through the tool option in the Quartus and choose Nios II Software Build Tools (SBT) for Eclipse. Thus, a new Nios Application and BSP from Template is created and linked to the .sopcinfo file generated by the Quartus software.

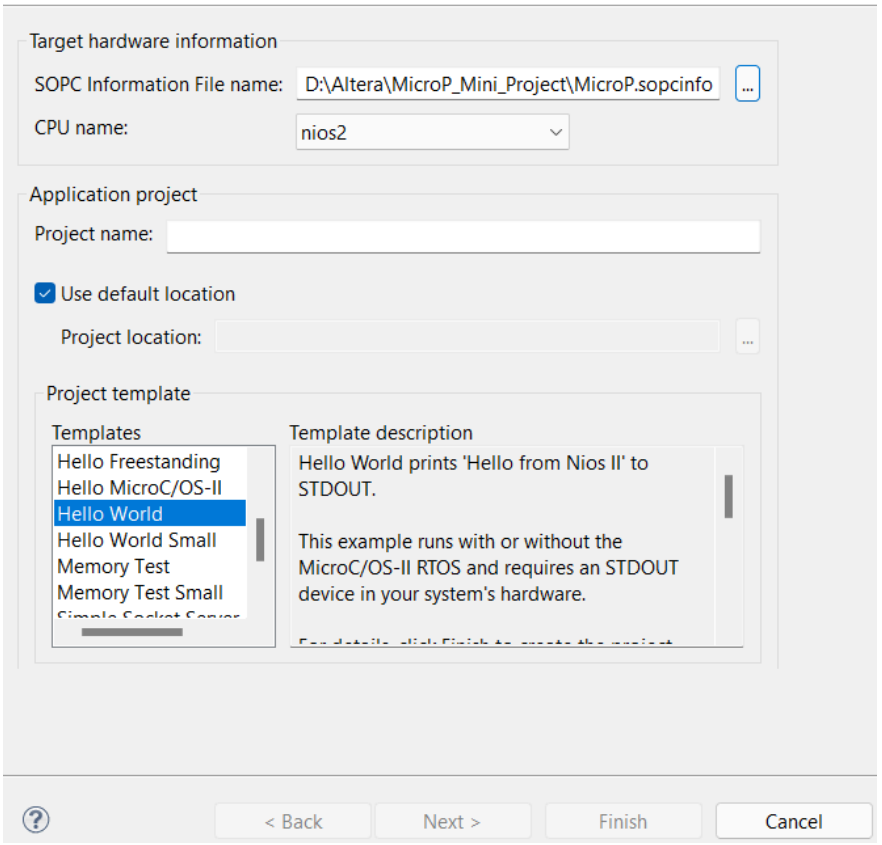


Figure 3.5.3.1: *Create New Application File in Eclipse*

Make sure the SOPC Information File name includes the correct .sopcinfo file.

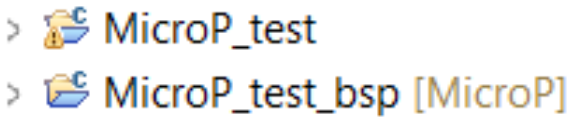


Figure 3.5.3.2: *Application and BSP files*

An application and BSP file will be generated in the project directory. Then, generate the project BSP (HAL library, system address etc) by clicking the BSP file and choose build project. After building the file, all the HAL libraries and a system.h file are generated.

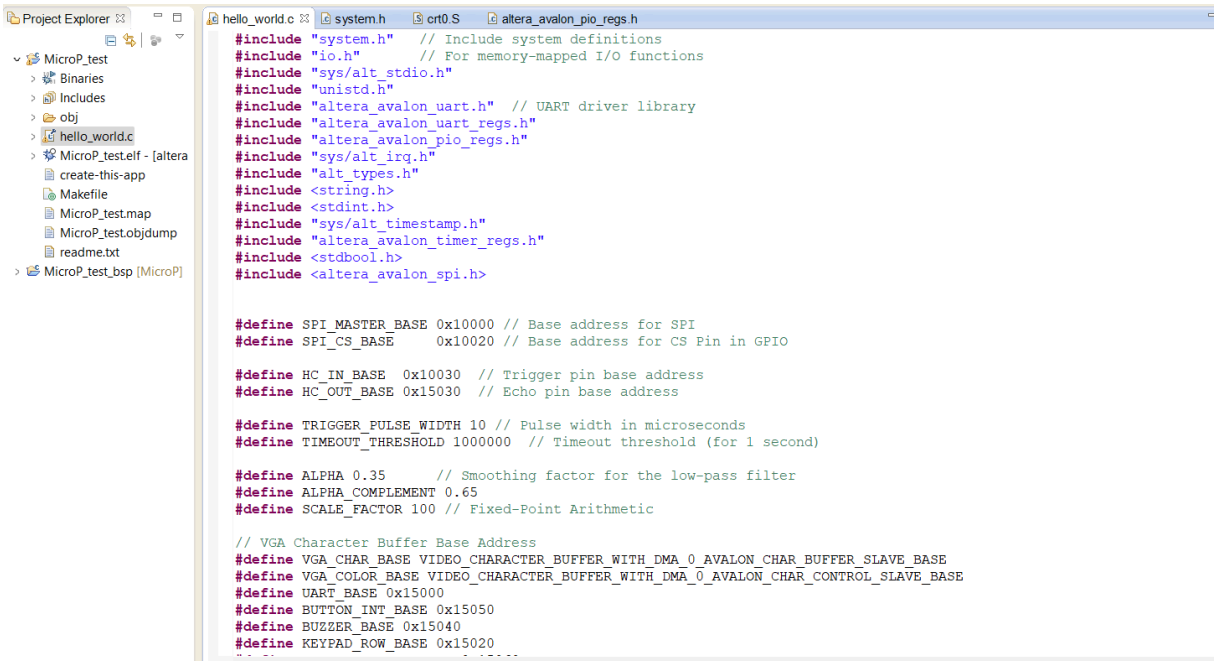


Figure 3.5.3.3: *C Programming in Eclipse*

C programming can be written under the application file, a .c file is used to include the C programming. After building the C programming with no errors, the code is uploaded into the FPGA board.

4.0 RESULT AND DISCUSSION

4.1 Program Flowchart

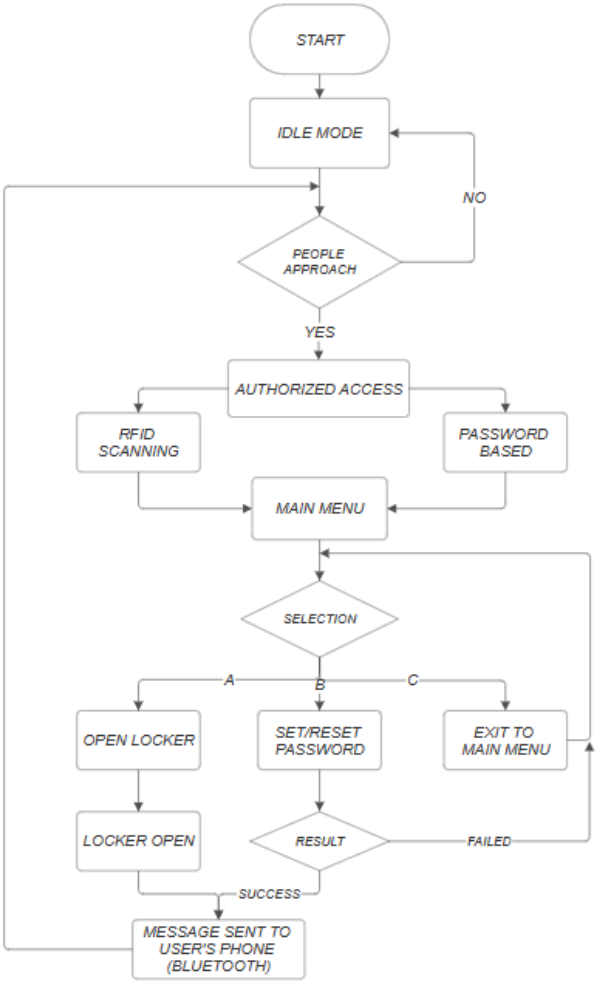


Figure 4.1.1: *Flow Chart of the Smart Locker System*

1. ***System Initialization and Idle Mode***

Initially, the program starts, and the system enters idle mode to conserve power. In this state, the monitor screen will remain blank until the ultrasonic sensor detects an approaching individual.

2. ***Detection and Activation***

When the ultrasonic sensor detects an individual approaching, the system activates and displays the ***Authorized Access Page***. The individual can choose between ***RFID Scanning*** or ***Password Entry*** to gain authorized access.

3. ***Authorization Process***

A. RFID Scanning

When the user places their RFID card on the module for scanning, the system compares the RFID tag's unique identifier (UID) with the stored records. Access is granted only if a match is found.

B. Password Entry

If the user presses the '*' key on the keypad, the system navigates to the ***Password-Based Verification Page***. The user is required to enter their password, which is then verified against the stored password in the database. Upon successful verification, access is granted.

4. Main Menu Navigation

Once access is granted, the system transitions to the ***Main Menu***. Here, the user can select further operations:

A. Open Locker

The locker opens, the green LED lights up to indicate the locker status, and a message, "Library Locker Opened" is sent to the user's phone through Bluetooth.

B. Set/Reset Password

If setting the password for the first time, the system will prompt the user to enter a password. For the password reset, the user must first correctly enter the previously used password. If verified, the user can input and confirm a new password. Upon success, the new password is displayed on the user's phone.

C. Return to Authorization Access

The system returns to the ***Authorization Access Page***, ready for the next operation.

4.2 Program Results

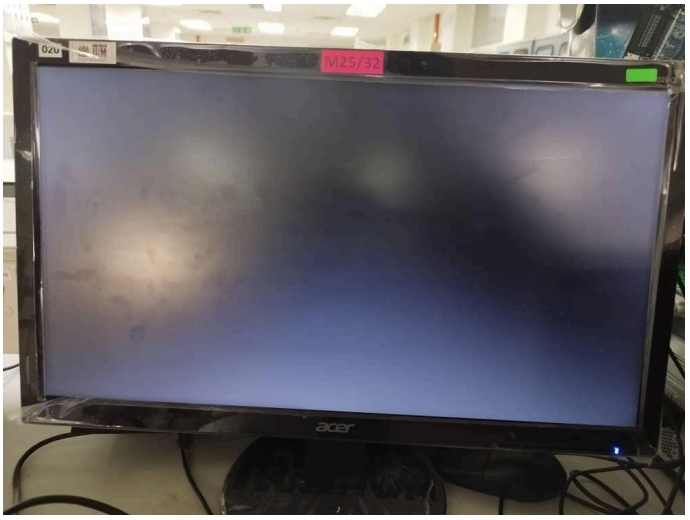


Figure 4.2.1: *Smart Locker System in Idle Mode*

Initially, when the program starts, and the ultrasonic sensor does not detect any approaching individuals, the monitor screen will remain blank to conserve power, as the system operates in idle mode.

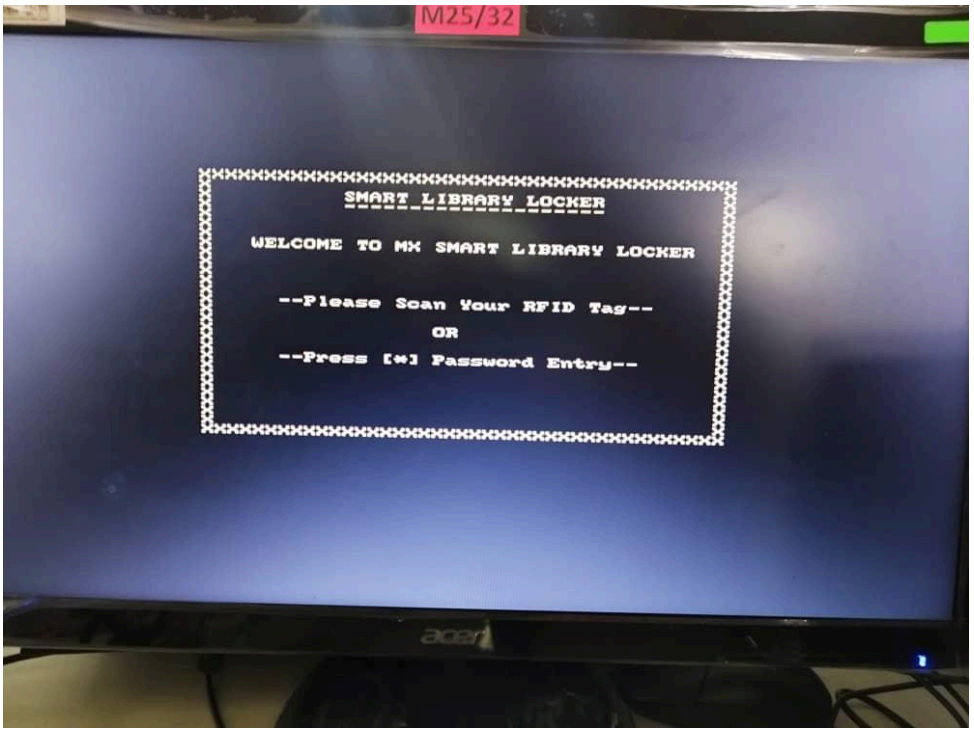


Figure 4.2.2: *Authorized Access*

When an individual approaches the screen, the ultrasonic sensor detects their presence and activates the system, displaying the authorized access page. The individual can proceed by using one of the available access methods (either ***RFID scanning*** or ***Password Entry***) to continue with further operations.

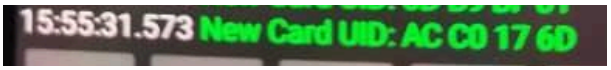


Figure 4.2.3: *RFID Card is Scanned*

If using RFID scanning, the system will compare the unique identifier (UID) of the RFID tag with the stored records and proceed with further operations only if a correct match is found. For password entry, the user is required to key in the previously set password, which will be verified against the password stored in the database. Upon a successful match, the user will be granted access to the main menu for further operations.

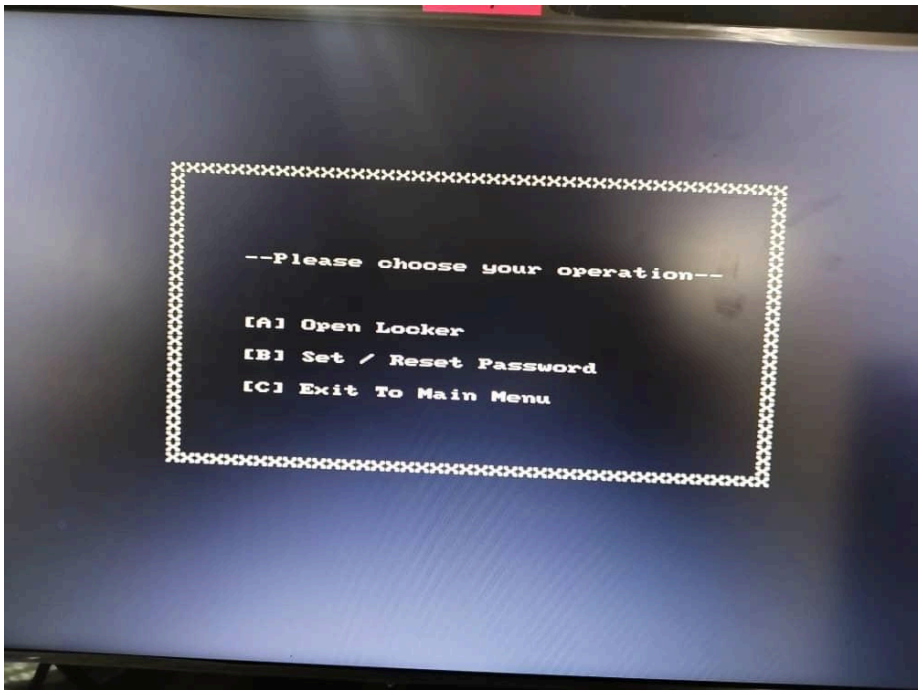


Figure 4.2.4: *Main Menu*

Once the user is granted permission, they will proceed to the main menu and can select further operations, as illustrated in Figure 4.2.3 above.

A. Open Locker

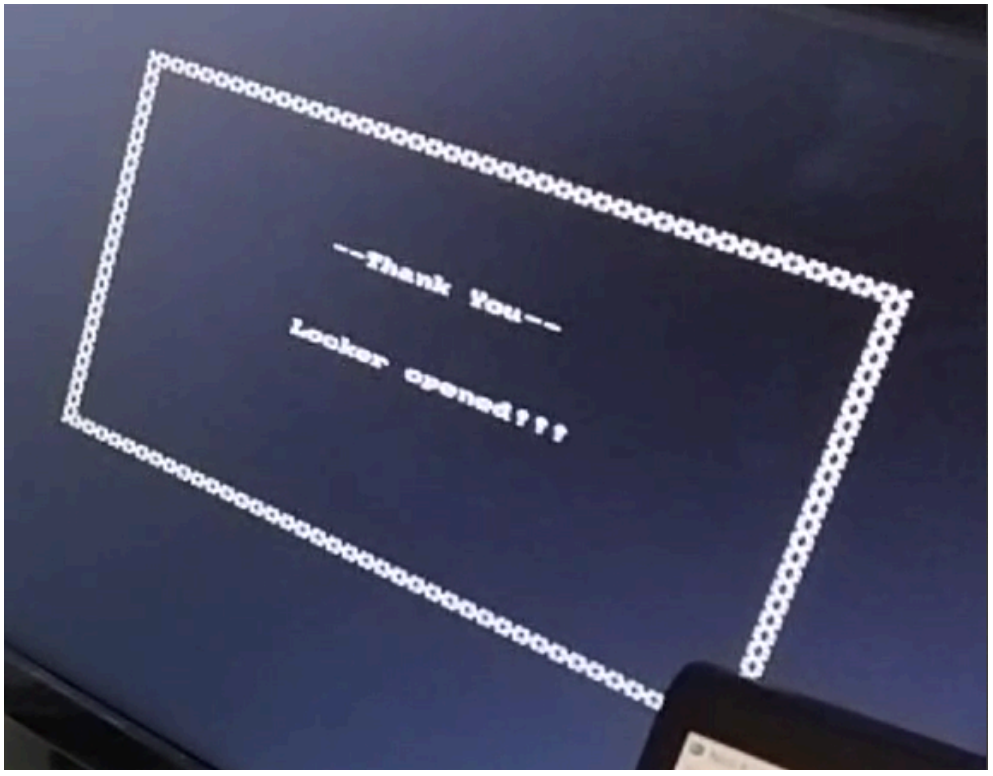
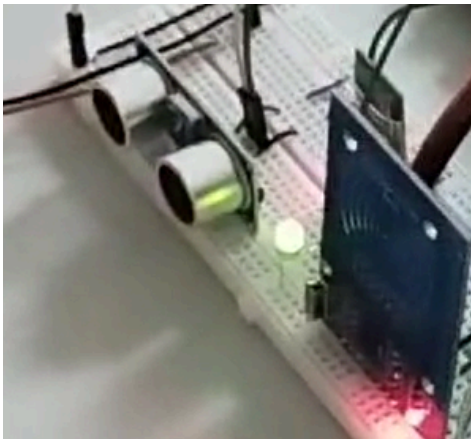


Figure 4.2.5: *Locker Opened*



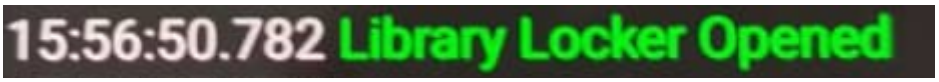


Figure 4.2.6 & Figure 4.2.7: *Green LED light up (TOP) ; Message Received in Phone (BOTTOM)*

When the user presses 'A' in the main menu, the locker will open, as shown in ***Figure 4.2.4***. At the same time, the green LED indicating the locker's status will light up, as depicted in ***Figure 4.2.5*** and a message "Library Locker Opened" will be sent to the user's phone via Bluetooth communication (***Figure 4.2.6***).

B. Set/Reset Password

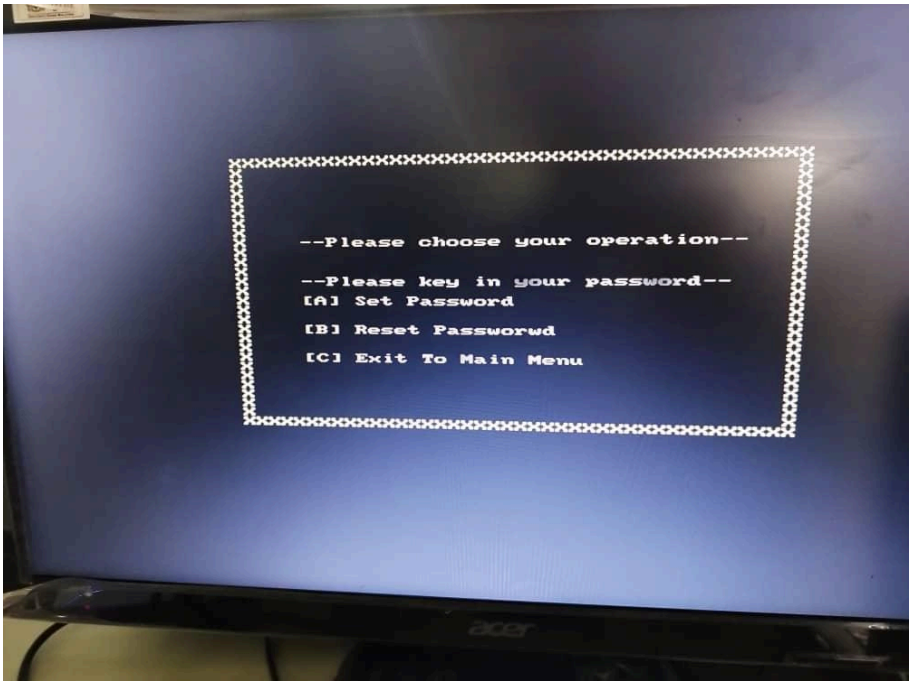


Figure 4.2.8: *Set / Reset Password Page*

If the user presses 'B' in the main menu, the user will come to the ***Set/Reset Password Page*** as shown above. Then, the user is able to set the password for the first time like the figure below.

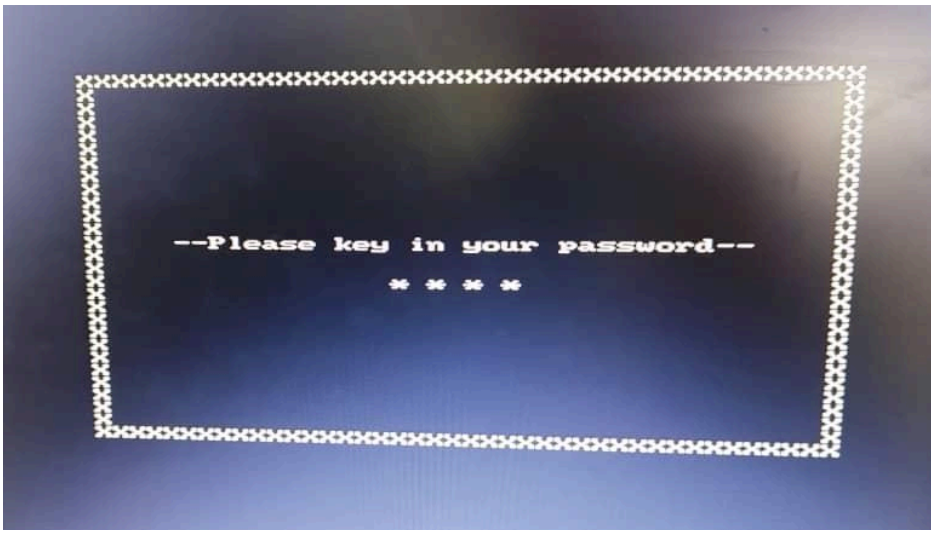


Figure 4.2.9: *Password Setting Page*

In another situation, if the user needs to reset the password, they must first correctly enter the previously used password before proceeding with the change (***Figure 4.2.9***). Only after the previous password is verified, the user will be allowed to set up a new password with

a double confirmation. Finally, upon successful password reset, the new password will be displayed on the user's phone via Bluetooth communication as shown below (***Figure 4.2.12***).

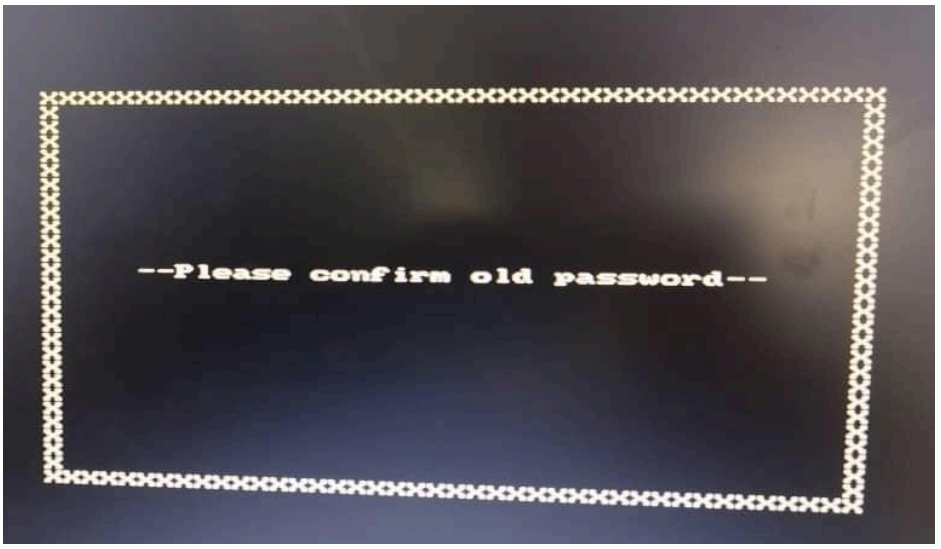


Figure 4.2.10: *Previous Password Verification*

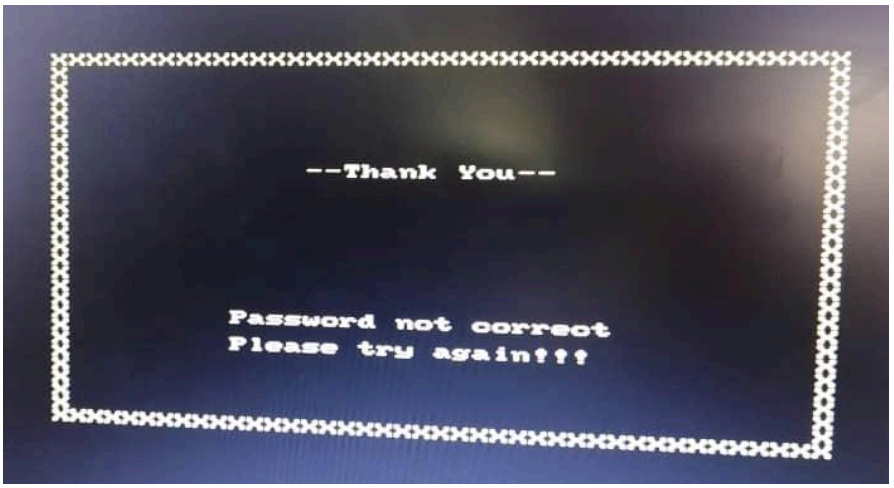


Figure 4.2.11*: Previous Password Verification Failed*

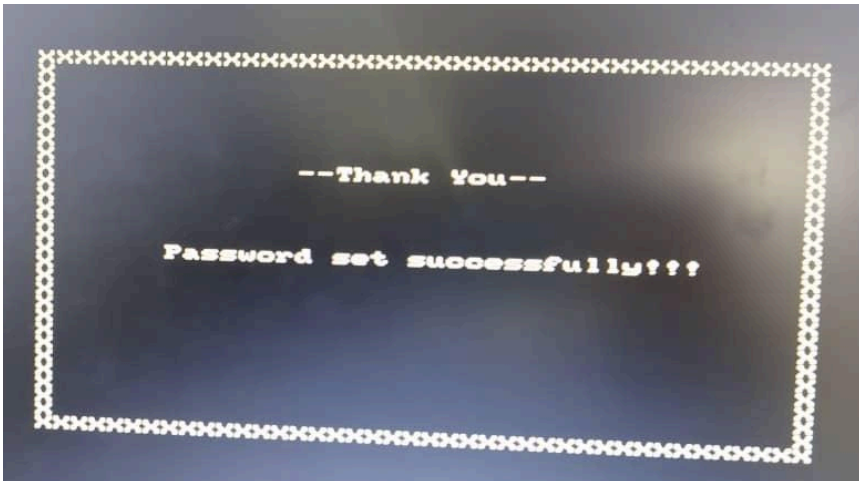


Figure 4.2.12*: Password Set Successfully*

***Figure 4.2.13**: Message sent to the user's phone*

C. Exit To Main Menu

***Figure 4.2.14**: Main Menu Page*

If the user selects 'C' in the main menu, the system will return to the authorization access page and await the next operation.

4.3 Module Functionality and Coding Implementation

4.3.1 RC522 - RFID Module

For the RC522 (RFID Module), SPI communication is the most commonly used interface for connecting the RFID module with microcontrollers and FPGA boards. Although the HAL library in the Eclipse IDE provides SPI functions for FPGA board implementation, there are no pre-existing libraries specifically designed for the RC522 module. Therefore, to establish a successful connection between the RFID module and the DE1-SoC board, the code logic must be manually implemented using direct register access to the RFID module through SPI communication interface.

```
// Function to clear register bits
void PCD_ClearRegisterBitMask(unsigned char reg, unsigned char mask) {
    unsigned char tmp = readRegister(reg);
    writeRegister(reg, tmp & (~mask));
}

// Function to set register bits
void PCD_SetRegisterBitMask(unsigned char reg, unsigned char mask) {
    unsigned char tmp = readRegister(reg);
    writeRegister(reg, tmp | mask);
}
```

`PCD_ClearRegisterBitMask(unsigned char reg, unsigned char mask)` function is used to clear (set to 0) specific bits in a given register of the RFID module. It first reads the current value of the register using ***`readRegister(reg)`***. Then, it computes a new value by performing a bitwise AND operation between the current register value and the complement of the provided mask ***(`~mask`)***. Hence, this function ensures that only the bits specified by the ***`mask`*** are cleared, while other bits remain unchanged. Finally, the updated value is written back to the register using ***`writeRegister`***.

`PCD_SetRegisterBitMask(unsigned char reg, unsigned char mask)` function is for setting specific bits in a given register. It starts by reading the current value of the register using ***`readRegister(reg)`*** same as the previous function. A bitwise OR operation is then

performed between the current register value and the ***`mask`***. This operation ensures that the bits specified by the ***`mask`*** are set to ***1***, while other bits remain unaffected. The resulting value is written back to the register using ***`writeRegister`***.

```
// Simplified initialization focusing on essential settings
void PCD_Init() {
    // Soft reset
    writeRegister(MFRC522_REG_COMMAND, PCD_RESETPHASE);
    usleep(50000);

    // Minimum required configuration
    writeRegister(MFRC522_REG_TXMODE, 0x00);    // Transmit data according to protocol
    writeRegister(MFRC522_REG_RXMODE, 0x00);    // Receive data according to protocol
    writeRegister(MFRC522_REG_MOD_WIDTH, 0x26); // Modulation width setting
    writeRegister(MFRC522_REG_TX_ASK, 0x40);    // Force 100% ASK modulation
    writeRegister(MFRC522_REG_MODE, 0x3D);      // CRC Initial value 0x6363

    // Enable antenna
    writeRegister(MFRC522_REG_TX_CONTROL, 0x83);
}
```

The ***`PDC_Init()`*** function initializes the RC522 RFID module by performing essential setup tasks. It begins with a soft reset by writing the ***`PDC_RESETPHASE`*** command to the ***`MFRC522_REG_COMMAND`*** register and introduces a 50ms delay to ensure the module resets properly. The function then configures the communication protocol by setting the ***`MFRC522_REG_TXMODE`*** and ***`MFRC522_REG_RXMODE`*** registers to ***`0x00`***. The modulation width is configured via ***`MFRC522_REG_MOD_WIDTH`***, and ***`MFRC522_REG_TX_ASK`*** is set to ***`0x40`*** to enforce ***100% Amplitude Shift Keying (ASK)*** modulation. The ***`MFRC522_REG_MODE`*** register is set to ***`0x3D`*** to initialize the CRC (Cyclic Redundancy Check, a type of error-detection technique) calculation with a standard 0x6363 value. Finally, the antenna is activated by setting the ***`MFRC522_REG_TX_CONTROL`*** register to 0x83, enabling the module to transmit and receive data. This minimal configuration ensures that the RC522 is ready for basic operation.

```
// Simplified card detection
int PICC_IsNewCardPresent() {
    // Clear registers
    writeRegister(MFRC522_REG_COMMAND, PCD_IDLE);
    writeRegister(MFRC522_REG_COMIRQ, 0x7F);
    writeRegister(MFRC522_REG_FIFO_LEVEL, 0x80);

    // Configure for REQA
    writeRegister(MFRC522_REG_BIT_FRAMING, 0x07);  // 7 bits

    // Send REQA
    unsigned char command = PICC_REQIDL;
    unsigned char irqEn = 0x77;
    unsigned char waitIRq = 0x30;

    writeRegister(MFRC522_REG_COMIEN, irqEn);
    writeRegister(MFRC522_REG_DIVIEN, 0x00);
    writeRegister(MFRC522_REG_COMIRQ, 0x7F);
    writeRegister(MFRC522_REG_DIVIRQ, 0x7F);

    // Write data to FIFO
    writeRegister(MFRC522_REG_FIFO_DATA, command);
```

```
    // Start transmission
    writeRegister(MFRC522_REG_COMMAND, PCD_TRANSCEIVE);
    writeRegister(MFRC522_REG_BIT_FRAMING, 0x87);  // Start transmission

    // Wait for completion
    unsigned long timeout = 2500;
    unsigned char irqReg;
    do {
        irqReg = readRegister(MFRC522_REG_COMIRQ);
        timeout--;
        usleep(1);
    } while ((timeout != 0) && !(irqReg & waitIRq));

    // Stop transmission
    writeRegister(MFRC522_REG_BIT_FRAMING, 0x00);

    if (timeout != 0) {
        unsigned char fifoLevel = readRegister(MFRC522_REG_FIFO_LEVEL);
        if (fifoLevel == 2) {
            return 1;
        }
    }

    return 0;
}
```

The ***`PICC_IsNewCardPresent()`*** function detects whether a new RFID card is present near the RC522 module. It achieves this by configuring the module for card detection, sending a command ***(`REQA`)***, and interpreting the module's response. The function provides a basic, low-level approach by directly accessing the RC522's hardware registers.

First of all, the function initializes the RC522 module by clearing communication registers, such as ***`MFRC522_REG_COMMAND`*** (*to set the module to idle mode*), ***`MFRC522_REG_COMIRQ`*** (to clear any previous interrupt flags), and ***`MFRC522_REG_FIFO_LEVEL`*** (to reset the FIFO buffer). It also configures the ***`MFRC522_REG_BIT_FRAMING`*** to set the communication frame to 7 bits, as required for the ***`REQA`*** command.

Then, the function sends the ***`REQA`*** command, which is specifically designed to detect cards in the IDLE state. The command is written to the FIFO buffer using the ***`MFRC522_REG_FIFO_DATA`*** register. Simultaneously, interrupt-related registers are configured to enable relevant communication interrupts. Once the configuration is complete, the module is instructed to start transmission by setting the ***`PCD_TRANSCEIVE`*** command in ***`MFRC522_REG_COMMAND`***.

After initiating the command, the function will enter a ***`while`*** loop to wait for a response from the card. It continuously checks the interrupt register (***`MFRC522_REG_COMIRQ`***) to detect when the desired response-related bits (***`waitIRq`***) are set. To prevent the function from being stuck indefinitely, a timeout mechanism ensures the loop exits if no response is received within the specified time frame.

Finally, if a response is detected within the specified time frame, the function reads the FIFO level register through ***(`MFRC522_REG_FIFO_LEVEL`)*** to confirm the expected 2-byte response from the ***`REQA`*** command. If the FIFO contains exactly 2 bytes, the function will conclude that a card is present and returns ***`1`***. If no valid response is received or the timeout is reached, it returns ***`0`***, indicating no card is detected.

```
int PICC_ReadCardSerial() {
    unsigned char command[2] = {PICC_ANTICOLL, 0x20};
    unsigned char result[5];  // 4 bytes UID + 1 byte BCC

    writeRegister(MFRC522_REG_BIT_FRAMING, 0x00);  // Clear bit frame adjustments
    writeRegister(MFRC522_REG_COLL, 0x00);         // All bits are valid

    // Send anticollision command
    writeRegister(MFRC522_REG_COMMAND, PCD_IDLE);
    writeRegister(MFRC522_REG_COMIRQ, 0x7F);
    writeRegister(MFRC522_REG_FIFO_LEVEL, 0x80);

    // Write command to FIFO
    writeRegister(MFRC522_REG_FIFO_DATA, command[0]);
    writeRegister(MFRC522_REG_FIFO_DATA, command[1]);

    // Start transmission
    writeRegister(MFRC522_REG_COMMAND, PCD_TRANSCEIVE);
    writeRegister(MFRC522_REG_BIT_FRAMING, 0x80);  // Start transmission

    // Wait for completion
    unsigned long timeout = 10000;
    unsigned char irqReg;
```

```
    do {
        irqReg = readRegister(MFRC522_REG_COMIRQ);
        timeout--;
        usleep(1);
    } while ((timeout != 0) && !(irqReg & 0x30));

    if (timeout != 0) {
        unsigned char fifoLevel = readRegister(MFRC522_REG_FIFO_LEVEL);
        if (fifoLevel == 5) {
            for (i = 0; i < 5; i++) {
                result[i] = readRegister(MFRC522_REG_FIFO_DATA);
            }

            // Calculate BCC
            unsigned char bcc = result[0] ^ result[1] ^ result[2] ^ result[3];

            if (bcc == result[4]) {
                uidLength = 4;
                for (i = 0; i < 4; i++) {
                    uid[i] = result[i];
                }
                return 1;
            }
        }
    }
```

The ***`PICC_ReadCardSerial`*** function is responsible for reading the unique identifier (UID) of an RFID card in proximity to the RC522 module. It uses the anti-collision mechanism to retrieve the UID, which ensures the proper identification of a card, even when multiple cards are present.

The function starts by initializing the RC522 for the anti-collision process. It configures the bit framing register ***(`MFRC522_REG_BIT_FRAMING`)*** to reset frame adjustments and sets the collision register ***(`MFRC522_REG_COLL`)*** to consider all bits valid for collision detection. The anti-collision command ***(`PICC_ANTICOLL`)*** and the desired response length ***(`0x20`)*** are prepared in an array and later written to the FIFO buffer. This setup ensures that the RC522 module is ready to initiate the anti-collision protocol.

The program is then configured to start communication by setting RC522 module to idle ***(`PCD_IDLE`)***, clearing interrupt flags ***(`MFRC522_REG_COMIRQ`)***, and resetting the FIFO buffer ***(`MFRC522_REG_FIFO_LEVEL`)***. The anti-collision command and response length are written to the FIFO buffer, and the transmission is started using the `PCD_TRANSCEIVE` command.

Once the transmission begins, the function enters a loop to wait for a response from the card. It continuously reads the interrupt register ***(`MFRC522_REG_COMIRQ`)*** to check for completion of the transmission. A timeout mechanism is implemented to ensure the function does not hang indefinitely if no response is received. If the response is completed within the timeout, the FIFO level register ***(`MFRC522_REG_FIFO_LEVEL`)*** is checked to verify that the expected 5 bytes of data (4-byte UID and 1-byte Block Check Character, BCC) have been received.

If the expected data is received, the function reads the 4 bytes from the FIFO buffer (from the RC522 module) and calculates the BCC by performing a bitwise XOR operation on the first 4 bytes of the UID. This BCC is then compared with the fifth byte to validate the integrity of the received UID. If the BCC matches, the UID is stored in the ***global `uid` array*** , and the function returns ***`1`***, indicating successful UID retrieval. If the validation fails or no response is received, the function returns ***`0`***, signaling an error or no card present.

```
void PICC_HaltA() {
   unsigned char command[2] = {PICC_HALT, 0x00};
   unsigned char result[1];
   unsigned char resultSize = sizeof(result);

   PCD_CommunicateWithPICC(PCD_TRANSCEIVE, 0x30, command, 2, result, &resultSize,
0);
}
```

The ***`PICC_HaltA`*** function sends a ***"Halt"*** command to a RFID card, instructing it to stop communication with the RFID reader as specified by the ***ISO/IEC 14443 standard***. It initializes the two-byte command array with the ***`PICC_HALT` instruction (`0x50`)*** and a ***mandatory (`0x00`) byte***.

The function uses ***`PCD_CommunicateWithPICC`*** to handle the low-level communication. This call transmits the ***"Halt"*** command to the card, specifying the command, its length, and placeholders for the result and result size. By executing this function, the communication session with the current card is terminated, freeing the RFID reader to handle other tasks or cards.

4.3.2 HC06 - Bluetooth Module

By implementing the inbuilt UART library provided by the HAL Library in Eclipse, connecting the Bluetooth module becomes straightforward. To establish communication, the TX pin of the Bluetooth module is connected to the RX pin of the FPGA board, while the RX pin of the Bluetooth module is connected to the TX pin of the FPGA board. With this setup, the functions provided below can easily send messages to a phone via Bluetooth communication.

$TX\ (Bluetooth\ Module)\ \Rightarrow\ RX\ (FPGA\ Board)$
$RX\ (Bluetooth\ Module)\ \Rightarrow\ TX\ (FPGA\ Board)$

```
// Function to send a single character over UART
void uart_send_char(char c) {
   while (!(IORD_ALTERA_AVALON_UART_STATUS(UART_BASE) &
ALTERA_AVALON_UART_STATUS_TRDY_MSK));  // Wait until the transmitter is ready
   IOWR_ALTERA_AVALON_UART_TXDATA(UART_BASE, c);  // Send the character
}
```

The function ***`uart_send_char(char c)`*** is responsible for transmitting a single character over UART. It begins by checking the status of the UART transmitter using the ***`IORD_ALTERA_AVALON_UART_STATUS`*** function. The `while` loop is for ensuring the function waits until the transmitter is ready to send data by checking the ***`ALTERA_AVALON_UART_STATUS_TRDY_MSK`*** bit. Once the transmitter is ready (status register is set), the character as function parameter is written to the UART transmit data register using the ***`IOWR_ALTERA_AVALON_UART_TXDATA`*** function, which initiates the hardware transmission of the character.

```
// Function to send a string over UART
void uart_send_string(const char* str) {
    while (*str) {
        uart_send_char(*str);
        str++;
    }
}
```

The function ***`uart_send_string(const char *str)`*** sends a string of characters over UART by iterating through each character in the string as string is an array. It uses a ***`while`*** loop that continues until the null terminator ***`\0`*** at the end of the string is encountered. For each character in the string, the function calls ***`uart_send_char(*str)`*** to send it over UART. The pointer ***`str`*** is then incremented to move to the next character in the string, ensuring that each character is sent sequentially until the entire string is transmitted.

4.3.3 VGA

The VGA interface in the project allows for displaying characters on a VGA screen with customizable colors. This functionality is achieved through specific functions for controlling the VGA's behavior by using the HAL libraries that are pre-built in the IP cores.

```
void set_vga_colors(char foreground, char background) {
    volatile char *vga_color_control = (volatile char *) VGA_COLOR_BASE;
    *vga_color_control = (foreground << 4) | background;
}
```

The function ***`set_vga_colors(char foreground, char background)`*** sets the foreground and background colors for the VGA display. In the above case, the foreground colour is set as white and the background colour is set as black. It does so by accessing the VGA color control register through its base address **VGA_COLOR_BASE**. The function takes two parameters: the desired foreground and background color codes. These values are combined into a single byte, where the foreground occupies the higher nibble and the background occupies the lower nibble, and is written to the control register.

```
void write_char_to_vga(int x, int y, char c) {
   // Calculate the position in the character buffer
        volatile char *vga_char_buf = (volatile char *) VGA_CHAR_BASE;
        int index = (y * 128)+ x;
            // Ensure the index is within the buffer range
        vga_char_buf[index] = c;
}
```

The function ***`write_char_to_vga(int x, int y, char c)`*** places a single character **c** at a specific (x, y) coordinate on the VGA display. It calculates the index in the VGA character buffer using the width of the screen (128 columns) and writes the character to the calculated position. The VGA buffer is accessed using the base address **VGA_CHAR_BASE**.

```
void clear_vga_fullscreen() {
   volatile char *vga_char_buf = (volatile char *) VGA_CHAR_BASE;
   for (y = 0; y < 60; y++) {
      for (x = 0; x < 128; x++) {
         vga_char_buf[y * 128 + x] = ' ';  // Fill with spaces
      }
   }
}
```

The function ***`clear_vga_fullscreen()`*** clears the entire VGA display by iterating through all positions on the screen, represented by 60 rows and 128 columns, and writing a space character (' ') to each position. This ensures that the screen is fully cleared for a fresh display.

```
for (x = 20; x < 61; x++) {
                    write_char_to_vga(x, 13, 'X');
          }

          for (x = 20; x < 61; x++) {
                    write_char_to_vga(x, 40, 'X');
          }

          for (y = 14; y < 40; y++) {
                    write_char_to_vga(20, y, 'X');
          }

          for (y = 14; y < 40; y++) {
                    write_char_to_vga(60, y, 'X');
          }
  a = 15;
  // Write simple text to the VGA screen
  write_char_to_vga(31, a, 'S');
  write_char_to_vga(32, a, 'M');
  write_char_to_vga(33, a, 'A');
  write_char_to_vga(34, a, 'R');
  write_char_to_vga(35, a, 'T');

  write_char_to_vga(37, a, 'L');
  write_char_to_vga(38, a, 'I');
  write_char_to_vga(39, a, 'B');
  write_char_to_vga(40, a, 'R');
  write_char_to_vga(41, a, 'A');
  write_char_to_vga(42, a, 'R');
  write_char_to_vga(43, a, 'Y');

  write_char_to_vga(45, a, 'L');
  write_char_to_vga(46, a, 'O');
  write_char_to_vga(47, a, 'C');
  write_char_to_vga(48, a, 'K');
  write_char_to_vga(49, a, 'E');
  write_char_to_vga(50, a, 'R');

  a= 16;
  write_char_to_vga(31, a, '-');
  write_char_to_vga(32, a, '-');
  write_char_to_vga(33, a, '-');
  write_char_to_vga(34, a, '-');
  write_char_to_vga(35, a, '-');
  write_char_to_vga(36, a, '-');
```

The function ***`enable_frame()`*** creates a bordered frame and displays a title on the VGA screen. It utilizes a combination of loops and specific character placements to achieve a visually appealing interface. The title "SMART LIBRARY LOCKER" is displayed within the rectangular border using multiple calls to ***`write_char_to_vga(x, y, c)`***. The characters are positioned starting at row 15 (**a = 15**) and centered horizontally by specifying the x-coordinates for each character. Above is the example by using the VGA IP cores to display characters on the monitor. Other VGA functions as shown in figure below used the same method to display different screens on the monitor.

```
//VGA
void set_vga_colors(char foreground, char background) {
void write_char_to_vga(int x, int y, char c) {
void clear_vga_fullscreen() {
void clear_vga_screen() {
void enable_frame() {
void welcoming_screen() {
void rfid_id_screen(char id[]) {
void rfid_not_match_screen(){
void keyin_pass_screen(int x){
void open_pass_screen() {
void locker_open_screen(){
void set_reset_screen(){
void pass_set_screen(){
void oldpass_screen(int x){
void keyin_newpass_screen(int x){
void pass_fail_screen(){
```

Figure 4.3.2.1: *Set and Clear RFID Register Bits*

4.3.4 Interrupt

```
void key_irq_init(){
	IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_INT_BASE, 0x1);
	alt_ic_isr_register(BUTTON_INT_IRQ_INTERRUPT_CONTROLLER_ID,
BUTTON_INT_IRQ, key_irq, 0, 0);
}
```

The `***key_irq_init()***` function initializes the interrupt mechanism for the button. It enables the interrupt to be activated for the button being pressed by setting the interrupt mask using `***IOWR_ALTERA_AVALON_PIO_IRQ_MASK()***` and registers the `***key_irq()***` ISR using `***alt_ic_isr_register()***`. This setup ensures that when a certain button is pressed, an interrupt will happen during the program execution and the `***key_irq()***` function will be activated.

```
void key_irq() {
   alt_printf("Interrupt Triggered!\n");
   IOWR_ALTERA_AVALON_PIO_DATA(BUZZER_BASE, 0xFF);
   IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_INT_BASE, 0x0); // Clear
interrupt flag
   if (status){
            IOWR_ALTERA_AVALON_PIO_DATA(BUZZER_BASE, 0x00);
            status = 0;
   }
   else
       status++;
}
```

The ***`key_irq()`*** is an ***interrupt service routine (ISR)*** which will be run only when a button press generates an interrupt. When the interrupt occurs, it outputs a message ("Interrupt Triggered"!) and activates a buzzer by writing *`0xFF`* to the buzzer's base address. It then clears the interrupt flag using ***`IOWR_ALTERA_AVALON_PIO_EDGE_CAP()`*** to allow future interrupts to be triggered. The function toggles the buzzer state based on the value of the ***`status`*** variable so that the technical assistance needs to manually turn off the buzzer by pressing again the button. If ***`status`*** is non-zero, the buzzer is turned off by writing *`0x00`* to the buzzer's base, and ***`status`*** is reset to 0. Otherwise, ***`status`*** is incremented to indicate a change.

4.3.5 Keypad

```
// Keymap for 4X4 Keypad
const char keyMap[4][4] = {
                {'1','2','3','A'},
                {'4','5','6','B'},
                {'7','8','9','C'},
                {'*','0','#','D'}
};


char scanKeypad(void){
        while (1) {  // Continuously scan until a key is detected
              for (row = 0; row < 4; row++) {
                 // Set the current row to LOW (active)
                 IOWR_ALTERA_AVALON_PIO_DATA(KEYPAD_ROW_BASE, ~(1 << row));

                 // Read the column states
                 alt_u32 colState = IORD_ALTERA_AVALON_PIO_DATA(KEYPAD_COLUMN_BAS

                 for (col = 0; col < 4; col++) {
                    // Check if the current column is LOW (key pressed)
                    if ((colState & (1 << col)) == 0) {
                       usleep(50000); // Debounce delay

                       // Wait for the key to be released
                       while ((IORD_ALTERA_AVALON_PIO_DATA(KEYPAD_COLUMN_BASE) &
(1 << col)) == 0);

                       // Reset the rows to default (all HIGH)
                       IOWR_ALTERA_AVALON_PIO_DATA(KEYPAD_ROW_BASE, 0xF);

                       return keyMap[row][col];
                    }
                 }
              }
        }
      return '\0';
}
```

The keypad scanning function continuously scans the keypad row by row by setting the current row to ***LOW*** while keeping all other rows ***HIGH***. If a key in the active row is pressed, the corresponding column will register as ***LOW***, indicating a key press. The inner *`for`* loop verifies this condition by checking the column states. To handle key bouncing, a 50ms delay is introduced. Additionally, a *`while`* loop ensures the function waits until the key is released before proceeding. Once the key press is processed, all rows are reset to HIGH

before returning the detected key based on the character set in the ***2D array 'keyMap`***. If no key is pressed, the function continues scanning indefinitely, returning the value ***`\0` (a null character) as a placeholder.***

4.3.6 Ultrasonic Sensor (HC-SR04)

```
// Function to send a trigger pulse
void send_trigger_pulse() {
    IOWR(HC_OUT_BASE, 0, 1);            // Set Trigger HIGH
    usleep(TRIGGER_PULSE_WIDTH);        // Wait for the pulse width
    IOWR(HC_OUT_BASE, 0, 0);            // Set Trigger LOW
}
```

The function ***`send_trigger_pulse()`*** is used to emit the ultrasonic wave through giving ***HIGH*** to the ***`Trigger`*** pin of the ultrasonic sensor with few milliseconds delay and following with ***LOW*** level to the ***`Trigger`*** pin to close the emission of ultrasonic wave.

```
// Function to measure echo pulse width using usleep
alt_u32 measure_echo_pulse() {
    long timeout = TIMEOUT_THRESHOLD;  // Timeout threshold to prevent infinite loop

    // Wait for Echo pin to go HIGH (start of pulse)
    while (IORD(HC_IN_BASE, 0) == 0 && timeout--) {
        usleep(1);  // Check the Echo pin every microsecond
    }

    if (timeout <= 0) {
        alt_printf("Timeout: Echo pin did not go HIGH.\n");
        return 0;  // Timeout occurred
    }
```

The ***`measure_echo_pulse()`*** function measures the duration of the echo pulse generated by an ultrasonic sensor through the input pin `Echo`. This pulse duration corresponds to the time it takes for the sound wave to travel to an object and back, which can then be used to calculate the distance.

$$Distance = \frac{Pulse\ Duration \times Speed\ of\ Sound}{2}$$

5.0 Conclusion

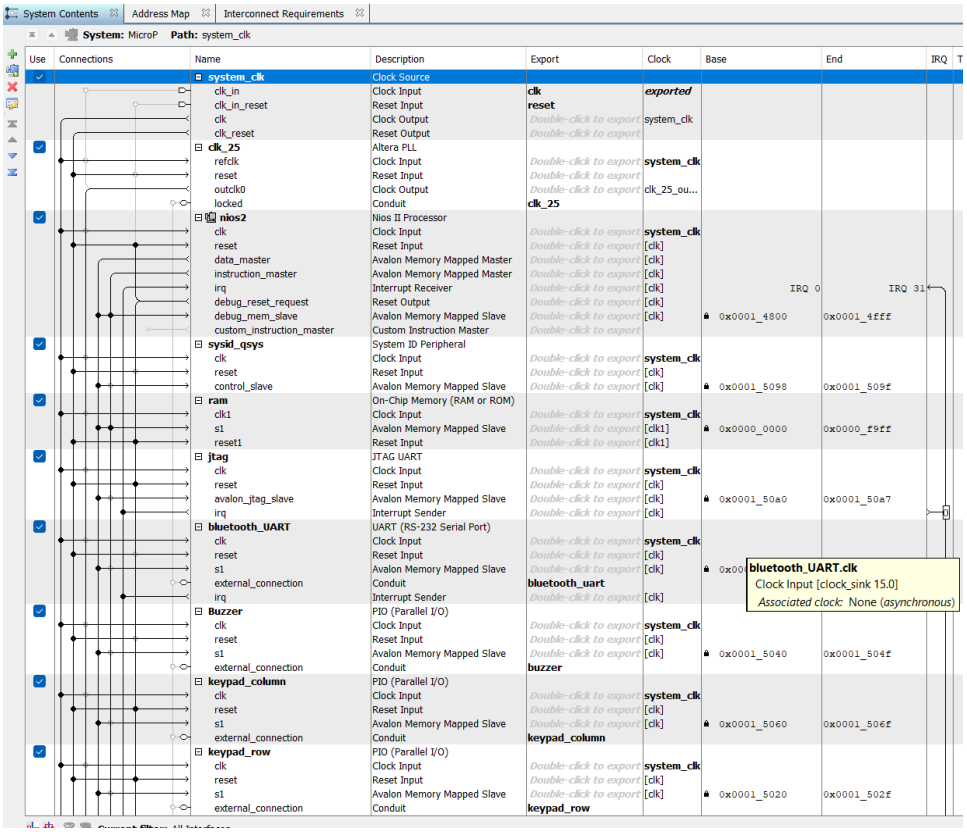
The Smart Library Locker successfully integrates RFID and password authentication with a monitoring system to provide a secure and efficient solution for accessing. The students access the library locker by just one tap with the RFID scanner using their student's card to achieve a more convenient accessibility. For security, the system sent and recorded the locker usage history to the student's phone through wireless communication. By utilising the password system, the system prevents the accessibility for the students who lose their card accidentally and acts as a backup method to access to the library locker. However, the project also contains rooms for future improvement such as the system can be expanded to support multi-locker configurations and develop a mobile app for notifications and history tracking. Through this project, we learned the importance of system-level design, troubleshooting hardware-software integration, and optimizing resource utilization on an FPGA platform.

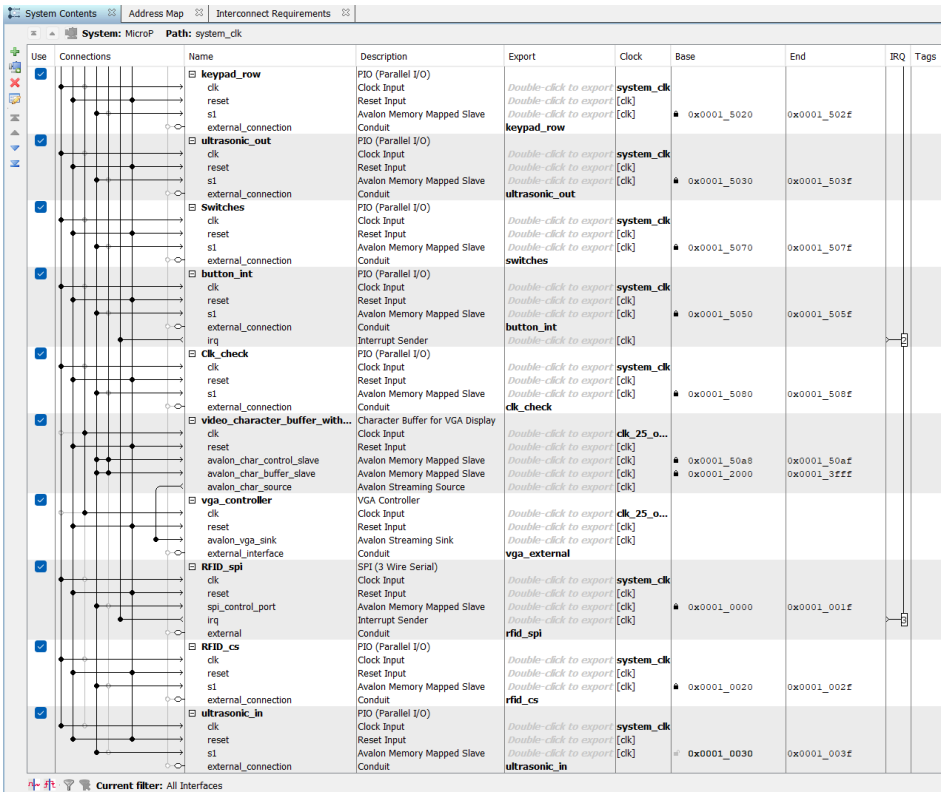
6.0 References

1. Oderuth, B. R., Ramkissoon, K., & Sungkur, R. K. (2019, September). Smart campus library system. In 2019 Conference on Next Generation Computing Applications (NextComp) (pp. 1-6). IEEE.
2. Rahayu, Y., Afif, L., & Soh, P. J. (2022). Design and development of smart lock system based QR-Code for library's locker at Faculty of Engineering, Universitas Riau.
3. Syed, Z., & Shaik, M. (2012). Fpga implementation of vga controller. In International Conference on Electronics and Communication Engineering (pp. 46-51).
4. Dakua, B. R., Hossain, M. I., & Ahmed, F. (2015). Design and implementation of UART serial communication module based on FPGA. Design and Implementation of UART Serial Communication Module Based on FPGA.
5. Oudjida, A. K., Berrandjia, M. L., Tiar, R., Liacha, A., & Tahraoui, K. (2009, December). Fpga implementation of i 2 c & spi protocols: A comparative study. In 2009 16th IEEE International Conference on Electronics, Circuits and Systems-(ICECS 2009) (pp. 507-510). IEEE.
6. Gaikwad, O., Prajwal, S., Manas, K., Mahesh, K., & Lalit, K. (2020). RFID Attendence using RC522. Int. J. Res. Appl. Sci. Eng. Technol, 8(5).

7.0 APPENDIX

QSYS Setup:





Verilog Code in Quartus Prime:

```
module MicroP_Mini_Project(
input CLOCK_50,
input reset,
input [7:0]SW,
input [3:0]keypad_column,
input call_button,
output clk_25_check,
output [3:0]keypad_row,
output vga_CLK,
output vga_HS,
output vga_VS,
output vga_BLANK,
output vga_SYNC,
output [7:0]vga_R,
output [7:0]vga_G,
output [7:0]vga_B,
output pll_locked,
input fpga_rx,
output fpga_tx,
output buzzer,
input  ultrasonic_in,
output ultrasonic_out,
input  spi_master_MISO,
output spi_master_MOSI,
output spi_master_SCLK,
output spi_master_SS_n,
output spi_master_SS_n_1
);
```

```
MicroP u0 (
    .clk_clk                (CLOCK_50),             //     clk.clk
    .reset_reset_n          (reset),                //     reset.reset_n
    .switches_export        (SW),                   //     switches.export
    .clk_check_export       (clk_25_check),         //
    .vga_external_CLK       (vga_CLK),              //        vga_external.CLK
    .vga_external_HS        (vga_HS),               //                    .HS
    .vga_external_VS        (vga_VS),               //                    .VS
    .vga_external_BLANK     (vga_BLANK),            //                    .BLANK
    .vga_external_SYNC      (vga_SYNC),             //                    .SYNC
    .vga_external_R         (vga_R),                //                    .R
    .vga_external_G         (vga_G),                //                    .G
    .vga_external_B         (vga_B),                //                    .B
    .clk_25_export          (pll_locked),           //
    .bluetooth_uart_rxd     (fpga_rx),              //    bluetooth_uart.rxd
    .bluetooth_uart_txd     (fpga_tx),              //                  .txd
    .keypad_column_export   (keypad_column),        //  keypad_column.export
    .keypad_row_export      (keypad_row),           //     keypad_row.export
    .button_int_export      (call_button),          //     button_int.export
    .buzzer_export          (buzzer),               //         buzzer.export
    .ultrasonic_in_export   (ultrasonic_in),        //  ultrasonic_in.export
    .ultrasonic_out_export  (ultrasonic_out),       // ultrasonic_out.export
    .rfid_spi_MISO          (spi_master_MISO),      //       rfid_spi.MISO
    .rfid_spi_MOSI          (spi_master_MOSI),      //                .MOSI
    .rfid_spi_SCLK          (spi_master_SCLK),      //                .SCLK
    .rfid_spi_SS_n          (spi_master_SS_n),      //                .SS_n
    .rfid_cs_export         (spi_master_SS_n_1)     //        rfid_cs.export
);

endmodule
```