Journal of Science and Engineering Research
Vol. 4, No. 1, 2025, pp.1-20
eISSN: 2957-7187

MicroBlaze Processor Based Embedded System II: Implementation of Bidirectional Communication and Control between FPGA Board and GUI

Shensheng Tang[1*] and Yi Xie[2]

[1] Department of Physics & Engineering, Bethel University, St. Paul, MN 55112, USA
[2] School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, 510006, China
[1]tang@bethel.edu
**Corresponding author's email: tang@bethel.edu*

DOI: http://dx.doi.org/10.56828/jser.2025.4.2.1
Article Info: Received: (October 17, 2025); Review Result: (November 20, 2025) Accepted: (December 21, 2025)

Abstract: In this paper, a MicroBlaze soft-core processor-based embedded system and a graphical user interface (GUI) are developed using VIVADO Design Suite and Visual Studio to implement different LED pattern control modes through Bidirectional Communication between the NEXYS A7 FPGA board and the GUI panel. The proposed system uses a UART controller for supporting Bidirectional Communication and two GPIO controllers for controlling the LED pattern modes, one connects to the onboard sixteen LEDs and the other connects the five onboard push buttons. All the controllers are connected to the MicroBlaze processor and collaborated with the developed SDK application program and C# GUI program, allowing the LED patterns to be controlled synchronously on the physical FPGA board and the GUI panel through either the physical onboard push buttons or the virtual buttons on the GUI. Experimental results are demonstrated with discussion. The MicroBlaze processor-based embedded system design and development method can be extended to a variety of applications from industrial automation to consumer electronics.

Keywords: MicroBlaze Processor, UART Serial Communication, C#, GUI, FPGA, VIVADO, SDK

1. Introduction

Embedded systems have become integral to modern technology, enabling intelligent control and interaction across a wide range of applications. Among the various platforms available, Field-Programmable Gate Array (FPGA) based systems offer a unique combination of flexibility, parallelism, and performance [1]. The MicroBlaze soft-core processor [2], developed by AMD/Xilinx, provides a customizable embedded processing solution within FPGA architectures, making it ideal for implementing complex control logic and communication protocols.

FPGA based platforms have emerged for implementing embedded systems across a wide range of applications, from signal processing and robotics to industrial automation and consumer electronics. The integration of processors within FPGA architectures—either as

hard-core processors like ARM Cortex-A series in Zynq devices [3] or soft-core processors such as Xilinx's MicroBlaze—has significantly expanded the design possibilities. Hard-core processors offer high performance and deterministic behaviour, making them suitable for compute-intensive tasks, while soft-core processors provide flexibility and customization, ideal for control-oriented applications. Systems utilizing soft-core processors like MicroBlaze have shown promise in educational environments and prototyping scenarios due to their reconfigurability and integration with development tools such as VIVADO [4] and SDK (Software Development Kit) [5]. Numerous studies have demonstrated the effectiveness of FPGA-based systems in real-time image processing [6][7], motor control [8], network packet inspection [9][10], and IoT device management [11][12].

In [6], a hardware-software co-designed FPGA-based multiprocessor system was developed for real-time image processing, utilizing five MicroBlaze soft-core processors in a master-slave configuration. Each processor is assigned a specific task and operates in parallel, enabling high-speed computation suitable for real-time image processing applications. In [7], an FPGA architecture was developed to extract and match image keypoints using a Scale-Invariant Feature Transform (SIFT) based algorithm and Euclidean distance, with the entire processing pipeline implemented within the FPGA fabric, eliminating the need for external computational hardware. In [8], an FPGA-based motor control system was proposed for industrial automation with the objective of overcoming the constraints of traditional digital signal processors and microcontrollers. Using a Proportional-Integral-Derivative (PID) algorithm and high-resolution high-resolution Pulse Width Modulation (PWM), the system controls multiple actuators in real time with low latency and high accuracy. In [9], an FPGA-based network intrusion detection system was proposed to use the Shift-And algorithm and a custom rule-matching module for detecting malicious network packets. The implemented system on a Xilinx Zynq-7030 FPGA demonstrates feasibility for real-time, resource-efficient detection. In [10], a novel FPGA-based architecture was presented for deep packet inspection using regular expression (RE) matching, crucial for network intrusion detection systems. To handle traffic beyond 100 Gbps, the design leverages approximate nondeterministic finite automata to reduce the required FPGA resource usage. In [11], an FPGA-based edge device was proposed to support IoT (Internet of Things) connectivity across heterogeneous devices by offloading key communication stack functions to hardware via SoC (System on Chip) technology, improving performance and interoperability. In [12], an FPGA-based IoT health monitoring system using wearable sensors and a reconfigurable smart interface was proposed to track vital signs — heartbeat, temperature, and blood pressure — and communicate patient data via SMS and the Blynk platform, offering a low-cost, real-time solution for smart healthcare.

These FPGA-based platforms often leverage embedded processors to manage peripheral interfaces, execute control algorithms, and facilitate communication protocols. Commonly used communication protocols include asynchronous mode such as UART (Universal Asynchronous Receiver/Transmitter) [13] and synchronous mode such as SPI (Serial Peripheral Interface) and I²C (Inter-Integrated Circuit) [14]. The UART interface will be used for implementing a bidirectional serial communication between an FPGA board and a graphical user interface (GUI). The implementation of a GUI panel can be by different ways such as C# programming [15][16] or Python programming [17]. In [18], an FPGA-based audio signal processing system was implemented using a ZedBoard with input/output circuits, a VIVADO project, and a C# GUI for serial communication. Finite Impulse Response (FIR) low-pass filtering was implemented in C/C++ using convolution. A modified mean normalization algorithm was applied to the filtered data. Testing with mixed music and

interference signals confirms successful real-time processing on the FPGA platform. In [16], an optimized discrete Fourier transform (DFT) algorithm was implemented on FPGA using Xilinx VIVADO HLS (High-Level Synthesis). Developed in C++, the design was verified via simulation, deployed as an IP (Intellectual Property) core on ZedBoard, and tested using an SDK program. A GUI from C# programming enables serial communication and spectrum visualization. Results confirm successful hardware execution. In [17], an FPGA-based logic analyser was developed with a GUI control panel implemented by Python programming for real-time data visualization and interaction.

This paper is the continuum of the previous work in [18], which developed a MicroBlaze processor based embedded system controlling different LED patterns on an NEXYS A7 FPGA development board [19] through the onboard five push buttons. In this work, we develop a GUI panel by Visual Studio [20] and build bidirectional serial communications between the FPGA board and the GUI to implement the controlling of LED patterns on both the physical board and the GUI. In other words, by operating either the five physical onboard push buttons or the five virtual buttons on the GUI panel, we can control the sixteen physical onboard LEDs and the sixteen virtual LEDs on the GUI to flash synchronously. We use two GPIO (general-purpose input/output) controllers to control the sixteen LEDs and the five push buttons respectively. We also add a UART controller on a VIVADO project to support the serial communication between the FPGA board and the GUI. The main contribution of the work includes (1) designing and developing a VIVADO project involving MicroBlaze soft-core processor, GPIO and UART controllers for the hardware part of the embedded system that can be run on the FPGA development board; (2) developing a Xilinx SDK application program by C/C++ language that can work with the designed hardware system to implement the onboard LED pattern control and the interface to the GUI; and (3) designing and implementing a GUI panel by C# programming on a host computer to control the LED patterns on both the physical board and the GUI panel simultaneously through the push buttons from either the physical board or the GUI panel.

The remainder of the paper is organized as follows: Section 2 describes the system and the design objectives. Section 3 presents the system hardware design and construction; Section 4 presents the system software development process through SDK; Section 5 shows the GUI development process using Visual Studio; Section 6 presents the integrated hardware and software system testing and experimental results; Finally, Section 7 concludes the paper.

2. System Description

The proposed embedded system consists of a MicroBlaze soft-core processor with associated parts like memory and clock modules, GPIO controllers connecting input and output devices, UART controller connecting GUI panel via serial port, and associated GUI program and SDK program running on a host computer, as shown in Figure 1. The processor serves as the brain of the embedded system coordinating other components. Input components (either the onboard push buttons or the virtual buttons on the GUI) are used to send commands to the processor for processing. Once processing is complete, the results are communicated to the required destination via the output component such as onboard LEDs or the virtual LEDs on the GUI.

The SDK program is designed for controlling the LED patterns on the physical FPGA board and interfacing the GUI panel via serial communication and is integrated directly with the hardware it controls. The GUI panel with serial port settings, virtual LEDs and virtual buttons can communicate with the FPGA board by sending commands to the FPGA board to

control the onboard LEDs and receiving commands from the FPGA board to control its virtual LEDs. The Bidirectional Communication between the FPGA board and the GUI panel are thus built with both onboard LEDs and virtual LEDs flashed synchronously through the push buttons from either the physical board or the GUI panel.

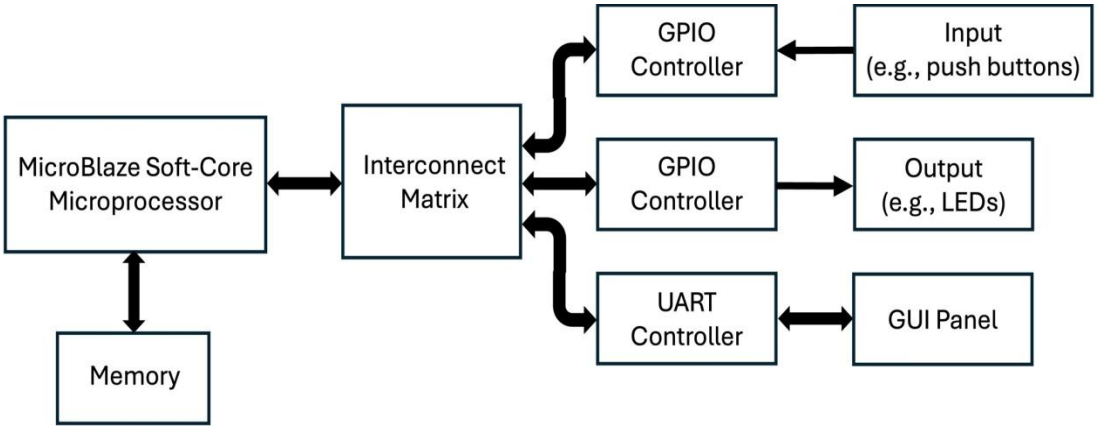

Figure 1: The block diagram of the proposed embedded system

Specifically, the MicroBlaze soft-core processor, associated with the developed SDK program and GUI program, receives commands from the either the onboard push buttons or the virtual push buttons (basically according to the representative value of each push button) and responses with an LED pattern. Each push button represents a specific integer value, i.e., the upper button represents value 2 (corresponding to the LED pattern data value 0x0002 in hexadecimal), the left button represents value 4 (or LED pattern data 0x0004), the right button represents value 8 (or LED pattern data 0x0008), the bottom button represents value 16 (or LED pattern data 0x0010). The central button represents value 1, however, since we define the central push button to turn all LEDs off, the corresponding LED pattern data will be 0x0000. The operation modes are given as follows:

- The SDK program and the GUI program will control the 16 onboard LEDs and the 16 virtual LEDs to flash synchronously from Led0 to Led15 with a certain time delay (e.g., 500 ms) and back to Led0 repeating.
- The onboard or virtual upper button when pressed will force the currently active LED—both physical and virtual—to reset to Led1 (corresponding to pattern data 0x0002). The system then enters a synchronized left-shift sequence, illuminating the next LED every 500 ms across both the FPGA board and the GUI panel. This process continues sequentially through Led15, after which the pattern wraps around to Led0, repeating the cycle indefinitely.
- Similarly, the onboard or virtual left button when pressed will force the currently active LED—both physical and virtual—to reset to Led2 (corresponding to pattern data 0x0004) and perform a similar process.
- The onboard or virtual right button when pressed will force the currently active LED—both physical and virtual—to reset to Led3 (corresponding to pattern data 0x0008) and perform a similar process.

- The onboard or virtual bottom button when pressed will force the currently active LED—both physical and virtual—to reset to Led4 (corresponding to pattern data 0x0010) and perform a similar process.
- The onboard or virtual central button when pressed will turn off all onboard LEDs and virtual LED indicators (corresponding to pattern data 0x0000).

3. System Hardware Design and Construction

The system hardware platform is developed through building a VIVADO project. The hardware system consists of the MicroBlaze processor IP and its accessory modules such as the MicroBlaze Debug Module IP, the Clocking Wizard IP, the Processor System Reset IP, and the MicroBlaze Local Memory IP; the AXI interconnect IP, two IPs of AXI GPIO controllers, and the AXI UART controller. The MicroBlaze is a soft-core microprocessor. The Debug Module provides debugging as necessary. The Clocking Wizard provides a local clock for the entire system. The MicroBlaze local memory provides the RAM for the processor. The AXI interconnect provides a means to connect various AXI-compliant components like GPIO and UART controllers within a system. The GPIO controllers are used to control the push buttons and the LEDs. The UART controller is used to support the Bidirectional Communication.

We construct the hardware platform on the NEXYS A7-100T FPGA board by AMD/Xilinx VIVADO Suite. After starting a VIVADO project, we first add the MicroBlaze IP to the workspace, as shown in Figure 2. Then we click the Run Block Automation option from the Designer Assistance message on the workspace window and configure the MicroBlaze processor by changing the Local Memory option to 32KB, the VIVADO will automatically generate a few associated IPs with connection including Clocking Wizard, Processor System Reset, MicroBlaze local memory, and MicroBlaze Debug Module. Next, we configure the Clocking Wizard IP by selecting the Board Interface of CLK_IN1 to sys_clock, EXT_RESET_IN to reset, the Reset Type in the Output Clocks tab to Active Low, and keep all the others by default.

Figure 2: Adding MicroBlaze IP

To interface the FPGA with external peripherals, two AXI GPIO IP blocks are instantiated in the VIVADO workspace—one for driving the 16 LEDs and the other for reading the 5 push button inputs. Each IP block is configured via the Board Interface settings: axi_gpio_0 is mapped to led_16bits, and axi_gpio_1 to push_button_5bits. Following configuration, the Run Connection Automation feature is invoked from the Designer Assistance prompt, selecting All Automation to automatically connect the S_AXI interfaces of the GPIO IPs to the M_AXI master port of the MicroBlaze processor. This automation instantiates an AXI

Interconnect IP block, which serves as a routing matrix, enabling seamless communication between the MicroBlaze processor and the programmable logic (PL) peripherals. The interconnect ensures proper address mapping and protocol compliance for AXI transactions, facilitating reliable data exchange between the processor and the GPIO modules.

We then add the AXI Uartlite IP on the workspace and click the Run Connection Automation feature to connect this IP to the system. We configure the UART IP by double-clicking on it and change the Baud Rate to 115200 bits per second, Data Bits to 8 bits, and Parity to No Parity on the IP Configuration tab. Finally, we click the Address Editor tab on the workspace to check the component memory addresses allocated by the system: axi_gpio_0 → 64 KB, axi_gpio_1 → 64 KB, axi_uartlite_0 → 64 KB, data local memory block → 128 KB, instruction local memory block → 128 KB. After tidying up the design schematic, the complete embedded system is shown in Figure 3.

After completing the embedded system design in VIVADO, a top-level HDL wrapper is generated to encapsulate the system hierarchy. The design flow proceeds with Run Synthesis, Run Implementation, and Generate Bitstream, producing a bitstream file (i.e., .bit file) that contains the configuration data required to program the FPGA with the specified hardware architecture. Subsequently, the hardware platform is prepared for software development by selecting Export Hardware from the VIVADO menu. This step generates a hardware description file that includes metadata such as IP block configurations, memory address mappings, and processor settings. The Launch SDK option is then used to open the software development environment, where embedded applications can be created, compiled, and deployed to the FPGA. This environment enables seamless integration between the hardware design and the software layer, facilitating real-time control and communication.

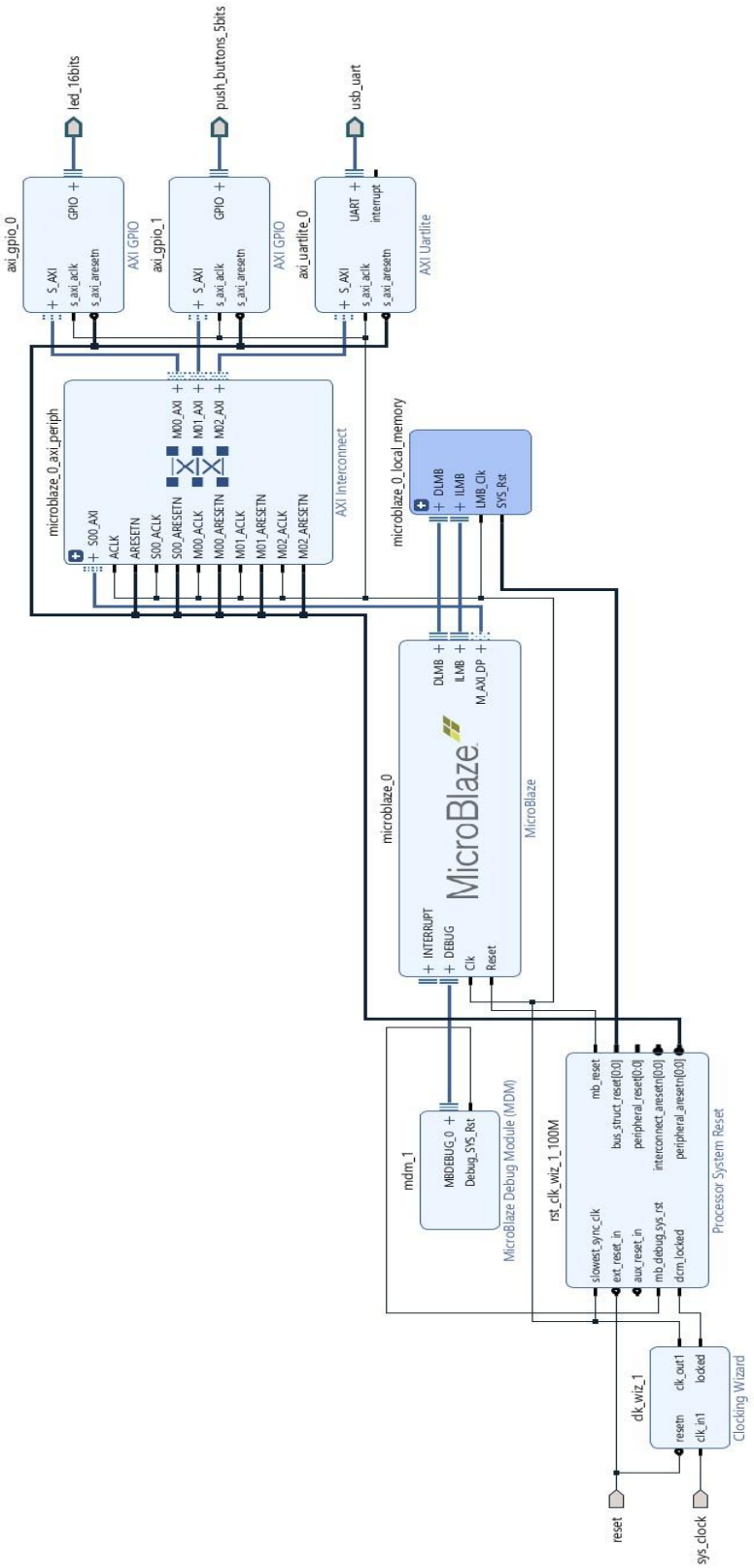

Figure 3: The MicroBlaze processor-based embedded system diagram

4. System Software Development using SDK

To enable software-level control of the hardware system designed in VIVADO, the Xilinx Software Development Kit (SDK)—or its successor, Vitis—is employed as the integrated development environment (IDE) for developing applications targeting the MicroBlaze soft-core processor. The processor receives input commands from both the physical push buttons on the FPGA board and virtual push buttons on the Python-based GUI panel. It controls the behavior of 16 physical LEDs and 16 corresponding virtual LEDs, necessitating a dedicated software application to manage synchronized interactions across hardware and software interfaces.

Upon launching SDK, a hardware platform is automatically generated based on the exported configuration from VIVADO. A new application project is created via File → New → Application Project, where the project name (e.g., LED_Ctrl_Gui) is specified and default settings are applied. A new Board Support Package (BSP) is created, and an initial template—such as Empty Application or Hello World—is selected to scaffold the project.

The core application, implemented in C and named Led_Ctrl_Gui.c, initializes the GPIO peripherals for both LEDs and push buttons, as illustrated in Figure 4. The program enters a continuous execution loop that reads the state of the physical push buttons and receives virtual button inputs via UART-based serial communication from the GUI, while writing the current LED pattern to both the physical and virtual LEDs synchronously. The control logic checks for button activation and determines whether the currently lit LED has reached the terminal position (Led15). If a button press is detected, the corresponding LED pattern is triggered on both the FPGA and GUI sides. If Led15 is active, the system resets the pattern to Led0 and resumes the shifting sequence, maintaining synchronized LED behavior across both platforms.

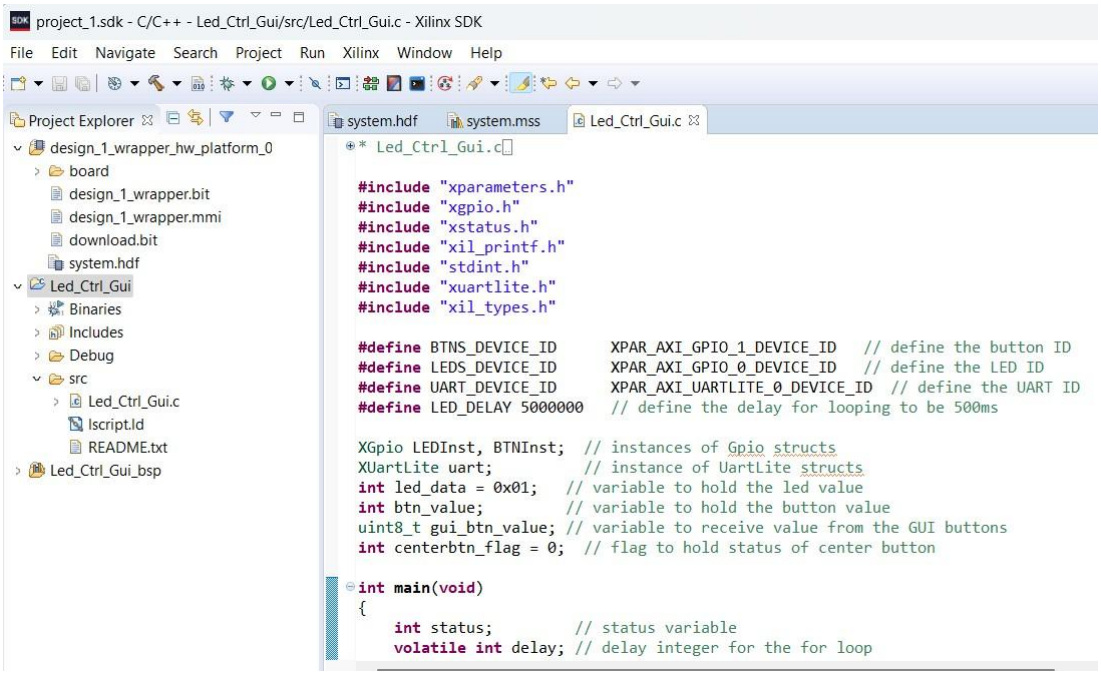

Figure 4: The SDK application project and its associated source program

A software flowchart illustrating the complete control logic and system functionality is provided in Figure 5.

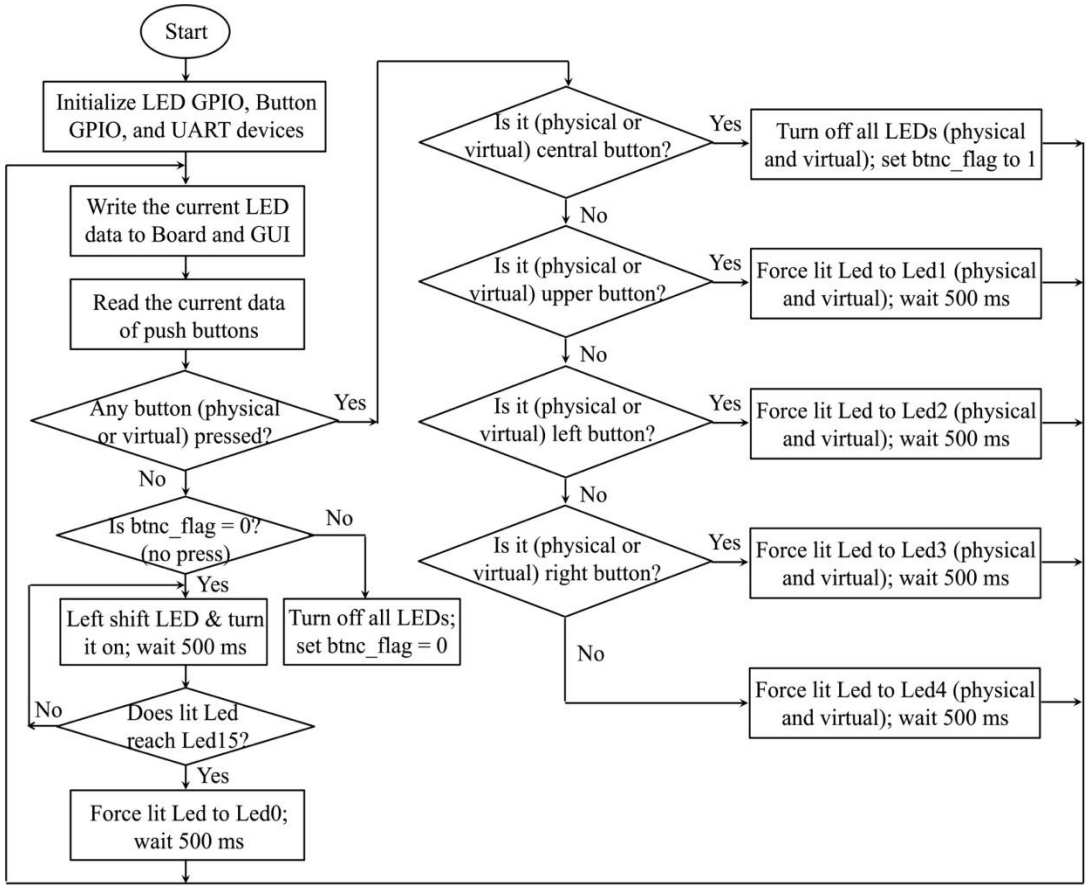

Figure 5: The software flowchart for the SDK application

5. GUI Development using Visual Studio

A graphical user interface (GUI) is designed using windows form application in .net framework Visual Studio [20]. We open Visual Studio and create a blank C# project of type Windows Forms App (.NET Framework) by changing the text property to “GUI for LED Control”. To add the Port Settings Menu, we enter the toolbox and search for MenuStrip. Click and drag to add to the design. Change the "text" property to “Port Settings”. Add a Picture Box from the toolbox to the design and add the university logo picture to the picture box property "Image". We then add five buttons from the toolbox to simulate the five onboard push buttons and change the Name and Text properties to "BTNU", "BTNL", "BTNR", "BTND", and "BTNC" respectively; and add sixteen progress bars with names from Led0 to Led15 and sixteen labels with corresponding names from L0 to L15. Finally, we add a progress bar with a label “Connection Status” to show the serial port status; and add a button with text "Disconnect” to terminate the serial communication if we like to perform the FPGA board operations independently. The completed GUI panel is shown in Figure 6.

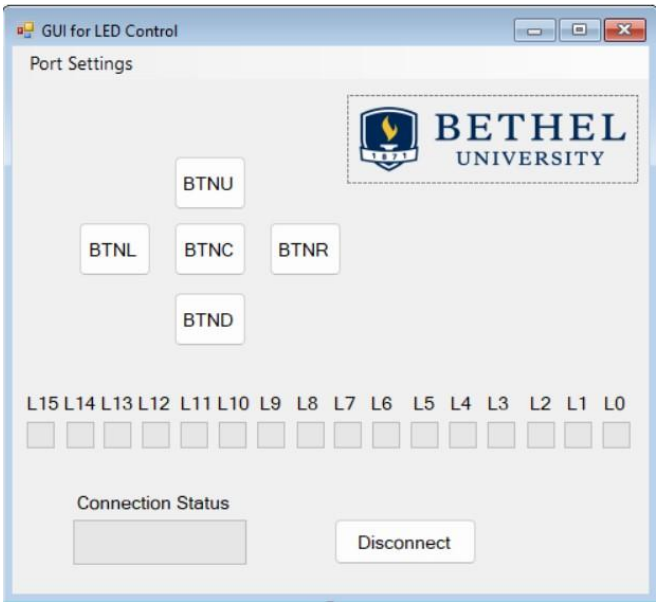

Figure 6: The primary GUI panel

To ensure the serial communications between the FPGA board and the GUI panel, we create another form with the text property "Serial Port Settings" to hold the serial port settings. This form includes six combo boxes, six labels and a button. The combo boxes and their corresponding labels implement the core serial communication settings: COM Port (serial port selection), Baud Rate (the number of bits transmitted per second), Data Bits (bits per character, usually 8), Stop Bits (indicating the end of a character, typically 1), Parity (error checking), and Flow Control (managing the rate of data transmission between transmitter and receiver). The button with the text changed to "Connect" is used to start the serial communication connection. By right clicking each component and selecting name and text properties with appropriate values, we obtain the serial port settings as shown in Figure 7.

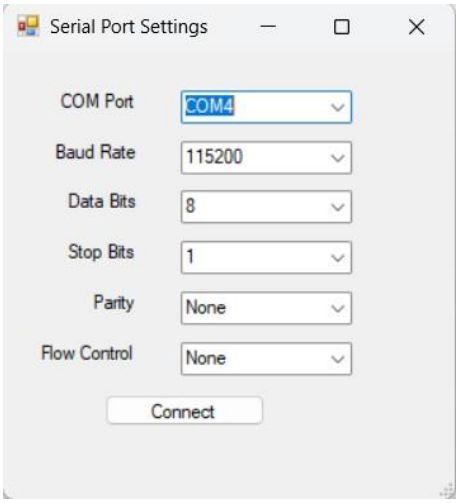

Figure 7: The serial port settings

After adding the necessary components to both forms, we need to write event handler code to each component by double clicking on each component. Due to space limitation, here we only show selected event handlers (serial port settings code, button click processing code, and LED controlling code) in Figures 8, 9, and 10 and provide the flowchart for the entire GUI program in Figure 11.

```
public partial class Form2 : Form
{
    public SerialPort serial = new SerialPort(); //create serial port variable for Form1 access
    public Form2()
    {
        InitializeComponent();
    }
    private void Form2_Load(object sender, EventArgs e)
    {
        string[] ports = SerialPort.GetPortNames();  //get available COM ports
        com_port_combobox.Items.AddRange(ports);     //add to COM port dropdown list
    }
    private void connect_button_Click(object sender, EventArgs e) //connect_button event handler
    {
        try
        {
            serial.PortName = com_port_combobox.Text;                   //add selected COM port
            serial.BaudRate = Convert.ToInt32(baud_rate_combobox.Text);
            serial.DataBits = Convert.ToInt32(data_bits_combobox.Text);
            serial.StopBits = (StopBits)Enum.Parse(typeof(StopBits), stop_bits_combobox.Text);
            serial.Parity = (Parity)Enum.Parse(typeof(Parity), parity_combobox.Text);
            serial.Open();      //open the selected serial port
            this.Close();
        }
        catch (Exception error) //if serial port is not opened and connected, show error message
        {
            MessageBox.Show(error.Message, "Connection Error",
                            MessageBoxButtons.OKCancel, MessageBoxIcon.Error);
        }

    }
}
```

Figure 8: The C# code for serial port settings

```
//event handler for BTNU click
private void BTNU_Click(object sender, EventArgs e)
{
    f2._serial.Write("\u0002"); //write 2 to the serial port
}

//event handler for BTNR click
private void BTNR_Click(object sender, EventArgs e)
{
    f2._serial.Write("\u0008"); //write 8 to the serial port
}
```

Figure 9: The C# code for selected button click (upper button and right button)

```
//event handler for printing received data to GUI LEDs
private void print_received_data(object sender, EventArgs e)
{
    int ledVal = int.Parse(data_in); //parse incoming string into integer for led value
    switch(ledVal) //select LED to turn on, each LED implemented by a progress bar with value 0 (off) or 100 (on)
    {
        case 0:   //if led value = 0, then turn all LEDs off
            LED0.Value = 0;  LED1.Value = 0;  LED2.Value = 0;  LED3.Value = 0; LED4.Value = 0;  LED5.Value = 0;
            LED6.Value = 0;  LED7.Value = 0;  LED8.Value = 0;  LED9.Value = 0; LED10.Value = 0; LED11.Value = 0;
            LED12.Value = 0; LED13.Value = 0; LED14.Value = 0; LED15.Value = 0;
            break;
        case 1:   //if led value = 1, then turn on LED0 only
            LED0.Value = 100; LED1.Value = 0;  LED2.Value = 0;  LED3.Value = 0; LED4.Value = 0;  LED5.Value = 0;
            LED6.Value = 0;   LED7.Value = 0;  LED8.Value = 0;  LED9.Value = 0; LED10.Value = 0; LED11.Value = 0;
            LED12.Value = 0;  LED13.Value = 0; LED14.Value = 0; LED15.Value = 0;
            break;
        case 2:   //if led value = 2, then turn on LED1 only
            LED0.Value = 0;  LED1.Value = 100; LED2.Value = 0;  LED3.Value = 0; LED4.Value = 0;  LED5.Value = 0;
            LED6.Value = 0;  LED7.Value = 0;   LED8.Value = 0;  LED9.Value = 0; LED10.Value = 0; LED11.Value = 0;
            LED12.Value = 0; LED13.Value = 0;  LED14.Value = 0; LED15.Value = 0;
            break;
        case 4:   //if led value = 4, then turn on LED2 only
            LED0.Value = 0;  LED1.Value = 0;  LED2.Value = 100; LED3.Value = 0; LED4.Value = 0;  LED5.Value = 0;
            LED6.Value = 0;  LED7.Value = 0;  LED8.Value = 0;   LED9.Value = 0; LED10.Value = 0; LED11.Value = 0;
            LED12.Value = 0; LED13.Value = 0; LED14.Value = 0;  LED15.Value = 0;
            break;
        case 8:   //if led value = 8, then turn on LED3 only
            LED0.Value = 0;  LED1.Value = 0;  LED2.Value = 0;  LED3.Value = 100; LED4.Value = 0;  LED5.Value = 0;
            LED6.Value = 0;  LED7.Value = 0;  LED8.Value = 0;  LED9.Value = 0;   LED10.Value = 0; LED11.Value = 0;
            LED12.Value = 0; LED13.Value = 0; LED14.Value = 0; LED15.Value = 0;
            break;
        ⋮
```

Figure 10: The C# code for LED lighting according to received LED value

6. System Testing and Results

This section presents the testing procedure and results of the fully integrated system, which includes the FPGA hardware, the SDK application, and the C# GUI. The testing process begins by opening the VIVADO project that has successfully generated the bitstream file. The SDK environment is then launched on the host computer.

Next, the FPGA board is connected to the computer via a USB cable, and the JTAG switch on the board is turned on. Upon powering the board, the red power LED should illuminate, indicating that the board is receiving power. Within the SDK interface, the FPGA is programmed by loading the configuration file (i.e., the bitstream generated by VIVADO). A successful configuration is confirmed when the green "done" LED on the FPGA board lights up.

Following this, the SDK application project, named Led_Ctrl_Gui, is executed by selecting the “Launch on Hardware (GDB)” command within the SDK window. This action initiates the application on the target FPGA board, as illustrated in Figure 12. Upon execution, both the physical onboard LEDs and the virtual LEDs in the GUI begin to shift left sequentially in synchrony, with a delay of 500 milliseconds between each state. Figure 13 captures a snapshot of the system when Led4 is illuminated. Once Led15 is reached, the sequence resets to Led0, and the process repeats continuously.

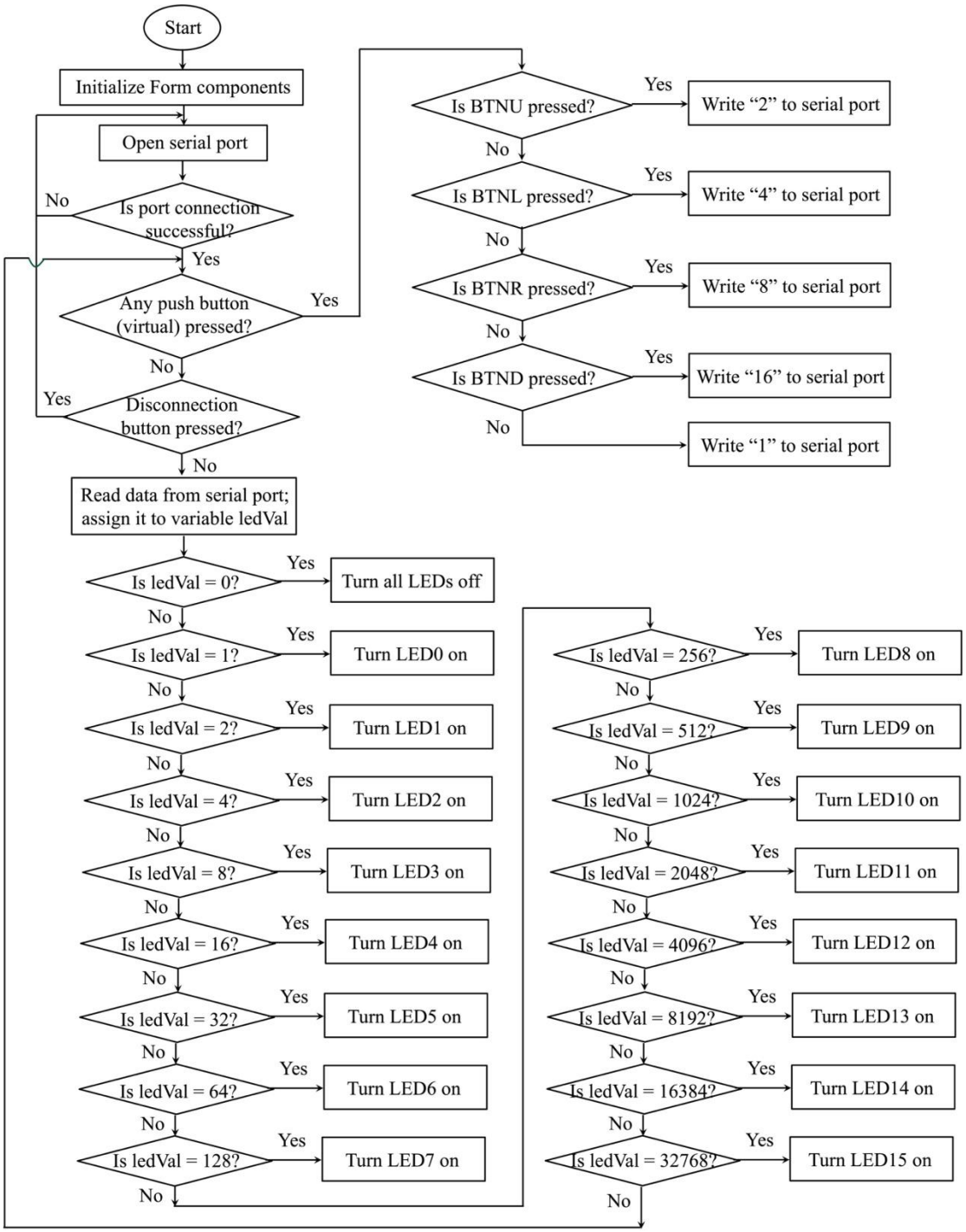

Figure 11: The software flowchart for the GUI operation

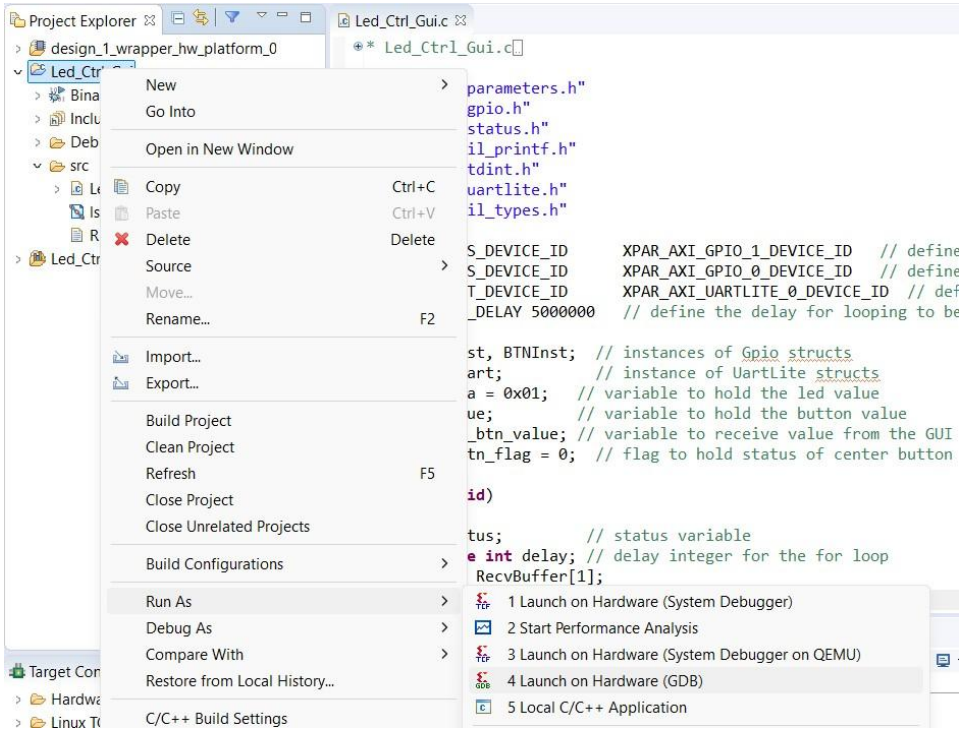

Figure 12: Launch application program on the target FPGA board

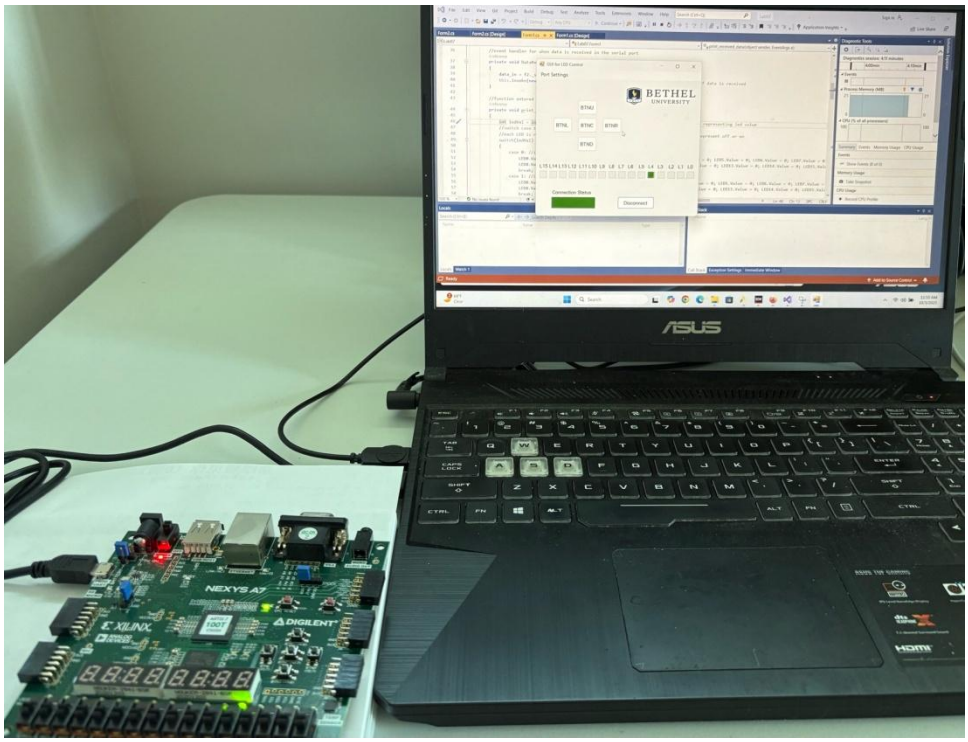

Figure 13 The regular LED operation on both FPGA board and GUI (snapshot status: Led4)

To validate the correct operation of the system, several snapshots were captured during runtime to illustrate the LED shifting behavior and the synchronization between the physical FPGA board and the virtual GUI interface. As shown in the following figures, the LEDs transition smoothly from one state to the next, with each LED lighting up sequentially from Led0 to Led15. The 500-millisecond delay between transitions ensures clear visibility of each state. The GUI accurately reflects the status of the physical LEDs, confirming successful serial communication between the C# application and the FPGA hardware. These results demonstrate the reliability and responsiveness of the system under real-time conditions.

Figures 14 through 18 illustrate the system's response to various user inputs, both from the physical FPGA board and the C# GUI panel. As shown in Figure 14, when the upper push button is activated—either by pressing the physical button on the FPGA board (left) or by clicking the corresponding button on the GUI (right)—the lit LED is immediately forced to Led1. This behavior aligns with the system design, where the upper button corresponds to the integer value 2 (binary: 0000_0000_0000_0010), which maps to Led1.

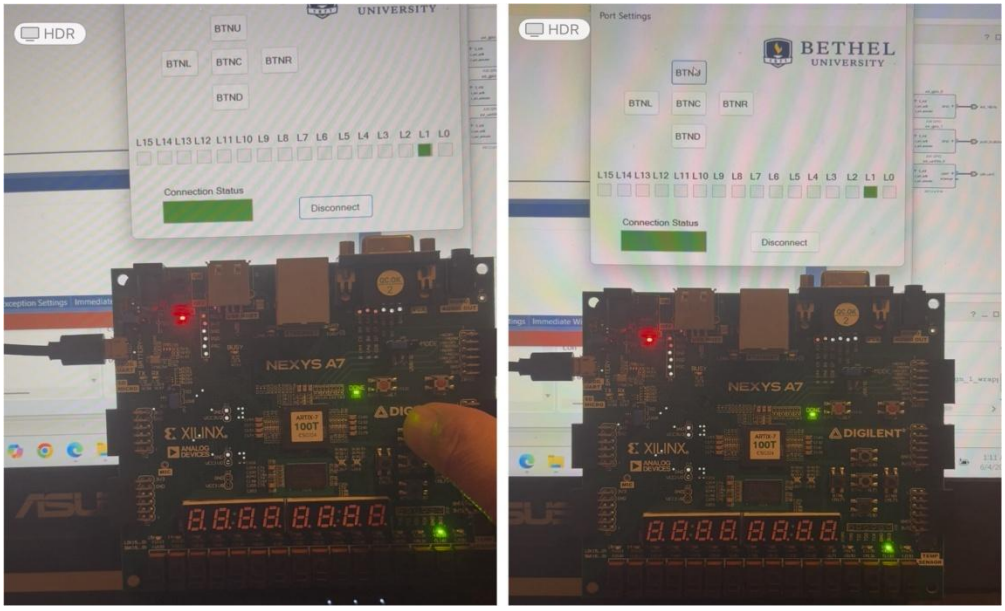

Figure 14: The snapshot of the lit LED being forced to Led1 when the upper button is either pressed from the physical board (left figure) or clicked from the GUI panel (right figure)

Similarly, Figure 15 demonstrates the effect of pressing the left push button from either the physical board (left) or the GUI panel (right), which corresponds to the integer value 4 (binary: 0000_0000_0000_0100). Upon activation, the lit LED is forced to Led2, and the sequential shifting resumes from that position. In Figure 16, the right push button—mapped to the integer value 8 (binary: 0000_0000_0000_1000)—forces the lit LED to Led3, with the left and right subfigures showing the results due to the operations from the physical board and the GUI panel, respectively.

Figure 17 presents the outcome of pressing the bottom push button from either the physical board (left) or the GUI panel (right), which represents the integer value 16 (binary: 0000_0000_0001_0000). As expected, the lit LED is forced to Led4, again with consistent behavior observed across both physical and virtual interfaces. Finally, Figure 18 shows the result of pressing the central push button, which is designated to turn off all LEDs. This

functionality is confirmed by the absence of any lit LEDs in both the physical board and GUI panel due to the button-press from the physical board (left) or the GUI panel (right), validating the correct implementation of the system's reset behavior.

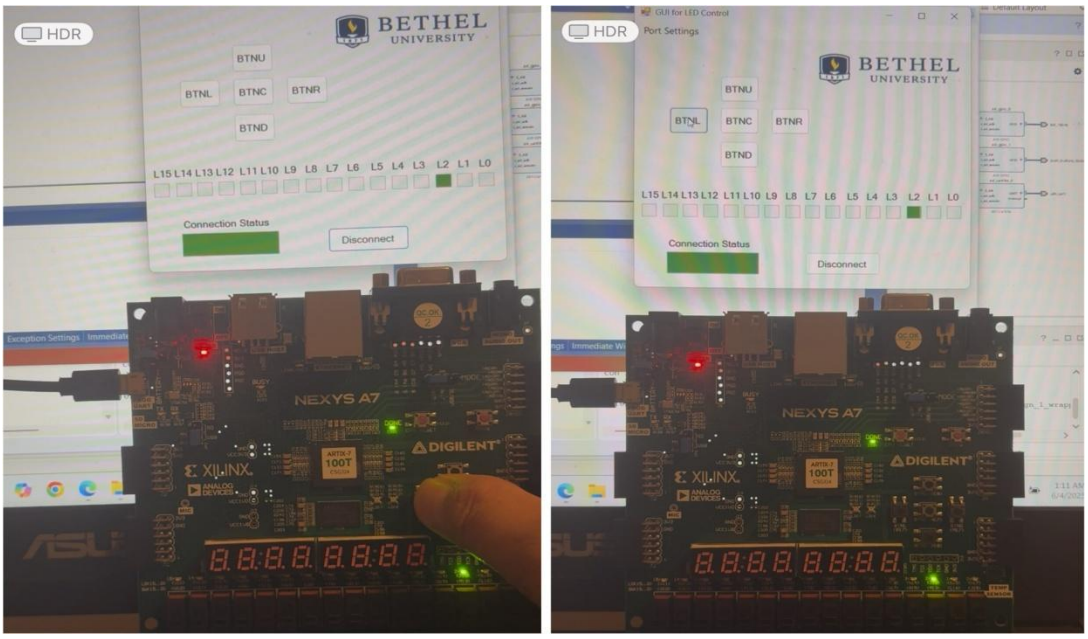

Figure 15: The snapshot of the lit LED being forced to Led2 when the left button is either pressed from the physical board (left figure) or clicked from the GUI panel (right figure)

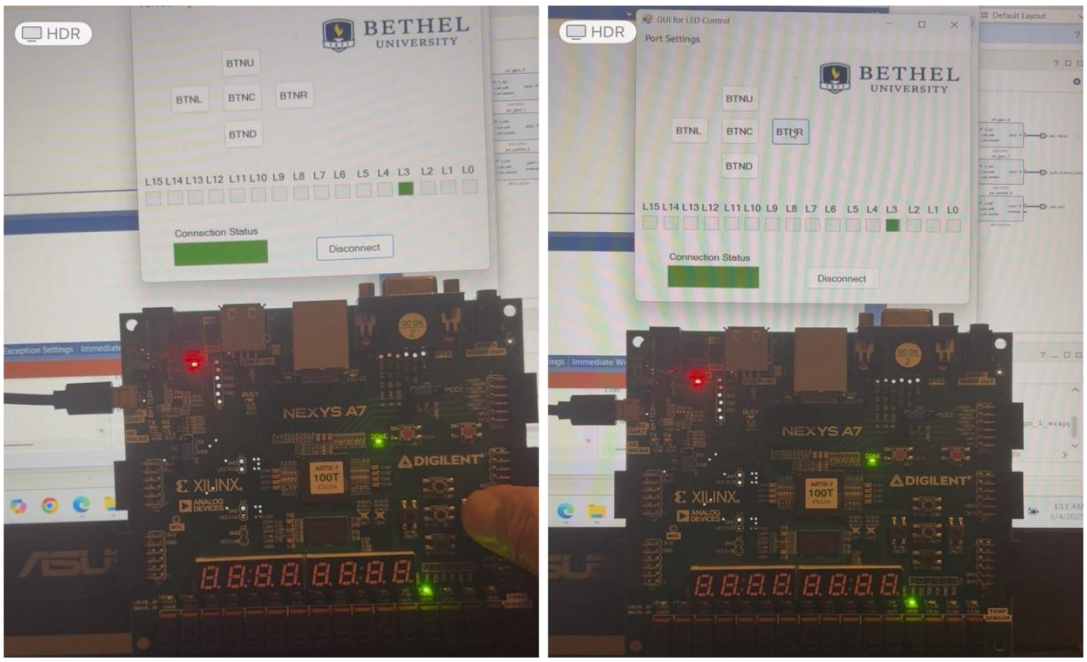

Figure 16: The snapshot of the lit LED being forced to Led3 when the right button is either pressed from the physical board (left figure) or clicked from the GUI panel (right figure)

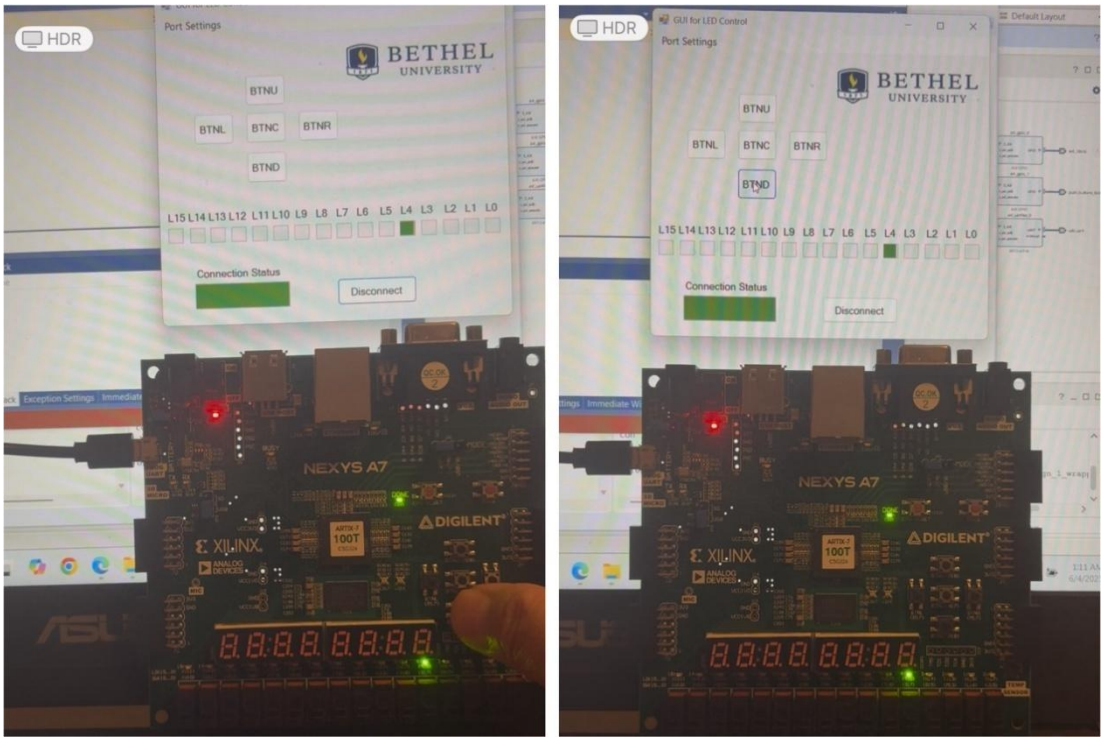

Figure 17: The snapshot of the lit LED being forced to Led4 when the bottom button is either pressed from the physical board (left figure) or clicked from the GUI panel (right figure)

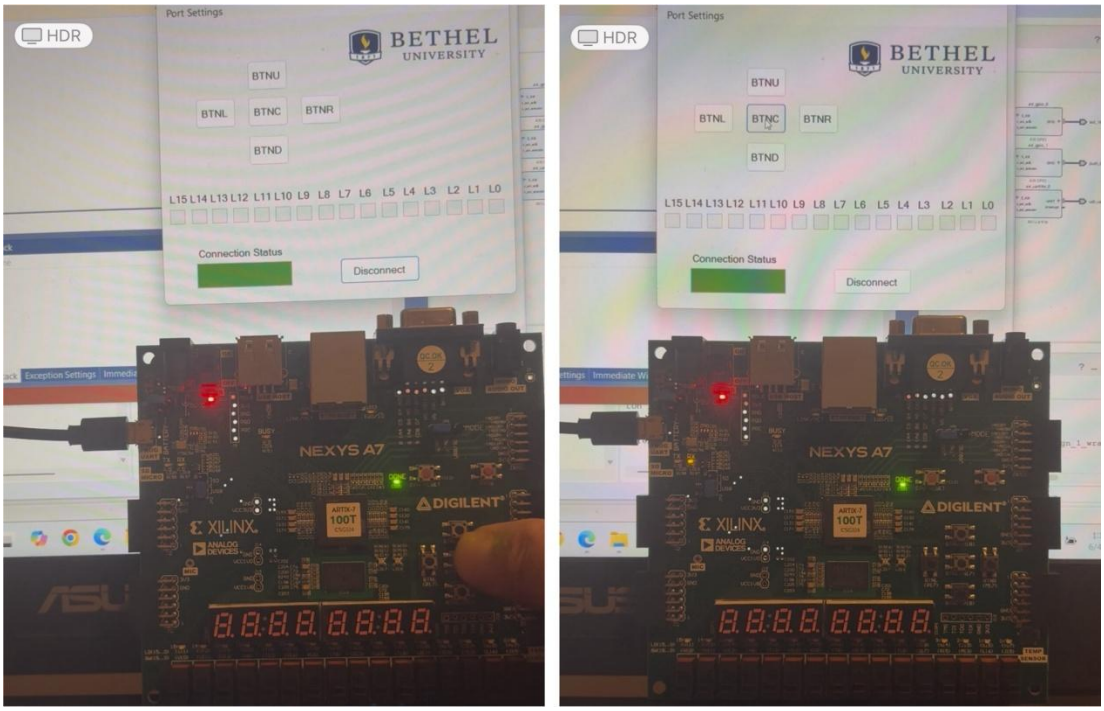

Figure 18: The snapshot of all the LEDs being turned off when the central button is either pressed from the physical board (left figure) or clicked from the GUI panel (right figure)

7. Conclusions

This paper presented the design and implementation of a MicroBlaze soft-core processor-based embedded system that enables synchronized LED pattern control on both a NEXYS A7 FPGA development board and a GUI panel. By integrating UART and GPIO controllers with the MicroBlaze processor and coordinating them through a custom SDK application and C# GUI program, the system achieves real-time bidirectional communication and responsive control through either physical onboard push buttons or virtual buttons on the GUI. The experimental results validate the system's functionality and demonstrate its reliability and flexibility. The button mapping logic, based on binary encoding of control signals, ensures precise LED control and seamless interaction between hardware and software components. The design methodology showcased in this work is scalable and adaptable, offering a robust foundation for future embedded system applications in diverse domains such as industrial automation, automotive systems, healthcare devices, consumer electronics, and communication technologies.

Acknowledgement

This work is supported in part by the PAC Research Grant (2024-2025) and Alumni Faculty Grant (2025-2026) from Bethel University, MN, USA.

References

[1] Brown, S., & Rose, J. (1996). FPGA and CPLD architectures: A tutorial. IEEE Design and Test of Computers, 13(2), 42–57.

[2] Lysecky, R., & Vahid, F. (2009). Design and implementation of a MicroBlaze-based warp processor. ACM Transactions on Embedded Computing Systems (TECS), 8(3), Article 22, 1–22. DOI:10.1145/1509288.1509294.

[3] Crockett, L. H., Elliot, R. A., Enderwitz, M. A., & Stewart, R. W. (2014). The Zynq book: Embedded processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC (Illustrated ed.). Strathclyde Academic Media.

[4] AMD/Xilinx, Inc. (2020). VIVADO design suite user guide: Using the VIVADO IDE (UG893, v2020.1).

[5] AMD/Xilinx, Inc. (2019). Getting started with Xilinx SDK (v2019.1). Available: https://docs.amd.com/v/u/en-US/ug1676-sdkdoc-platformstudio-documentation

[6] Masud, N., Nasir, J., Nazir, M. S., & Aqil, M. (2015). FPGA-based multiprocessor embedded system for real-time image processing. Proceedings of the 15th International Conference on Control, Automation and Systems (ICCAS), Busan, Korea (South), 436–438. DOI:10.1109/ICCAS.2015.7364955.

[7] Bundschuh, S., Kunze, J., & Kuhnert, K.-D. (2024). Implementation of an FPGA-based system to process images and match keypoints on high-resolution pictures. Electronics, 13(23), 4774. DOI:10.3390/electronics13234774.

[8] Vaithianathan, M., Udkar, S., Roy, D., Reddy, M., & Rajasekaran, S. (2024). FPGA-based motor control systems for industrial automation. Proceedings of the International Conference on Sustainable Communication Networks and Application (ICSCNA), Theni, India, 249–254. DOI:10.1109/ICSCNA63714.2024.10864026.

[9] Thinh, T. N., Hieu, T. T., Dung, V. Q., & Kittitornkun, S. (2012). An FPGA-based deep packet inspection engine for network intrusion detection system. Proceedings of the 9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, Phetchaburi, Thailand, 1–4. DOI:10.1109/ECTICon.2012.6254301.

[10] Ceska, M., Vojtěch, H., Holík, L., Korenek, J., Lengál, O., Matousek, D., Matoušek, J., Semric, J., & Vojnar, T. (2019). Deep packet inspection in FPGAs via approximate nondeterministic automata. Proceedings of the IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 109–117. https://api.semanticscholar.org/CorpusID:129945670

[11] Gomes, T., Pinto, S., Gomes, T., Tavares, A., & Cabral, J. (2015). Towards an FPGA-based edge device for the Internet of Things. Proceedings of the IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, 1–4. DOI:10.1109/ETFA.2015.7301601.

[12] Kumar, R. T., Abinaya, S. P., Prakash, D., Janaki, N., Sivarajan, S., & Mani, P. (2024). FPGA interfaced IoT system for smart medical robot monitoring system. Proceedings of the 2nd International Conference on Computer, Communication and Control (IC4), Indore, India, 1–6. DOI:10.1109/IC457434.2024.10486285.

[13] Huang, W., & Sheng, G. (2024). Analysis and research on UART communication protocol. Proceedings of the 4th Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS), Shenyang, China, 768–771. DOI:10.1109/ACCTCS61748.2024.00140.

[14] Leens, F. (2009). An introduction to I2C and SPI protocols. IEEE Instrumentation & Measurement Magazine, 12(1), 8–13. DOI:10.1109/MIM.2009.4762946.

[15] Tang, S., Moua, S., Xie, Y., & Zheng, Y. (2022). FPGA-based implementation of an audio signal processing system on Zedboard. Journal of Smart Technology Applications (JSTA), 3(1), 1–20.

[16] Tang, S., Sinare, M., & Xie, Y. (2022). FPGA-based DFT system design, optimization and implementation using high-level synthesis. International Journal of Computer Applications in Technology (IJCAT), 69(1), 47–61.

[17] Samarasinghe, G. S. (2023). FPGA-based logic analyzer. International Journal of Mechanics of Solids, 4(1), 15–20.

[18] Tang, S. (2025). MicroBlaze processor-based embedded system I: Implementation of LED patterns using an Artix-7 FPGA board. Journal of Science and Engineering Research, 4(1), 1–16.

[19] Digilent, Inc. (n.d.). NEXYS A7 reference manual. Available: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[20] Chowdhury, K. (2019). Mastering Visual Studio 2019: Become proficient in .NET Framework and .NET Core by using advanced coding techniques in Visual Studio (2nd ed.). Packt Publishing..

This page is empty by intention.