Experiment 4: Design of a LED controller

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | 邹易航 | **Class** | AI1 |
| **Date** | 2024.12.18 | **Score** |  |
| **Examiner** |  |  |  |

**I. Experiment procedure**

**1.1 Design the VHDL code to describe the LED controller**

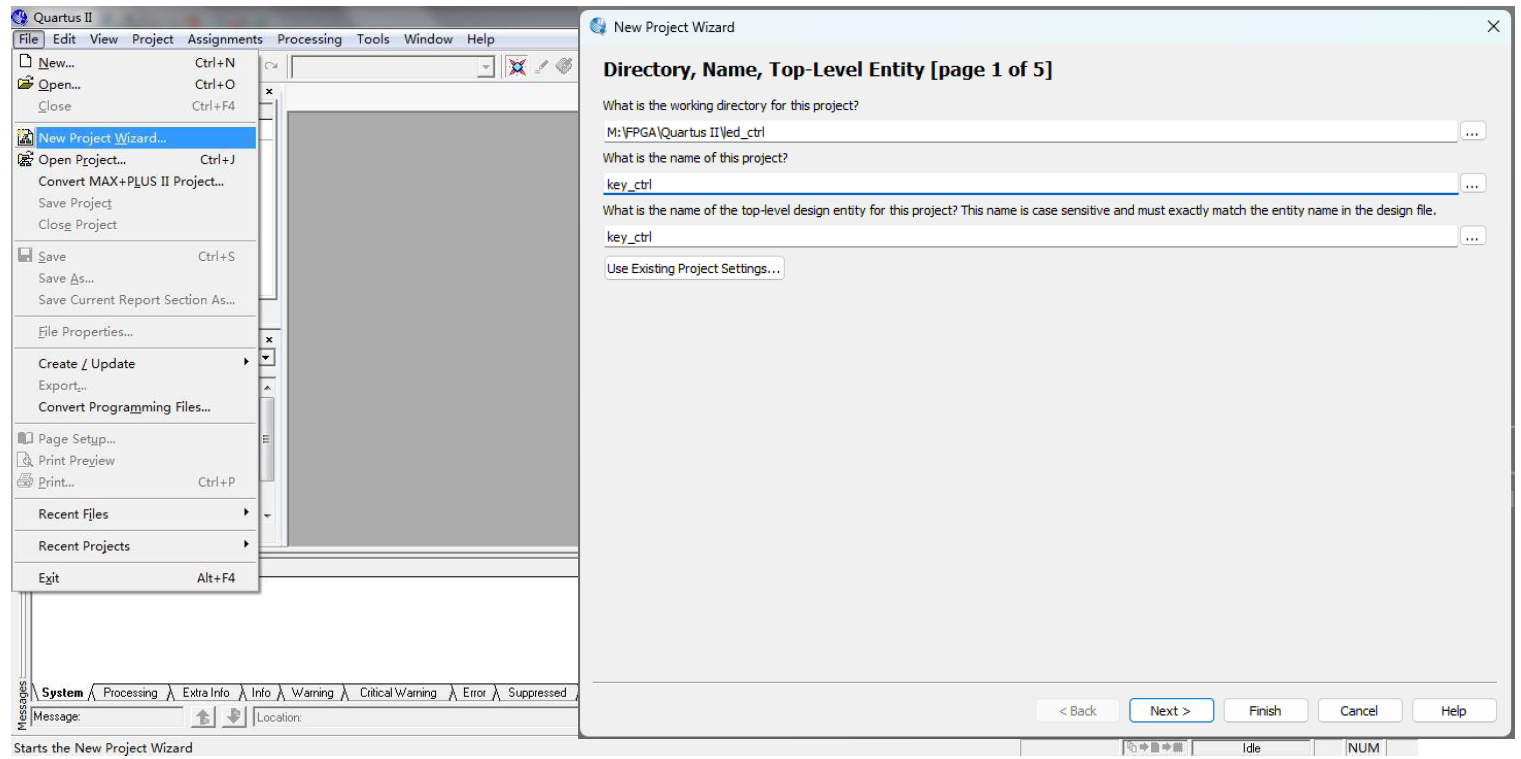


Figure 1: Create new project

The screenshot displays the "New Project Wizard" window from the Quartus II software, typically used for FPGA and ASIC designs. In this initial setup screen, users are prompted to enter several essential details to start a new project. The visible fields include the working directory for the project, the project name, and the name of the top-level design entity. It is critical that the name of the top-level design entity is case-sensitive and matches exactly with the entity name in the design file. This window represents the first step in a five-step process, as indicated by "Page 1 of 5" at the top, which guides users through the complete setup of their project. The interface offers various menu options like 'File', 'Edit', 'Project', and 'Tools' at the top, each providing functionalities necessary for effective project management and execution. The layout is designed to be user-friendly, ensuring that users can easily navigate through the settings to configure their project.

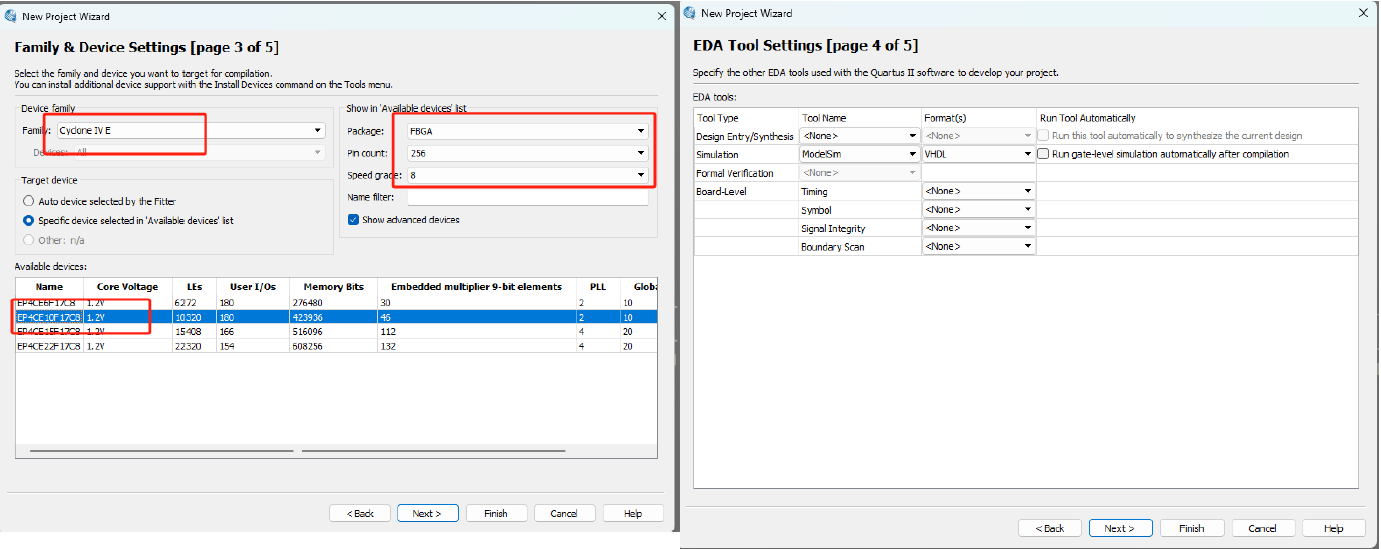


Figure 2: Choose device and EDA tool settings

The screenshot shows two critical configuration steps in the Quartus II "New Project Wizard" for setting up an FPGA design project. In the "Family & Device Settings" (Page 3 of 5), the user selects the FPGA device family and specific target device that best fits the project's requirements. For example, choosing "Cyclone V" as the device family and a specific model like EP3C25 ensures the FPGA meets the necessary specifications for the intended application.

The "EDA Tool Settings" (Page 4 of 5) section allows the user to configure the software tools for design entry, simulation, and timing analysis. Setting up tools such as ModelSim for VHDL simulation is crucial for verifying the design's functionality and performance before the actual hardware implementation. This step ensures that the project setup is comprehensive, facilitating a smooth transition from design to testing and final implementation.

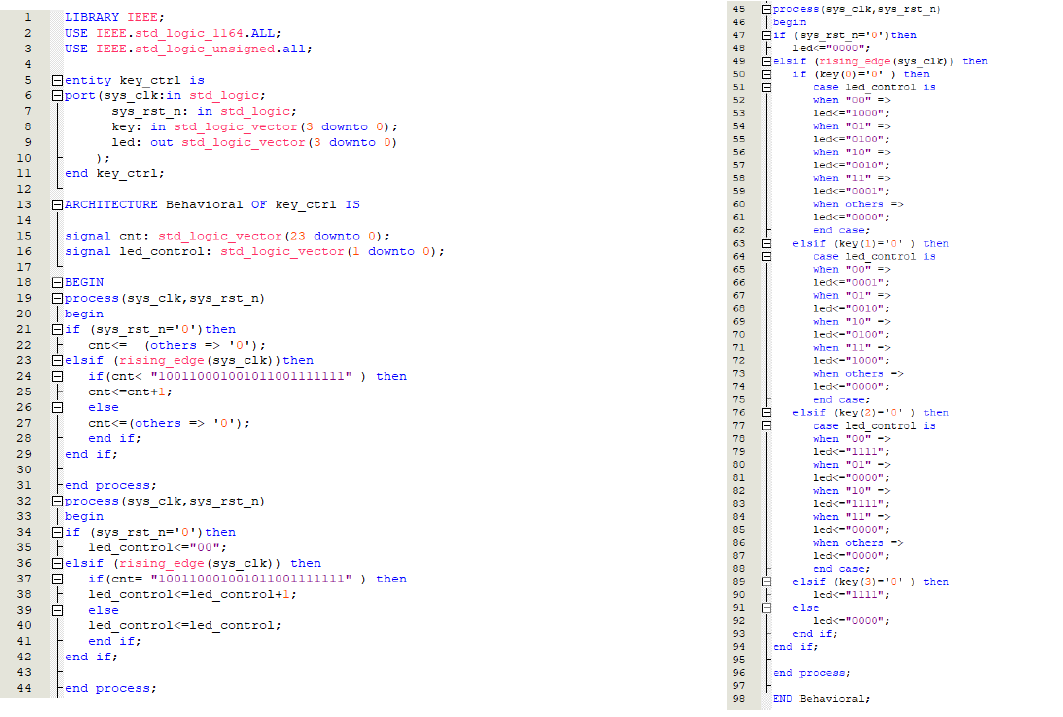


Figure 3: Code of LED controller

The provided VHDL code snippet illustrates a digital logic implementation for an FPGA, designed to control LED behavior based on system inputs and timing conditions. The code is organized into two main processes, both triggered by changes in system clock or reset conditions.

The first process manages a counter that is reset to zero upon a system reset or incremented upon detecting specific binary patterns with each clock cycle. This allows the system to track occurrences or time intervals based on predefined conditions.

The second process controls an LED based on the value of this counter. When the system is reset, the LED control signals are set to an initial state, and as the counter reaches certain thresholds, these signals are adjusted to reflect different LED states. This setup enables the LED to display varying patterns or signals in response to the internal counter's value, facilitating dynamic visual feedback based on the system's operational status. This configuration is typical in embedded systems where visual indicators are required to monitor or demonstrate the device's state.

**1.2 Design a testbench to test the function of the control system by pressing the reset button and 4 control buttons respectively**

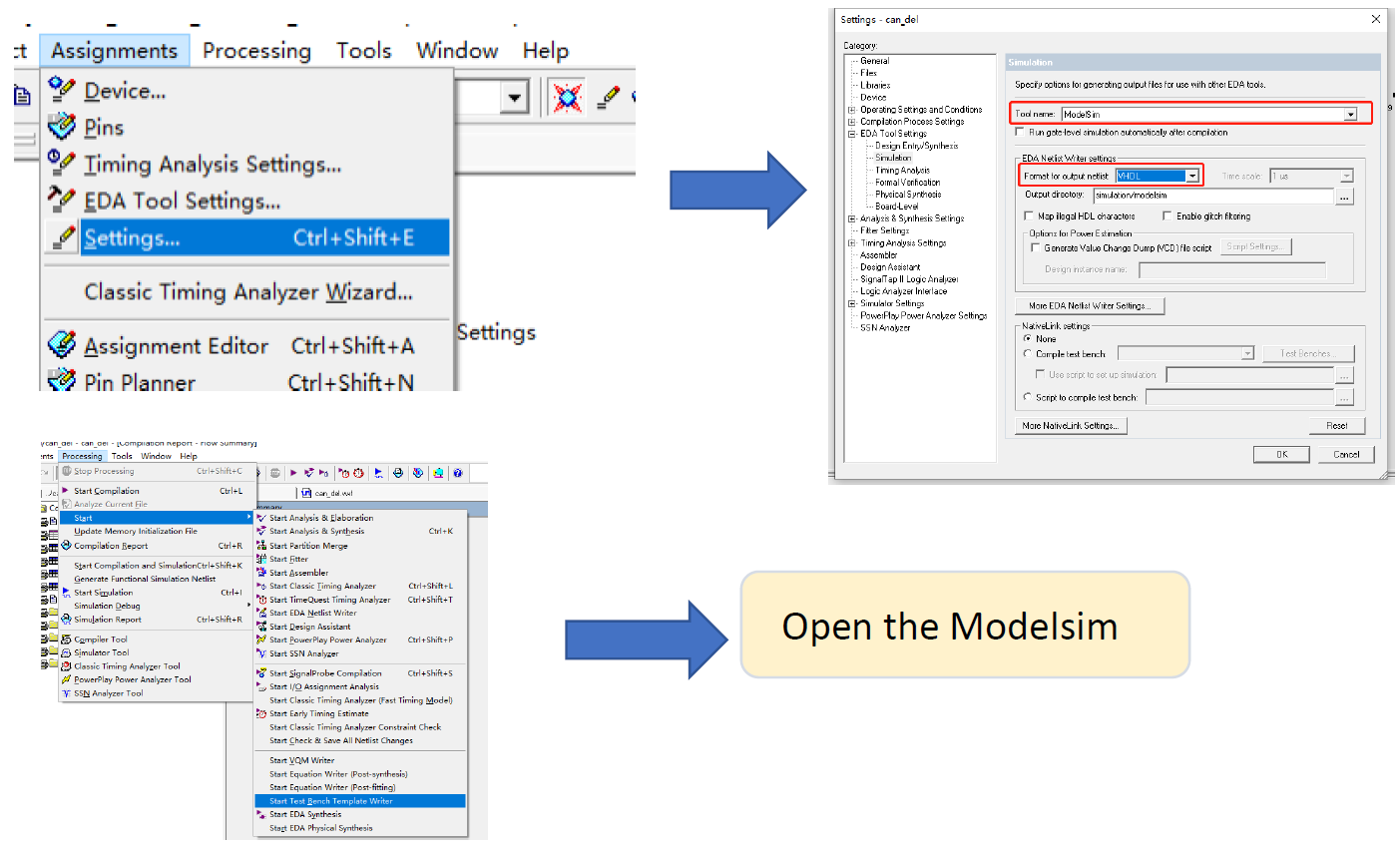


Figure 4: Create testbench template

This screenshot illustrates the steps required to configure and launch a simulation tool within a typical FPGA design workflow using the Quartus II software. The process starts from the main menu where the user navigates to "Tools" and then selects "EDA Tool Settings" from the dropdown menu. This action opens a settings dialog where specific simulation options can be configured. Here, the user has selected ModelSim as the simulation tool, specifying VHDL as the format for netlist output.

Once the settings are configured, the user can launch the ModelSim software directly from Quartus II to simulate the VHDL code and verify the FPGA design. This integration between Quartus II and ModelSim simplifies the simulation and testing process, allowing for an efficient validation of logic designs before hardware implementation. This setup is crucial for detecting and fixing design issues early in the development cycle.

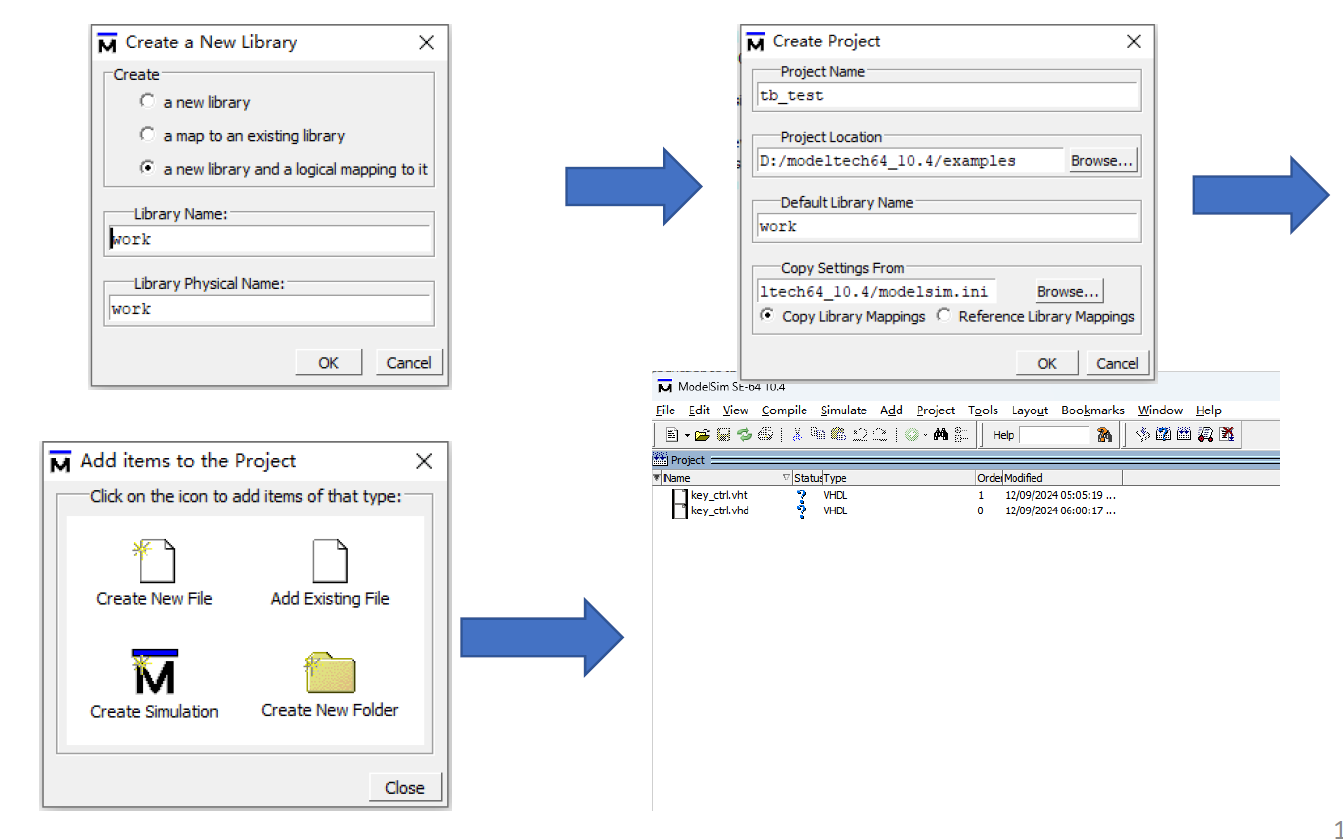


Figure 5: Create project in modelsim

The screenshot outlines the process of setting up a simulation project in ModelSim, a popular tool for simulating and testing VHDL and Verilog designs. The sequence starts with the creation of a new simulation library, where the user specifies the library name as "work," which is a common default in VHDL and Verilog environments. This step ensures that all compiled design units are stored in a designated library, making them easily accessible for simulation.

Next, the user creates a new project in ModelSim by naming it and specifying the project's location on the computer. The default library name "work" is again specified, linking the project to the library created in the previous step. This setup allows the user to manage all related design files and simulations in an organized manner.

The final step shows the main interface of ModelSim with the project loaded. Here, the user can add VHDL files to the project, either by creating new files or importing existing ones. Once the files are added, they can be compiled and simulated within the ModelSim environment to test and verify the FPGA design. This workflow is crucial for identifying design errors early, optimizing the design, and ensuring its operational integrity before implementation on physical hardware.

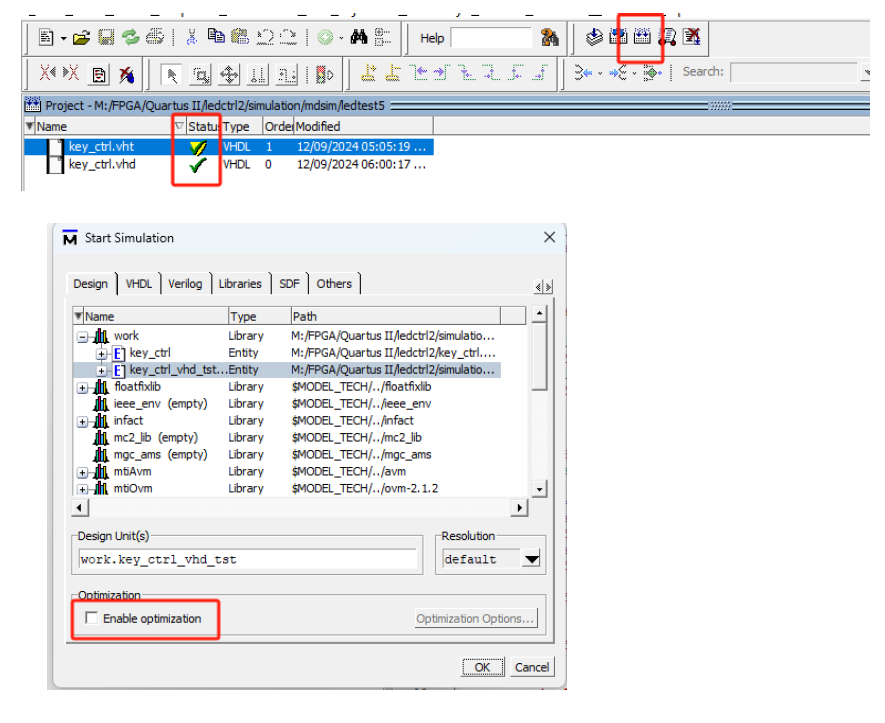


Figure 6: Compile all files in ModelSim

Next, the process involves setting up and running a simulation in ModelSim. The project structure shows that VHDL files have been successfully added and compiled, indicated by the check marks beside the file names within the 'work' library. This library organizes and makes the design units accessible for simulation.

In the "Start Simulation" dialog, the specific design entity to simulate, such as 'key\_ctrl\_vhd\_tst,' is selected from the library. Additional options, like enabling optimization, allow users to adjust the simulation for performance or accuracy, depending on the requirements.

The toolbar provides tools for managing simulations, such as recompiling, running, or debugging, ensuring iterative validation of the design. This step is critical for testing the VHDL code functionality and confirming that the design meets the intended specifications before implementation on hardware.

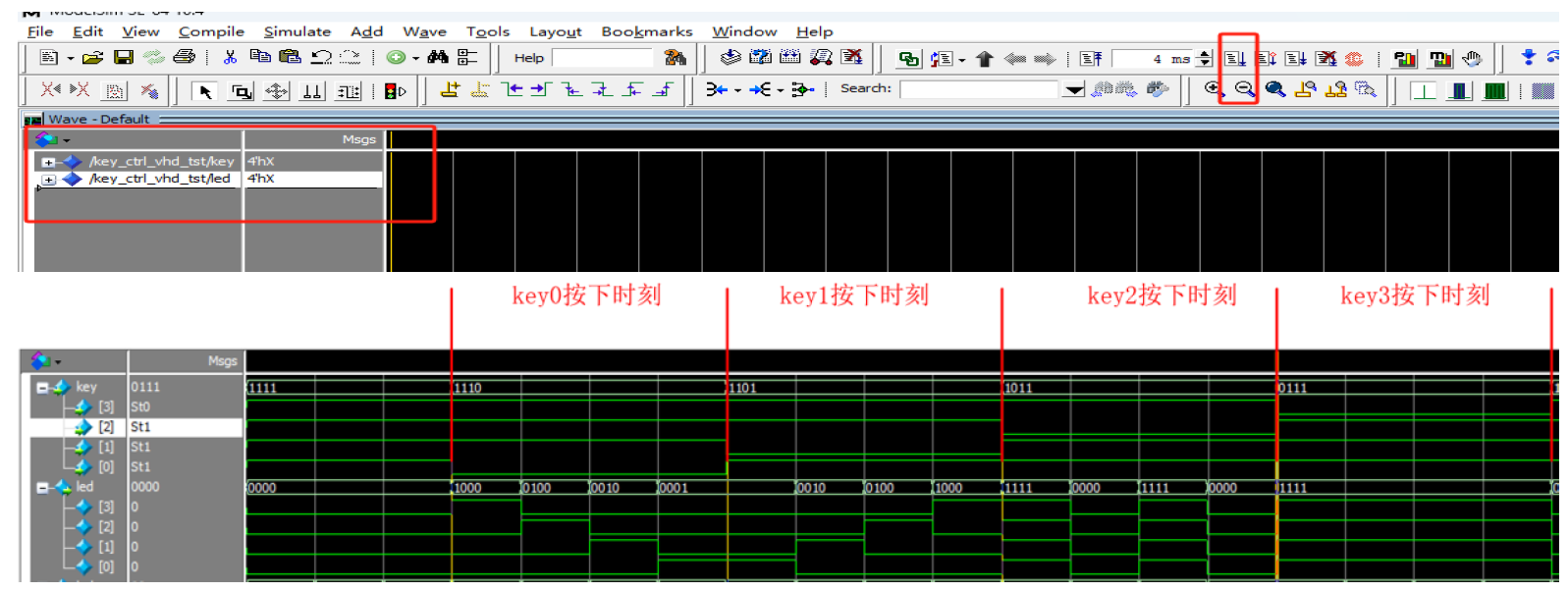


Figure 6: The waveform of the simulation

In the ModelSim environment shown, the waveform viewer displays the simulation results of a VHDL design. The interface highlights the waveforms for various signals, allowing for detailed analysis of each signal's behavior over time. Key signals such as key0, key1, key2, and key3 are marked to illustrate specific test conditions or changes within the simulation timeline. Each signal waveform provides a visual representation of how data values change, represented by different levels in the waveform.

This simulation setup enables the user to pinpoint exact moments where signal transitions occur, crucial for debugging and verifying that the VHDL logic performs as expected. By observing these changes, users can validate the timing and logical correctness of their FPGA designs, ensuring that each part of the circuit responds appropriately to test inputs. The tool's functionality to zoom, measure time intervals, and annotate specific points helps in refining the design by providing a clear and detailed view of each signal's behavior during the simulation.

**1.3 Program the VHDL code to the FPGA and test its function by pressing the corresponding buttons**

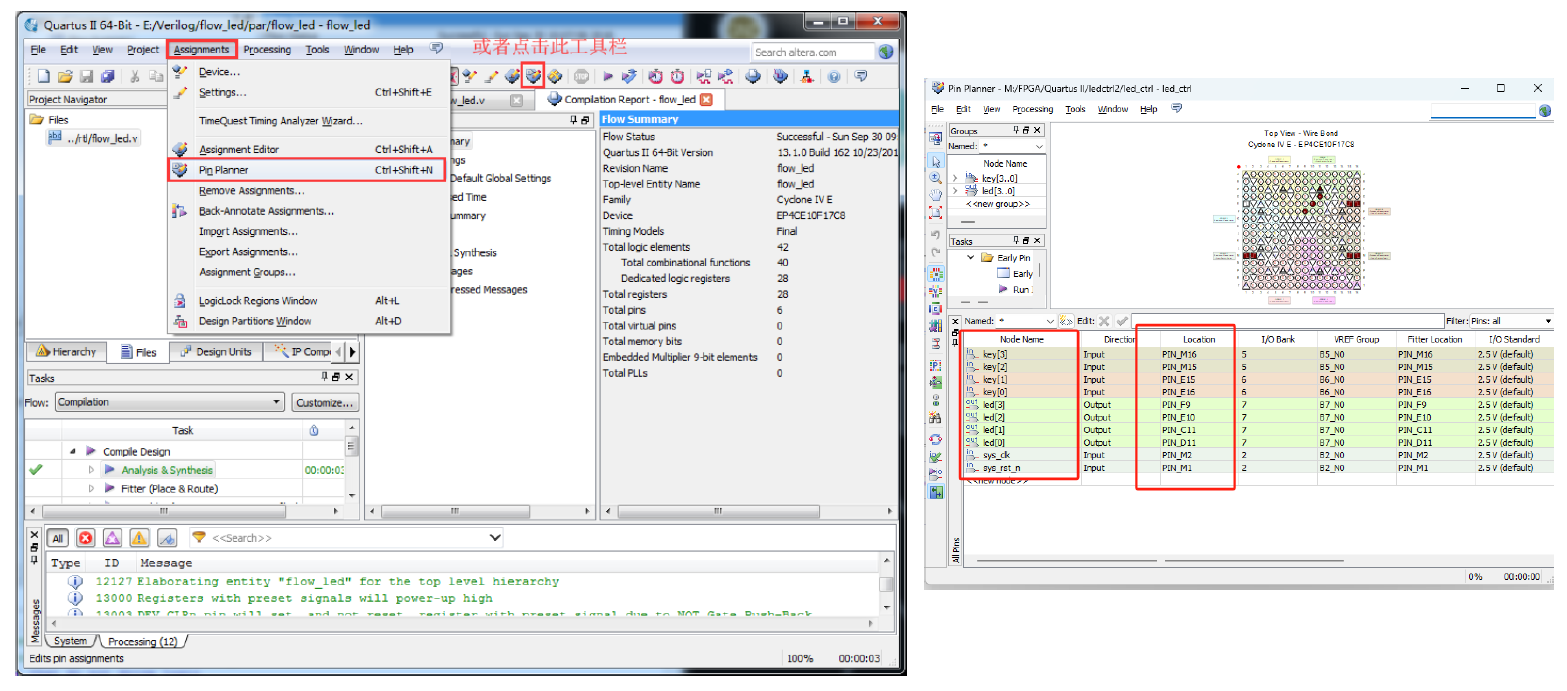


Figure 7: Assign FPGA pins to the nodes

The image presents an advanced view from the Quartus II software, detailing key aspects of FPGA design and implementation phases. On the left side of the interface, we see the project navigator and compilation report, which are essential for managing various project files and observing the results after synthesis and implementation. The "Tasks" window at the bottom-left corner, and the compilation tasks listed, reflect different stages of the FPGA design flow including compilation, synthesis, and timing analysis.

The right portion of the image showcases the Pin Planner and the assignment details for an FPGA device. This planner is crucial for configuring and assigning physical pins on the FPGA to match the requirements of the VHDL code. Each signal name is matched with a specific pin location on the FPGA, along with other settings such as the I/O standard and current strength settings, which are vital for ensuring correct electrical behavior and interface compatibility with other hardware components.

The use of tools like the Pin Planner aids in visualizing the physical layout of the FPGA pins, which is represented by the colorful grid in the middle. This helps designers ensure that pin assignments are correctly planned according to the physical package of the FPGA, which can prevent errors in the physical deployment of the design. The table on the bottom right shows detailed information about each pin's function, location, and electrical characteristics, providing a comprehensive overview to guide the physical setup and integration of the FPGA in real-world applications.

Overall, these tools and views in Quartus II form a critical part of the FPGA design process, allowing for detailed planning, configuration, and verification before the actual hardware programming and deployment.



Figure 8: The correct relation of signals and pins

The image illustrates the use of the Quartus II Pin Planner tool, which is vital in defining and organizing the input and output pin assignments for an FPGA project. The left side of the image provides a detailed view of the Pin Planner. Each signal, whether it’s an input or an output, is associated with a specific pin on the FPGA device, as listed in the table with clear indications of their respective locations on the FPGA package.

For instance, inputs like sys\_clk, sys\_rst\_n, and key[0] to key[3] are mapped to specific pins with precise location details such as M2 for the system clock or E16 for the key[0] input. Outputs like led[0] to led[3] are similarly assigned to pins like D11 and C11, specifying that these pins will control LEDs.

On the right, a complementary table displays the same information but is formatted for clarity in documentation or presentation purposes. It categorizes the pins into inputs and outputs and provides additional details such as the signal name, type, pin location, and a description that might include the signal's purpose or characteristics like the clock frequency for sys\_clk.

This configuration ensures that every signal is correctly and efficiently routed to the appropriate physical pin on the FPGA, facilitating the device's operational integrity and compatibility with other hardware components. The visual layout in the Pin Planner aids in verifying that all pins are correctly assigned and that the design conforms to the physical constraints and requirements of the FPGA hardware. This step is crucial for avoiding common pitfalls in hardware design such as pin conflicts or misassignments that can lead to project delays or failures.

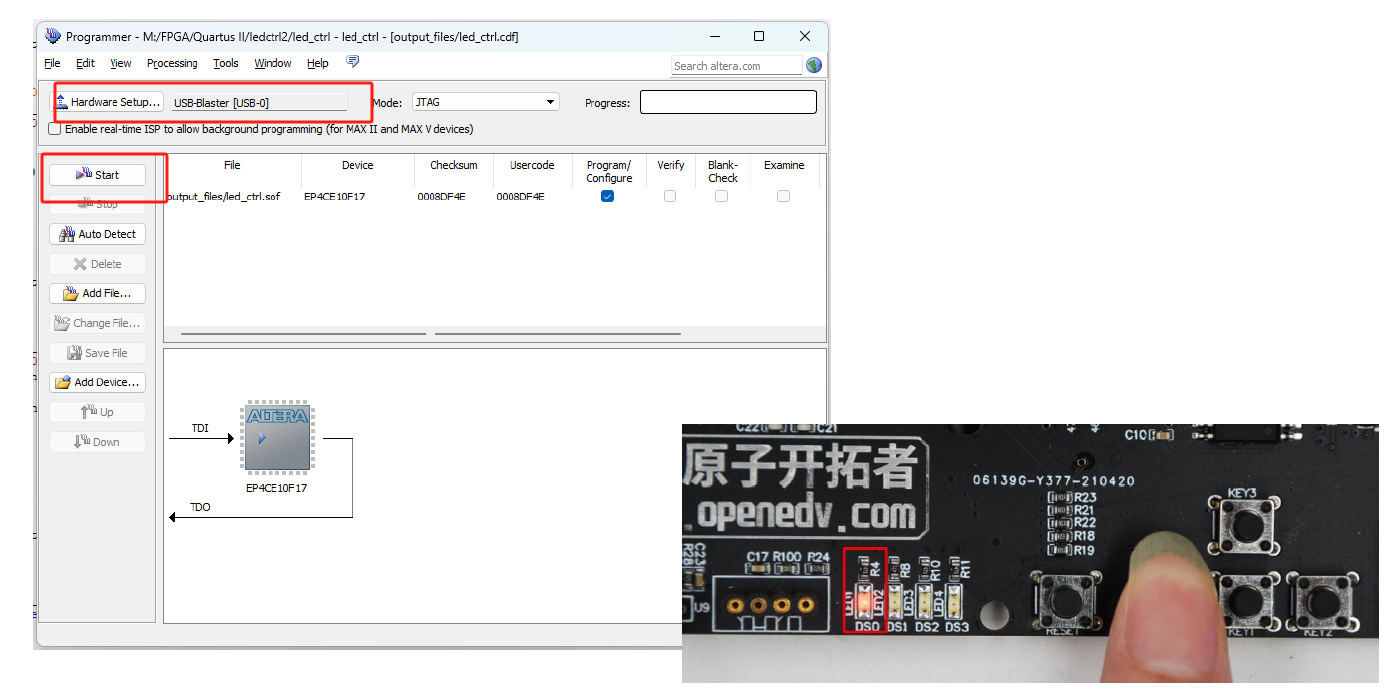


Figure 9: Download and Simulate the VHDL code on FPGA

In the image, the process of programming and verifying an FPGA using Quartus II software and a hardware programming device is depicted. The left part of the image shows the "Programmer" interface in Quartus II, where the user can set up and initiate the programming of FPGA devices. The interface features options to choose the hardware setup (in this case, USB-Blaster), specify the programming file, and initiate the process by clicking on the "Start" button. This part of the software allows for configuration and programming of the FPGA via JTAG, a standard debugging interface.

On the right side of the image, an actual FPGA development board is shown, specifically an OpenEduv development board. This hands-on component of FPGA programming involves physically interacting with the board. Here, a user is pressing a button on the board, which could be part of a procedure to manually reset the board or initiate a specific function during the programming or testing phase.

This complete setup illustrates a typical workflow in FPGA development where the software and hardware are interactively used to upload designs, test them, and make adjustments as necessary. The ability to directly interact with the hardware while managing settings via the software is crucial for practical FPGA development, enabling real-time testing and debugging to ensure that the FPGA operates as intended in real-world applications.

**II. Experimental record**

**2.1 Design the VHDL code to describe the LED controller**



Figure 10: VHDL code of LED controller (first part)



Figure 11: VHDL code of LED controller (second part)

Designing the VHDL code to describe an LED controller involves creating a structured approach to manage various inputs and outputs using digital logic. The code segment here encapsulates the entire behavior of an LED control system using VHDL, as typically implemented in FPGA-based designs.

**Entity Declaration:**

The VHDL entity, named key\_ctrl, includes interface definitions such as:

* sys\_clk: System clock signal.
* sys\_rst\_n: Active-low system reset.
* key: Four-bit vector for input keys.
* led: Four-bit vector for controlling individual LEDs.

**Behavioral Architecture:**

This section details the logic used to control LED states based on the input from the keys under the influence of a clock signal.

1. **Initialization and Control Logic**:
   * **Counter Management**: A counter (cnt) is used to execute actions based on clock events. The counter increments conditionally upon the detection of a specific binary pattern. If the reset is active or the pattern does not match, the counter resets.
   * **LED Control**: Based on the value of the counter or specific key inputs, the LEDs are manipulated. For instance, each key corresponds to a specific LED pattern.
2. **Key-Based LED Control**:
   * The process sections conditionally check the state of each key input. Depending on which key is pressed (key(0) to key(3)), different case statements dictate the pattern displayed on the LEDs.
   * For each key, various patterns can be assigned, such as turning all LEDs on, setting them in a sequence, or turning them all off, enhancing the interactive aspect of the design.

**Detailed LED Control Logic:**

* Each key press is evaluated at the rising edge of the clock. Depending on which key is active, a different LED control pattern is assigned. This decision-making is implemented using nested case statements within each key's conditional block, providing a clear and organized method to handle multiple scenarios.
* The design allows for dynamic changes and can be extended or modified for more complex scenarios, such as creating patterns or sequences based on combinations of key presses.

This VHDL code effectively demonstrates the use of conditional logic, binary pattern matching, and synchronized operations based on system clock signals, encapsulating the core functionalities needed in an LED control system tailored for FPGA applications. The implementation showcases the integration of input handling, process synchronization, and output management in a single cohesive design.

**2.2 Design a testbench to test the function of the control system by pressing the reset button and 4 control buttons respectively**

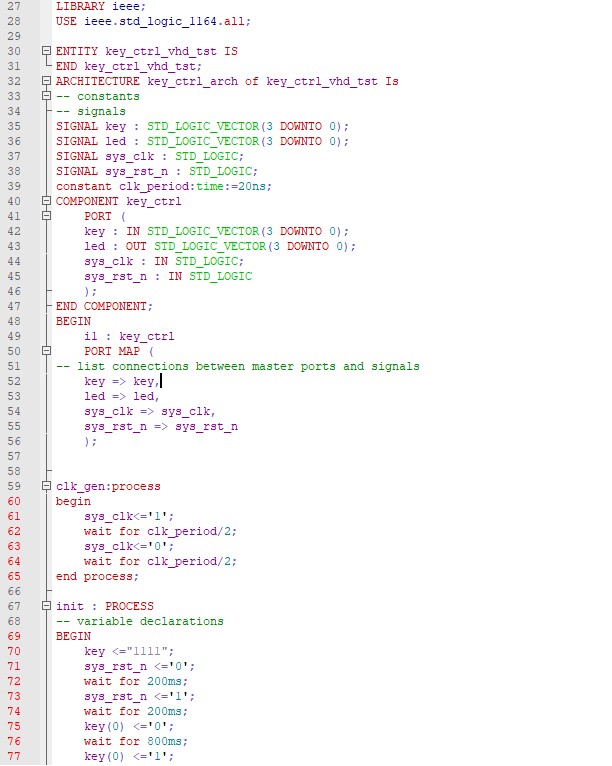


Figure 12: VHDL code of testbench (first part)

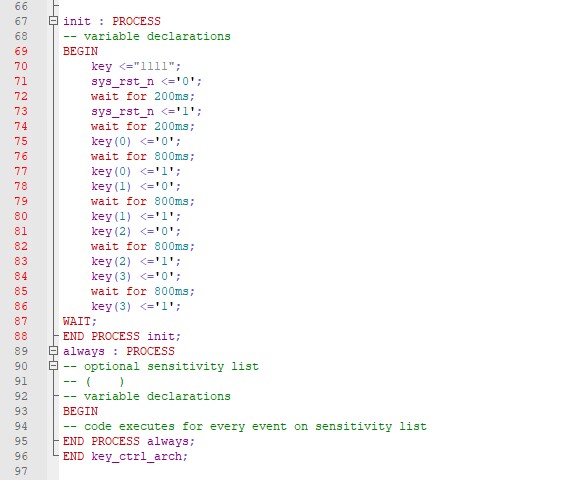


Figure 13: VHDL code of testbench (second part)

Designing a testbench to verify the functionality of the LED control system involves simulating the behavior of the control system by systematically pressing the reset button and the four control buttons. This testbench aims to ensure that the control system reacts as expected to the inputs under test conditions.

**Testbench Setup**

The testbench, defined as key\_ctrl\_vhd\_tst, includes the following components:

1. **Signal Declarations**:
   * key: A 4-bit signal to represent the state of the four control buttons.
   * led: A 4-bit signal to observe the output of the LED controller.
   * sys\_clk: A clock signal to drive the timing of the simulation.
   * sys\_rst\_n: A reset signal to test the initialization and reset behavior of the control system.
2. **Component Instantiation**:
   * The LED controller (key\_ctrl) is instantiated within the testbench, connecting its inputs and outputs to the testbench's signals. This allows the testbench to directly manipulate the controller's inputs and monitor its outputs.

**Simulation Processes**

1. **Clock Generation Process (clk\_gen\_process)**:
   * Generates a continuous clock signal with a defined period, crucial for testing the synchronous behavior of the control system.
2. **Initialization and Test Sequence (init process)**:
   * **Reset Simulation**: Initially sets all keys to '1' (inactive state) and activates the reset signal. After a brief period, the reset is deactivated, setting the stage for further testing.
   * **Sequential Key Press Simulation**: Simulates pressing each key by toggling the respective bit from '1' to '0' and back to '1'. Each key press is maintained for a specified duration to ensure the control system has sufficient time to process the input and update the LED outputs accordingly.
   * The delays between actions allow the system to stabilize and reflect the changes, ensuring that the control logic for each button is correctly implemented.

**Testbench Execution**

* The testbench is configured to run these processes automatically upon simulation start, cycling through the reset and each button press. This automated sequence helps verify that the LED outputs are correctly toggled in response to each input, demonstrating the system's reliability and correct functionality under controlled test conditions.

This testbench effectively mimics user interactions with the control system, providing a comprehensive test scenario to verify each aspect of the system’s response to inputs. It highlights potential issues in the control logic or timing, essential for debugging and validating the control system before deployment in a real-world application.



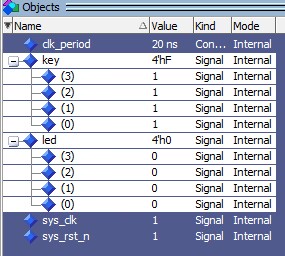




Figure 13: Steps before simulation

The screenshot displays a successful compilation and simulation status of the VHDL file key\_ctrl.vhd as indicated by the timestamps next to the file names, showing recent modifications and accesses. It reveals the debug interface where various internal signals and constants are listed under the "Objects" section, which have been set up and are being monitored during simulation.

Notably, the clk\_period is defined as 20 ns, establishing the clock frequency used in the simulation. The key and led signals are shown with their respective values at this instance in the simulation, with all keys set to '1' (inactive) and all LEDs set to '0' (off), demonstrating the initial or default state of the system before any input actions are simulated. The sys\_clk and sys\_rst\_n signals are also visible, indicating that the system clock is running and the system is not in reset.

The simulation has run for a significant duration, specifically up to 4000 ms, allowing for ample time to observe the behavior of the control system under various input conditions. This setup confirms that the objects have been correctly added to the test environment and the simulation has proceeded long enough to produce meaningful results, reflecting the system's response to the defined test inputs and conditions.

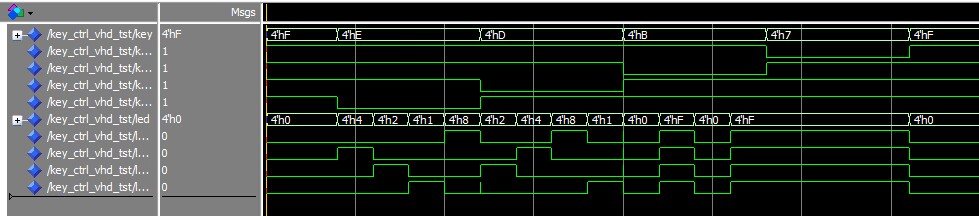


Figure 14: Simulation result

The waveform displayed in the screenshot illustrates the simulation results of the key\_ctrl\_vhd\_tst testbench, highlighting how the system handles input from keys and controls the LEDs accordingly. In the simulation, each of the top four traces represents the state of a key, showing a clear sequence of activation (state changing from '1' to '0') and deactivation. This action simulates the pressing and releasing of each button. Correspondingly, the lower four traces represent the LED outputs, which react to the key presses. Each LED's state changes in sync with the related key press, indicating a direct and correct response from the system to the inputs.

This direct correlation between key activation and LED response validates the VHDL code's logic, demonstrating that the system processes inputs and produces outputs as designed. The minimal delay between key presses and LED reactions ensures the system's responsiveness meets the necessary performance criteria for real-time operations. The correctness of the LED responses to the respective key presses confirms that the control logic is both effective and efficiently implemented within the system. This simulation verifies that the designed system behaves as expected, providing confidence in its functionality for further development stages or final deployment in hardware applications.

**2.3 Program the VHDL code to the FPGA and test its function by pressing the corresponding buttons**



Figure 15: Assign nodes to the FPGA pins

The image displays the process of programming the VHDL code into an FPGA and setting up the node assignments for testing the functionality of the design by pressing corresponding buttons. This setup is crucial for ensuring that each input and output is correctly mapped to specific pins on the FPGA, which aligns with the physical hardware configuration.

In the table shown, each node (representing inputs like keys and outputs like LEDs) is assigned to a specific pin on the FPGA. The assignment details include the direction (input or output), the specific pin location (like PIN\_M16, PIN\_E15), and the electrical properties such as the I/O standard set at 2.5V, current strength at 8mA, and slew rate configurations. This detailed mapping ensures that the VHDL design's logical connections are perfectly aligned with the physical pin layout on the FPGA, facilitating accurate and reliable function when the FPGA is subjected to real-world use.

For instance, the keys (key[0] to key[3]) are mapped to pins like PIN\_F16 through PIN\_M16, ensuring that when these physical buttons are pressed, the corresponding signals are correctly read by the FPGA. Similarly, outputs for the LEDs (led[0] to led[3]) are mapped to different pins like PIN\_D11 and PIN\_F9, ensuring that the output signals drive the LEDs correctly. This pin assignment is integral to the system's functionality, as it ensures that the inputs and outputs interact with the correct components of the integrated circuit, thus validating the entire design's operation through physical tests. This setup is not only a testament to the thoroughness of the FPGA programming process but also crucial for field testing where the FPGA's response to real interactions is observed and analyzed.

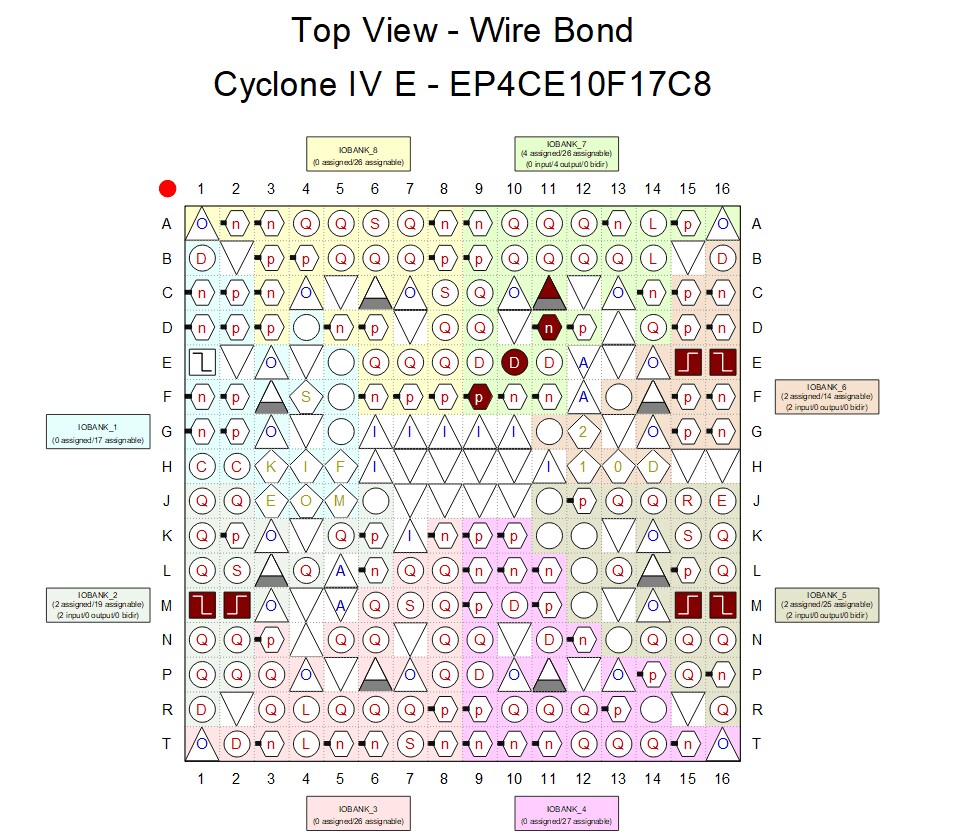


Figure 16: FPGA pins top view

The image presents a top view of the pin layout for a Cyclone IV E FPGA, specifically the EP4CE10F17C8 model. This diagram is essential for hardware developers as it clearly illustrates the distribution and function of each pin across the FPGA's physical surface. Each pin is color-coded and marked with symbols to indicate its specific function, such as input, output, power, or ground, and is further categorized into various I/O banks, each optimized for specific electrical characteristics and functions. This layout aids in effectively mapping and assigning the VHDL code's signals to the correct physical pins, ensuring that the programmed logic corresponds accurately to the FPGA's architecture. Such diagrams are crucial during the hardware setup phase to facilitate accurate connections and functional testing of the FPGA-based designs, allowing developers to verify the correct implementation of their designs in real-world applications.

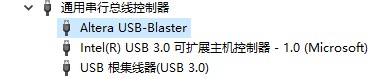


Figure 17: Altera USB-Blaster

Next, the Altera USB-Blaster is successfully recognized, indicating the system is ready for FPGA programming and configuration.

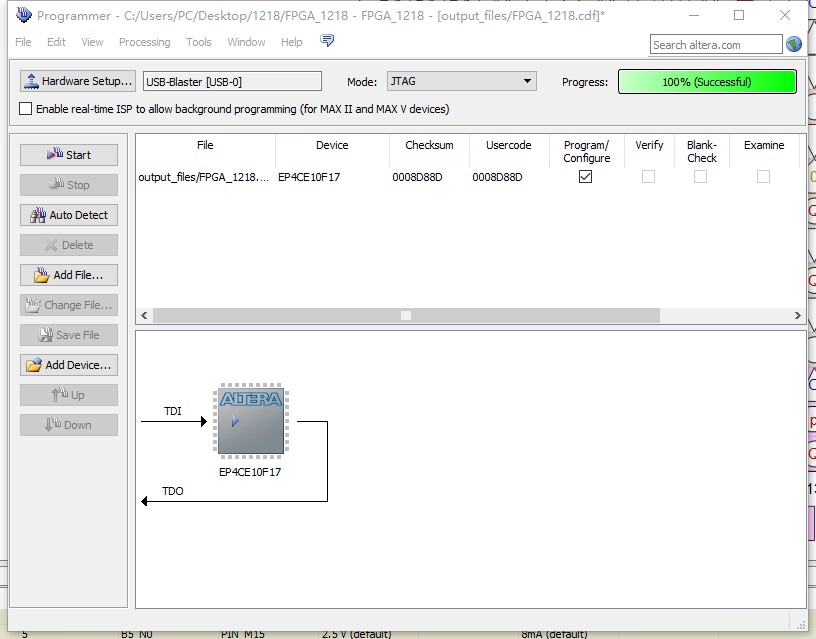
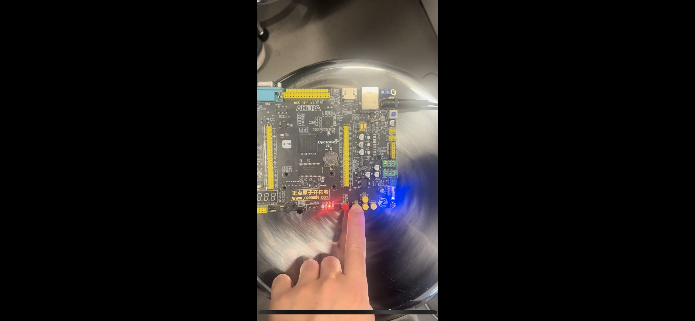


Figure 18: Loading the files into FPGA

Next, the FPGA programming process is completed successfully with 100% progress, confirming the configuration file has been loaded into the device via JTAG.

**2.4 FPGA results**

****

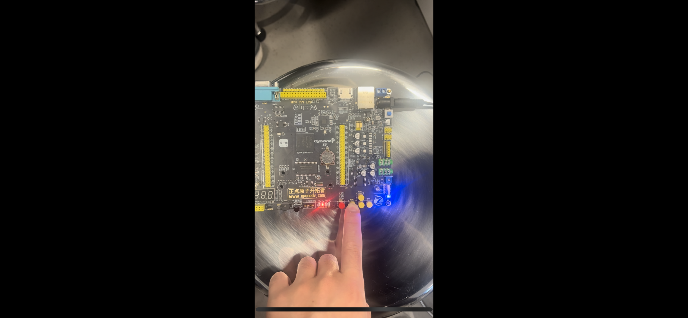
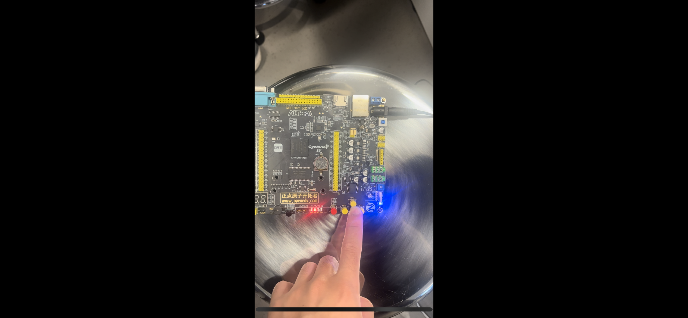
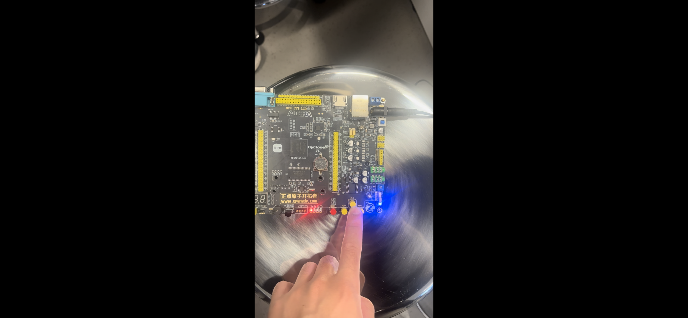
****

Figure 19: FPGA results of pressing the left button

**Button 1 (Left): LEDs light up sequentially from right to left**

According to the code, when Button 1 (key[0]) is pressed, the value of led\_control is updated in a state machine manner. With each update, the LEDs light up sequentially from right to left (e.g., 0001 -> 0010 -> 0100 -> 1000). This behavior is implemented using a case statement, where the current state of led\_control determines the next state. The observed behavior of the LEDs lighting up sequentially from right to left matches the logic designed in the code.



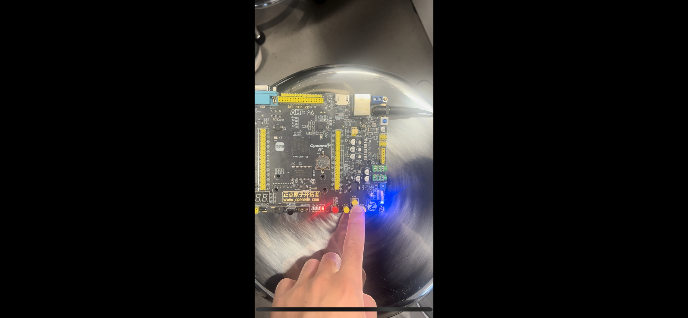
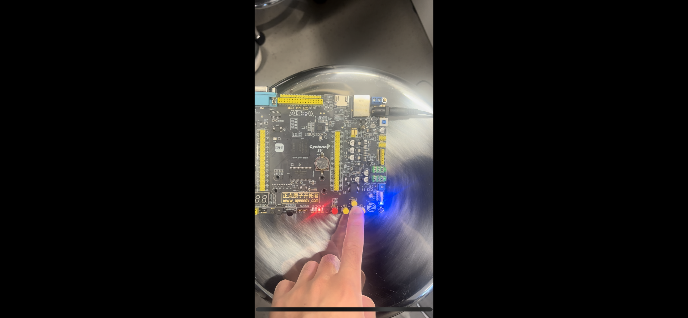
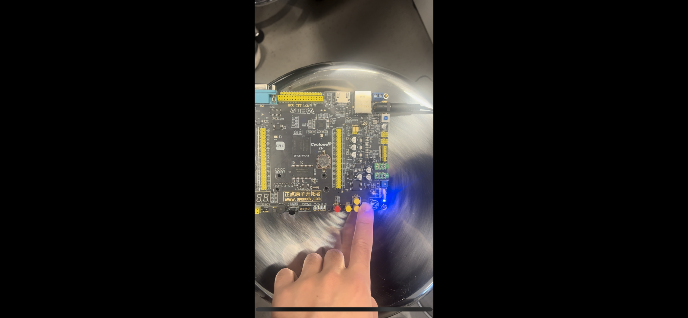
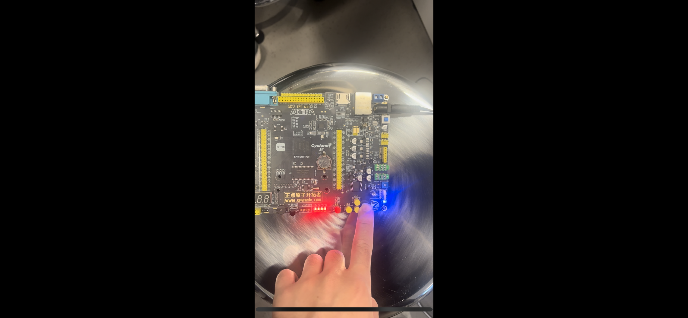


Figure 20: FPGA results of pressing the down button

**Button 2 (Down): LEDs light up sequentially from left to right**

When Button 2 (key[1]) is pressed, the code uses another case statement to update the led\_control value, but this time, the LEDs light up sequentially from left to right (e.g., 1000 -> 0100 -> 0010 -> 0001). This reverse direction of the sequence is a mirrored effect of Button 1, and the LED behavior observed during the operation perfectly matches the logic defined in the code.



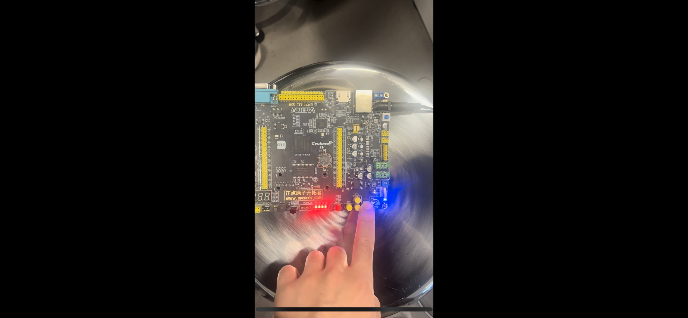


Figure 21: FPGA results of pressing the right button

**Button 3 (Right): LEDs cycle between all ON and all OFF**

Upon pressing Button 3 (key[2]), the code switches the led\_control value to a looping state, where all LEDs light up simultaneously and then turn off simultaneously (e.g., 1111 -> 0000 -> 1111). This is achieved by directly setting led\_control to either all ON or all OFF in the case statement, without shifting operations. The observed result, where the LEDs alternately light up and turn off in sync, matches the intended logic in the code.

****

Figure 21: FPGA results of pressing the up button

**Button 4 (Up): All LEDs remain ON**

When Button 4 (key[3]) is pressed, the code directly sets led\_control to 1111, which causes all LEDs to remain lit. This implementation does not rely on a state machine or looping logic but instead fixes the LED state to always ON. The observed behavior, where all LEDs remain fully lit, aligns perfectly with the straightforward logic implemented in the code.

**III. Analysis and discussion**

The overall experiment involves the complete process of designing, testing, and implementing an LED control system using VHDL, from writing the code to deploying it on an FPGA. Initially, the VHDL code was developed to describe the functionality of the LED controller. This code utilized a modular approach, defining processes to handle the clock signal, input key presses, and LED output logic. Each button's function was implemented to control the LEDs in distinct patterns, such as sequential lighting from right to left, left to right, simultaneous on-off cycles, and constant illumination. The logical design was organized and efficient, ensuring proper interaction between the input and output signals.

Following the code development, a testbench was written to simulate the behavior of the design. The testbench included signal declarations, clock generation, and sequential testing of all button presses to verify the design’s logic. By simulating each scenario in ModelSim, the expected outputs were observed in the waveform, which corresponded precisely to the intended LED behavior. The testbench simulation allowed for comprehensive debugging and validation of the VHDL code before implementing it on hardware. This step proved essential to ensure the design met all functional requirements without hardware-related variables interfering during the testing phase.

Finally, the design was deployed onto an FPGA for hardware testing. The physical setup involved mapping the input keys and LED outputs to specific FPGA pins using the Pin Planner, ensuring proper correspondence between the design and the board’s hardware configuration. After successful programming using the Altera USB-Blaster, the system was tested by pressing each button on the FPGA. The LEDs responded as designed, demonstrating the various lighting patterns for each button press. The sequential lighting, simultaneous blinking, and constant illumination behaviors observed on the FPGA matched the simulated results, confirming the correctness of the VHDL design.

This experiment highlights the effectiveness of combining VHDL coding, simulation, and FPGA implementation in digital design. The iterative process of coding, simulating, and hardware testing ensures that potential issues are identified and resolved in a systematic manner. The testbench simulation provided a crucial intermediate step for debugging and validating the logic before deploying it on hardware. Furthermore, the FPGA implementation verified the practical functionality of the design, showcasing the importance of both simulation and real-world testing in FPGA-based development workflows.

**VI. Other attachments**

**The RTL block view of the LED controller**

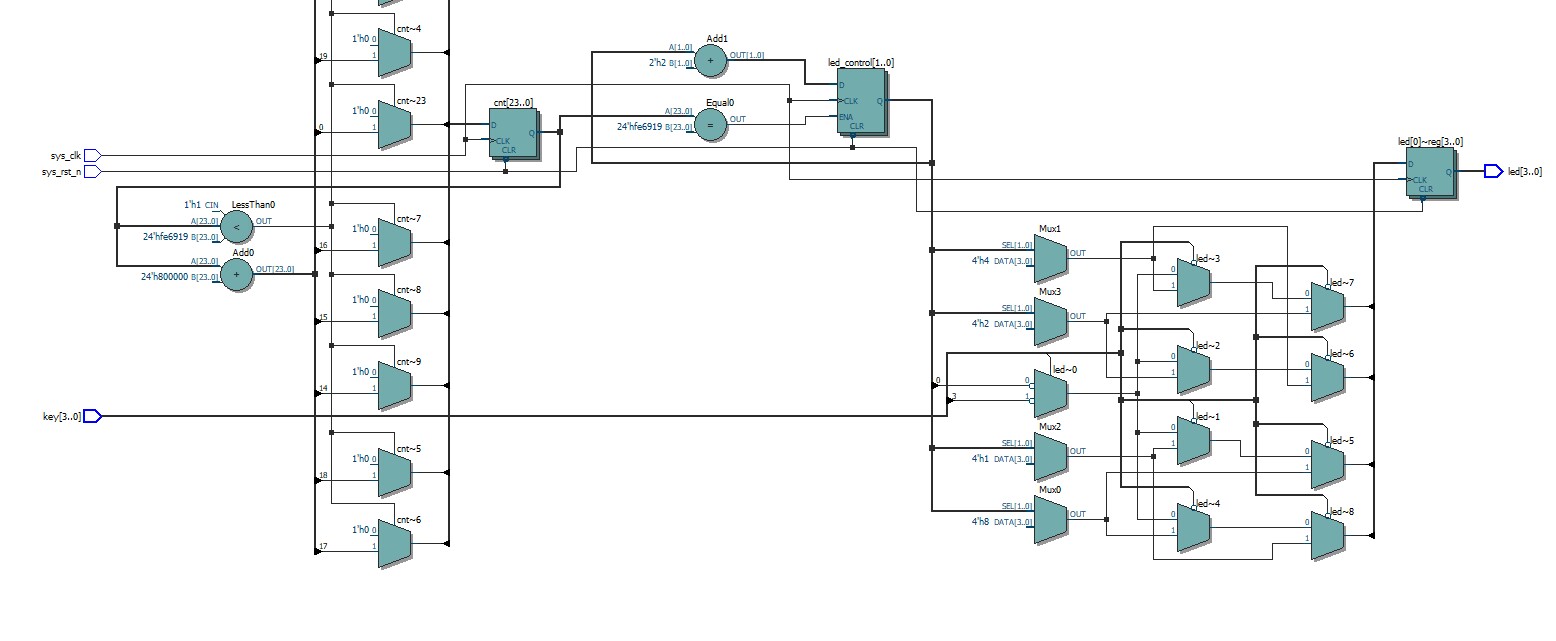


Figure 22: The RTL block view of LED controller

The RTL (Register Transfer Level) view provides a graphical representation of the synthesized VHDL code, showcasing how the design's logical components are implemented in hardware. In this RTL view, the major functional blocks, including flip-flops, multiplexers, comparators, and logic gates, are connected to realize the LED control system described in the VHDL code.

Starting from the inputs, the sys\_clk and sys\_rst\_n signals are routed into clock and reset components, ensuring synchronized operations and proper initialization of the design. The key inputs are used to control the behavior of the system, and their signals are passed through decision-making logic (such as multiplexers and combinational gates) to determine the appropriate LED control patterns.

The counter is a central component in this design, represented by a set of flip-flops or registers, which increment based on the clock signal and are reset when specific conditions are met. The counter output feeds into comparators, which evaluate the counter value against predefined thresholds to control the LED output patterns. For example, conditions like sequential lighting or simultaneous blinking are determined by the logic connected to the counter.

The LED control signals (led[0] to led[3]) are generated by a combination of multiplexers and logic gates. These components implement the case statements in the VHDL code, allowing the design to toggle LED states based on the pressed keys. Each multiplexer selects the appropriate state for the LEDs depending on the counter value and input conditions.