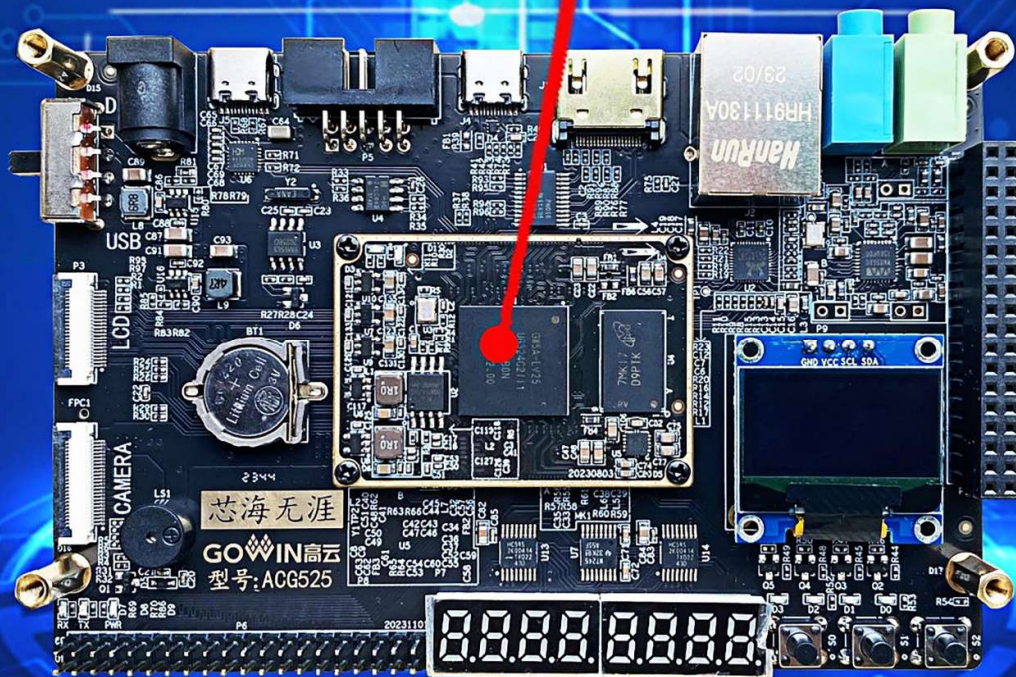
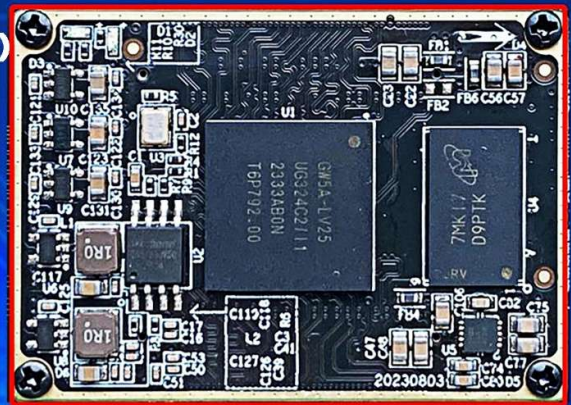


FPGA ACG525

High Quality China FPGAs Chip Instruction Book - 13 Chapters Basic

- **FLEXIBLE HARDWARE
(CORE BOARD + CARRIER BOARD)**
- **EXTENSIVE DOCUMENTATION
(PDF 1,000+ PAGES)**
- **TRAINING SUPPORT
(MANY BASIC EXAMPLES)**
- **REAL-WORLD APPLICATIONS
(ENTERPRISE CASE STUDIES)**



PREFACE

Field Programmable Gate Arrays, or FPGAs, have been available for nearly forty years. Over these forty years, FPGAs have evolved from having only 64 programmable logic elements initially to over ten million programmable logic elements today. They have progressed from basic logic glue functions to digital communications, industrial control, image processing, and now to data centers and artificial intelligence applications. From early simple programmable logic elements, they have incorporated memory and DSP units, then integrated high-performance memory controllers and high-speed transceivers, and now encompass high-performance processor systems. Modern FPGAs are no longer purely field-programmable gate arrays but are highly integrated hybrid-structure devices.

Domestic FPGA manufacturers started relatively late in the R&D and production of FPGA devices, mostly after 2010. However, precisely because of this, domestic manufacturers have had ample technology and experience to reference during FPGA device R&D. They can fully learn from the R&D and production experiences of major foreign manufacturers and, combining this with domestic application links, create products that are closer to the local application environment.

Currently, among domestic FPGAs, manufacturers that have released and mass-produced various FPGA devices include Gowin Semiconductor, Anlogic, PangoMicro, Intelligence Silicon, Efinix, Fudan Micro, and Hercules Micro. These manufacturers possess independent intellectual property rights in FPGA technology and provide their own developed full-flow FPGA development and verification software and hardware tools.

After nearly a decade of development, domestic FPGAs have become quite mature and capable. Domestic FPGAs have passed the early stages of infancy and stumbling and have entered a period of rapid development. Various domestic FPGA manufacturers are striving to exert force, releasing new series of devices based on 22nm and 28nm processes. The new series of devices integrate more functions and offer stronger performance. This indicates that domestic FPGA manufacturers have already secured a place in the field of mid-to-high-end FPGA devices. It is believed that in the next few years, domestic mid-to-high-end FPGA devices will increasingly appear in applications that could previously only be satisfied by FPGAs from major foreign manufacturers.

However, some practical problems facing domestic FPGAs cannot be ignored, the most important of which is the application ecosystem. Since domestic FPGAs started late, coupled with the lead of foreign FPGA chips in technology accumulation and application ecology, it is difficult for domestic FPGAs to be effectively promoted in the general market. Major customer application scenarios are singular, and small customers lack the ability to utilize them. Therefore, domestic FPGAs were previously mostly applied within one or two large customers or industries. In the general market, even if small and medium-sized users are willing to understand and try them, due to the immaturity of the application ecosystem and the scarcity of reference materials, it is difficult to effectively advance customer project progress.

Starting from the end of 2020, a chip shortage swept the globe. A large number of chips that were originally very cheap and common began to suffer from shortages and price increases. 6The

shortage and price increase of FPGA devices were particularly staggering, with the entire chip shortage and price hike lasting for over 2 years.

In a situation where foreign devices were unaffordable or unavailable, the public's attention once again focused on domestic devices, and everyone began to settle down to study and use domestic FPGAs. Only then did they discover that domestic FPGAs have not only caught up with foreign mid-to-low-end FPGA devices in terms of function and performance, but their functional innovations tailored to the domestic application environment were even more surprising. For example, for image and data acquisition fields, domestic FPGAs often package SDRAM or DDR2 memory together, so we do not need to design external memory chips on the circuit board ourselves, which can greatly reduce our hardware development difficulty. In addition, SOC FPGA devices integrating Cortex-M1, Cortex-M3, 8051, or RISC-V hard-core CPUs allow a system that originally required 2 chips to be completed with a single domestic FPGA. These innovative designs better meet the usage needs of domestic industrial customers.

During this domestic FPGA application migration, we deeply participated in the project migration work of multiple domestic FPGA chips for several customers. In the process of using domestic FPGAs, our biggest feeling was that domestic FPGA hardware functions are strong, but development materials are few and application cases are scarce. For beginners, getting started directly without teaching and reference materials is indeed quite difficult. Plus, in our customer groups, customers often encountered problems that were hard to solve when studying domestic FPGAs, calling on us to launch domestic FPGA teaching materials. Therefore, we determined to create a set of domestic FPGA teaching development boards, a set of domestic FPGA learning kits that beginners can refer to during the introductory and advanced stages.

Coinciding with this time, Gowin Semiconductor invited us to join the Gowin FPGA University Program and become an official partner to jointly develop a teaching development board based on Gowin Semiconductor's newly launched 22nm Arora V series FPGA chip. After fully evaluating the performance and functions of Gowin Semiconductor's devices, as well as the EDA software, we found this to be a combination very suitable for Chinese people to learn FPGA. The software functions are rich and complete, the operation is simple and clear, IP resources are abundant, and official documentation is detailed. So we hit it off and immediately launched the project.

Lasting 9 months, with the concerted cooperation of Gowin Semiconductor's original factory FAEs, distributor FAEs, and nearly ten people from the Xiaomeige FPGA team, the ACG525 teaching development board was successfully listed.

The ACG525 development board has abundant hardware resources, perfect teaching materials, and comprehensive routine source codes. Moreover, both hardware and software materials are fully open source. It is your high-quality partner for industrial product pre-research, scientific research project development, and learning FPGA from scratch.

Next, let us embark on the study tour of Gowin FPGA together.

Chapter 1: Gowin Software Installation Guide	7
1.2 Installation Step Description	7
1.3 Apply for License	13
Chapter 2: Develop a project	14
2.1 Start Gowin EDA	14
2.2 Creating A Project:	14
2.3 Adding Source Files	17
2.4 RTL Description & Analysis Process:	20
2.5 Analysis & Synthesis	20
2.8 Board level verification:	24
2.9 Store the program on FPGA Memory	26
2.10 Chapter Summary	27
Chapter 3: Gowin & Modelsim for simulation	29
3.1 Modelsim Compiling Gowin Device Libraries	29
3.2 Modelsim Simulation Verification Flow	35
3.3 Reflection and Summary	46
Chapter 4: Using Gowin Online Logic Analyzer	47
4.1 Introduction to Gowin Online Logic Analyzer	47
4.2 GAO File Configuration	47
4.3 Place and Route	58
4.4 Downloading fs File	58
4.5 Waveform Capture	58
4.6 Chapter Summary	61
Chapter 5 Sequential Logic Circuit Design: Counter	62
5.1 Counter Working Principle	62
5.2 Counter Verilog Implementation	63
5.3 Functional Simulation Verification	66
5.4 Board-Level Debugging and Verification	72
5.5 Reflection and Summary	80
Chapter 6: Serial Port Transmit Module Design and Verification	81
6.1 Asynchronous Serial Communication Principle and Circuit Design	81
6.2 UART Asynchronous Serial Communication Transmit Module Design and Implementation	86

6.3 Stimulus Creation and Simulation Testing	92
6.4 Serial Port Transmit Test Module	93
6.5 Board-Level Verification	95
6.6 Reflection and Summary	99
Chương 7 Serial Port Receive Module Design and Verification	100
7.1 Serial Port Receive Principle Analysis	100
7.2 UART Asynchronous Serial Communication Receive Module Design and Implementation	101
7.3 Simulation Testing	108
7.4 Serial Port Receive Test Module	109
7.5 Board-Level Verification	111
7.6 Reflection and Summary	114
Chapter 8: Serial Port Controlling LED Light	115
8.1 Design Analysis	115
8.2 Defining Serial Port Transmission Protocol Frame	116
8.4 Board-Level Verification	123
8.5 Summary	126
Chapter 9: Multi-Byte Serial Port Transceiver Module Design and Verification	127
9.1 Multi-Byte Serial Port Transmission Principle and Design	127
9.1.1 Multi-Byte Serial Port Transmission Principle	128
9.2 Multi-Byte Serial Port Reception Principle and Design	133
9.3 Creating Top-Level Encapsulation for Testing	140
9.4 Board-Level Verification	141
9.5 Summary	143
Chapter 10: State Machine Design Example	144
10.1 State Machine Working Principle	144
10.2 String Detection State Machine Implementation	146
10.3 Stimulus Creation and Simulation Testing	149
10.4 String Detection Board-Level Testing	151
10.5 Summary	153
Chapter 11: Independent Key Debounce Design and Verification	155
11.1 Key Physical Structure and Circuit Design	155
11.2 Hardware Circuit Implementation of Key Debounce	156

11.3 State Machine Implementation of Key Debounce.....	159
11.4 Testbench Design and Simulation Testing.....	164
11.5 Summary.....	167
Chapter 12: Basics of Modular Design: Adder/Subtractor Counter.....	168
12.1 Module Function Partitioning	168
12.2 Stimulus Creation and Simulation Testing	170
12.3 Board-Level Verification	172
12.4 Common Problem Explanation.....	174
12.5 Summary.....	174
Chapter 13: 8-Bit 7-Segment Digital Tube Driver Design and Verification	175
13.1 Digital Tube Driver Principle	175
13.2 Digital Tube Dynamic Scanning Driver Design.....	177
13.3 74HC595 Driver Module Design.....	181
13.4 Digital Tube Display Module Simulation Testing	184
13.5 Board-Level Debugging and Verification	184
13.6 Common Problem Explanation.....	186
13.7 Summary.....	186

Chapter 1: Gowin Software Installation Guide

1.1 Download Gowin Designer EDA

You can obtain the Gowin software through various channels, including but not limited to the Gowin official website and download links provided by various vendors. The Gowin official website link is as follows:

<http://www.gowinsemi.com.cn>

After entering the official website, find the "Gowin Yunyuan Software" column under the "Support" or "Developer Zone" section to see the download link for the Gowin software. Click to download the installation package, which includes Commercial and Educational versions. The Commercial version requires applying for a license, while the Educational version does not. We use the Commercial version of the software in our development process, and the following software installation instructions will be based on this.

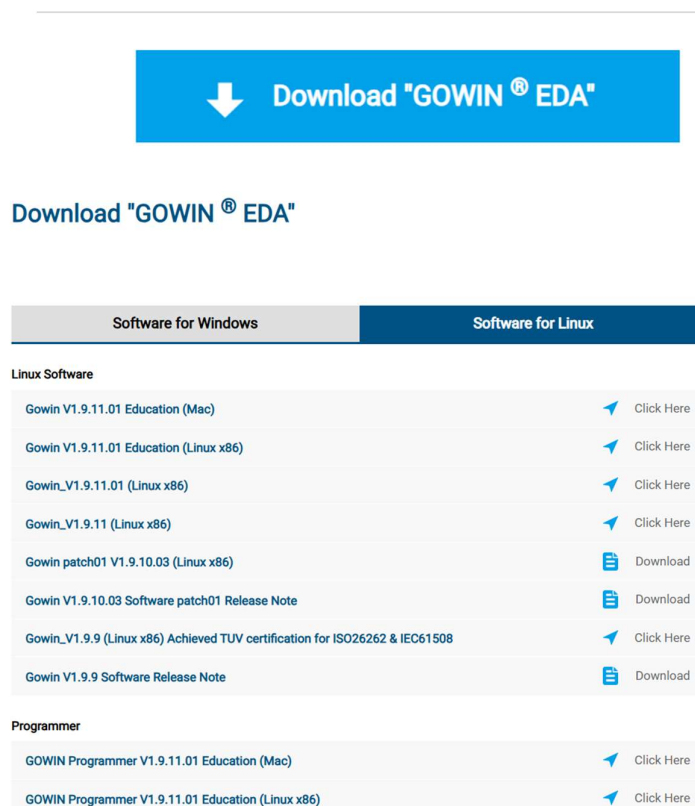


Figure 1-1 Download EDA Package

1.2 Installation Step Description

During installation, first unzip the Gowin software installation package to a path that does not contain Chinese characters, spaces, or special characters. After decompression, the files will

appear as shown in Figure 1-1 below



Figure 1-2 Gowin Installation Package

Double-click the .exe file in the figure, and the interface shown in Figure 1-4 below will appear. Click "Next" to proceed to the next step.

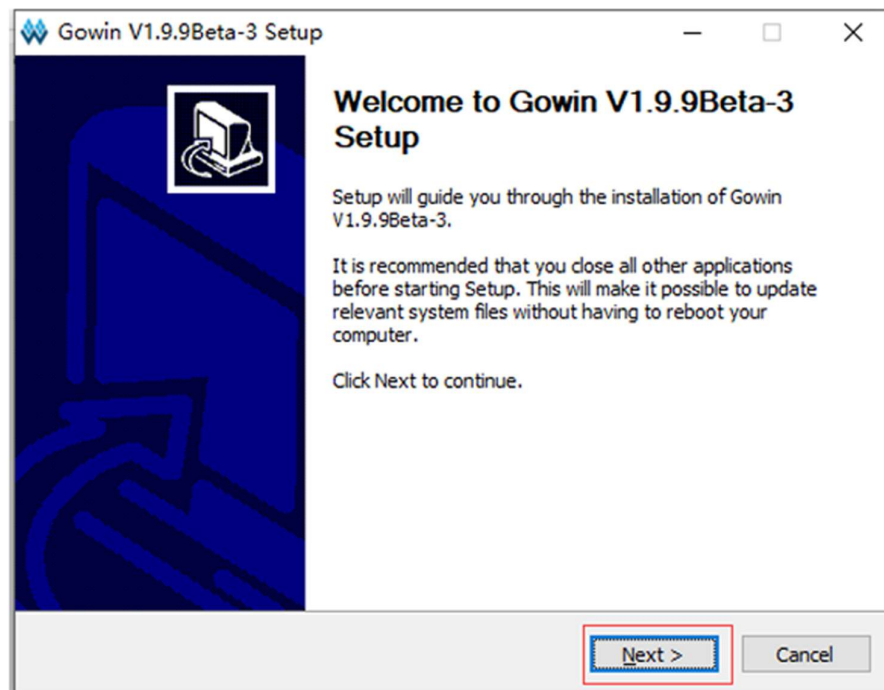
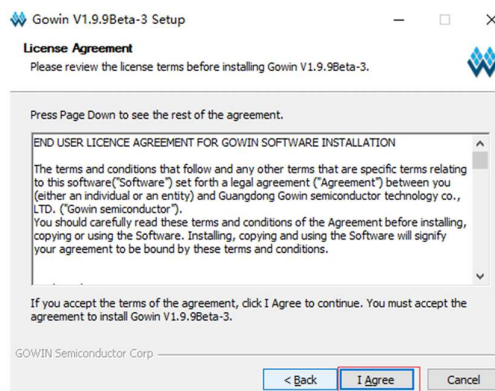


Figure 1-4 Installation Startup Window

Afterwards, the interface shown in Figure 1-5 below will pop up. Select "I Agree" in the popped-up interface.



Hình 1-5 Installation Guide

After clicking "I Agree", the interface shown in Figure 1-4 below will pop up. Click "Next" to proceed to the next step.

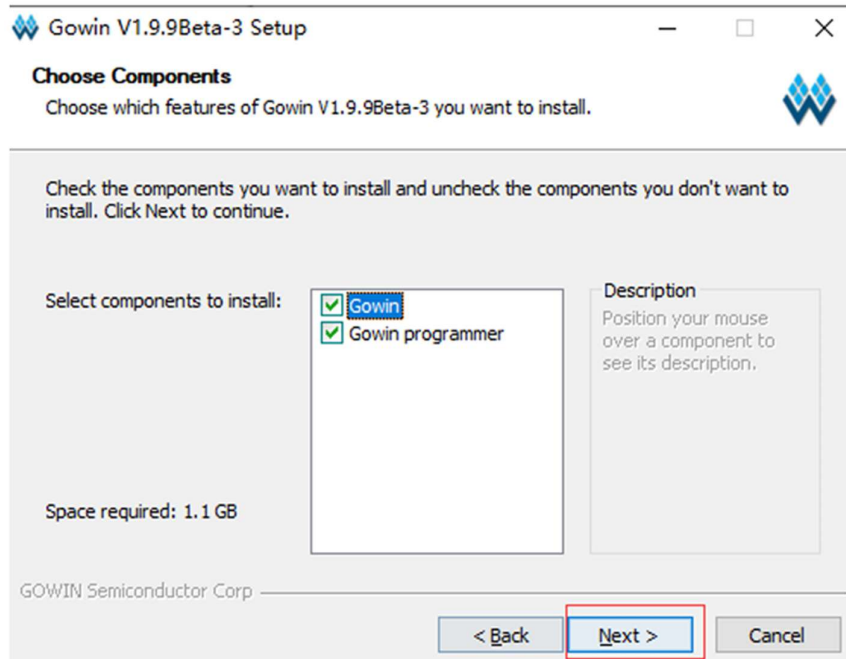


Figure 1-6 Component Selection Interface

Then Window 1-7 be shown, to select the path and Directory to in stall on PC Hard Driver, Click **Install**

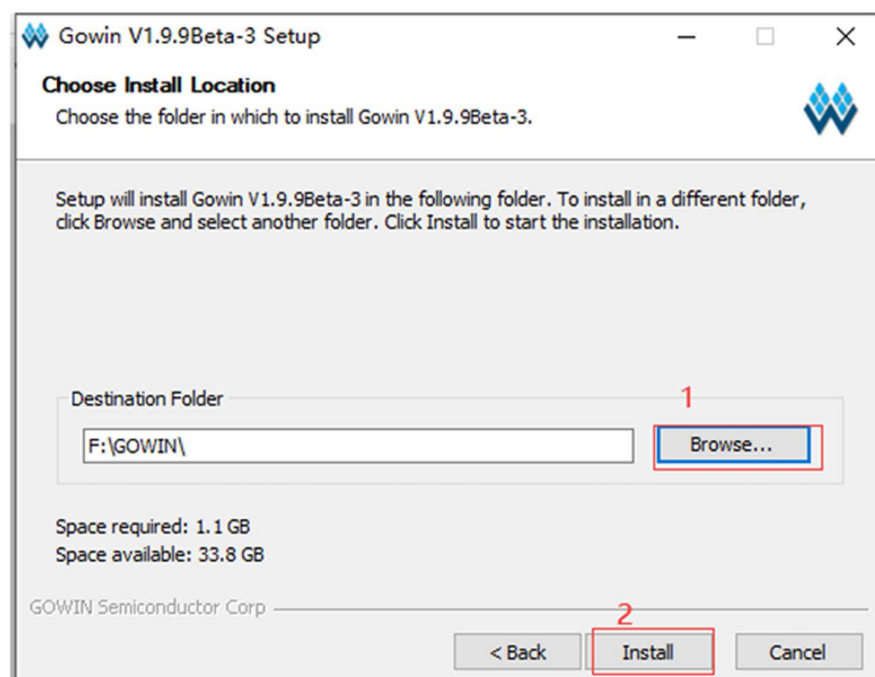
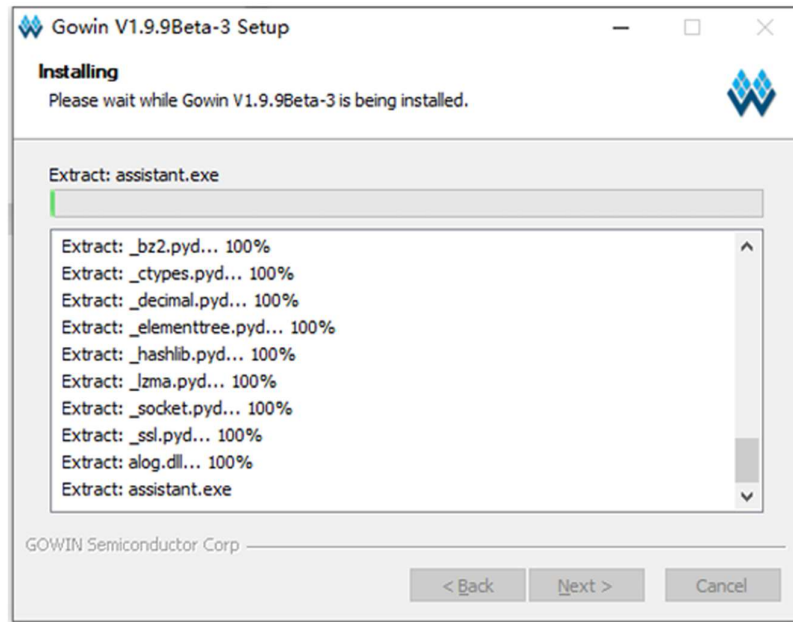


Figure 1-7 Set up the Path for installing

Then the interface shown in Figure 1-7 will appear. Click "Browse" to select the path where the software needs to be installed (note that the path cannot contain Chinese characters), and then click "Install" to start the installation.



Hình 1-8 Installing process

After the installation is complete, the interface shown in Figure 1-7 below will pop up. Click "Finish" to complete the installation of the Gowin software.

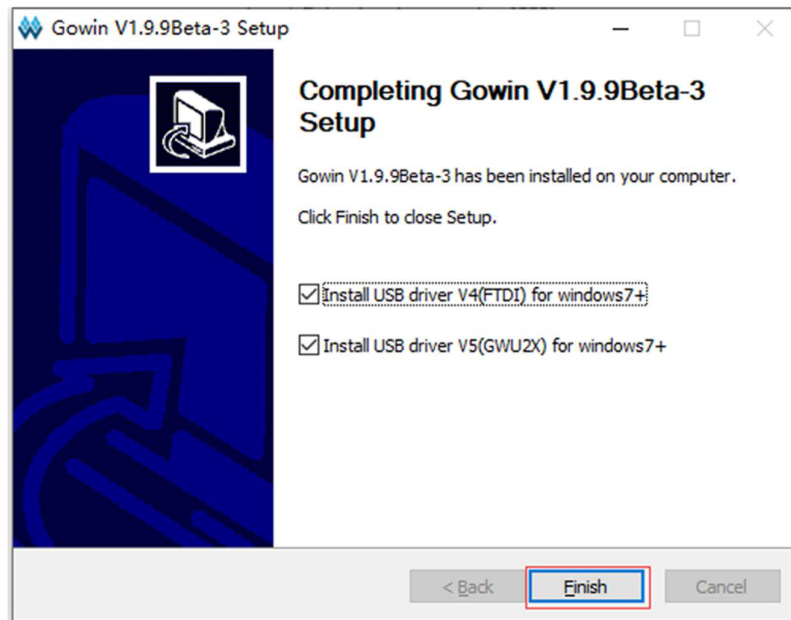


Figure 1-9 Installing Complete interface

After clicking Finish, the interface shown in Figure 1-10 below pops up. Click Browse to select the GWU2

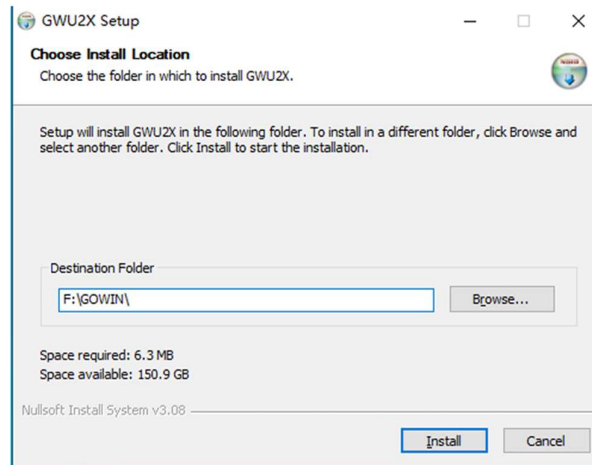


Figure 1-10 Install GWU2X driver

Wait for the installation to complete, then click Close, as shown in Figure 1-11 below:

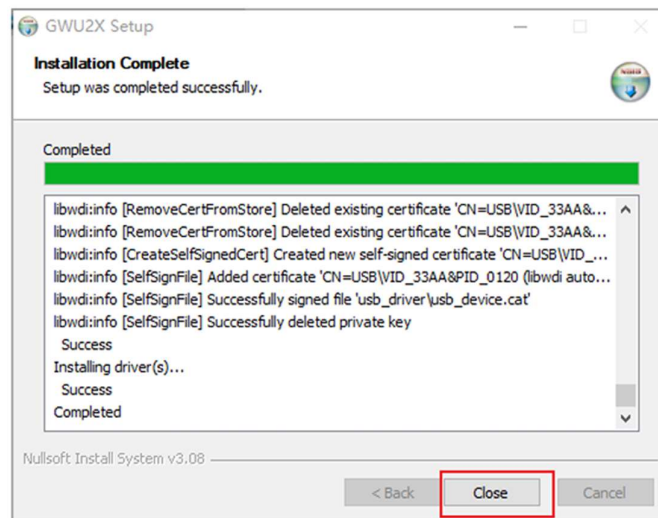


Figure 1-11 Install GWU2X complete

Then, we can see the driver installation startup interface shown in Figure 1-12 below

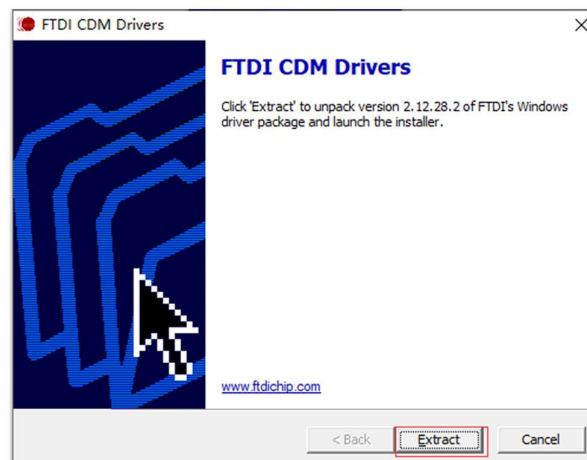


Figure 1-12 Install Driver for FTDI

Then the driver installation wizard will pop up, as shown in Figure 1-13 below. Click "Next"

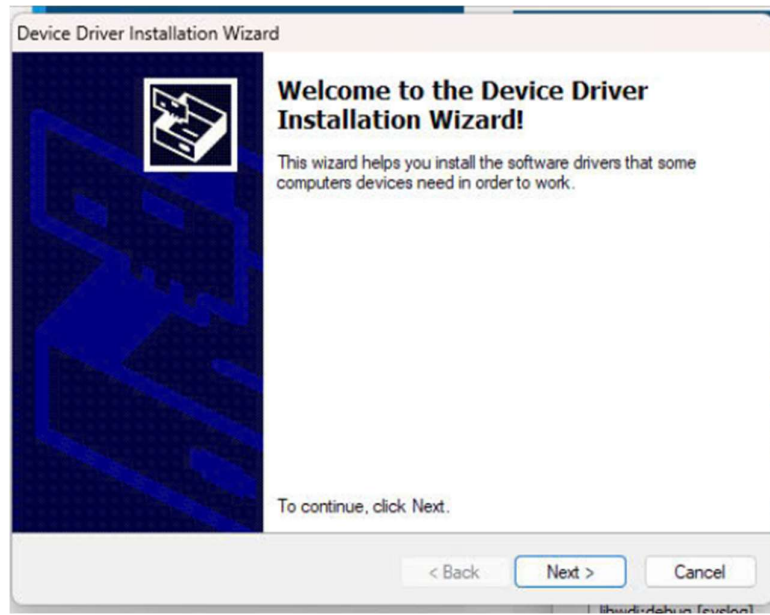


Figure 1-13 Program Installation Wizard Prompt Interface

Subsequently, the license agreement interface pops up, as shown in Figure 1-14. Click "I accept this agreement"

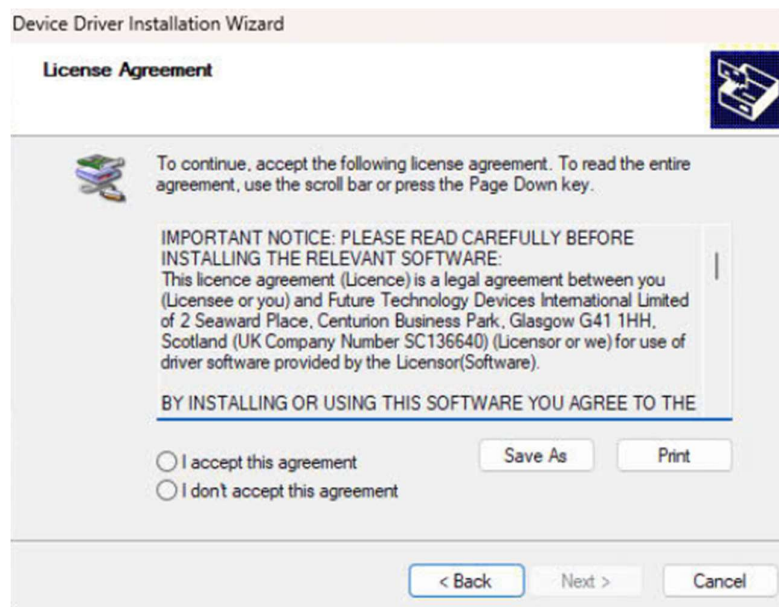


Figure 1-14 License Agreement Interface

Then wait for the installation to complete, as shown in Figure 1-15 below.

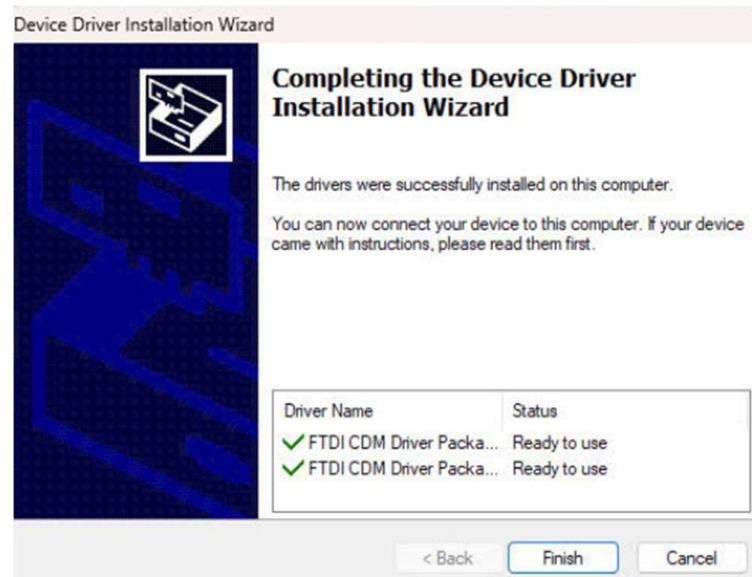


Figure 1-15 Waiting for GWU2X Installation to Complete

1.3 Apply for License

After the Gowin software is installed, there will be a shortcut to the Gowin software on the desktop. Double-click it, and the interface shown in Figure 1-16 below will pop up. You need to find the path of our License and associate it

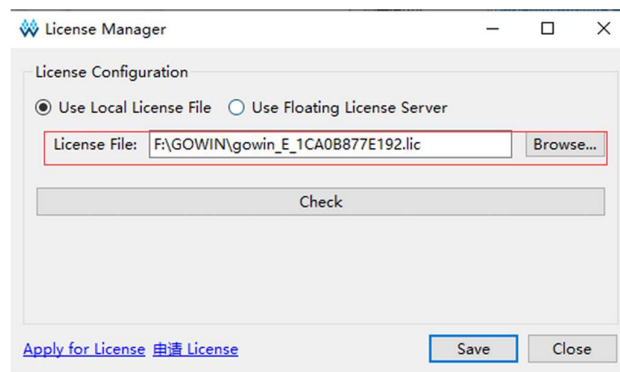


Figure 1-16 Importing License File in License Manager

After Checking and confirming there are no issues, click Save to save it. Afterward, the software can be opened and used.

Chapter 2: Develop a project

Project sample code |----ch2_led_flash

Chapter Overview

This chapter's experiment will guide users to become familiar with the basic development flow of the Gowin development software. The entire flow includes creating a project, analysis and synthesis, physical constraints, place and route, board-level verification, and program configuration.

2.1 Start Gowin EDA

After the Gowin software is installed, a shortcut will be created on your desktop, as shown in Figure 2-1 below. Double-click the



Figure 2-1 Gowin Desktop Shortcut

2.2 Creating A Project:

The steps to create a design project mainly include the following:

1. Start the Gowin Integrated Development Environment (IDE) and enter the Gowin startup interface, as shown below.

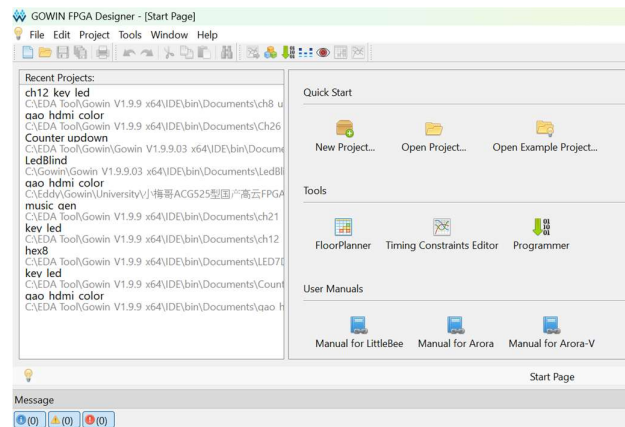


Figure 2-2 Gowin Designer start up window

Start up window have 4 parts:

- (1) Recent project
- (2) Quick start:
- (3) Tool: Including Floor Planner, Timing Constraints Editor, Programmer.
- (4) User Manual: Link to Gowin Website.

Click to **new project** in **Quick Start** to create new project.

2. New Project Wizard will open, displaying the New dialog box as shown in Figure 2-3 below. Click the OK button.

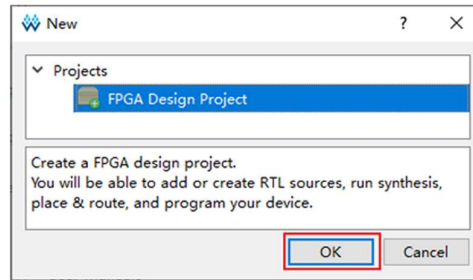


Figure 2-3 Gowin Create Project Navigation Interface

3. Enter the Project Name dialog box, as shown below. Fill in the project name and project path. Note that the project naming cannot contain Chinese characters or spaces, otherwise errors will occur in subsequent processes. The file path should not be too long; Windows systems limit path length to 260 characters, and Linux systems to 4096 characters. Determine whether to check "Use as default project location" according to the actual situation. If checked, the next time a project is created, it will default to the project path set this time. After setting, click Next.

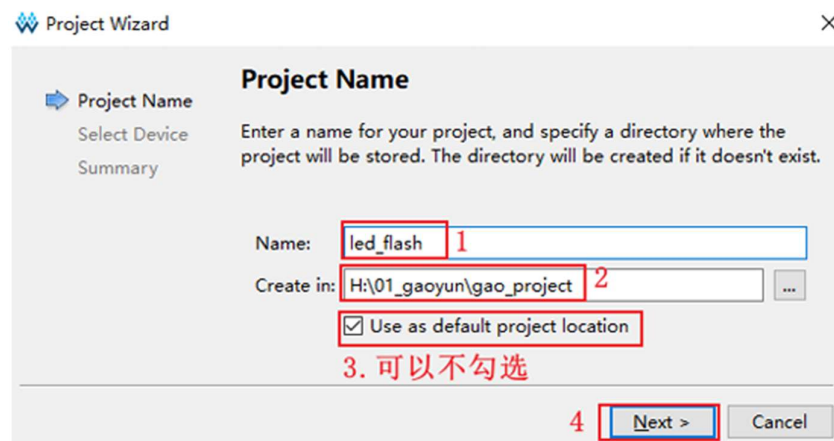


Figure 2-4 Gowin Create Project Name Interface

4. Subsequently, the Select Device dialog box appears, as shown in Figure 2-5 below. Here we need to select the specific FPGA chip model used in the design. Judge based on the series, package, speed grade, etc., of the FPGA used on the board. Users can also judge based on the markings on the FPGA chip. After selection, click **Next**

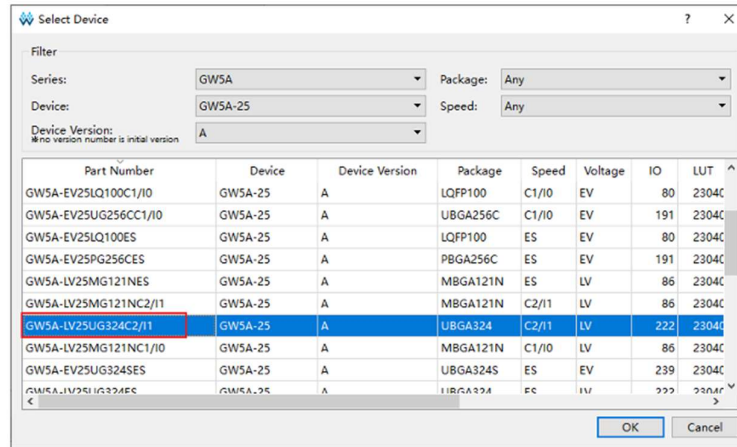


Figure 2-5 Gowin Select FPGA Device Type

- Finally, the Summary interface appears, as shown in Figure 2-6 below. Click Finish to complete the project creation.

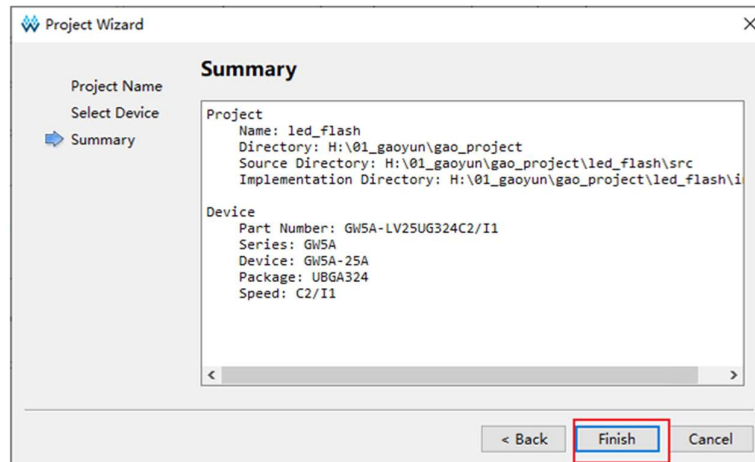


Figure 2-6 Project Creation Completion Interface

- Next, we will enter the Gowin project design interface, as shown in Figure 2-7 below. The main design interface mainly includes modules such as Design, Process, Hierarchy, and Console. The following will guide everyone to familiarize themselves with the Gowin tool design interface.

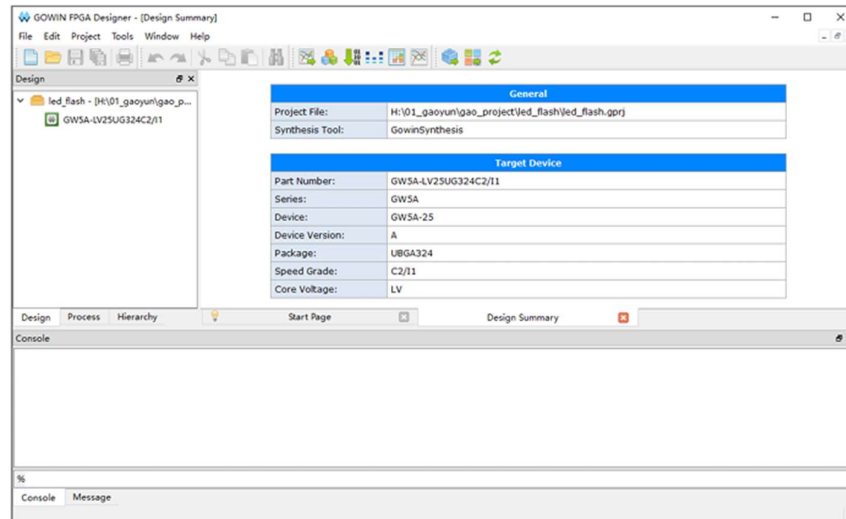


Figure 2-7 Gowin Project Design Interface

2.3 Adding Source Files

After completing the project creation using the Gowin tool, we can add source files to the project in Gowin. Next, we will demonstrate this process by creating a Verilog source file.

1. Click the icon on the toolbar or click File->New... in the menu bar sequentially. After clicking, the file type selection interface appears as shown below. Here we select the Verilog File type, and then click OK.

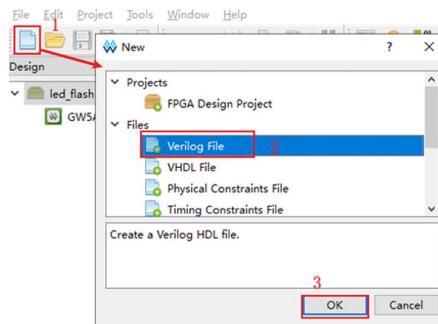


Figure 2-8 Adding a Verilog Source File

2. Subsequently, the New Verilog file interface appears. In this dialog box, name the file and choose the file save path. It is saved by default in the src folder of the project directory folder. Then check "Add to current project", as shown in Figure 2-9 below. Finally, click OK.

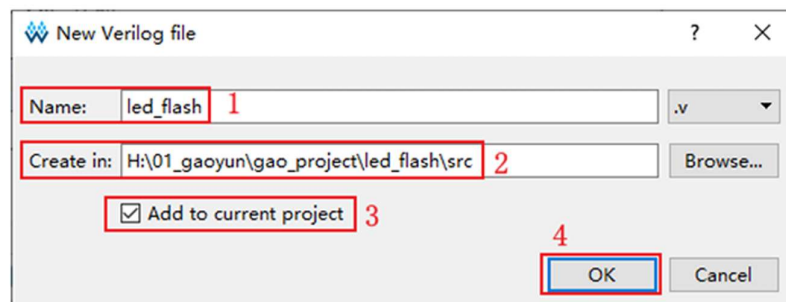


Figure 2-9 New Verilog file Configuration Interface

After the source file is added, we can see a new folder appeared in the Design box:

Verilog Files. Under this folder is the Verilog source file we just created, as shown in Figure 2-10 below.

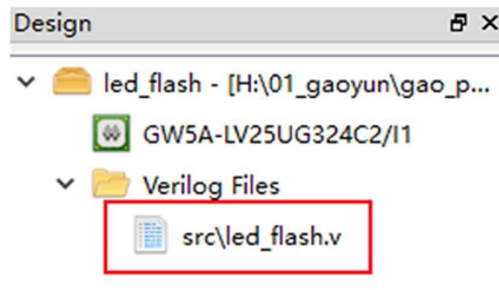
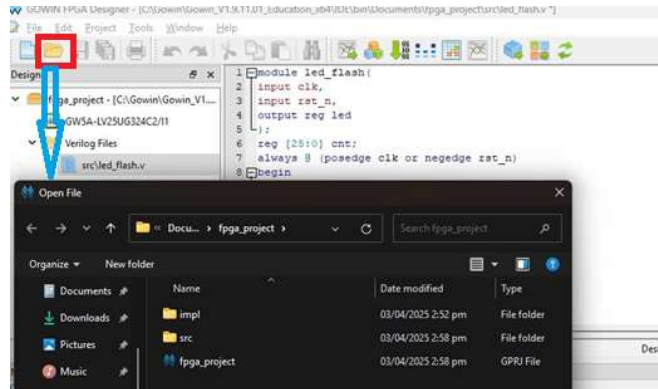


Figure 2-10 Schematic of Source File Addition Completion

3. Click the led_flash.v file and write the code for this experiment project. The function implemented in this experiment is to toggle the state of the LED light, which is achieved by designing a counter, as follows:

```
module led_flash(  
    input clk,  
    input rst_n,  
    output reg led  
);  
    reg [25:0] cnt;  
    always @ (posedge clk or negedge rst_n)  
    begin  
        if (!rst_n)  
            cnt <= 26'd0;  
        else if (cnt < 26'd49_999_999)  
            cnt <= cnt + 1'b1;  
        else  
            cnt <= 26'd0;  
    end  
    always @ (posedge clk or negedge rst_n)  
    begin  
        if (!rst_n)  
            led <= 1'b0;  
        else if (cnt == 26'd49_999_999)  
            led <= ~led;  
        else  
            led <= led;  
        end  
    endmodule
```

4. After editing is complete, press Ctrl+S to save the source file. The source file is now successfully added. The above description completes the creation of a new source file. If there is already a written Verilog source file, it can be added by clicking the icon in the toolbar. After clicking, the Open File dialog box will pop up. Then select the file to be added. Note that the file to be added should be in the project directory.



5.

Hình 2-11 adding Existing File

2.4 RTL Description & Analysis Process:

1.1.1 Introduction to RTL Description and Analysis Function

When we use Verilog language to describe a certain circuit, there may be grammatical or logical errors. Grammatical errors will be prompted by the Gowin software during compilation. However, logical errors cannot be detected by the Gowin software, such as connection errors between modules. The RTL description and analysis function allows viewing the RTL structure and syntax of the project, which enables analysis and correction of logical errors.

1.1.2 RTL Description and Analysis Process

The Gowin software provides users with the function to view RTL design analysis.

Click Tools->Schematic Viewer->RTL Design Viewer sequentially, and you can see the schematic netlist after the RTL description and analysis stage, as shown in Figure 2-12 below. The netlist is represented by logic symbols unrelated to FPGA underlying components. Readers can see the schematic after the synthesis stage in the subsequent process. The logic components in the schematic after the synthesis stage are constructed from the underlying components in the FPGA.

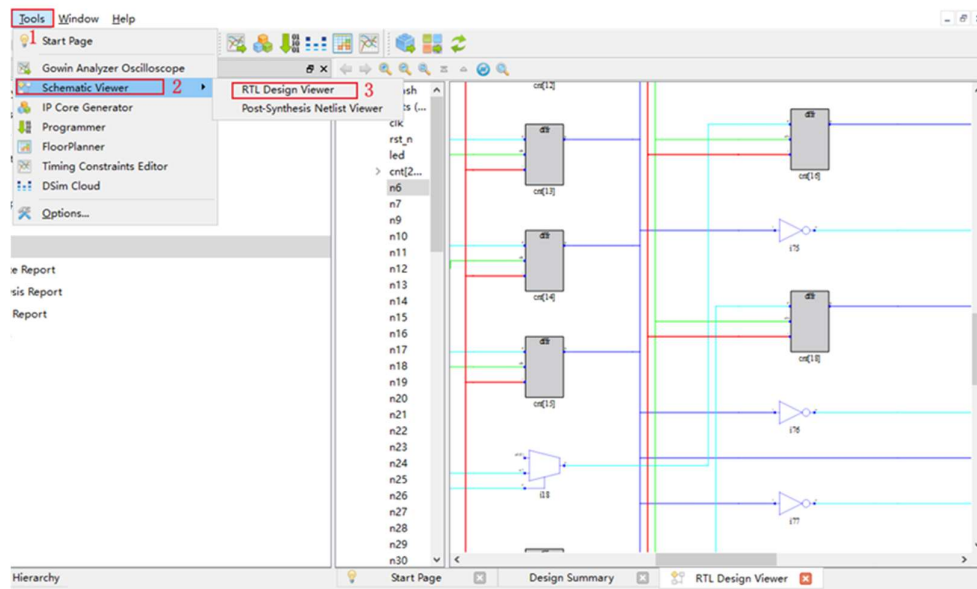


Figure 2-12 Viewing RTL Netlist

2.5 Analysis & Synthesis

We can perform analysis and synthesis by clicking Process->Synthesize sequentially, or directly clicking the icon in the toolbar. After analysis and synthesis, a circuit netlist based on FPGA underlying resources is generated, as shown in Figure 2-13 below. If no errors appear, the generation is successful. If errors occur, the user should modify them according to the prompts.

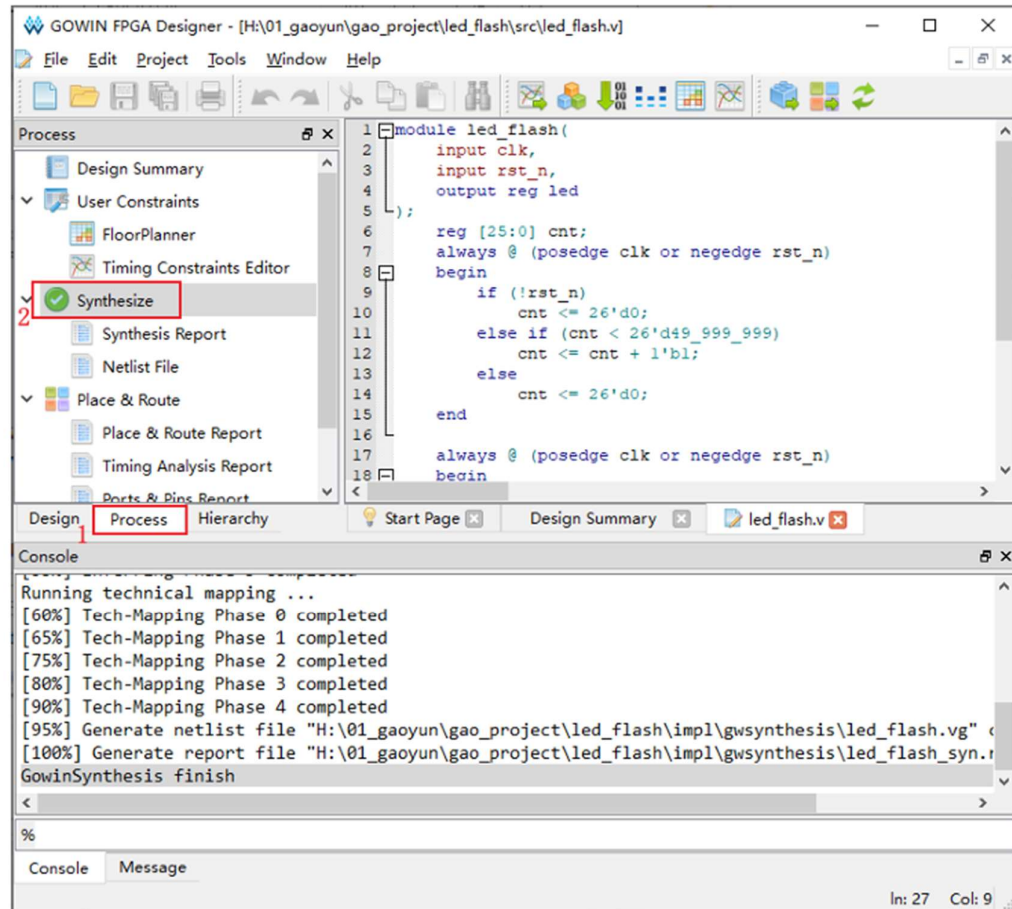


Figure 2-13 Synthesis

After synthesis is complete, the Hierarchy window will automatically display the resource information in the current project design, as shown in Figure 2-14 below.

Hierarchy							
- + Update							
Unit	File	Register	LUT	ALU	DSP	BSRAM	SSRAM
led_flash	src\led_flash.v	27 (27)	66 (66)	0 (0)	0 (0)	0 (0)	0 (0)

Design Process **Hierarchy**

Hình 2-13 Figure 2-14 Resource Information Display Window

In the Hierarchy window, place the mouse over the led_flash.v file, right-click the file, and you can set this file as the top-level file, as shown in Figure 2-15 below.

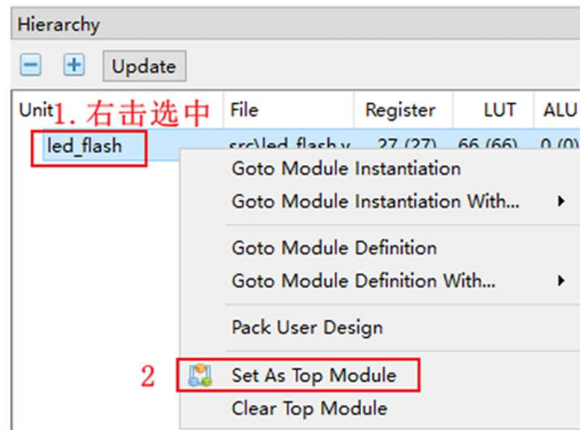


Figure 2-15 Set Top Module

After setting is complete, the file name will become bold. Then re-analyze and synthesize, and you can see the **Hierarchy** window as shown in Figure 2-16 below.



Hình 2-16 Thiết lập Top Module thành công

Physical Constraint We can perform physical constraints by clicking 2.6

Process->FloorPlanner, or directly clicking the icon in the toolbar, as shown in Figure 2-17 below. Figure 2-17 Physical Constraint Operation Method, Then the dialog box shown in Figure 2-18 below pops up. Since we have not created a new .cst file to store physical constraints, a dialog box will pop up asking if we need to create a .cst file. Click OK here.

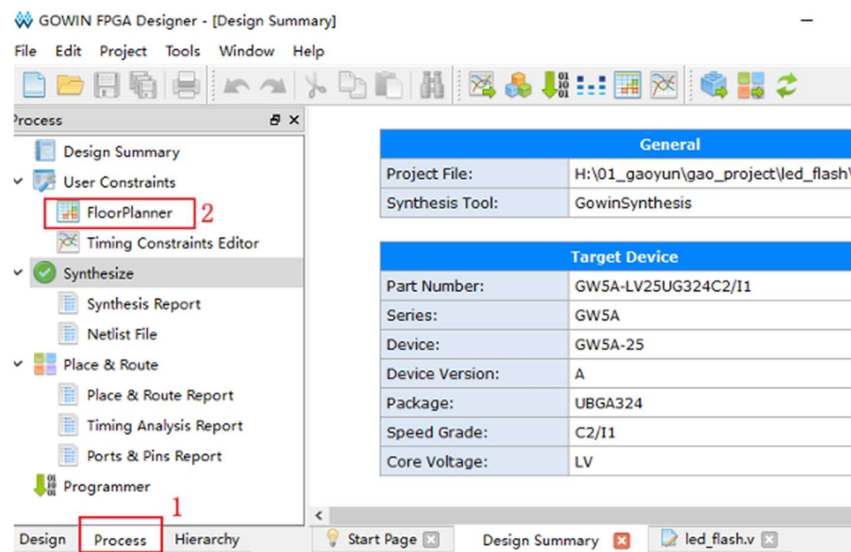
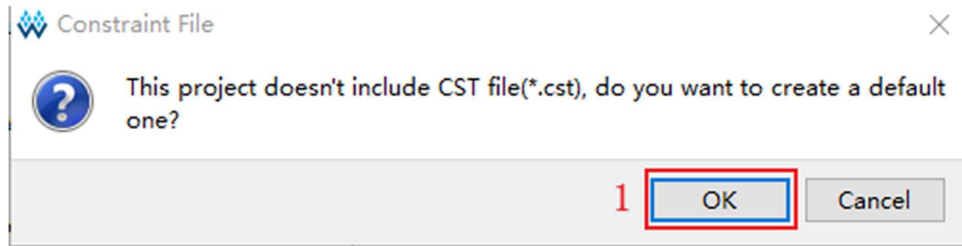


Figure 2-17 Physical Constraint Operation Method



Hình 2-18 xác nhận tạo CTS file (*.cts)

Enter the FloorPlanner interface, click I/O Constraints to perform I/O constraints, and then assign pins and level standards according to your own board, as shown in Figure 2-19 below

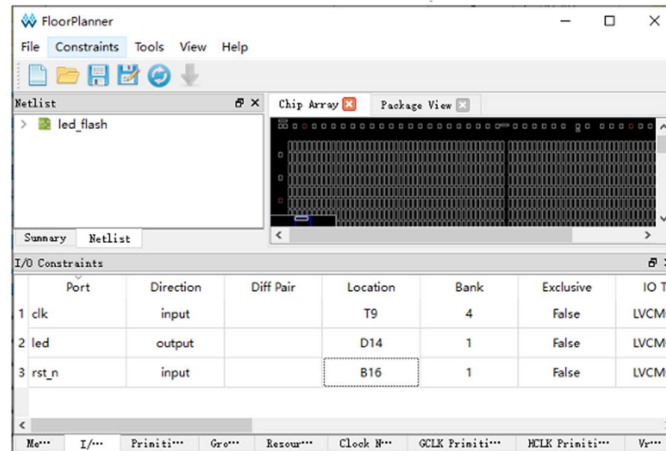
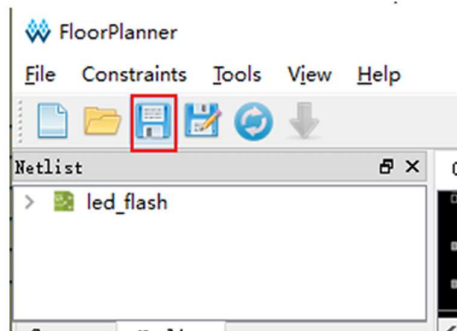


Figure 2-19 I/O Constraint Interface

After constraints are complete, save the file as figure 2-20 bellow.



Hình 2-20 Save constraint File

After constraints are completed, click Design->led_flash.cst to view the file content, as shown in.

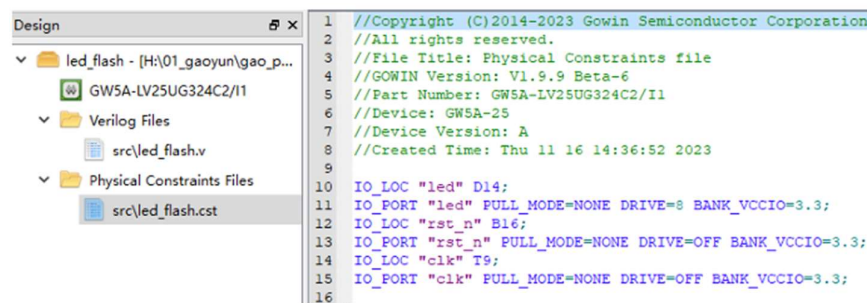


Figure 2-21 View content of Physical Constraint

From the figure above, it can be seen that the content in the file matches what we set in **FloorPlanner**, indicating that our physical constraints are successful

2.7 Place & Route

We perform place and route by clicking **Process -> Place & Route**. After successful place and route, we can see **"Bitstream Generate Complete"**, representing that the fs file has been generated successfully, as shown in Figure 2-22 below. If it fails, the user should modify it according to the prompt information given in the Console.

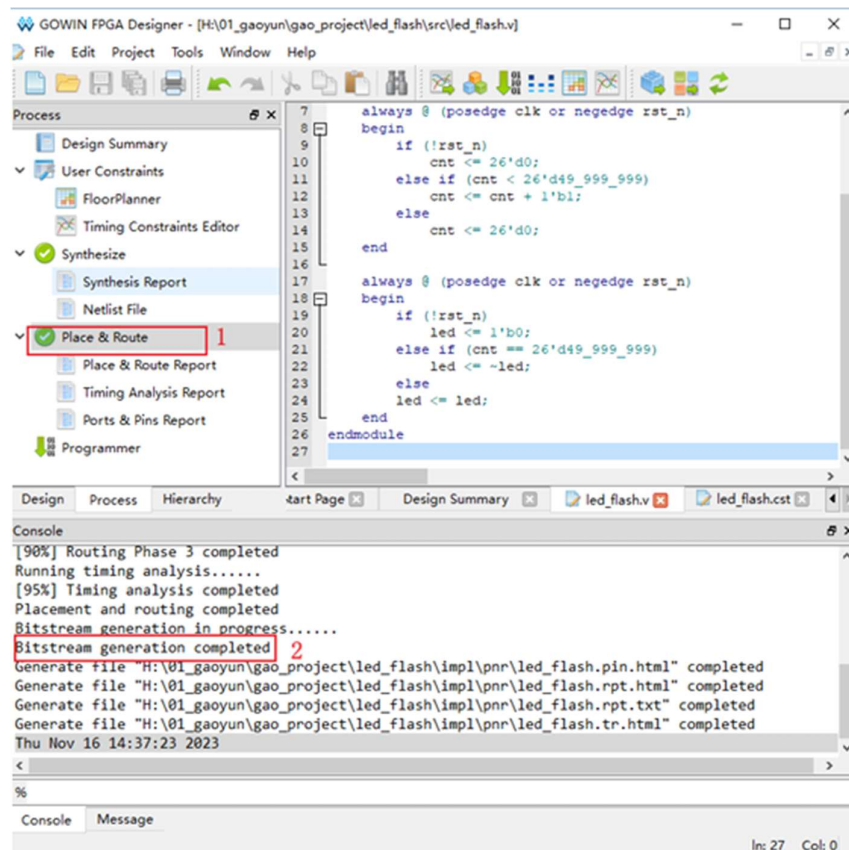


Figure 2-22 Place & Route Successful

2.8 Board level verification:

We generated the fs file required for the experiment through the previous steps. Next, we need to download the fs file to the FPGA chip to complete the FPGA configuration.

1. Connect the downloader and power supply to the development board in sequence, as shown in Figure 2-23 below.



Hình 2-23 Hardware Connection

2. Then power on the development board, click Process->Program Device. The interface shown in Figure 2-24 pops up, indicating that the device has been detected, then click save. If it displays No USB Cable Connection, please check your hardware connection.

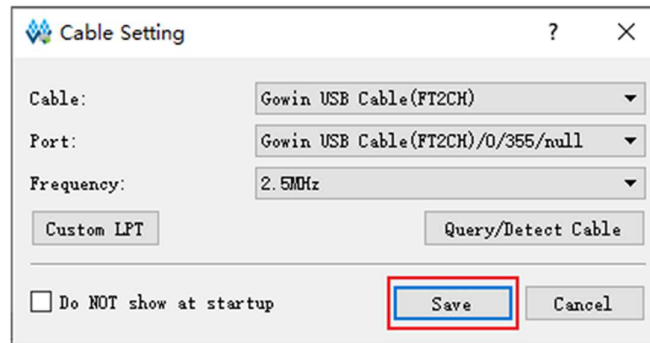
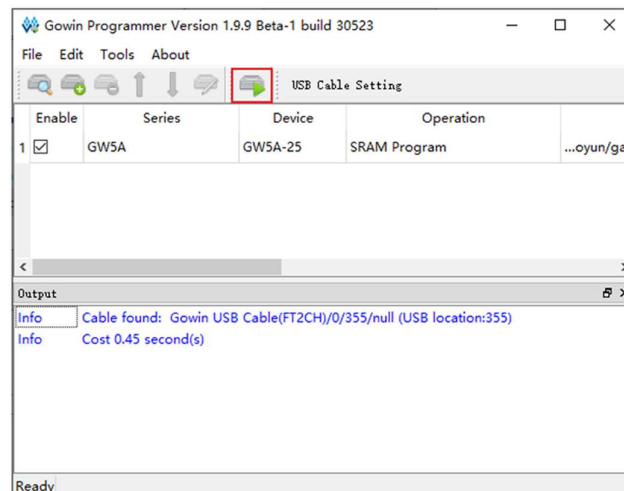


Figure 2-24 Device Detected Prompt

3. Click the icon in Gowin Programmer to download the fs file to the FPGA, as shown in Figure 2-25 below. After the download is successful, information will be shown in Figure 2-26



Hình 2-25 Download file fs

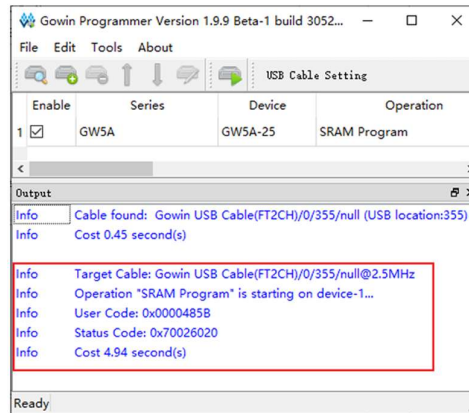


Figure 2-26 Download file successful

4. Observe the phenomenon: At this time, we can see that the corresponding LED0 light on the development board is flashing, indicating that our experiment was successful. At this time, we can see that after the program is downloaded, all 4 LED lights are on, but we only control one of them. This is because in the Gowin software, uncontrolled pins are pulled up by default, meaning they will output a high level, causing the uncontrolled LED lights to turn on. This can be set in the software, as shown below.

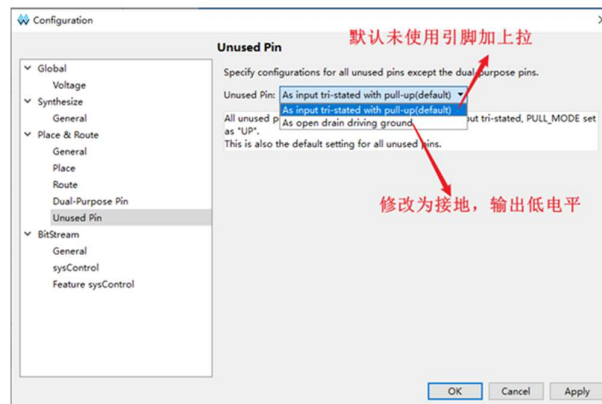
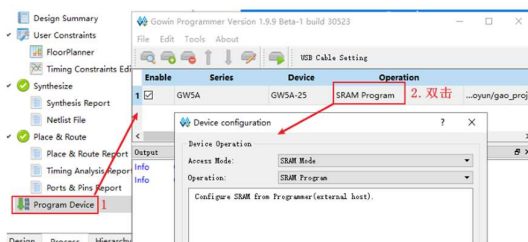


Figure 2- 27 Modifying Default Level of Unused Pins

2.9 Store the program on FPGA Memory

The program downloaded through the previous section will be lost after the development board is powered off and restarted. To achieve non-volatile storage after power-off, next we will teach everyone how to perform configuration (solidification) of the Gowin FPGA chip.

1. Click Program Device to enter the Gowin Programmer interface.
2. Double-click SRAM Program in the Operation column to enter the Device configuration interface, as shown in Figure 2-28 below.



Hình 2- 28 Figure 2-28 Operation Diagram to Enter Device Configuration Interface

In the Device configuration interface, configure Access Mode to External Flash Mode 5AT, and Operation to exFlash Erase, Program 5AT, then click Save, as shown in Figure 2-29 below

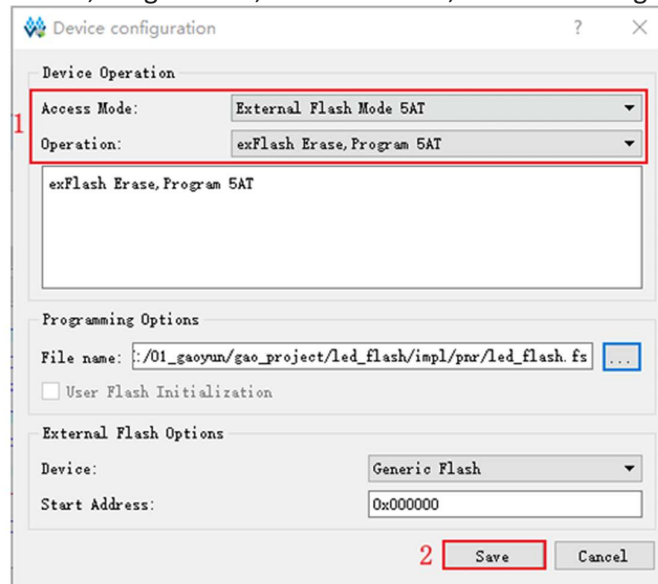


Figure 2-29 Set storage program memory

Click the icon to download the program. After successful download, the following information will be printed, as shown in Figure 2-30.

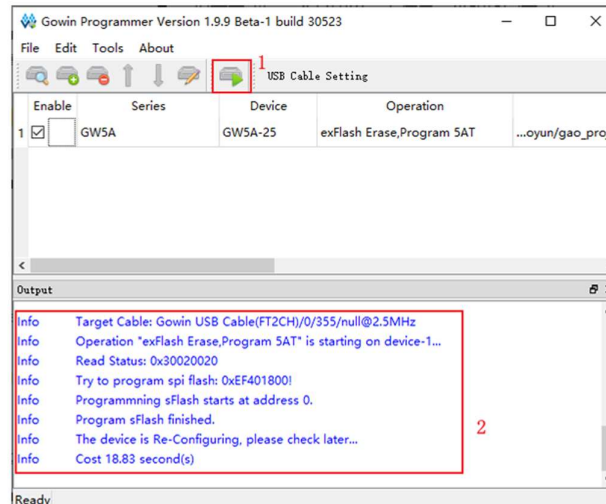


Figure 2-30 Programing Successful

Then re-power the development board, and you can see that the board is running the program solidified in this experiment.

2.10 Chapter Summary

This chapter's experiment guided everyone to become familiar with the basic development flow of the Gowin software. For the details that should be paid attention to in each process, this chapter provided certain descriptions. Some omissions will be supplemented when used in subsequent chapters.

As the first chapter for starting experiments, this chapter introduced the entire design process step by step. Users must operate and familiarize themselves with the entire process according to the flow, understanding the purpose and function of each step to avoid project errors or even having to restart the entire project due to missing steps.

Chapter 3: Gowin & Modelsim for simulation

Sample Practice Code |----Project 2 |----ch3_led_flash_sim

Chapter Overview

This chapter will guide everyone to learn and master Modelsim simulation, how to compile Gowin libraries in Modelsim, and perform Gowin and Modelsim joint simulation.

3.1 Modelsim Compiling Gowin Device Libraries

Gowin's IP CORE or primitives are only used within the corresponding development platform. Third-party tools like

Modelsim cannot acquire the internal operating logic results of Gowin's IP CORE, and therefore cannot directly perform simulation. Gowin library compilation is required. This chapter first leads everyone to learn how to compile Gowin libraries. This step needs to be performed when using Gowin FPGA for the first time. The compiled libraries can be used at any time subsequently, without compiling again before every simulation.

This chapter's experiment requires that Modelsim is already installed on your computer. If Modelsim is not yet installed, users can go to our forum to download and install it. The download link is as follows: <http://www.corecourse.cn/forum.php?mod=viewthread&tid=28768>

3.1.1 Creating a New Modelsim Library

- a. Open the ModelSim software, click the menu bar [File] -> [Change Directory] option sequentially to enter the path switching page, as shown in Figure 3-1 below.

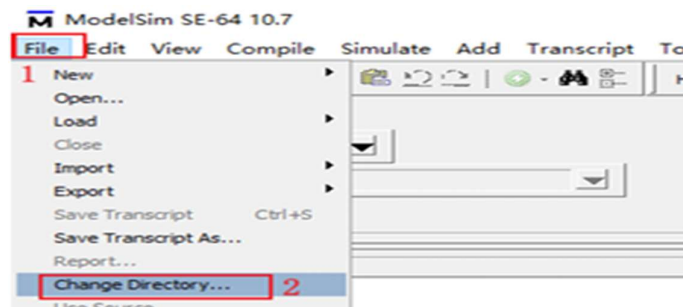


Figure 3-1 Enter Path Change Directory

- b. In the opened folder selection page, locate the Modelsim software installation directory (e.g., mine is G:\Modelsim), as shown in Figure 3-2 below, then click Select Folder

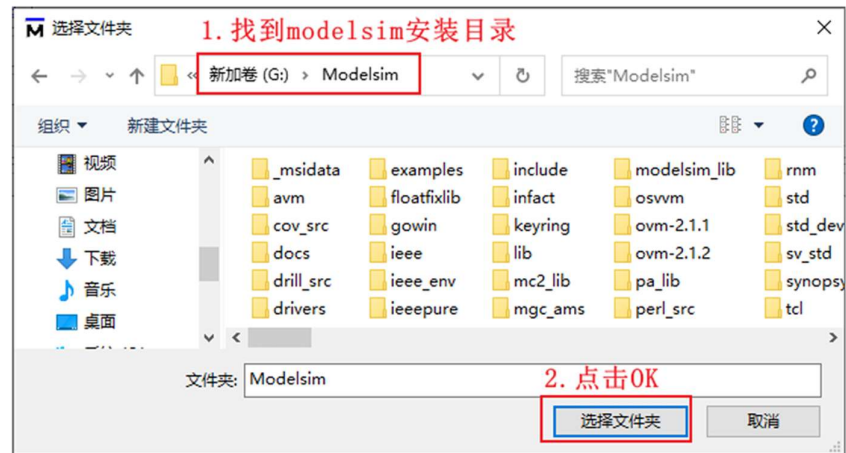


Figure 3-2 Select Directory install Modelsim

c. Click on Menu **File** -> **New** -> **library** , mở trang tạo thư viện như Hình 3-3 bên dưới

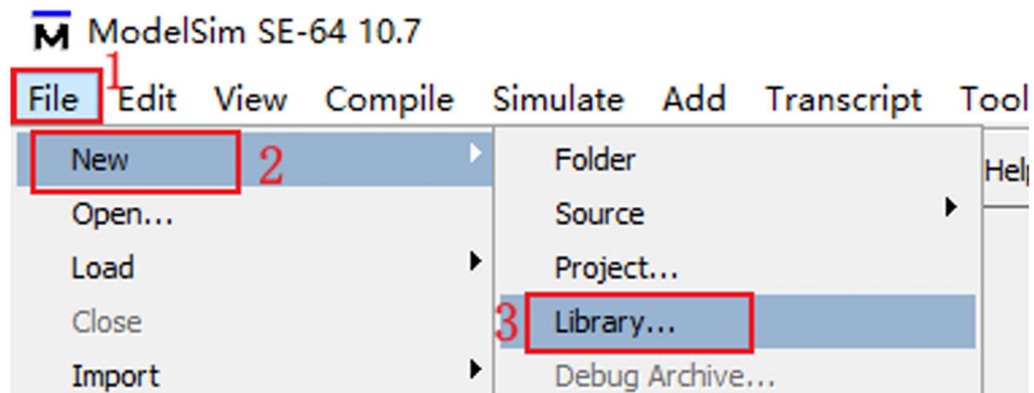


Figure 3-3 Opening create New Library Interface

d. In the opened interface, make sure to first select the Library drop-down as the gw5a we just created, and then modify the file path to the library path of the Gowin software's gw5a series devices, specifically: "\\Gowin\\Gowin_V1.9.9Beta-1\\IDE\\simlib\\gw5a", as shown in Figure 3-7 below.

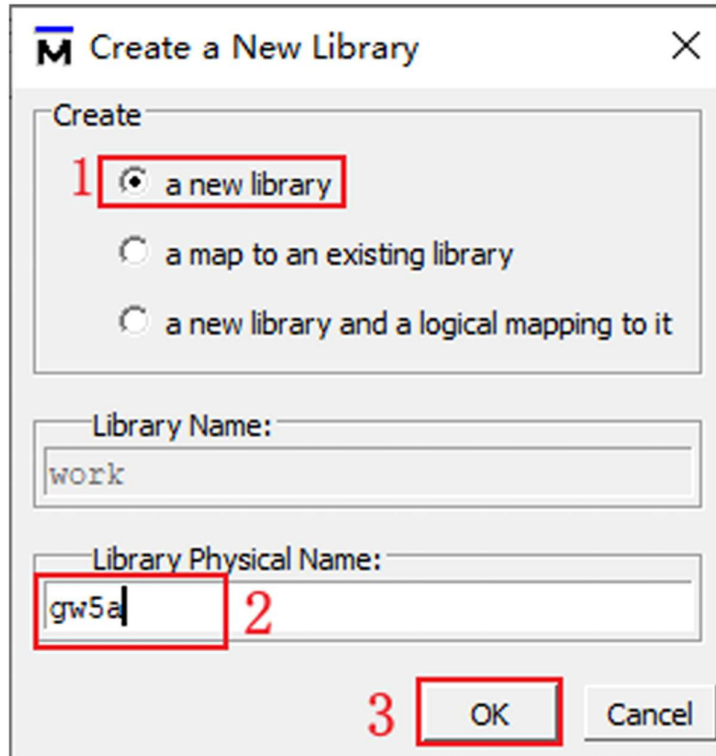
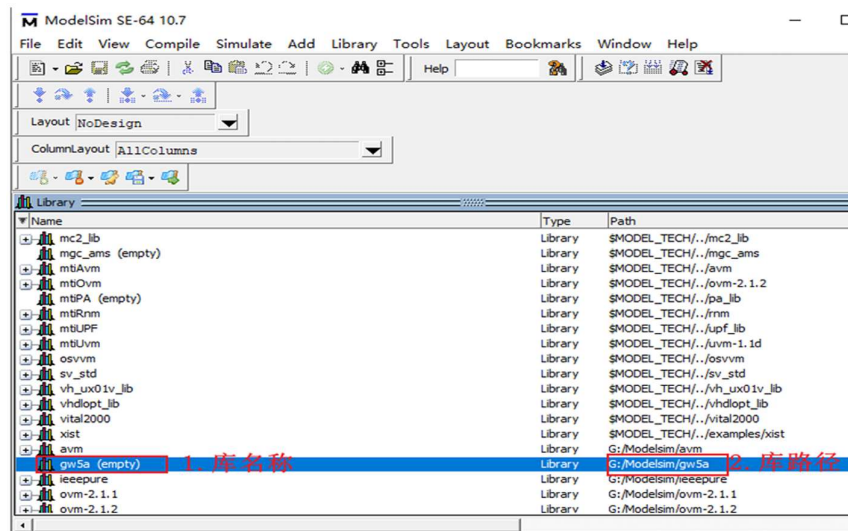


Figure 3-4 Create new Library name

- e. If the gw5a (empty) library appears in the library list, and the path is the location set previously (under the project root directory), it indicates that the library creation is complete, as shown in Figure 3-5 below.



Hinh 3-5 Create new library complete

3.1.2 Compiling Gowin Device Library Files

1. Click **[Compile]** -> **[Compile]** in the menu bar sequentially to open the compile library file interface, as shown in Figure 3-6 below

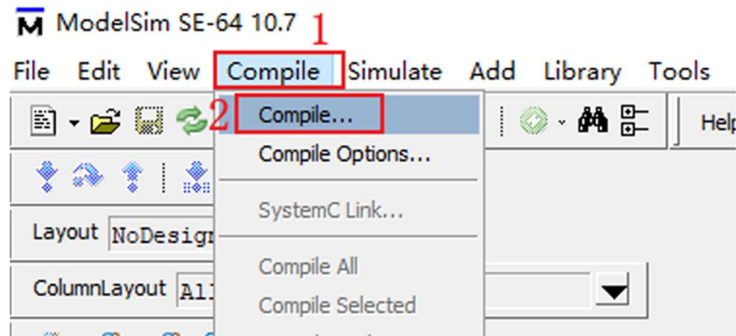


Figure 3-6 Open Opening Compile Library interface

2. In the opened interface, make sure to first select the Library drop-down as the gw5a we just created, and then modify the file path to the library path of the Gowin software's gw5a series devices, specifically: "\\Gowin\\Gowin_V1.9.9Beta-1\\IDE\\simlib\\gw5a", as shown in Figure 3-7 below.

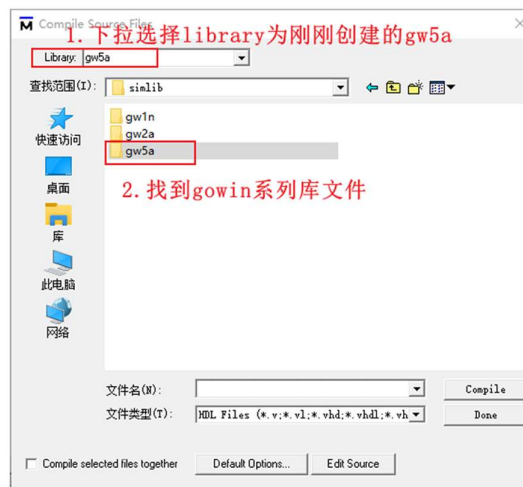


Figure 3-7 Find Library Gowin FPGA

3. Under this path, find the prim_sim.v file, and then click the Compile button to start compiling, as shown in Figure 3-8 below. At this time, Modelsim's transcript window will quickly refresh various compilation process information. After compilation is complete,

click the Done button to finish compiling

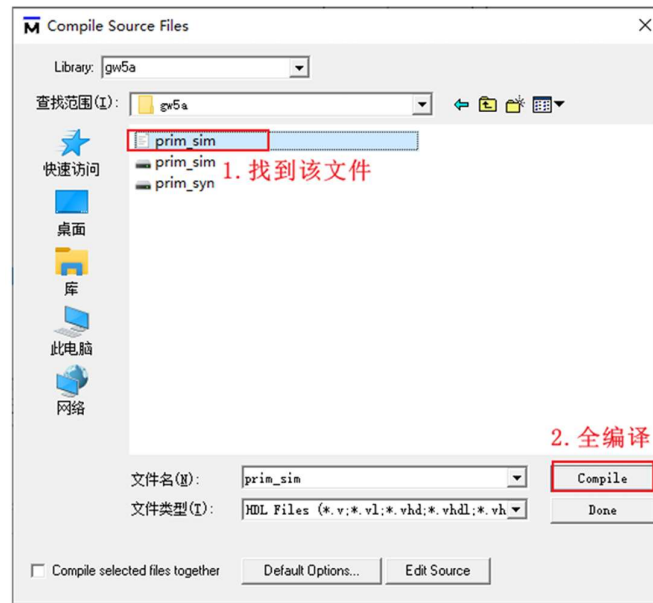


Figure 3-8 Find library file for compile

4. After compilation is complete, the Transcript interface will display 0 Errors, 0 Warnings, and various components will appear under the gw5a library, as shown in Figure 3-9 below.

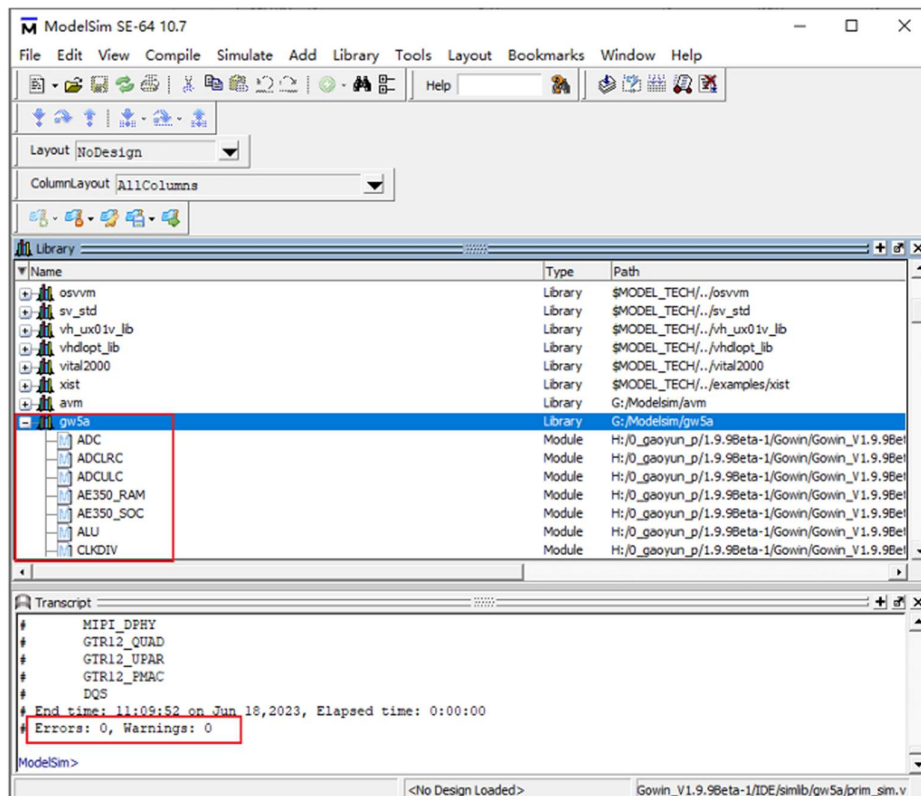


Figure 3-9 Gowin Series Device Library Compilation Complete

3.1.3 Adding Library to Modelsim Default Library List

For the library just compiled to be automatically found and added for use by the Modelsim software subsequently, it needs to be added as one of Modelsim's default libraries. The addition method is very simple, as shown below:

1. Find the Modelsim installation location (e.g., my computer is G:\Modelsim). Under this path, there is a file named modelsim.ini. This is a read-only file. Right-click the Modelsim.ini file, click Properties, and in the properties settings interface, uncheck its read-only attribute, as shown in Figure 3-10 below.

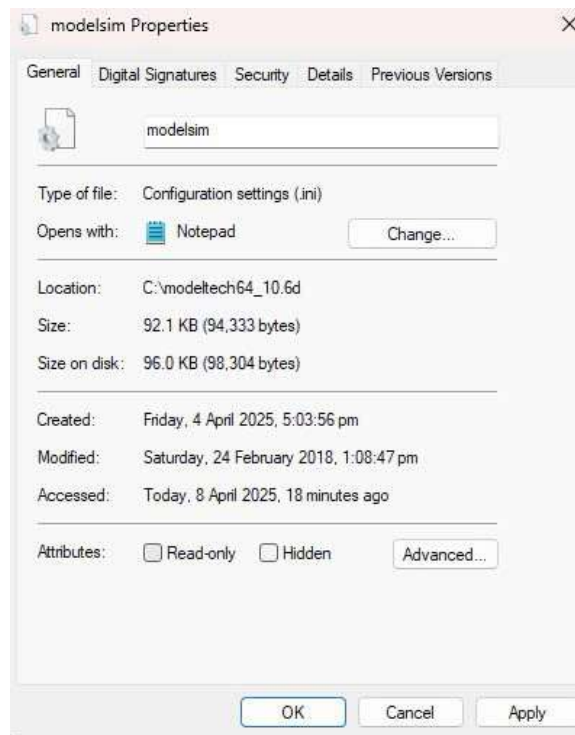


Figure 3-10 Removing Read-only Attribute of modelsim.ini

2. Then open the file with Notepad or notepad++. Around line 17 of the file, add the gw5a library specification statement "gw5a = \$MODEL_TECH/../../gw5a" and save it, as shown below.

```
File Edit View
; vsim modelsim.ini file, version 10.4
[Version]
INIVersion = "10.6d"

; Copyright 1991-2018 Mentor Graphics Corporation
;
; All Rights Reserved.
;
; THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF
; MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE TERMS.
;

[Library]
std = $MODEL_TECH/../../std
ieee = $MODEL_TECH/../../ieee
vital2000 = $MODEL_TECH/../../vital2000
gw5a=$MODEL_TECH/../../gw5a
;
; VITAL concerns:
;
; The library ieee contains (among other packages) the packages of the
; VITAL 2000 standard. When a design uses VITAL 2000 exclusively, it should use
Ln 1, Col 1 94,358 characters 100% Unix (L
```

Figure 3-11 Adding gw5a Library File Path

3. Restore the read-only attribute of modelsim.ini.

At this point, the method on how to use Modelsim to compile the library files for Gowin 5A series FPGAs has been introduced. Devices of the 1N and 2A series can also be compiled by referring to the above method. Next, we can use Modelsim to create a simulation project and simulate the design project.

3.2 Modelsim Simulation Verification Flow

In the section "Gowin Software Basic Development Flow", we explained the basic development flow of Gowin FPGA and lit up the LED light to make it flash. This section will explain the flow of Gowin FPGA joint simulation verification with Modelsim based on the project led_flash.

3.2.1 Writing Verilog Simulation Design Code

1. First open the project, click the icon to create a new Verilog File, or use the Ctrl+N shortcut, as shown in Figure 3-12 below.

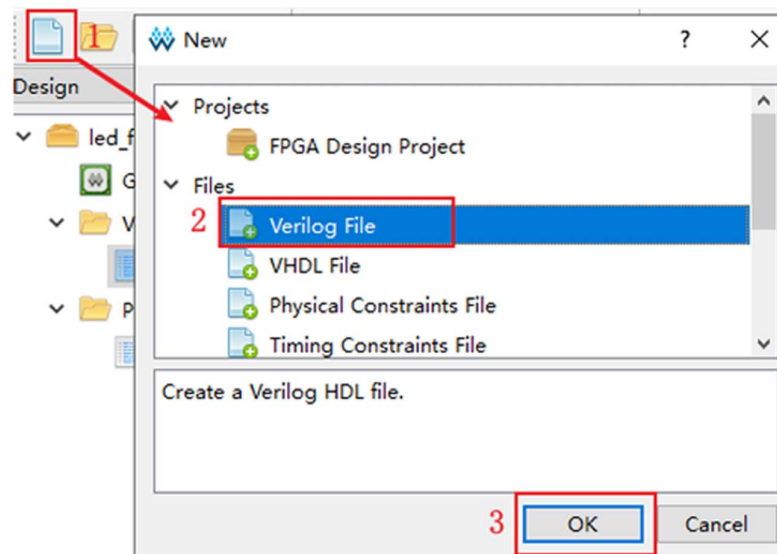


Figure 3-12 Creating a New Verilog File

2. Then in the popped-up New Verilog File, name the file led_flash_tb. Keep the default file save path unchanged, as shown in Figure 3-13 below.

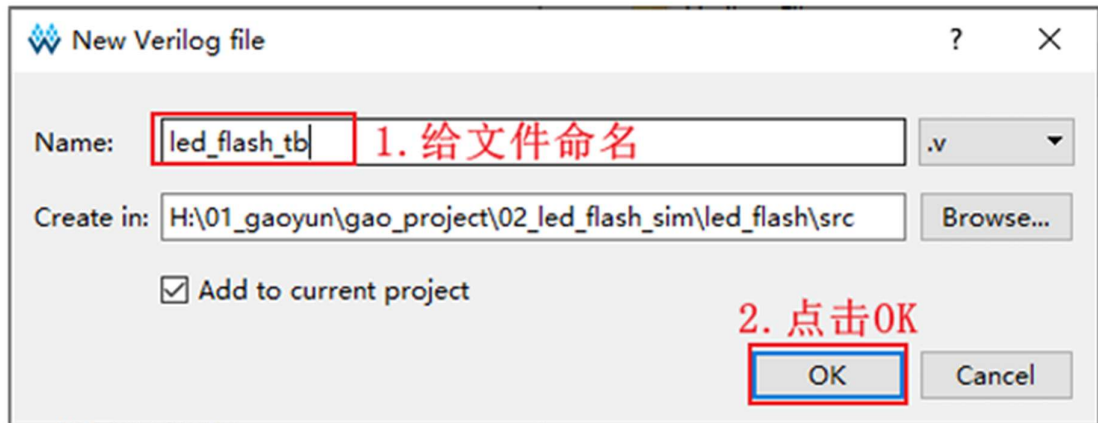


Figure 3-13 Naming Simulation File

3. Enter the code content in the opened file. For the simulation file of LED flashing, the following

```
`timescale 1ns/1ns
module led_flash_tb();

    reg Clk;
    reg Rst_n;
    wire Led;

    led_flash led_flash(
        .clk(Clk),
        .rst_n(Rst_n),
        .led(Led)
    );

    initial Clk = 1;
    always#10 Clk = ~Clk;

    initial begin
        Rst_n = 0;
        #201;
        Rst_n = 1;
        #200;
        #100000000;
    end

endmodule
```

code can be used:

4. After saving the led_flash_tb file, perform analysis and synthesis to check for grammatical errors. After the simulation file design is completed, we can use Modelsim to perform functional simulation testing of the project.

3.2.2 Creating Modelsim Project and Adding Simulation Files

1. Open Modelsim and create a new project in Modelsim. Select File->New->Project, as shown in Figure 3-14 below.

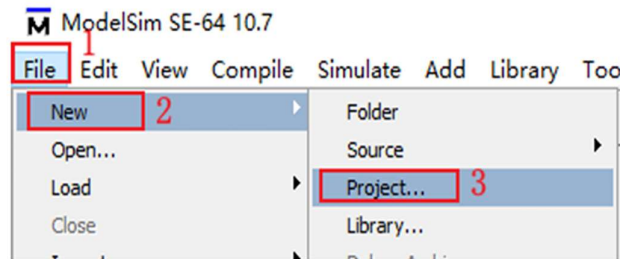


Figure 3-14 Creating Modelsim Project

2. Enter the new project setup interface. Fill in the project name in the "Project Name" column. Here we name the project corresponding to the project name "led_flash". "Project Location" is the project path. Click "Browse" to set it. As needed, we save the simulation project in a newly created sim folder under our design project, and select it, as shown in Figure 3-15 below.

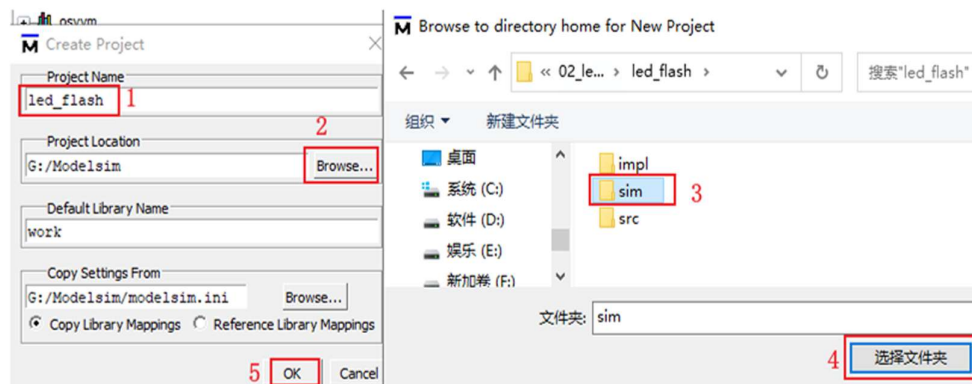


Figure 3-15 Modelsim Project Settings

3. After clicking OK, the interface shown in Figure 3-16 below appears. It can be seen from the figure that there are four operations to choose from: Create New File, Add Existing File, Create Simulation, and Create New Folder. Here we select "Add Existing File".

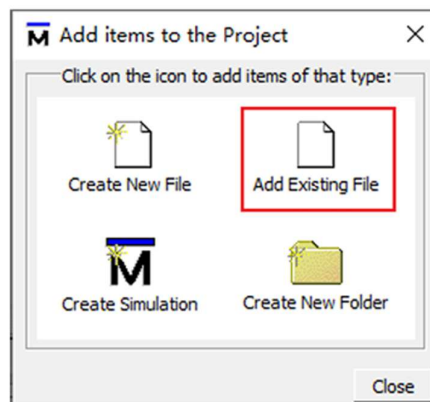


Figure 3-16 Add Items to the Project Dialog Box

4. Subsequently, in the popped-up dialog box, select Browse to choose the project files required for this experiment: "led_flash.v" and "led_flash_tb.v", as shown in Figure 3-17 below.

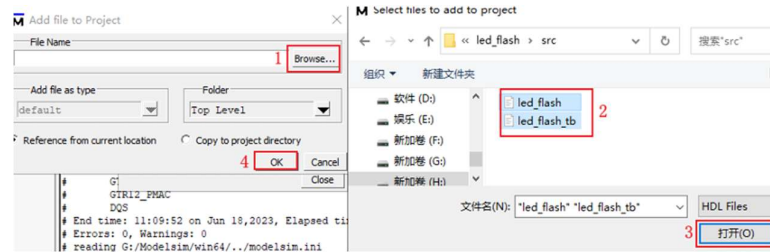


Figure 3-17 Selecting Files to Add

After addition is complete, we can see the added project files in the Modelsim software, as shown Figure 3-18 bellow

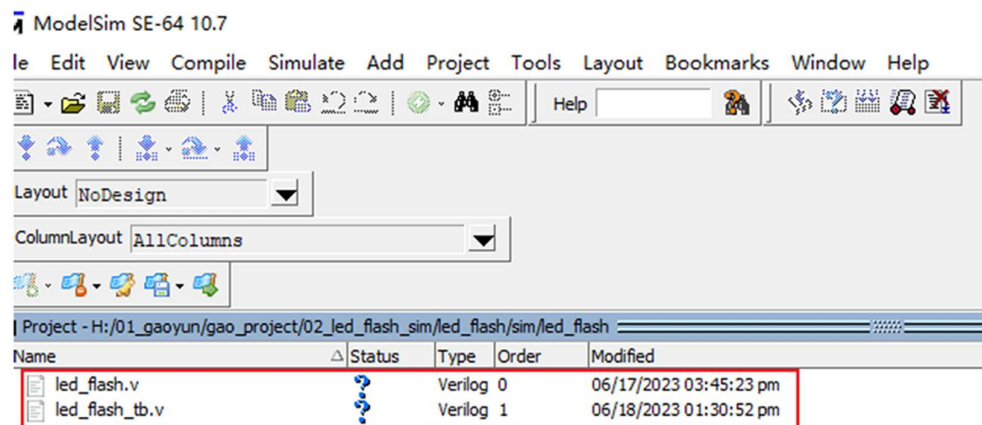


Figure 3-18 Schematic after Successful File Addition

3.2.3 Compiling Simulation Files

There are two ways to compile: The first is Compile Selected, which requires selecting one or several files first. Executing the command completes the compilation of the selected files. The second is Compile All, which compiles all files in the project in compilation order. Here, we choose to compile all files. Select Compile->Compile All, as shown in Figure 3-19 below:

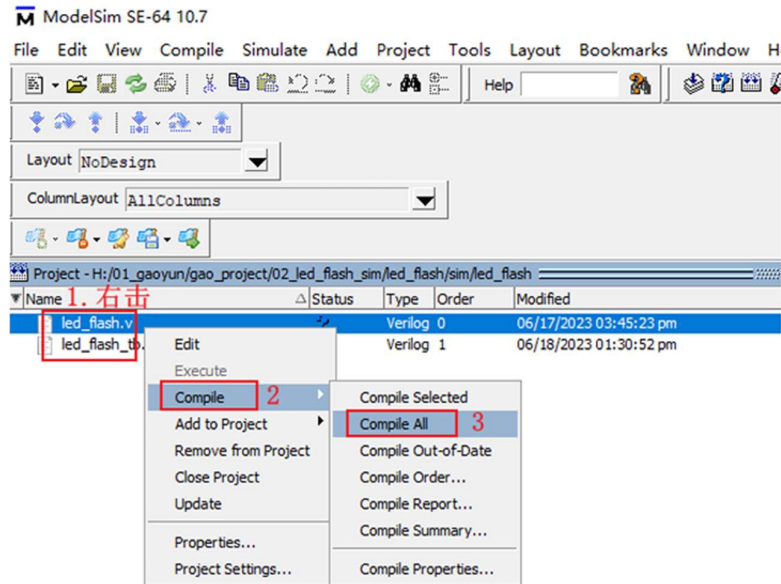


Figure 3-19 Compiling All Project Files

After compilation is complete, the Status after the file changes from "?" to "v", indicating successful compilation. The "Transcript" below also prompts that the file compilation was successful, as shown in Figure 3-20 below.

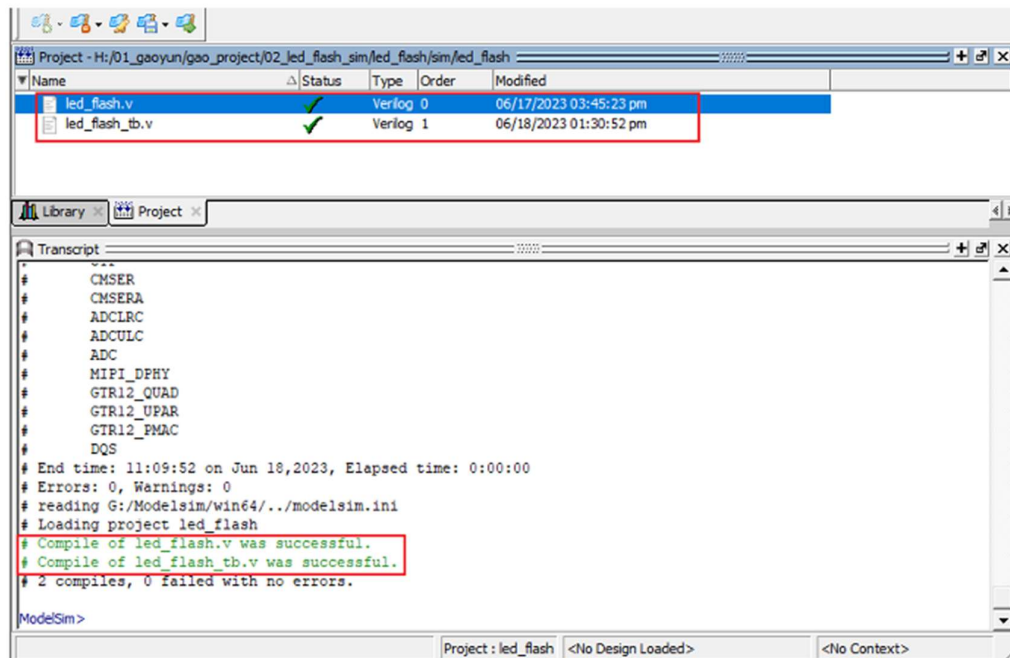


Figure 3-20 Schematic of Successful File Compilation

Note that there are two states undesirable in design: Compilation Error (displaying a red "x") and Compilation with Warning (a yellow triangle symbol appears after the checkmark).

Compilation error means Modelsim cannot complete the file compilation work. Usually, this situation is because the compiled file contains obvious grammatical errors. Modelsim will identify these grammatical errors and prompt the user, who can modify them according to Modelsim's prompt information.

3.2.4 Configuring Simulation Environment

After compilation is complete, we will proceed to configure the simulation environment.

We right-click in the project status bar, and then select "Add to Project" ->

"Simulation Configuration..." and click it, as shown in Figure 3-21 below.

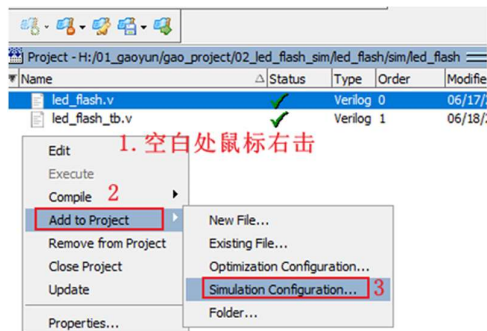


Figure 3-21 Entering Simulation Environment Configuration Operation Schematic

Enter the Add Simulation Configuration page. In the Design tab page, we select the "led_flash_tb" module in the work library as the design top layer. Click to copy the module name as the naming for "Simulation Configuration Name", ensuring the naming remains consistent, as shown in Figure 3-22 below. In complex project designs, we can design multiple different simulation configuration top layers to perform simulation testing on the project.

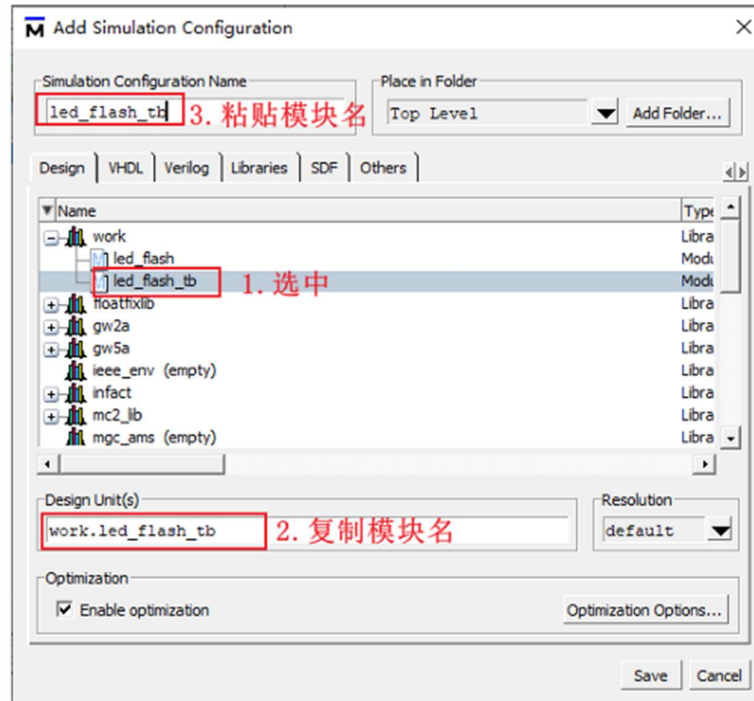


Figure 3-22 Simulation Environment Interface Configuration 1

Click "Optimization Options...". In the "Optimization Options.." settings column, select "Apply full visibility to all modules(full debug module)", and click "OK", as shown in Figure 3-23 below.

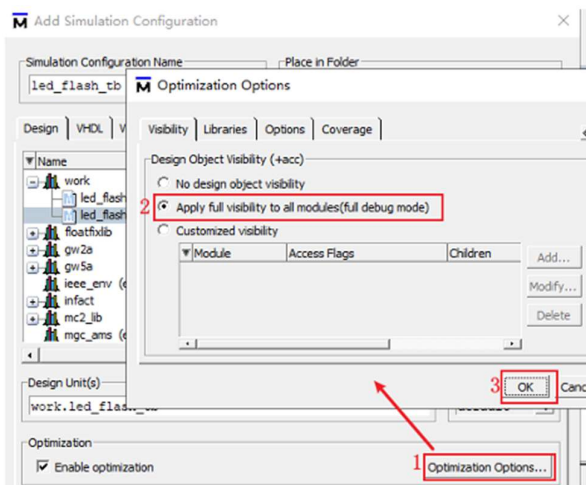


Figure 3-23 Simulation Environment Interface Configuration 2

Click the "Libraries" settings column. In the "Search libraries(-L)" column, click "Add..." to add our newly created Gowin library file "gw5a". In "Search Libraries First(-Lf)", also select the library file "gw5a". Finally, click "Save" to save the settings, as shown in Figure 3-24 below.

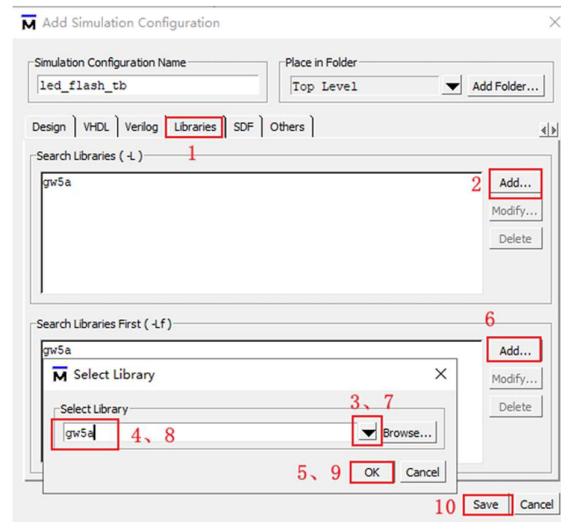


Figure 3-24 Simulation Environment Configuration Interface 3

Through the above steps, the configuration of the simulation environment is completed. After saving the configuration, we can see that the simulation configuration file "led_flash_tb" is generated in the "project" column, as shown in Figure 3-25 below.

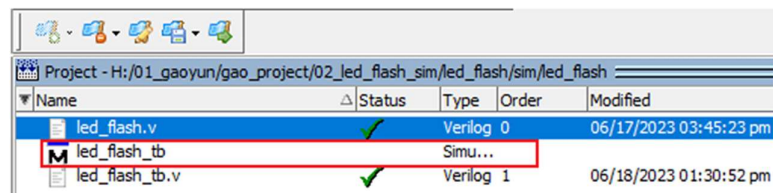


Figure 3-25 Schematic of Simulation File Generation Success

3.2.5 Observing Simulation Interface Waveforms

1. We first need to double-click the newly generated simulation file "led_flash_tb" to enter the "sim" interface, as shown in Figure 3-26 below.

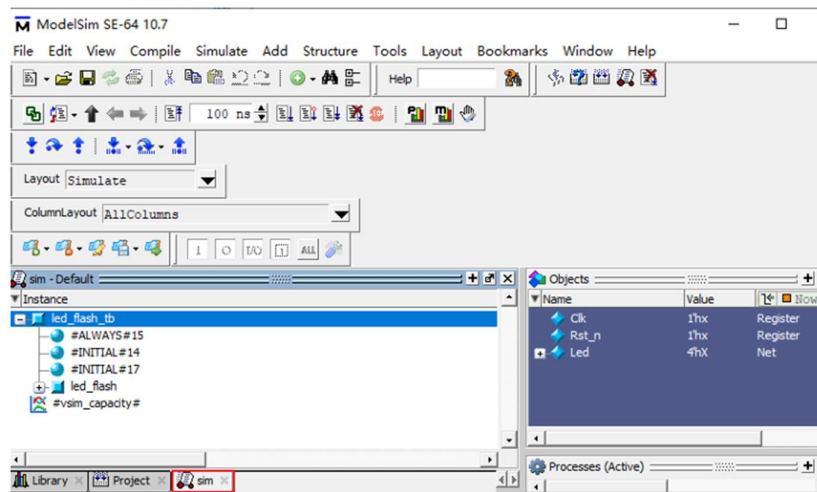


Figure 3-26 Entering sim Interface

2. In the "sim" interface, we can add the waveforms of the modules we want to observe. Select the module, right-click and choose "Add Wave", as shown in Figure 3-27 below.

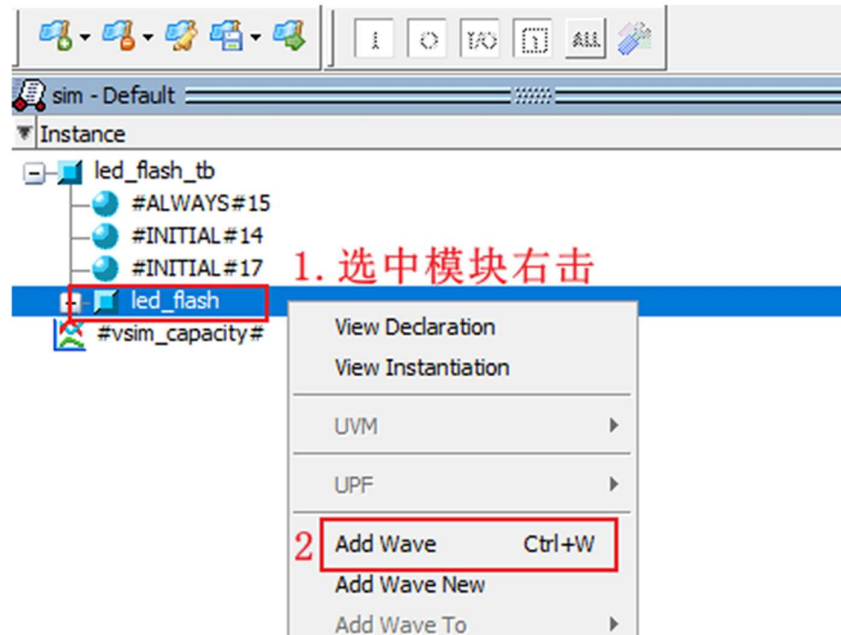


Figure 3-27 Adding Module Waveforms to Observe

3. After adding the waveforms, return to the "Library" column, right-click "work" and click Update to update the "led_flash_tb" file in the "work" column, as shown below.

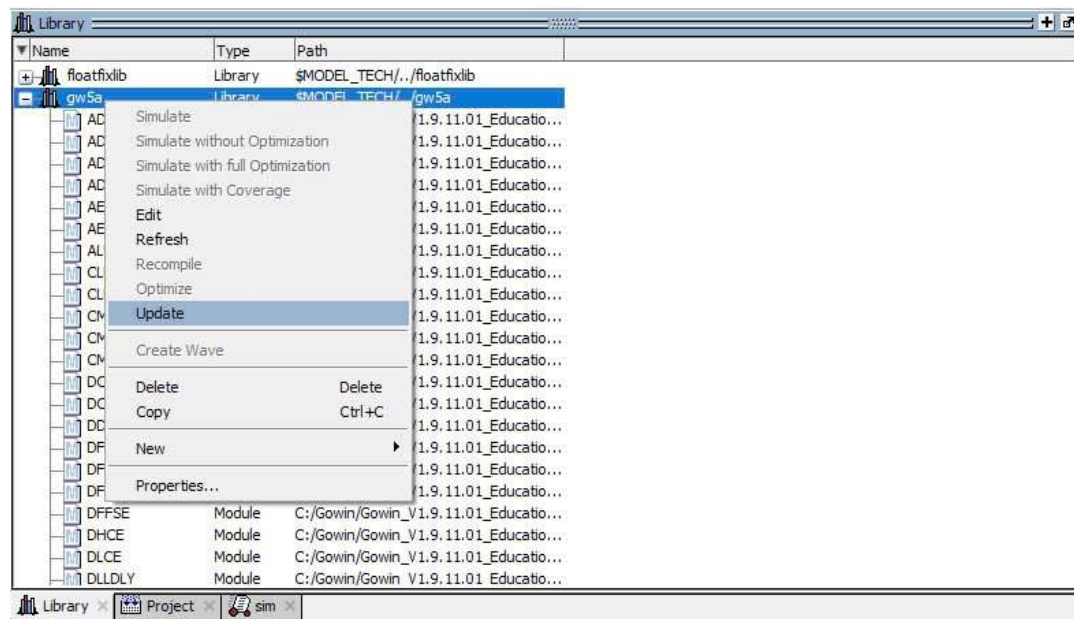


Figure 3-28 Updating work Column

4. Click Run all to start running the simulation, as shown in Figure 3-29 below.

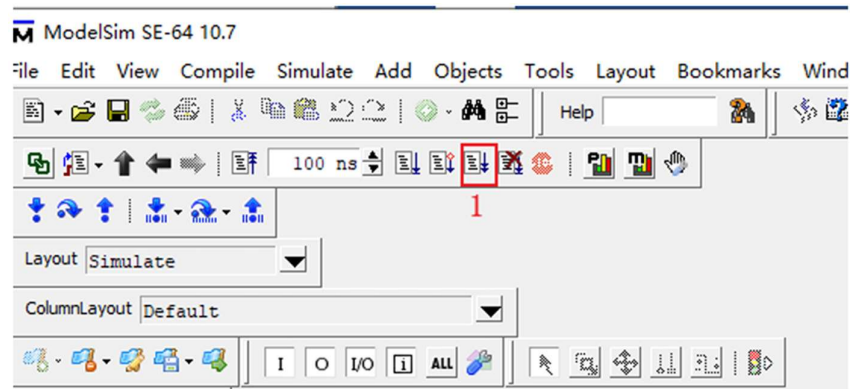


Figure 3-29 Starting Simulation

5. Then it will jump to the Wave window. Wait for the program to run for a while, and we can see the waveform changes, as shown in Figure 3-30 below. From the figure below, it can be seen that the state of led has toggled according to the count value of cnt.

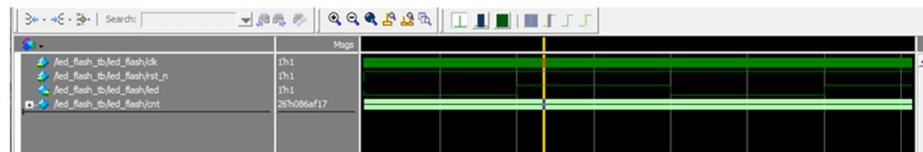


Figure 3-30 Observing Waveforms

If you want to stop running, click the [Stop] icon in the toolbar above the Wave window. To restart, click [Restart]. At this time, clicking "Zoom Full" allows quick observation of the waveforms. If you want to understand the function of each button in the toolbar, move the mouse to the button, and you can see the explanation, such as the Continue Run Simulation button, as shown in Figure 3-31 below:



Figure 3-31 Wave Interface Tool Button Function View

During the development process, after changing the design file, click save and check for syntax errors, then re-run the program. In Modelsim, we enter the "Library" interface and re-"Recompile" "led_flash_tb" and "led_flash" under the "Work" directory to reload the simulation waveforms.

3.2.6 Saving Waveform Files

When using Modelsim for simulation, for some time-consuming simulations, you can save the simulation waveform results for convenient opening and viewing next time.

First, step one: save dataset sim. Open the sim window, click file->save dataset sim or in the sim interface click the Save icon to save the .wlf file and name the file. The operation is as shown below.

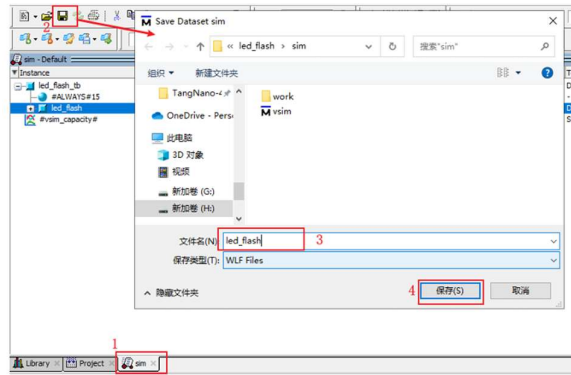


Figure 3-32 Saving .wlf File

Second step, save the waveform in the Wave interface. After clicking the save icon, the waveform file "wave.do" will be saved by default in the project's simulation folder sim. The operation is as shown in Figure 3-33 below.

3.2.7 Reopening Simulation Project

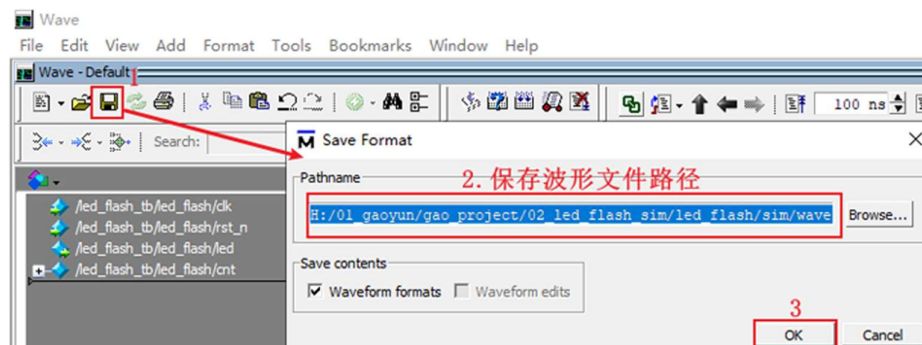


Figure 3-33 Saving Waveform File

After closing the simulation, if the user needs to reopen the simulation, just open the ".mpf" file, as shown in Figure 3-34 below.



Figure 3-34 Reopening Simulation Project

When opening ".mpf" for the first time, the system will not default to open with Modelsim. Readers need to manually set the opening method for the ".mpf" file. Readers with multiple versions of Modelsim installed also need to pay attention to setting the corresponding version of the Modelsim software. The version installed for the experiment is modelsim-win64-10.7-se.

If the waveform file was saved previously, after opening the Modelsim software, click file->all file to open the saved .wlf file. The operation is as shown in Figure 3-35 below.

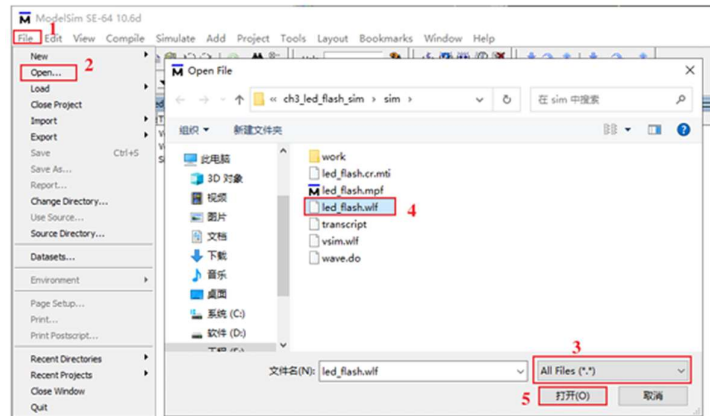


Figure 3-35 Opening Saved .wlf File

Subsequently, click file->load->Macro file to open the saved .do file, as shown in Figure 3-36 below. After opening, you can see the waveform file we saved previously.

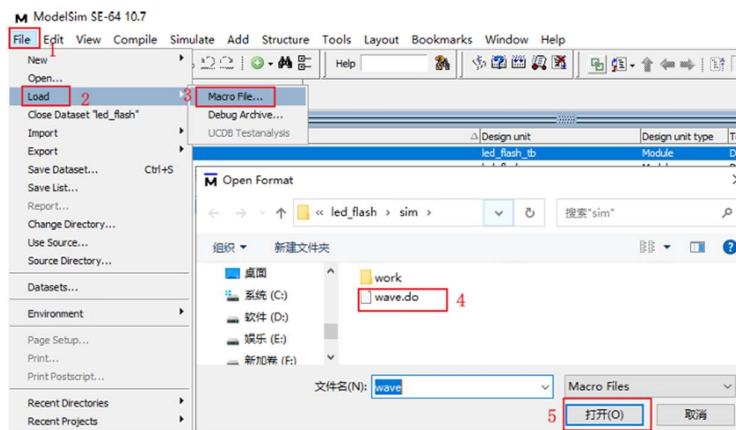


Figure 3-36 Opening Saved Waveform File

3.3 Reflection and Summary

This chapter's experiment guided everyone to learn how to use Modelsim to compile the device library of Gowin FPGA and how to use Modelsim for simulation. In the source code of this simulation experiment, the time for the LED light state change is 1S. It takes a while to wait for the simulation waveform to change. If you want to see the waveform change faster, you can reduce the count of the counter cnt to see the waveform change more quickly.

Chapter 4: Using Gowin Online Logic Analyzer

The following table:

Project Source Code, |----02_Design Examples, |----ch4_led_flash_gao

Chapter Overview

This chapter's experiment builds on the previous experiments to guide everyone in learning the use of the Gowin Online Logic Analyzer. Here we only explain the basic usage flow of this function. More functions will be explained when needed in later experiments.

4.1 Introduction to Gowin Online Logic Analyzer

The function of Gowin Semiconductor's Online Logic Analyzer (Gowin Analyzer Oscilloscope, hereinafter referred to as GAO) is to help users analyze the timing relationships between signals in the design more simply, quickly perform system analysis and fault location, and improve design efficiency.

Working principle of GAO: When the FPGA is working, it utilizes unused memory resources in the device to save signals to the memory in real-time according to the trigger conditions set by the user. The signal status is read in real-time through the JTAG interface and displayed on the Gowin interface. GAO includes a signal configuration window and a waveform display window. The signal configuration window is mainly used to insert positioning information into the design. This type of positioning information is mainly based on sampling clocks, trigger units, and trigger expressions. The waveform display window connects Gowin and the target hardware through the JTAG interface, intuitively displaying the data of the sampled signals set in the signal configuration window through waveforms.

GAO supports RTL-level signal capture and post-synthesis netlist-level signal capture, and provides two versions: Standard GAO and Lite GAO. Standard GAO can support up to 16 functional cores, each core can be configured with one or more trigger ports, supporting multi-level static or dynamic trigger expressions. Lite GAO configuration is simple, no need to set trigger conditions, and can capture the initial value of signals, facilitating users to analyze the working status at the moment of power-on.

4.2 GAO File Configuration

4.2.1 Creating New GAO Configuration File

Open the Gowin project, click the icon, and in the popped-up New dialog box, choose to create a new GAO Config File, as shown in Figure 4-1 below. Click OK.

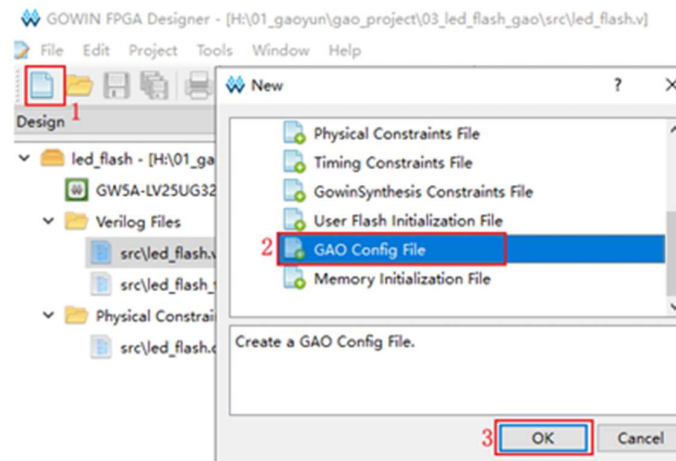


Figure 4-1 Creating New Gao Configuration File

The New GAO Wizard dialog box pops up. There are two options in the Type column: "For RTL Design" and "For Post-Synthesis Netlist". The "For RTL Design" type is used to capture RTL signals before synthesis optimization, generating a configuration file with the extension .rao; the "For Post-Synthesis Netlist" type is used to capture Netlist signals after synthesis optimization, generating a configuration file with the extension .gao. Here we select the "For RTL Design" type Standard GAO configuration file, as shown in Figure 4-2 below.

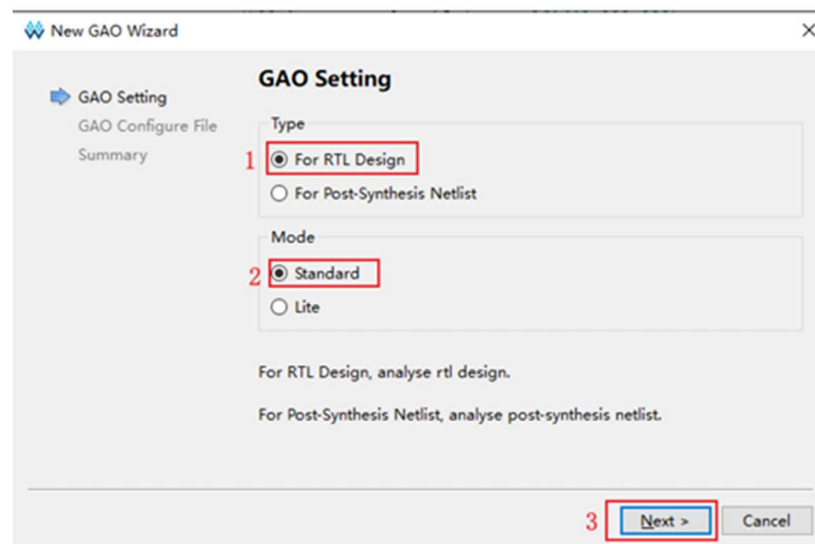


Figure 4-2 GAO File Configuration Interface 1

Then enter the configuration file name in the "Name" edit box in the popped-up window. It defaults to remain consistent with the project name. Here we follow the default name and click the "Next" button, as shown below.

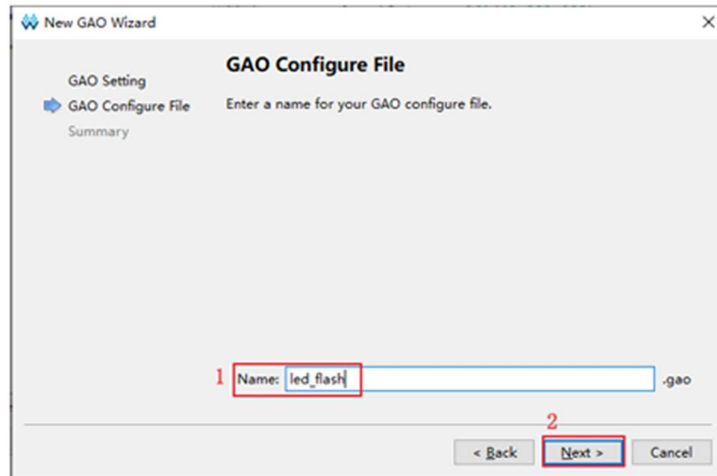


Figure 4-3 GAO File Configuration Interface 2

Finally, the Summary dialog box pops up. Click Finish to complete the creation of the GAO file.

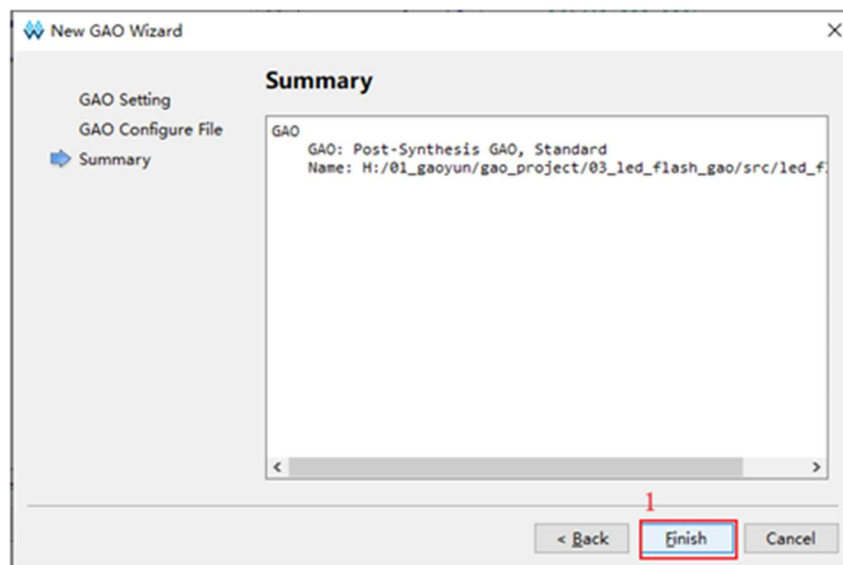


Figure 4-4 GAO File Configuration Interface 3

4.2.2 Starting GAO Configuration Window

After the GAO file is created, GAO Config Files will appear in the Design window. Below this column is the GAO file we just created. Double-click to enter the GAO configuration window, as shown in Figure 4-5 below. The GAO configuration window includes Ao Core for configuring the number of functional cores and the signal configuration view of the corresponding Core. The Core signal configuration view includes the Trigger Options view for configuring signal trigger conditions and the Capture Options view for configuring signal sampling conditions.

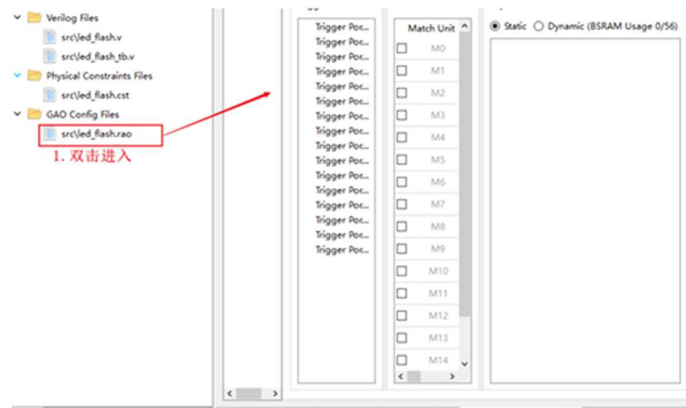


Figure 4-5 Entering GAO Configuration Window

Note: If the warning shown in Figure 4-6 appears when double-clicking the .gao file, we need to first analyze and synthesize (Synthesize), and then double-click the .gao file to enter the GAO configuration window.

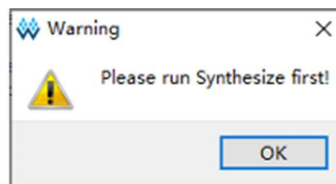


Figure 4-6 Prompt Warning Appears

4.2.3 Configuring Standard Mode GAO

The Standard Mode GAO configuration window is used to configure the number of functional cores, signal trigger conditions, and signal sampling conditions.

1. Configuring Functional Core Quantity

The Ao Core view is used to display and manage the number of functional cores used in the current project. The default is Core 0, supporting up to 16 Cores. You can click Add to add or Remove to remove anywhere in the Ao Core view, as shown in Figure 4-7 below. Here we use the default Core 0.

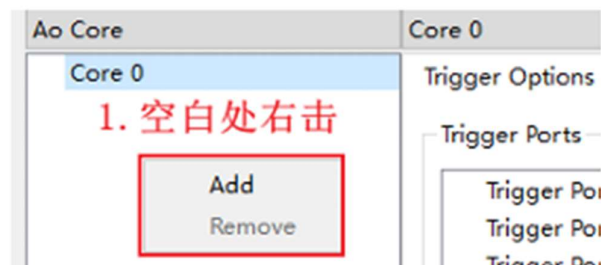


Figure 4-7 Schematic of Adding or Removing Core

2. Configuring Trigger Ports

The Trigger Ports view of Core 0 is used to configure the trigger ports of the functional core. It can be seen that there are a total of 16 trigger ports, and the width range of each trigger port is 1~64. Here we use port 0. Double-click it to enter the port configuration interface Trigger Port, as shown in Figure 4-8 below. MSB and LSB in the figure represent the high bit and low bit of the trigger port respectively.

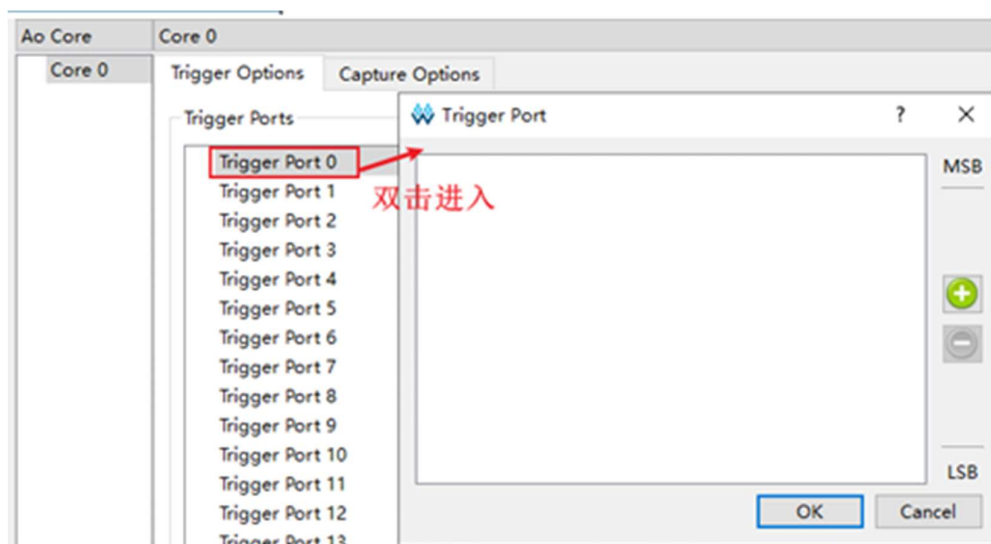


Figure 4-8 Schematic of Entering Trigger Port Configuration Interface

Click to enter the Search Nets dialog box. In the Name column, enter the trigger method you want to add. For example, in this experiment, the desired trigger method is the LED light status signal led. Enter led, and then click Search, as shown in Figure 4-9 below.

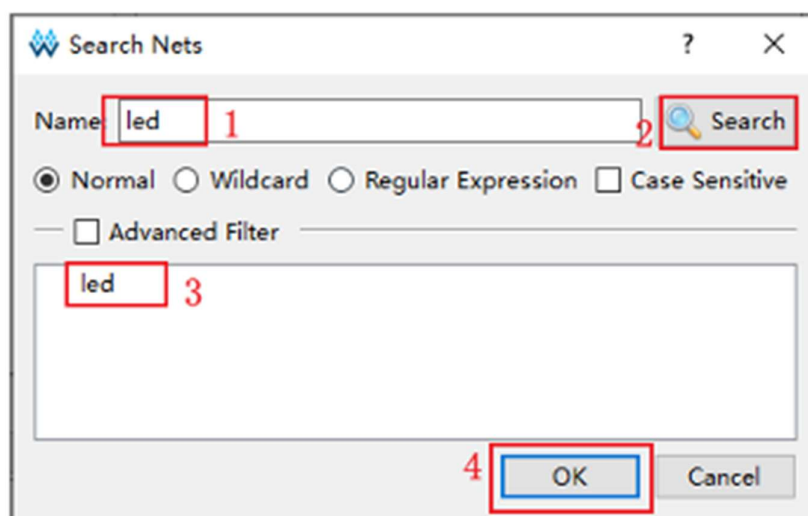


Figure 4-9 Selecting Trigger Signal

Then return to Trigger Port and click OK, as shown in Figure 4-10 below.

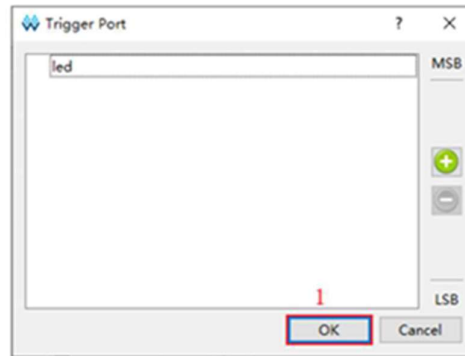


Figure 4-10 Trigger Signal Addition

3. Configuring Match Units

The Match Units view is used to configure the matching units of the trigger ports. Up to 16 trigger matching units correspond to M0~M15. The matching unit is the minimum unit for the GAO functional core to implement trigger conditions. The functional core processes the trigger port signals designed by the user through the matching unit. When the trigger port signal meets the requirements, triggering can be achieved. Here we select matching unit M0. Double-click the matching unit, and configure the trigger conditions in the popped-up "Match Unit Config" dialog box, as shown in Figure 4-11 below.

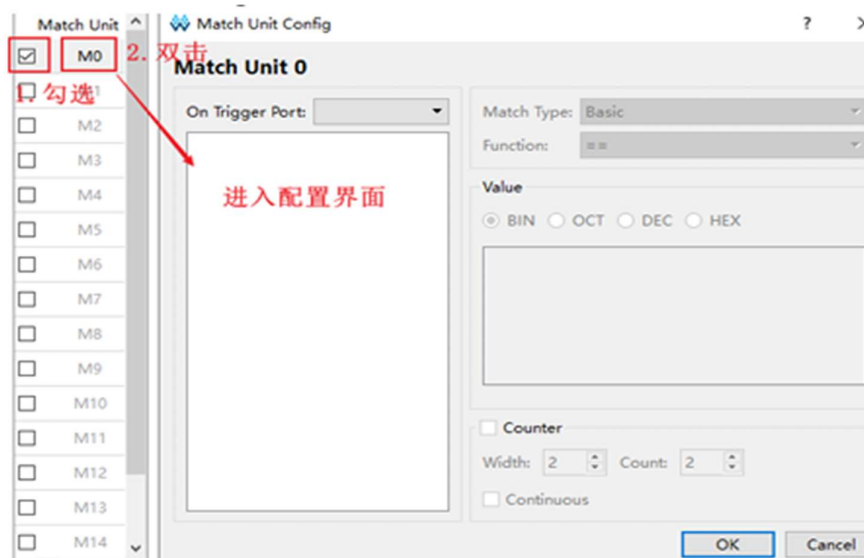


Figure 4-11 Entering Trigger Condition Configuration Interface

In the popped-up "Match Unit Config" dialog box, click the "On Trigger Port" drop-down box and select trigger port Trigger Port 0 in the drop-down list. The dialog box also contains "Match Type", "Function", and "Value" (used to set Bit Value). The descriptions for these items are shown in Table 4-1 below.

Table 4-1 Match Types Supported by Trigger Match Units

Type	Bit Value	Match Functions	Descriptions
Basic	0, 1, X	==, !=	Used for general signal comparison, a resource-saving type
Basic w/edges)	0, 1, X, R, F, B, N	==, !=, kiểm tra biên độ (transition)	Edge detection, Used when control signal transitions need to be considered
Extended	0, 1, X	==, !=, >, >=, <, <=	Used when address or data signal values need to be
Extended w/edges	0, 1, X, R, F, B, N	==, !=, >, >=, <, <=, kiểm tra biên độ	<=, Edge detection, Used when both values and transitions of address or data signals need to be considered.
Range	0, 1, X	>, >=, <, <=, kiểm tra trong/ngoài phạm vi	Range detection, Out of range detection, Used when values of address or data signals within a specific range need to be considered
Range w/edges	0, 1, X, R, F, B, N	>, >=, <, <=, kiểm tra phạm vi, kiểm tra biên độ	Range detection, Out of range detection, Edge detection, Used when values and transitions of address or data signals within a specific range need to be considered.

Note: In Bit Values, "0" represents low level 0, "1" represents high level, "X" represents either, "R" represents rising edge 0->1 change,

"F" represents falling edge 1->0 change, "B" represents either rising or falling edge transition, "N" represents no logic level transition.

Each trigger match unit has a counter used to set data sampling to start after the trigger condition is met N times. N is the counter value. This is related to whether the "Counter" checkbox is checked. Checking the "Counter" checkbox allows setting the use of the counter. If the counter is not used, data collection defaults to start after matching 1 time.

Users can configure the items in the "Match Unit Config" dialog box according to their own needs. Here we just use the default configuration. The final configuration is shown in Figure 4-12 below.

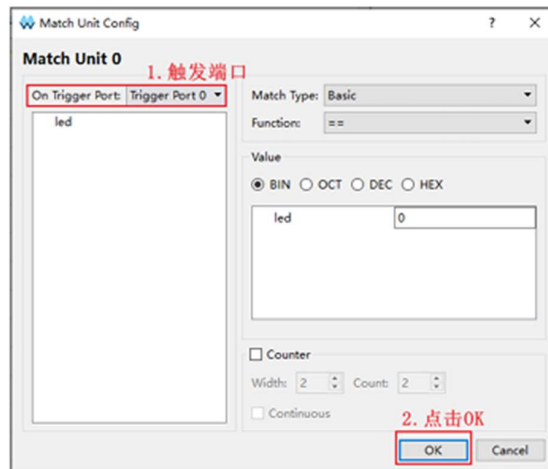


Figure 4-12 Configuring Trigger Condition

4. Configuring Trigger Expressions

The Expressions view is used to set trigger expressions. A functional core has up to 16 trigger expressions.

Right-click anywhere in the Expressions view and select "Add" to pop up the Expression dialog box, as shown in Figure 4-13 below.



Figure 4-13 Entering Expressions Dialog Box

Here we only have one trigger match unit M0. Click M0, then click OK, as shown in Figure 4-14 below.

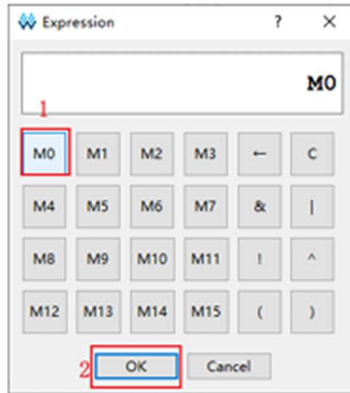


Figure 4-14 Selecting Trigger Expression

After adding, it is shown in Figure 4-15 below. Users can double-click to modify.



Figure 4-15 Trigger Expression Added Successfully

5. Configuring Sampling Signals

Click "Capture Options" to enter the sampling signal configuration view. The Capture Options view is mainly used to configure signal sampling information such as sampling clock, storage depth, sampling data signals, and display the number of BSRAM resources used by the Capture Signals of the current Ao Core. The Capture Options view is shown in Figure 4-16 below.

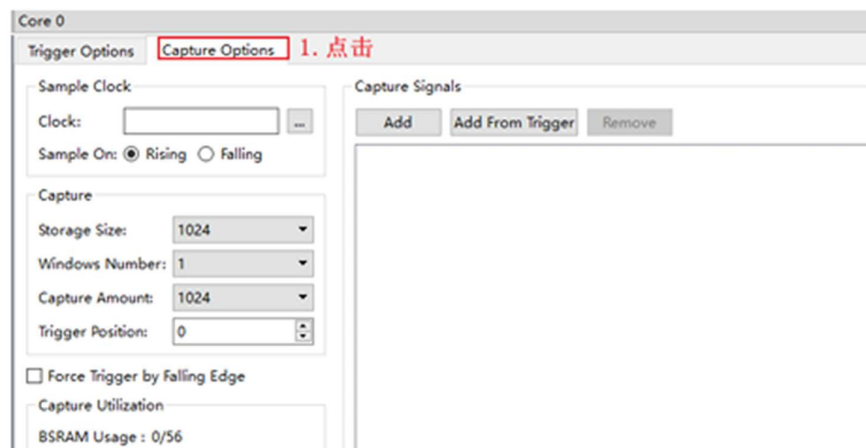


Figure 4-16 Capture Options View

First, configure the sampling clock Clock. Click the icon after Clock to enter the Search Nets interface. Directly enter the sampling clock clk in the file box, then click Search. After successful search, add it, as shown in Figure 4-17 below.

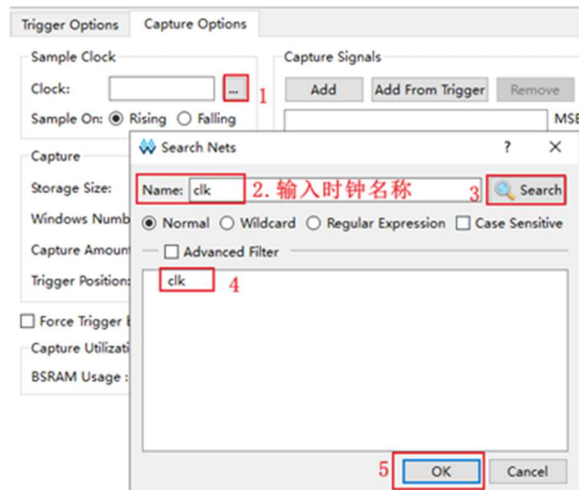


Figure 4-17 Adding Sampling Clock

Then the Storage Depth (Storage Size) is 4096, the number of acquisition windows is 1, the Sampling Length (Capture Amount) is also 4096, and the Trigger Position is 0, as shown in Figure 4-18 below. Storage depth refers to the allowable memory address length for data sampling. Sampling length refers to the actual sampling memory address length used for each acquisition buffer page. The sampling length cannot exceed the set storage depth. For example, if we set the storage depth to 8192 and the sampling length to 4096, the actual number of data points sampled during capture is 4096. Usually when setting, we set the storage depth and sampling depth to be consistent.

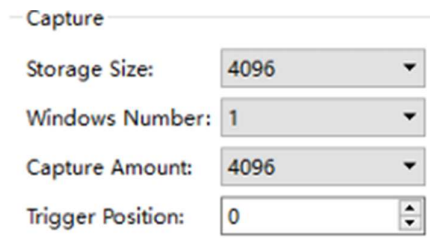


Figure 4-18 Setting Capture Options

Select the data signal to be sampled. Click the "Add" button to select the signal required by the functional core to sample and store data as the sampling data signal. Then the Search Nets dialog box pops up. Select the required data port signal and click "OK" to complete the configuration. Here bus signals can also be added. Here we add the cnt[25:0] signal for observation, as shown in Figure 4-19 below.

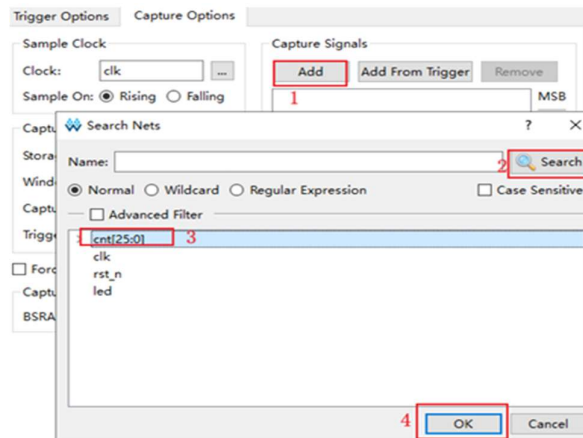


Figure 4-19 Adding Signals to Observe

There is also a button "Add From Trigger" in the figure above. This button directly uses the trigger port sampling trigger signal as the sampling data signal. One or more trigger ports can be selected in the list below Add From Trigger to use the selected trigger port acquisition signals as sampling data signals, as shown below.

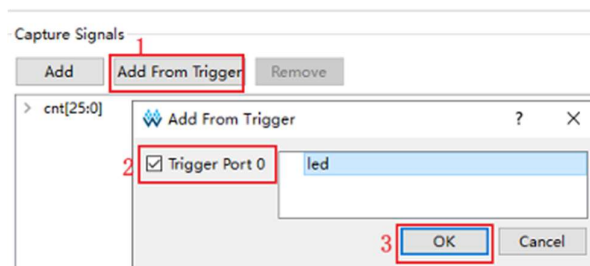


Figure 4-20 Selecting Trigger Signal as Sampling Signal

After adding, as shown in Figure 4-21 below, after selecting the signal, you can choose "Remove" to clear the signal.

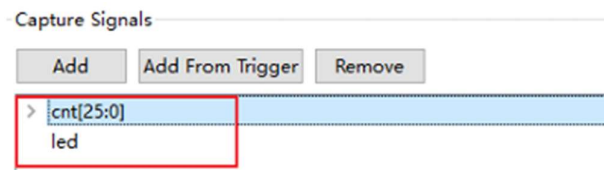


Figure 4-21 Sampling Signal Configuration Complete

At this point, the GAO file configuration is complete. Ctrl+S to save. If no errors are reported during saving, it means the configuration is error-free. If errors occur, please check your steps according to the operation steps to see if they are complete, if any link was missed, or if the configuration is wrong.

4.3 Place and Route

Every time after modifying the GAO file, you need to click "Place & Route" to re-place and route, as shown in Figure 4-22 below.

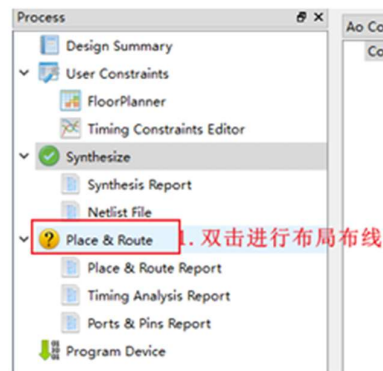


Figure 4-22 Re-Place and Route

4.4 Downloading fs File

Click to download the fs file. The operation is as shown below.

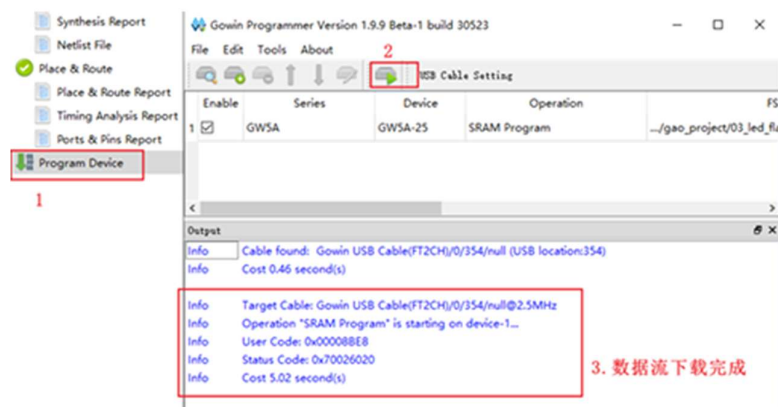


Figure 4-23 fs File Download Complete

4.5 Waveform Capture

Click the button in the toolbar to enter the Gowin online logic analysis interface, as shown in Figure 4-24 below.

Make sure that the Cable at the top is Gowin USB Cable(FT2CH). If not modified, it will not trigger, causing the waveform to not display.

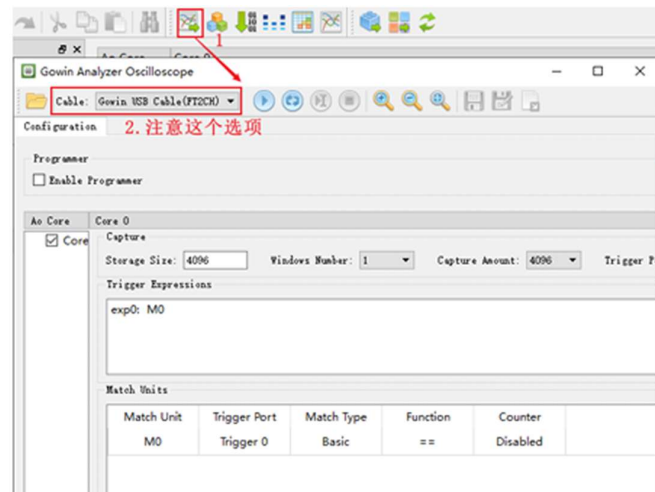


Figure 4-24 Entering Gowin Online Logic Analysis Interface

From the above interface, we can see that the sampling configuration of Core 0 can still be modified. Here we do not modify it, and directly click the icon above to observe the waveform. The waveform is shown in Figure 4-25 below.

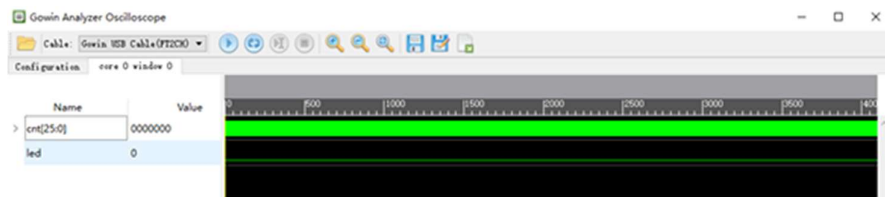


Figure 4-25 Waveform Interface Acquired by Gowin Online Logic Analyzer

At this time, click the button above to observe the change of cnt, as shown in Figure 4-26 below. It can be seen that cnt keeps accumulating. led does not change because the count value of cnt is too large to observe the change.

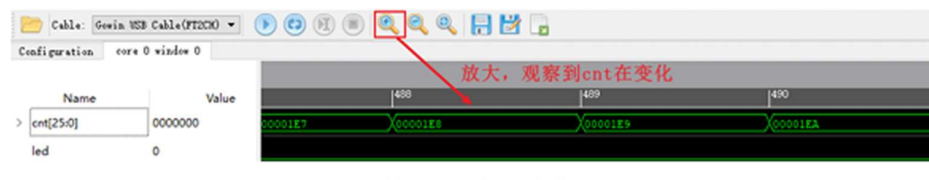


Figure 4-26 Observing cnt Change

If the user wants to see the change of led, the value of cnt can be reduced, for example, set to 299, as shown in Figure 4-27 below.

```

reg [25:0] cnt;
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        cnt <= 26'd0;
    else if (cnt < 26'd299)
        cnt <= cnt + 1'b1;
    else
        cnt <= 26'd0;
end

always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        led <= 1'b0;
    else if (cnt == 26'd299)
        led <= ~led;
    else
        led <= led;
end

```

Figure 4-27 Modifying cnt Parameter

After modification, the sampling signal cnt in the GAO file also needs to be modified, as shown in Figure 4-28 below. The modification here is because the actual count used is only bits 0~8 of cnt.

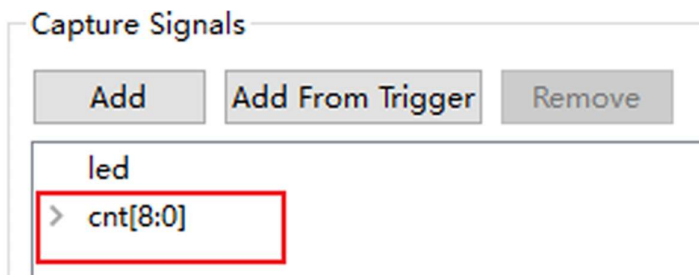


Figure 4-28 Modifying Sampling Signal cnt

Recompile and download the fs file. At this time, we cannot see the change of the LED light with the naked eye. Capture the waveform through the Gowin logic analyzer as shown in Figure 4-29 below. From the figure, it can be seen that the state of the LED light has been changing.

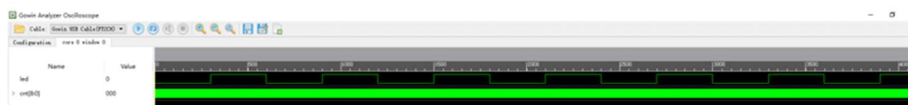
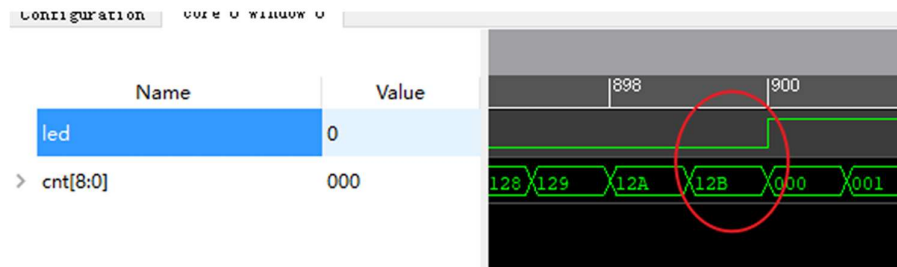


Figure 4-29 led Change After Modifying cnt

We zoom in on one of the edge changes as shown below. From the figure, it can be seen that after cnt counts to 12B, the LED changes. Converted to decimal, it corresponds to 299, which is consistent with our code design.

Figure 4-30 Waveform Edge Change Waveform Diagram



At this point, the basic usage flow of the Gowin online logic analyzer has been introduced. There are still many functions not used in this experiment, which will be supplemented after being used later.

4.6 Chapter Summary

In the development of FPGA, the use of online logic analyzers is very extensive. This chapter's experiment guided everyone to become familiar with how to use the online logic analyzer in the Gowin software. It is hoped that users will become familiar with the entire operation according to our steps so that they can use it in their own designed projects later.

Chapter 5 Sequential Logic Circuit Design: Counter

Project Source Code, |----02_Design Examples, |----ch5_bin_counter

Chapter Overview

Sequential logic circuits are circuits where the steady-state output at any time depends not only on the current input but also on the state formed by the previous input. This is contrary to combinational logic circuits, where the output of combinational logic is only a function of the current input. In other words, sequential logic possesses storage elements to store information, whereas combinational logic does not.

There are many types of sequential logic circuits. This section will use the most commonly used counter as an example to learn simple sequential logic circuit design and compare it with combinational logic circuits. Here, a counter is designed to toggle the state of the LED on the ACG525 development board every 500ms. The crystal oscillator on the ACG525 development board is 50MHz, meaning the clock period is 20ns. Thus, it can be calculated that $500\text{ms} = 500_000_000\text{ns} / 20\text{ns} = 25_000_000$, which means the counter needs to count 25_000_000 times. This requires a counter of at least 25 bits ($2^{25} > 25_000_000 > 2^{24}$). And whenever the count reaches the limit, it needs to be cleared and re-counted.

5.1 Counter Working Principle

The core element of a counter is the flip-flop. Its basic function is to count pulses. The maximum number of pulses it can remember is called the modulus/value of the counter. Counters are commonly used in frequency division, timing, etc. There are many types of counters. According to different counting methods, they can be divided into binary counters, decimal counters, and arbitrary modulus counters. According to the source of the flip-flop clock pulse signal, they can be divided into synchronous counters and asynchronous counters. According to the counting increment/decrement, they can be divided into up-counters, down-counters, and reversible counters.

Verilog HDL is called a Hardware Description Language because we are not doing ordinary programming like C, but writing an actual hardware circuit. Below we will design a counter to control an LED to flash. As mentioned above, the counter consists of an adder, a comparator, a register, and a multiplexer, as shown in Figure 5-1 below.

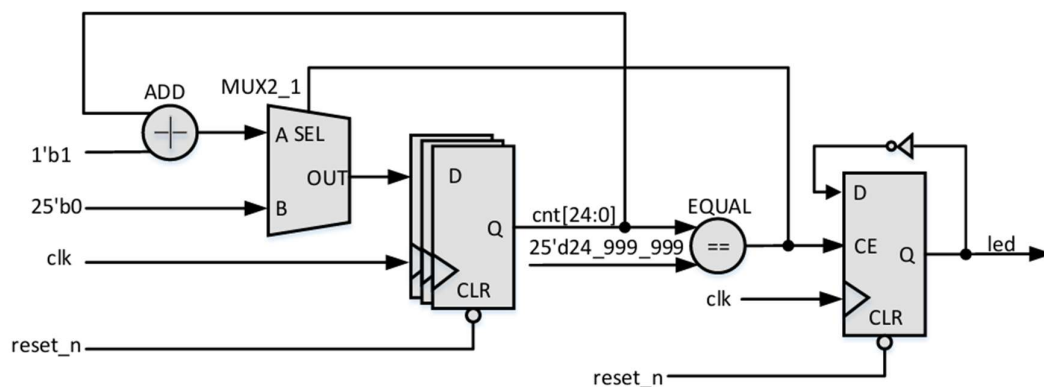


Figure 5-1 Counter Controlling LED Logic Circuit Diagram

The LED hardware circuit on the development board is shown in Figure 5-2 below. It can be seen that when the control terminal outputs a high level, the LED is on, and when it outputs a low level, the LED is off. Here, we only need to toggle the control terminal of the LED whenever the counter value is full to achieve the required LED blinking.

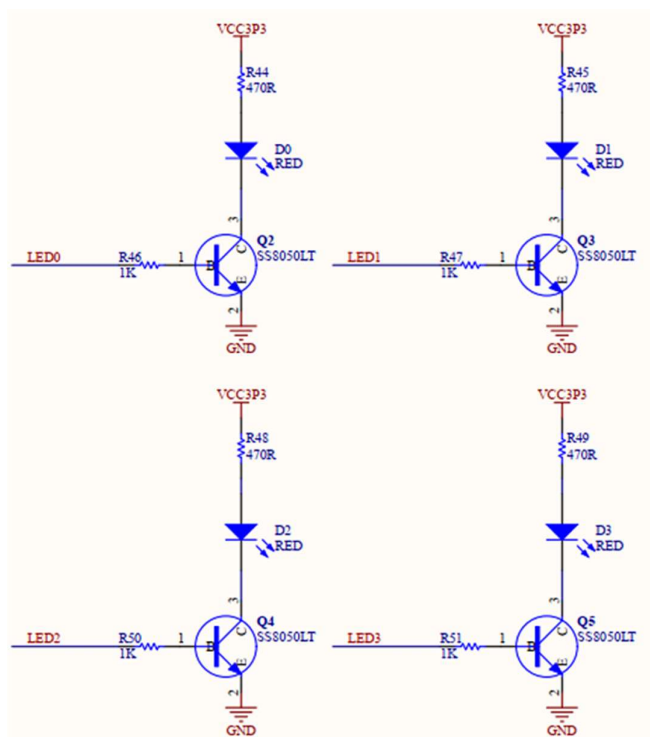


Figure 5-2 Schematic Diagram of LED Circuit on Development Board

5.2 Counter Verilog Implementation

From the above analysis, the module interface diagram of this design can be derived as shown in Figure 5-3 below. The functional description of each port is shown in Table 5-1 below.

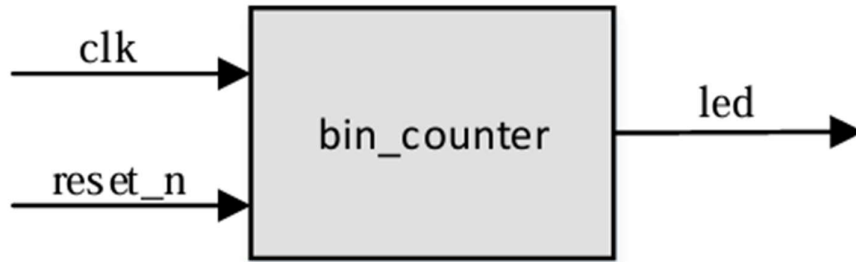


Figure 5-3 Counter Module Diagram

Tên cổng	I/O	Mô tả chức năng cổng
clk	I	Xung clock hoạt động của module, tần số 50MHz
reset_n	I	Tín hiệu reset cho module, reset mức thấp
led	O	Đầu ra điều khiển LED, mức thấp thì LED sáng

Table 5-1 Counter Module Port Function Description

5.2.1 New Project Creation

According to the relevant chapters described earlier, create a new project named bin_counter. Double-click the Gowin software icon on the desktop to open the Gowin development environment, and proceed as follows:

- (1) Start Page — Click New Project.
- (2) New Page — Click OK.
- (3) Project Name Interface — Project name bin_counter, set project path, click Next.
- (4) Select Device Interface — Select the chip model, judge based on the FPGA chip on your board (the model is marked on the chip), then click Next.
- (5) Summary Interface — Click Finish to complete project creation.

The created project is shown in Figure 5-4 below.

General	
Project File:	H:\01_gaoyun\gao_project\05_bin_counter\bin_counter\bin_counter.gprj
Synthesis Tool:	GowinSynthesis
Target Device	
Part Number:	GW5A-LV25UG324ES
Series:	GW5A
Device:	GW5A-25
Package:	UBGA324
Speed Grade:	ES
Core Voltage:	LV

Figure 5-4 Schematic of Project Creation Completion

5.2.2 Code Design

From the analysis of the counter module block diagram and the counter module port function table, the port list can be derived.

module bin_counter

```
(
    clk,
    reset_n,
    led
);

input clk;
input reset_n;
output led;
reg led;
```

endmodule

From the experimental principle, it can be seen that a counter is needed. Since the counter starts counting from 0 instead of 1, it is cleared when the count value reaches 25'd24_999_999 instead of 25'd25_000_000. Here, the maximum value of the counter is defined using a parameter. The advantage of using parameter definition is that by modifying the value defined in the parameter, all parameters used in the design can be modified. The specific code is as follows.

```
parameter MCNT = 24_999_999;

reg [24:0]cnt; // Định nghĩa thanh ghi đếm (counter)

//quá trình bộ đếm

always@(posedge clk or posedge reset)
```

```

if(reset)
    cnt <= 25'd0;
else
    if(cnt == MCNT)
        cnt <= 25'd0;
    else
        cnt <= cnt + 1'b1;

```

When the counter counts to the preset value, let the led invert once to achieve the purpose of blinking.

// led output control process

```

always@(posedge clk or posedge reset)
if(reset)
    led <= 1'b1;
else
    if(cnt == MCNT)
        led <= ~led;
    else
        cnt <= led;

```

5.3 Functional Simulation Verification

Analyze and synthesize the above design content until there are no errors or warnings. Create a new counter simulation file. The specific steps are as follows.

- (1) Click the icon in the toolbar to create a new simulation file.
- (2) Select Verilog File in the New popup window and click OK.
- (3) In the New Verilog file popup window, name the file bin_counter_tb in the Name column, keep others as default, and click OK. The simulation file is created successfully.

As shown in Figure 5-5 below, the newly created simulation file appears in Design->Verilog Files. Double-click the file name to enter the editing window.

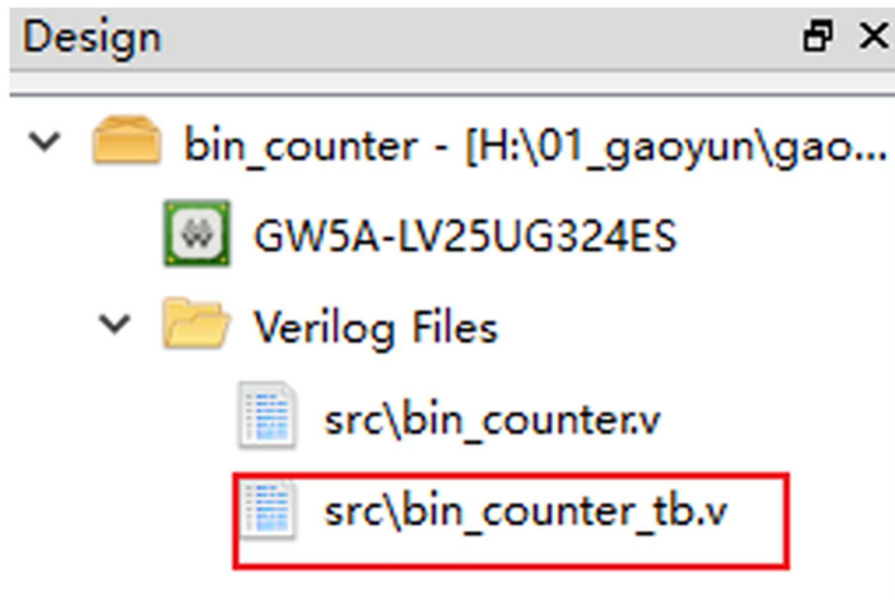


Figure 5-5 Schematic of Simulation File Creation Success

The simulation stimulus generates a clock `clk` with a period of 20ns and a reset signal, and instantiates the `bin_counter.v` to be tested.

```
`timescale 1ns/1ns
`define CLOCK_PERIOD 20
module bin_counter_tb();
    reg clk;
    reg reset_n;
    wire led;

    bin_counter bin_counter_inst (
        .clk(clk),
        .reset_n(reset_n),
        .led(led)
    );

    initial clk = 1;
    always #(`CLOCK_PERIOD/2) clk = ~clk;
    initial begin
        reset_n = 1'b0;
```

```

#(`CLOCK_PERIOD *200 + 1);

reset_n = 1'b1;

#2000000000;

$stop;

end

```

endmodule

After the simulation file is added, perform analysis and synthesis to check for grammatical errors. After checking that there are no errors, use Modelsim to perform simulation verification on the project.

5.3.1 Creating Modelsim Simulation Project

Create a Modelsim simulation project, follow the operations below:

- (1) Under the project directory, create a new sim folder to store simulation project files.
- (2) Click the Modelsim icon to enter the Modelsim software.
- (3) Click File->New->Project sequentially to create a new simulation project.
- (4) In the Create Project popup, name the project bin_counter in the Project Name column, click Browse to select the sim folder under the project directory for storage, and click OK.
- (5) The Add items to the Project dialog box pops up. Click Add Existing File->Browse to add the simulation files required for this experiment: bin_counter.v and bin_counter_tb.v, then click OK.
- (6) Click Compile->Compile All to compile all files.

After compilation is successful, it is shown in Figure 5-6 below. If errors occur, please modify them according to the prompts.

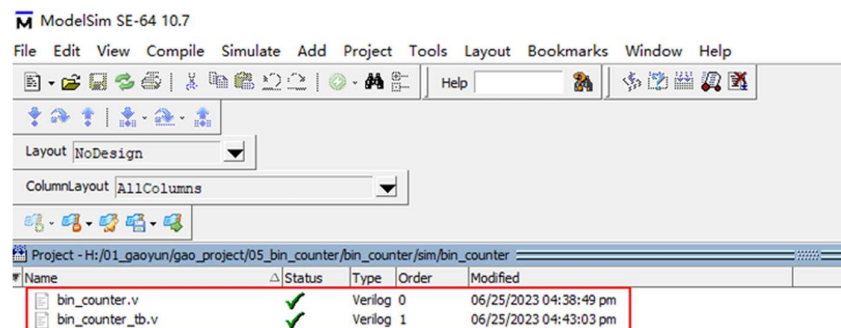


Figure 5-6 Schematic of File Compilation Success

5.3.2 Configuring Simulation Environment

After the simulation project is created and compiled, the next step is to configure the simulation environment. The operations are as follows:

(1) We right-click in the project status bar, and then select "Add to Project" ->

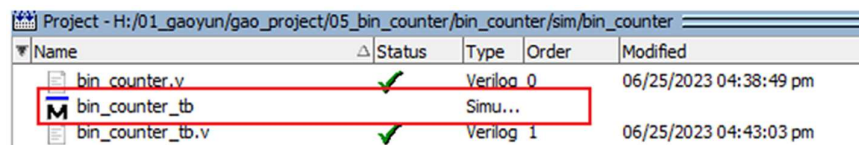
"Simulation Configuration...".

(2) Enter the Add Simulation Configuration page. In the Design tab page, we select the "bin_counter_tb" module in the work library as the design top layer. Click to copy the module name as the naming for "Simulation Configuration Name", ensuring the naming remains consistent.

(3) Click "Optimization Options...". In the "Optimization Options.." settings column, select "Apply full visibility to all modules(full debug module)", and click "OK".

(4) Click the "Libraries" settings column. In the "Search libraries(-L)" column, click "Add..." to add our newly created Gowin library file "gw5a". In "Search Libraries First(-Lf)", also select the library file "gw5a". Finally, click "Save" to save the settings.

Through the above steps, the configuration of the simulation environment is completed. After saving the configuration, we can see that the simulation configuration file "bin_counter_tb" is generated in the "project" column, as shown in Figure 5-7 below.



Name	Status	Type	Order	Modified
bin_counter.v	✓	Verilog	0	06/25/2023 04:38:49 pm
bin_counter_tb		Simu...		
bin_counter_tb.v	✓	Verilog	1	06/25/2023 04:43:03 pm

Figure 5-7 Schematic of Simulation File Generation Success

5.3.3 Observing Simulation Interface Waveforms

We first need to double-click the newly generated simulation file "bin_counter_tb" to enter the "sim" interface. In the "sim" interface, we can add the waveforms of the modules we want to observe. Select the module bin_counter_tb, right-click and select "Add Wave", enter the Wave window, click the icon in the toolbar to run at full speed. After the simulation stops, click the icon. The waveform is as shown in Figure 5-8.

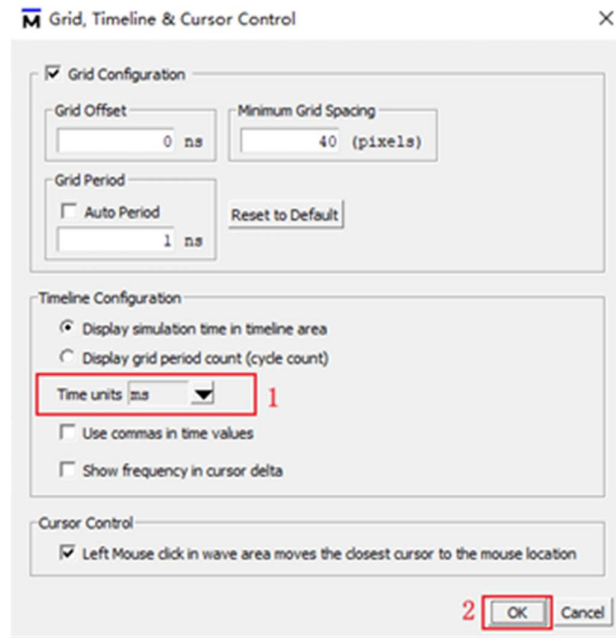


Figure 5-10 Simulation Display Time Unit Setting Interface

After changing the display time unit, observe the waveform high/low level time. As shown in Figure 5-11 below, the display unit has been changed to ms, and the high/low level time is 500ms. It is the same as above, just the display time unit has changed.

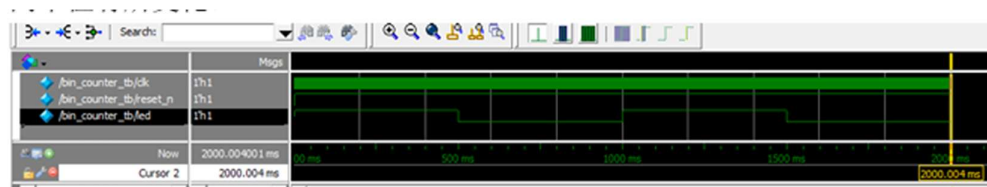


Figure 5-11 Waveform Diagram After Changing Simulation Display Time Unit

When performing the above functional simulation, it can be found that the computer needs to run for a long time. This is because the count value of the counter is too large. The maximum simulation count value can be changed to shorten the simulation time. Two methods of modification are introduced here. The first method is to modify the maximum count value defined by parameter in the design file bin_counter.v to 24_999. The specific code is as follows.

```
parameter MCNT = 24_999;
```

The second method is to modify the maximum value of the counter to 24_999 in the simulation file bin_counter_tb.v. The specific code is as follows.

```
defparam bin_counter_inst.MCNT = 24_999;
```

Through the above two methods, the simulation time can be shortened while verifying the correctness of the design. Comparing the above different methods, it can be found that the second

method is relatively better. The second method can achieve the purpose of shortening simulation time only by designing the simulation code without changing the design code. The modified simulation waveform is shown in Figure 5-12 below. The high/low level change time in the waveform diagram becomes 0.5ms, which is 1000 times shorter than 500ms, which also indicates that the functional simulation is correct. It should be noted that after modifying the code, you need to recompile and run the simulation in Modelsim.

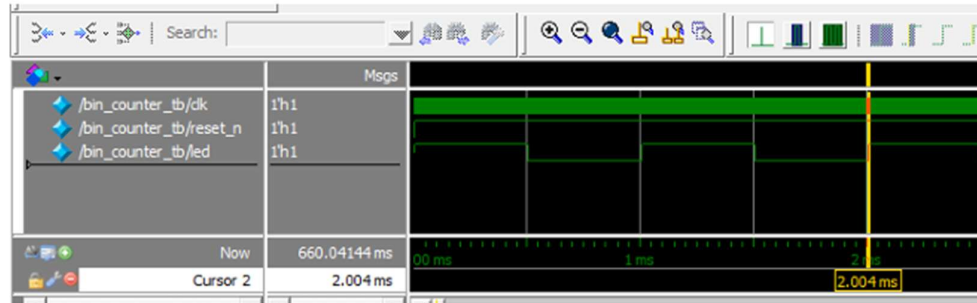


Figure 5-12 Functional Simulation Waveform After Reducing Count Value

5.4 Board-Level Debugging and Verification

The board-level verification part of this experiment mainly verifies the following goals:

1. Whether the generated fs file can be correctly downloaded to the development board.
2. After the download is complete, whether the LED light on the development board flashes every 1 second.

System Hardware Required:

3. ACG525 Development Board.
4. Gowin downloader and download cable.
5. 12V power supply.
6. A PC with complete development functions that meets the experiment requirements.

5.4.1 Adding I/O Constraints

Click FloorPlanner on the Process interface on the right side of the Gowin software to create a new .cst file to store I/O constraints, as shown in Figure 5-13 below.

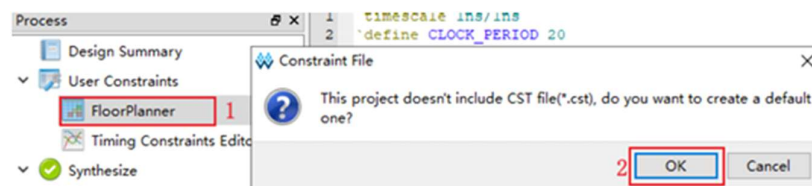


Figure 5-13 Creating New I/O Constraints

After entering the FloorPlanner interface, click I/O Constraints below to enter the I/O constraint interface. Assign pin positions according to the schematic. After addition is complete, it is shown in Figure 5-14 below.

I/O Constraints								
	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive
1	clk	input		T9	4	False	LVC MOS33	OFF
2	led	output		D14	1	False	LVC MOS33	8
3	reset_n	input		B16	1	False	LVC MOS33	OFF

Figure 5-14 Pin Assignment Interface

Thus, pin constraints are added. But at this time, the constraint content is saved in memory and has not been written to the file. Click the Save button on the toolbar, or directly press Ctrl+S.

After closing the FloorPlanner interface, click Design on the right side. You can see our newly added .cst file. Double-click to view the file content, as shown in Figure 5-15 below.

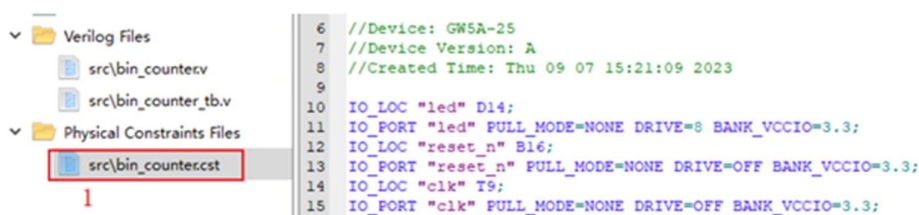


Figure 5-15 I/O Constraint File

5.4.2 Adding Timing Constraints

We need to click "Timing Constraints Editor" in the Process interface for timing constraints. At this time, it will pop up that there is no constraint file in this project, do you want to create a constraint file? Click OK, as shown in Figure 5-16 below.

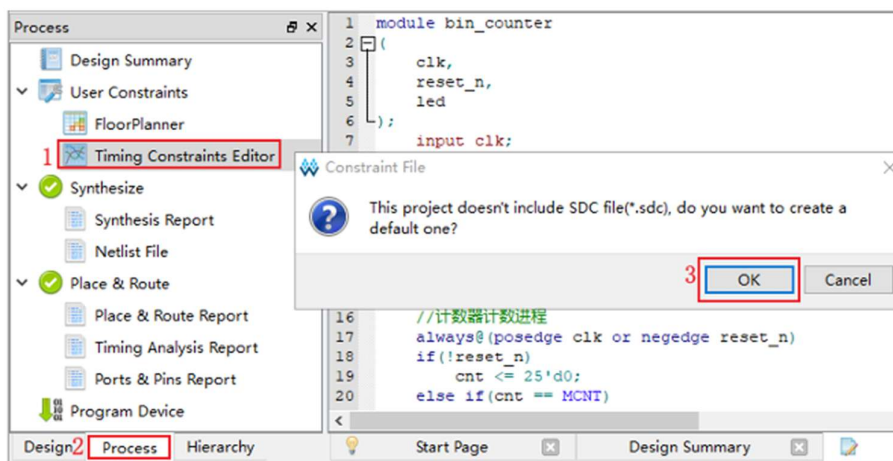


Figure 5-16 Creating Timing Constraint File

Then enter the timing constraint interface, as shown in Figure 5-17 below.

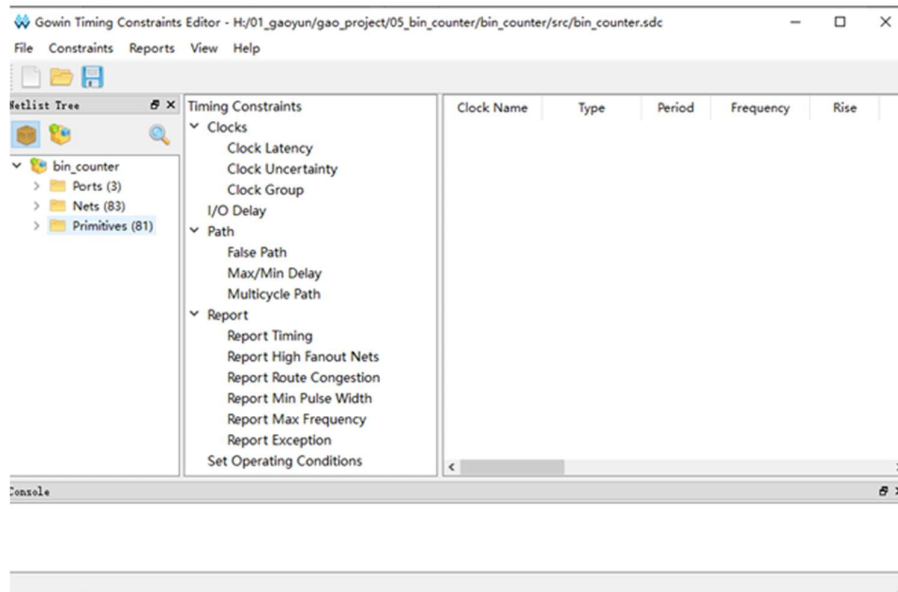


Figure 5-17 Timing Constraint Interface

Select Clocks under Timing Constraints, right-click in the blank space on the right, and select Create Clock, as shown in Figure 5-18 below.

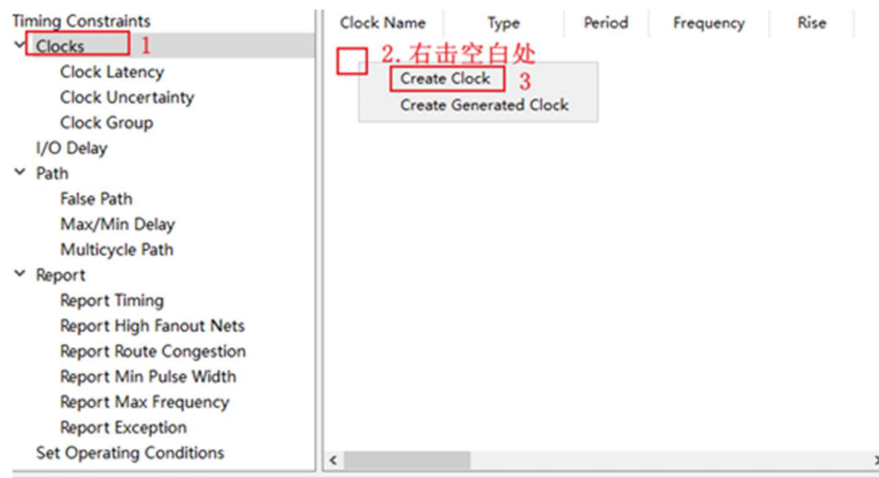


Figure 5-18 Steps to Create Clock Constraint

The Create Clock interface pops up, as shown in Figure 5-19 below.

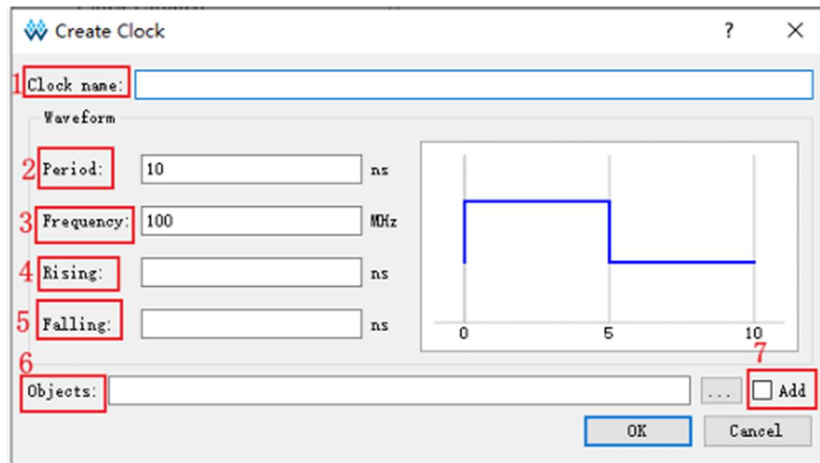


Figure 5-19 Create Clock Interface

Brief description of the parameters shown in the figure above:

1. Clock Name: Clock name, supports identifiers starting with letters or underscores.
2. Period: Period, default 10, floating-point type, precise to thousandths.
3. Frequency: Frequency. Default 100, floating-point type, precise to thousandths.
4. Rising: Rising edge time, floating-point type, precise to thousandths.
5. Falling: Falling edge time, floating-point type, precise to thousandths.
6. Objects: Specify the target, use the selection tool for automatic content filling.
7. Add: Check when adding multiple clocks on the same source.

We click the icon to select the constraint clock settings, and the window shown in Figure 5-20 below pops up.

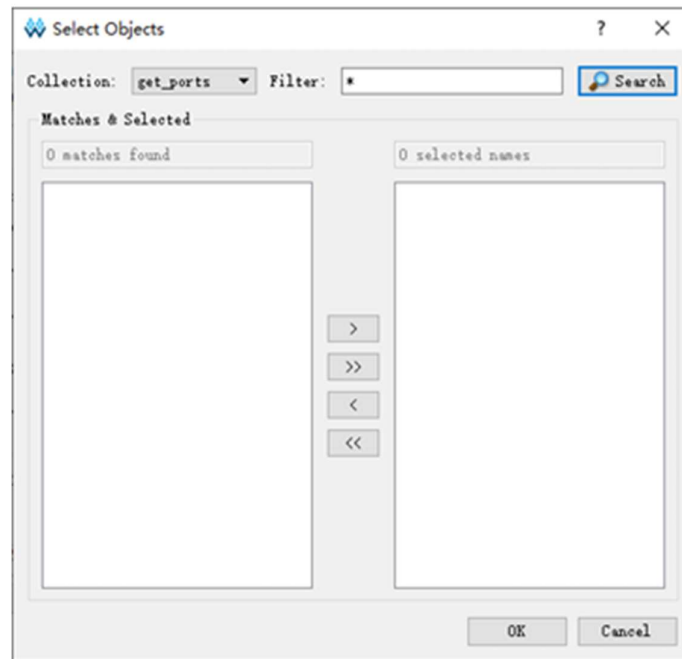


Figure 5-20 Select Objects Interface

The clock that needs to be constrained here is the 50M system clock coming from the pin. Click Search->clk-> ->OK sequentially. The operation is as shown in Figure 5-21 below.

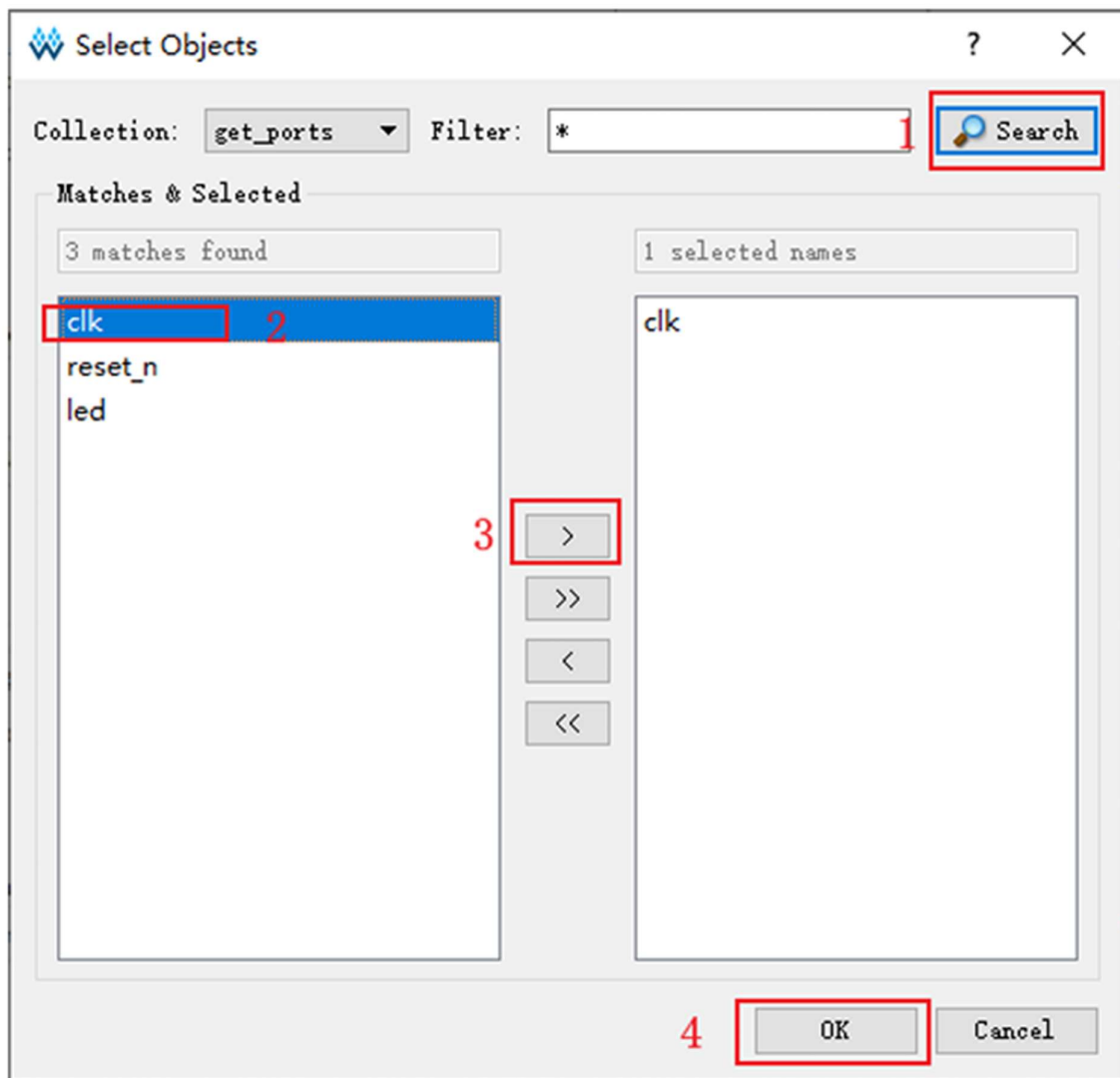


Figure 5-21 Selecting Clock to Constrain

Then return to the Create Clock interface. Name the clock clk, period 20ns, frequency 50Mhz, Rising 0, Falling. The final Create Clock interface is shown in Figure 5-22 below.

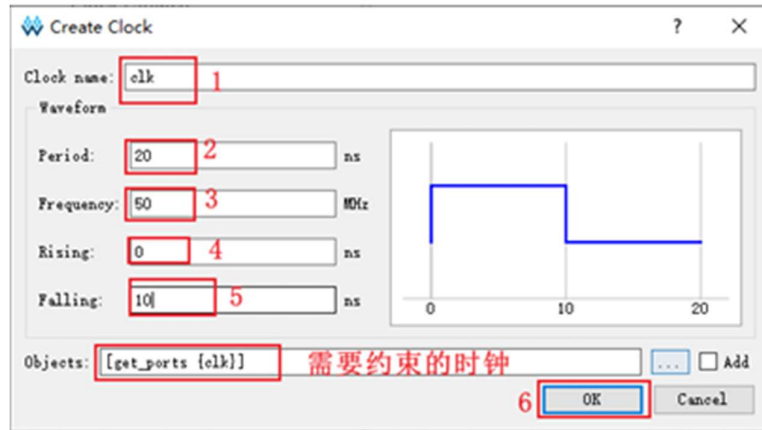


Figure 5-22 Create Clock Configuration Interface

After adding, you can see the clock constraint we just added in the blank space on the right. Click to save, as shown in Figure 5-23 below.

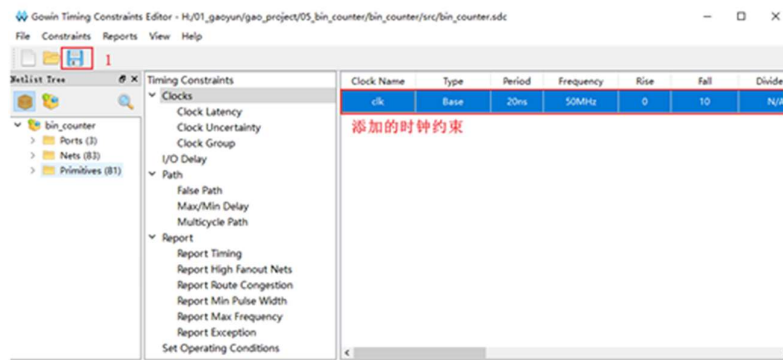


Figure 5-23 Saving Timing Constraints

Through the above operations, the timing constraint of the 50Mhz clock is completed. Close the Gowin Timing Constraints Editor interface. At this time, in the Design interface, you can see the timing constraint file bin_counter.sdc we added. Double-click to view the constraint file content, as shown in Figure 5-24 below.

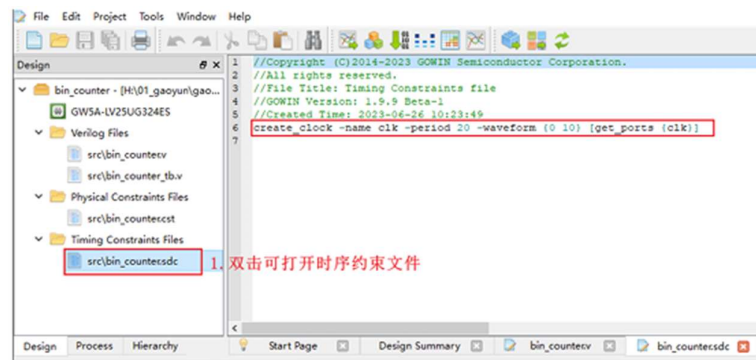


Figure 5-24 Method to View Timing Constraint File

5.4.3 Place and Route

Click Place & Route in Process to perform place and route. After successful place and route, the fs file will be generated. The Console prompt is as shown in Figure 5-25.

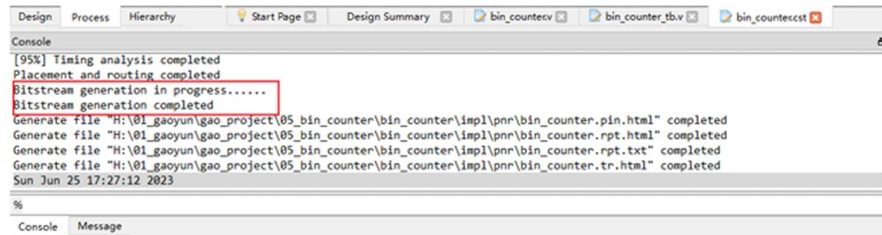


Figure 5-25 Schematic of fs File Generation Success

5.4.4 Hardware Connection

Connect the downloader and power supply to the development board in sequence. The overall hardware connection is shown in Figure 5-26 below.



Figure 5-26 Hardware Connection Diagram

5.4.5 Downloading fs File

Click Program Device to enter the download interface. Then the Cable Setting interface pops up. Click Save, then click the icon to download the file. After the download is complete, it is shown in Figure 5-27 below.

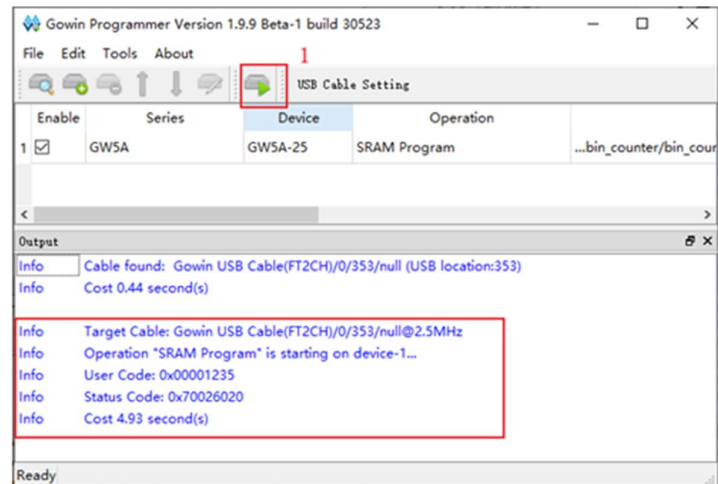


Figure 5-27 Schematic of Successful Download

After Program is complete, you can see LED0 toggling on/off every 1 second. The experiment proves that the design meets expectations.

5.5 Reflection and Summary

This chapter used a counter as an example to learn simple sequential logic design and briefly introduced the method of clock constraints. Readers should use this as a basis to design a logic circuit to drive the LED light to flash at different frequencies, and perform simulation and board-level verification.

Chapter 6: Serial Port Transmit Module Design and Verification

Project Source Code, |----02_Design Examples, |----ch6_uart_tx

Chapter Overview

In today's electronic systems, it is often necessary to send and receive data within the board, between boards, or between lower-level computers and upper-level computers. This requires both parties to jointly follow certain communication protocols to ensure the correctness of data transmission. Common protocols include UART (Universal Asynchronous Receiver/Transmitter), IIC (Inter-Integrated Circuit), SPI (Serial Peripheral Interface), USB2.0/3.0 (Universal Serial Bus), and Ethernet. Among these protocols, the most basic one is UART. Due to its simple circuit structure and low cost, it is widely used where cost-effectiveness is a priority.

This chapter will learn the principle of UART communication and its hardware circuit design, and use FPGA to implement the data transmission part design in UART communication.

6.1 Asynchronous Serial Communication Principle and Circuit Design

6.1.1 RS232 Communication Interface Standard

Universal Asynchronous Receiver/Transmitter (UART) is an asynchronous transceiver. During data transmission, it converts parallel data into serial data for transmission, and during data reception, it converts the received serial data into parallel data. It can achieve full-duplex transmission and reception. It includes interface standards and bus standards such as RS232, RS449, RS423, RS422, and RS485. In other words, UART is a general term for asynchronous serial communication. RS232, RS449, RS423, RS422, and RS485 correspond to the interface standards and bus standards of various asynchronous serial communication ports. They specify the electrical characteristics, transmission rate, connection characteristics, and mechanical characteristics of the interface of the communication port.

The RS-232 to be focused on in this chapter is a serial data communication interface standard formulated by the Electronic Industries Alliance (EIA) of the United States. The full name of the original number is EIA-RS-232 (referred to as 232, RS232), which is widely used for computer serial interface peripheral connections. Its DB9 interface pin definition is shown in Figure 6-1 below, and the pin functions are shown in Table 6-1 below. If multiple UART interfaces exist in the system, they can be referred to as COM1, COM2, etc., respectively.

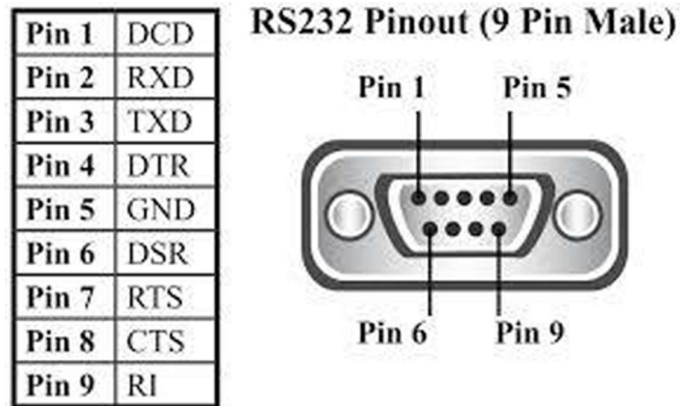


Figure 6-1 RS232 DB9 Pin Definition

Table 6-1 Pin Functions

Chân	Ký hiệu	Ý nghĩa	Décription
Pin1	CD	Carrier Detect	Modem notifies computer that a carrier is detected
Pin2	RXD	Receiver	Receive Data
Pin3	TXD	Transmit	Transmit Data
Pin4	DTR	Data Terminal Ready	Computer tells modem it is ready to transmit
Pin5	GND	Ground	Ground
Pin6	DSR	Data Set Ready	Modem tells computer everything is ready;
Pin7	RTS	Request to send	Computer requests modem to submit data;
Pin8	CTS	Clear to Send	Modem notifies computer it can transmit data;
Pin9	RI	Ring Indicator	Modem notifies computer there is an incoming call.

6.1.2 UART Key Parameters and Timing Diagram

UART communication requires multiple settings before use. The most common settings include data bits, baud rate, parity type, and stop bits.

Data bits: This parameter defines the number of data bits sent during a single UART data transmission from start to stop. Can be selected as: 5, 6, 7, or 8 (default).

Baud Rate (Baud): Refers to the baud rate sent from one device to another, that is, the number of data bits that can be communicated per second. Typical baud rates are 300, 1200, 2400, 9600, 19200, 115200, etc. Generally, devices at both ends of the communication must be set to the same baud rate, but some devices can also be set to automatically detect the baud rate.

Parity Type: Used to verify the correctness of data. Parity is generally not used. If used, it can be either Odd parity or Even parity. In even parity, because the parity bit will be set to 1 or 0 accordingly (usually the highest bit or the lowest bit), the data will be changed so that the number of "1"s in all transmitted digits (including data bits and check bit) is even; in odd parity, the number of "1"s in all transmitted digits (including data bits and check bit) is odd. Parity can be used by the receiver to check if an error occurred during transmission. If the number of "1"s in a byte is incorrect, an error must have occurred during transmission. If the parity is correct, then either no

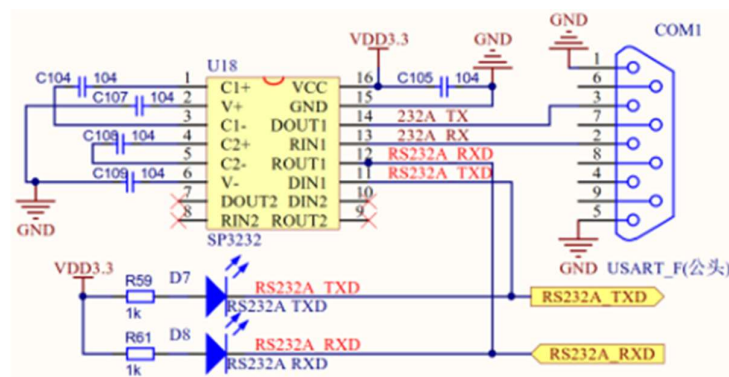


Figure 6-3 RS232 to TTL Circuit Diagram

Nowadays, system integration is getting higher and higher. The RS232 interface of DB9 occupies too much PCB area. Most systems have switched to USB to serial port solutions. The ACG525 development board also adopts the USB to serial port solution, and its circuit diagram is shown in Figure 6-4 below.

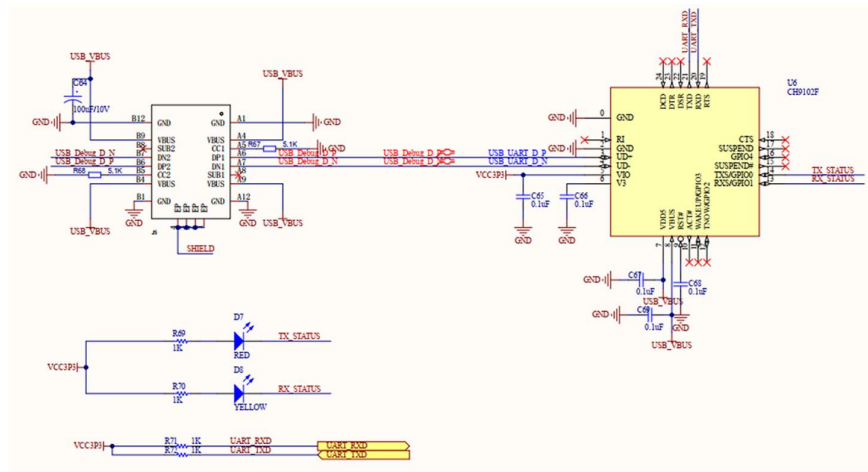


Figure 6-4 ACG525 Development Board Serial Port Circuit Diagram

CH9102F is a USB bus adapter chip that implements USB to high-speed asynchronous serial port. It provides common MODEM contact signals for expanding asynchronous serial ports for computers, or directly upgrading ordinary serial devices or MCUs to the USB bus. Its features are as follows:

1. Full-speed USB device interface, compatible with USB V2.0.
2. Built-in firmware, emulates standard serial port, used to upgrade original serial peripherals, or add extra serial ports via USB.
3. Supports installation-free operating system built-in CDC class drivers or multi-function high-rate VCP vendor drivers.
4. Hardware full-duplex serial port, built-in independent transceiver buffer, supports communication baud rate 50bps ~ 4Mbps.
5. Serial port supports 5, 6, 7, or 8 data bits, supports odd parity, even parity, blank, mark, and no parity.
6. Supports common MODEM contact signals RTS, DTR, DCD, RI, DSR, CTS. Supports CTS and RTS hardware automatic flow control.
7. Supports half-duplex, provides sending status TNOW to support RS485 switching.
8. Supports RS232 interface through external level conversion device.
9. CH9102F serial port I/O independent power supply, supports 5V, 3.3V, and 2.5V or even 1.8V power supply voltage.
10. CH9102X serial port I/O supports 3.3V signals.
11. Built-in power-on reset, built-in clock, no external crystal required.
12. CH9102F built-in EEPROM, can configure chip VID, PID, maximum current value, vendor and product information strings, and other parameters.

13. Chip built-in Unique ID (USB Serial Number).
14. Provides QFN24 and QFN28 lead-free packages, RoHS compliant.

6.1.4 UART Driver Installation

When using the serial port for the first time, we need to install the driver first. The operation steps are as follows:

1. First go to the official website of Nanjing Qinheng Company to download the driver installation software. The official website link is as follows:

<https://www.wch.cn/>

After entering the official website, find the USB Chip on the homepage, then enter the new interface, select USB to UART, CH9102 in sequence. After clicking, enter the CH9102 introduction interface, scroll down, find CH343SER.EXE to download. The entire operation steps are shown in Figure 6-5 below.



Figure 6-5 Downloading Serial Port Driver Installation Software from Official Website

2. Double-click to open the software. If a security warning dialog box prompts, just click Run, as shown in Figure 6-6 below.



Figure 6-6 Running Software

3. After opening, the interface is as shown in Figure 6-7 below. Just click Install.

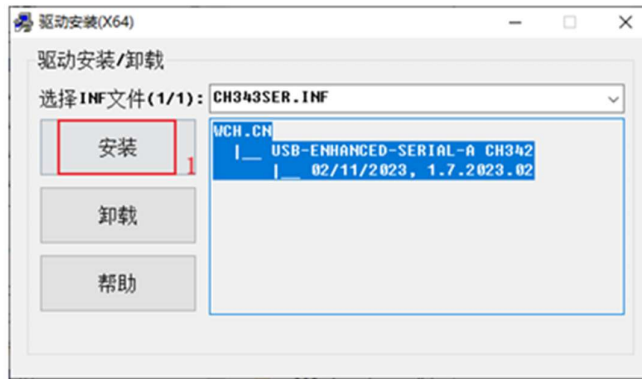


Figure 6-7 Installing Driver

4. Wait for the installation. After successful installation, a pre-processing installation success prompt dialog box will pop up as shown below. At this time, it means our driver installation is successful. Click OK.

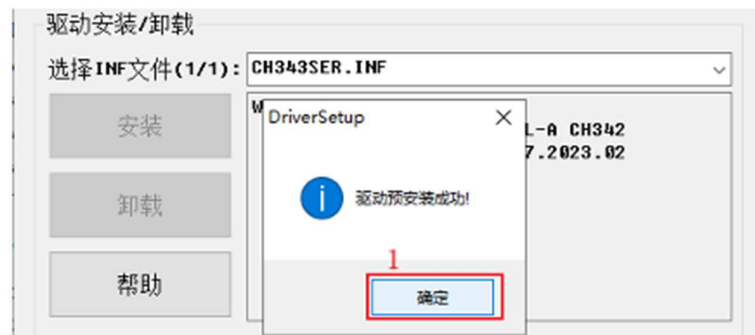


Figure 6-8 Driver set up successful

6.2 UART Asynchronous Serial Communication Transmit Module Design and Implementation

Based on the above principles, the overall block diagram of the serial transmit module to be implemented in this chapter is shown in Figure 6-9 below. Its interface list is shown in Table 6-2 below.

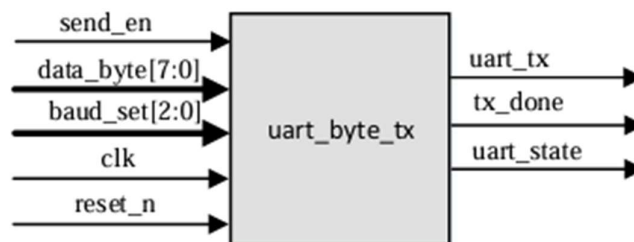


Figure 6-9 Serial Port Transmit Module Overall Design Block Diagram

Table 6-2 Module Interface List

Sinal	I/O	Mfunction Description
clk	I	Module global clock 50MHz
reset_n	I	Module global reset signal
data_byte	I	8bit data to be transmitted
send_en	I	Transmit enable signal
baud_set	I	Baud rate setting signal
uart_tx	O	Serial transmit signal output
tx_done	O	Transmission done signal, one clock cycle high level
uart_state	O	Transmit status, 1 when transmitting

According to the functional requirements, the serial port transmit module can be further detailed as the detailed structure diagram shown in Figure 6-10 below. The function of each sub-module is shown in Table 6-3 below. The green boxes represent registers with a single structure to achieve stable data input and output.

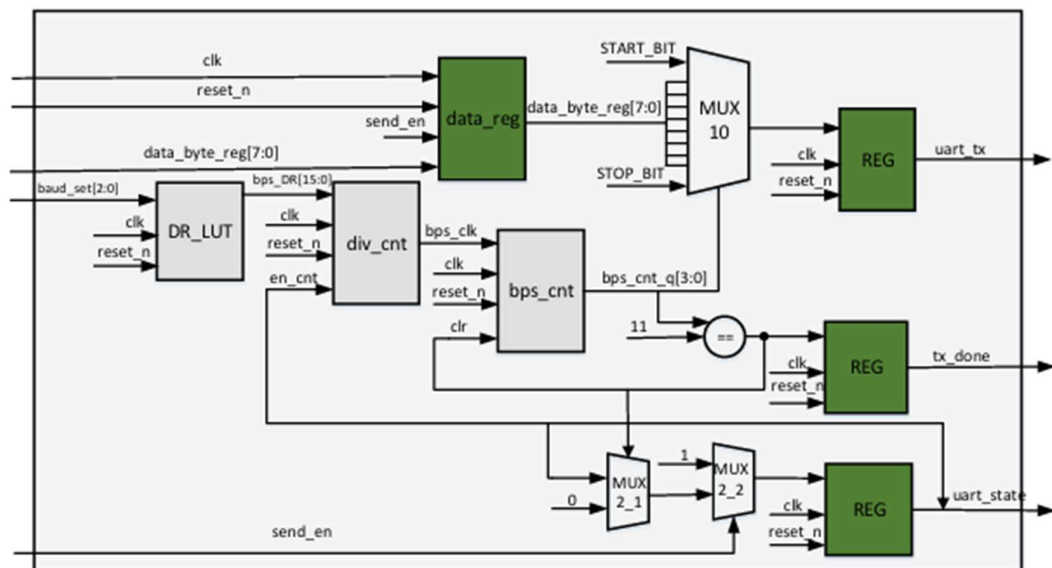


Figure 6-10 Serial Port Transmit Module Structure Diagram

Table 6-3 Sub-module Function Description

Name	Function Description
div_cnt	Generates baud rate clock
Bps_cnt	Counts baud rate clocks
DR_LUT	Look-up table to adapt to different baud rates
MUX10	Determines the bit to be sent based on bps_cnt value
Toán tử “==”	operator, Judges whether a transmission is completed
MUX2	MUX2, Module enable signal

6.2.1 Baud Rate Clock Generation Module Design

From the principle part, we know that the baud rate is one of the parameters that need to be set in UART communication. In the baud rate clock generation module, the relationship between the count value required by the counter and the baud rate is shown in Table 6-4 below, where the system clock period is System_clk_period, which is 20ns here. If the clock frequency connected to this module is other values, this parameter needs to be modified according to the specific frequency value.

Table 6-4 Baud Rate Calculation

baud_set	Baud rate	Baud period	Clock count calculation	Divisor Value
0	9600	104167ns	104167/System_clk_period	5208-1
1	19200	52083ns	52083/System_clk_period	2604-1
2	38400	26041ns	26041/ System_clk_period	1302-1
3	57600	17361ns	17361/ System_clk_period	868-1
4	115200	8680ns	8680/ System_clk_period	434-1

This module is designed to ensure module reusability. When different baud rates are needed, only the count value of the baud rate clock counter needs to be set differently. This can be implemented using a look-up table. The following design code only includes settings for 5 baud rates. If other baud rates are needed, this can be modified according to actual usage.

```

always@(posedge clk or posedge reset_n)
if(reset_n)
    bps_DR <= 16'd5207;
else begin
    case(baud_set)
        0:bps_DR <= 16'd5207;
        1:bps_DR <= 16'd2603;
        2:bps_DR <= 16'd1301;
        3:bps_DR <= 16'd867;
        4:bps_DR <= 16'd433;
        default:bps_DR <= 16'd5207;
    endcase
end
always@(posedge clk or posedge reset_n)
if(reset_n)
    div_cnt <= 16'd0;
else if(uart_state) begin
    if(div_cnt == bps_DR)
        div_cnt <= 16'd0;
    else div_cnt <= div_cnt + 1'b1;
end

```

```

end
else
    div_cnt <= 16'd0;
// bps_clk gen
always@(posedge clk or posedge reset_n)
if(reset) bps_clk <= 1'b0;
else if(div_cnt == 16'd1)
    bps_clk <= 1'b1;
else bps_clk <= 1'b0;

```

Baud rate generation essentially involves using a timer to time and generate a clock signal with the same frequency as the corresponding baud rate clock. For example, if we use a baud rate of 115200bps, we need to generate a clock signal with a frequency of 115200Hz. How to generate such a 115200Hz clock signal? Here, we first calculate the period of the 115200Hz clock signal. 1 second is 1000_000_000ns, so the baud rate clock period $T_b = 1000000000/115200 = 8680.6\text{ns}$. That is, one period of 115200 bps is 8680.6ns. So, we only need to set our timer timing time to 8680.6ns. Whenever the timing time is reached, generate a high pulse signal with the length of a system clock cycle. The system clock frequency is 50MHz, meaning the period is 20ns. So, we only need to count 8680/20 system clocks to obtain 8680ns timing. $\text{bps } 115200 = T_b/T_{\text{clk}} - 1 = T_b \cdot f_{\text{clk}} - 1 = f_{\text{clk}}/115200 - 1$. Correspondingly, the calculation of other baud rate timing values is the same.

To control the baud rate externally, a 3-bit baud rate selection port: baud_set is used in the design. By giving this port different values, different baud rates can be selected. The principle of controlling different baud rates with this port is simple, it is a multiplexer. The multiplexer sets different bit rate clock frequencies by selecting different timer maximum count values. The relationship between the baud_set value and each baud rate is shown below.

Table 6-5 Correspondence between Baud Rate and baud_set Parameter Value

Code	Baud Rate
000	9600 bps
001	19200 bps
010	38400 bps

6.2.2 Data Output Module Design

By counting the baud rate clock, the data transmission loop status is determined. The code is as follows:

```
//bps counter
always@(posedge clk or posedge reset_n)
if(reset_n) bps_cnt <= 4'd0;
else if(bps_cnt == 4'd11)
    bps_cnt <= 4'd0;
else if(bps_clk)
    bps_cnt <= bps_cnt + 1'b1;
else bps_cnt <= bps_cnt;
```

Also, to enable the module to control or call other modules, a byte transmission end signal is generated here. After the transmission of one data bit is completed, the tx_done signal outputs a high level for one clock.

```
always@(posedge clk or posedge reset_n)
if(reset_n)
    tx_done <= 1'b0;
else if(bps_cnt==4'd11)
    tx_done <= 1'b1;
else
    tx_done <= 0;
```

Generate the data transmission status signal, that is, the uart_state signal is high level during normal transmission, and low level otherwise. The circuit structure implemented here also has a priority order, but it is fundamentally different from the C language. MUX2_1 and MUX2_2 in Figure 6-10 are the if—else if—else circuit structures implemented in the following design.

```
always@(posedge clk or posedge reset_n)
if(reset_n)
    data_byte_reg<=8'd0;
else if(send_en)
    uart_state <= 1'b1;
else if (bps_counter==4'd11)
```

```

        uart_state <= 0;

else

        uart_state = uart_state;

```

Since the serial port is an asynchronous transceiver, to ensure that the transmitted data is stable when the clock arrives, the input data also needs to be registered here.

```

always@(posedge clk or posedge reset_n)

if(send_en)

    data_byte_reg <= data_byte;

else

    data_byte_reg <= data_byte_reg;

```

6.2.3 Data Transmission State Control Module

There is also a 10-to-1 multiplexer in the module structure diagram 6-10, which functions to determine the state of data transmission based on the value of bps_cnt. As shown in the timing diagram 6-2, at different baud rate clock count values, there are corresponding transmission data.

```

always@(posedge clk or posedge reset_n)
if(reset_n)
    uart_tx <= 1'b1;
else begin
    case(bps_cnt)
        0: uart_tx <= 1'b1;
        1: uart_tx <= START_BIT;
        2: uart_tx <= data_byte_reg[0];
        3: uart_tx <= data_byte_reg[1];
        4: uart_tx <= data_byte_reg[2];
        5: uart_tx <= data_byte_reg[3];
        6: uart_tx <= data_byte_reg[4];
        7: uart_tx <= data_byte_reg[5];
        8: uart_tx <= data_byte_reg[6];
        9: uart_tx <= data_byte_reg[7];
        10: uart_tx <= STOP_BIT;
        default: uart_tx <= 1'b1;
    endcase
end

```

At this point, the code design is complete. Perform analysis and synthesis on the design until there are no errors.

6.3 Stimulus Creation and Simulation Testing

The schematic generated from the above analysis is basically consistent with the module structure diagram we designed, but this does not mean that the designed module's function will meet our requirements. This requires functional simulation of the design module to verify the correctness of the module function. First, design the simulation testbench file. In the simulation file, a clock clk with a period of 20ns is generated as the working clock of the serial port module, and then the reset signal, enable signal, and data to be transmitted are generated. Here, all data changes are staggered by 1ns with the system clock, so that the timing relationship between input/output data and the clock can be seen more clearly.

```
initial clk = 1;
always #(`CLK_PERIOD/2) clk = ~clk;
initial begin
    reset_n = 1'b0;
    data_byte = 8'd0;
    send_en = 1'd0;
    baud_set = 3'd4;

    #(`CLK_PERIOD*500 + 1);
    reset_n = 1'b1;
    #(`CLK_PERIOD*50);

    // Send first byte
    data_byte = 8'haa;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;

    @(posedge tx_done);
    #(`CLK_PERIOD*5000);

    // Send second byte
    data_byte = 8'h55;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;

    @(posedge tx_done);
    #(`CLK_PERIOD*5000);
    $stop;
End
```

Then open Modelsim to create a new function and get the waveform file shown in Figure 6-11 below. It can be seen that before the reset and enable signals are valid, the output signal uart_tx is 1. After the reset ends and enables, the output signal starts to send a set of serial data. And when the data to be sent is 0xaa, the serial output signal is sequentially 1, 0 (start bit), 01010101b (LSB), 1 (stop bit); when the data to be sent is 0x55, the output signal is sequentially 1, 0 (start bit), 10101010b (LSB), 1 (stop bit). At the same time, during the transmission of serial data, uart_state is

1 when in the transmitting state. After each byte of data is sent, tx_done generates a clock cycle pulse, which is consistent with the design expectations, and the simulation passes.

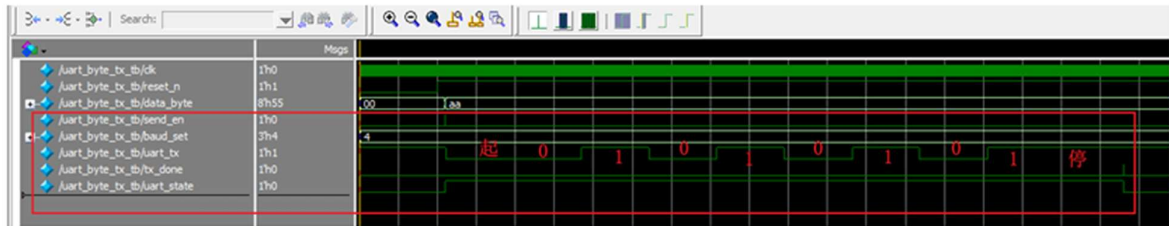


Figure 6-11 Single byte Data Transmission Simulation Waveform Diagram

6.4 Serial Port Transmit Test Module

Before performing board-level verification on the single-byte serial transmit module, we need to write a serial transmit test module to control when the single-byte serial transmit module sends data and the content of the data sent. Here our design idea is: use a counter with a count time of 1S. When the count time is up, generate the send_en signal. After a transmission is completed and tx_done is generated, the data to be sent data_byte increments by 1. In this way, we can observe that the serial port sends a byte of data every 1S and the data is incrementing. baud_set is set to 3'd0, corresponding to a baud rate of 9600. The code for the serial transmit test module is as follows:

```
module uart_tx_test(

    input clk,

    input reset_n,

    output uart_tx,

    output led

);

    parameter MCNT = 49_999_999; //Định thời gian 1 giây

    reg [7:0] data_byte;

    reg send_en;

    // Thiết kế định bộ thời gửi dữ liệu đều đặn

    reg [25:0] cnt; // định nghĩa thanh ghi bộ đếm

    wire tx_done;

    // tiến trình bộ đếm
```

```

always@(posedge clk or negedge reset_n)

if(!reset_n)

    cnt <= 25'd0;

else if(cnt == MCNT)

    cnt <= 25'd0;

else

    cnt <= cnt + 1'b1;

always@(posedge clk or negedge reset_n)

if(!reset_n)

    send_en <= 1'b0;

else if(cnt == MCNT)

    send_en <= 1'b1;

else

    send_en <= 1'b0;

always@(posedge clk or negedge reset_n)

if(!reset_n)

    data_byte <= 8'b0;

else if(tx_done)

    data_byte <= data_byte + 1'b1;

else

    data_byte <= data_byte;

uart_byte_tx uart_byte_tx(

    .clk(clk),

    .reset_n(reset_n),

    .data_byte(data_byte),

    .send_en(send_en),

```

```

.baud_set(3'd0),

.uart_tx(uart_tx),

.tx_done(tx_done),

.led_state(led)

);

```

Endmodule

6.5 Board-Level Verification

The board-level verification part of this experiment mainly verifies the following goals:

1. Whether the generated fs file can be correctly downloaded to the development board.
2. Open the serial port debugging assistant on the computer, and you can see that the received data increments from 00 every 1S.

System Hardware Required:

1. ACG525 Development Board.
2. Gowin downloader and download cable.
3. 12V power supply.
4. One Type-C cable.
5. A PC with complete development functions that meets the experiment requirements.

6.5.1 Adding I/O Constraints

Click FloorPlanner on the Process interface on the right side of the Gowin software to create a new .cst file to store I/O constraints, as shown in Figure 6-12 below.

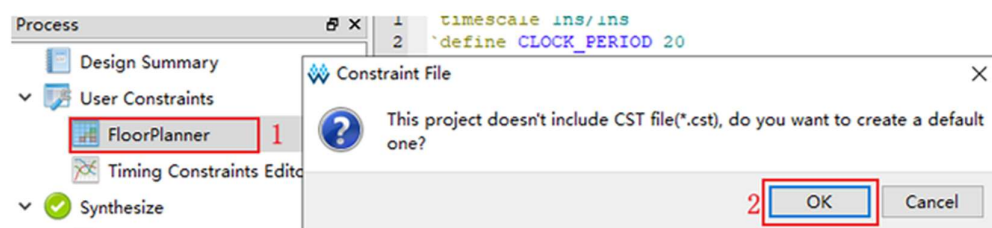


Figure 6-12 Creating New I/O Constraints

After entering the FloorPlanner interface, click I/O Constraints below to enter the I/O constraint interface. Assign pin positions according to the schematic. After addition is complete, it is shown in Figure 6-13 below.

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	clk	input		T9	4	False	LVC MOS33
2	led	output		V12	4	False	LVC MOS33
3	reset_n	input		B16	1	False	LVC MOS33
4	uart_tx	output		V8	5	False	LVC MOS33

Figure 6-13 Pin Assignment Interface

Thus, pin constraints are added. But at this time, the constraint content is saved in memory and has not been written to the file. Click the Save button on the toolbar, or directly press Ctrl+S.

6.5.2 Place and Route

Click Place & Route in Process to perform place and route. After successful place and route, the fs file will be generated. The Console prompt is as shown in Figure 6-14.

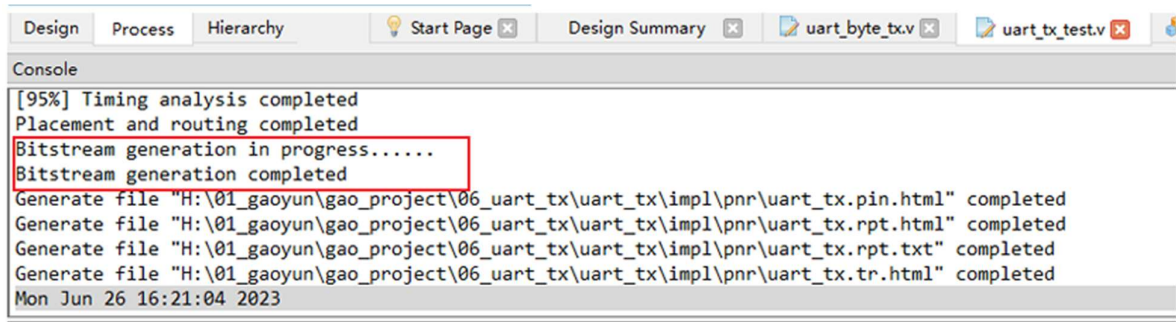


Figure 6-14 Schematic of fs File Generation Success

6.5.3 Connect with the Hardware

Connect the downloader, power supply, and serial cable to the development board in sequence. The overall hardware connection is shown in Figure 5-26 below. You can choose to power via power supply or USB, users can choose by themselves.

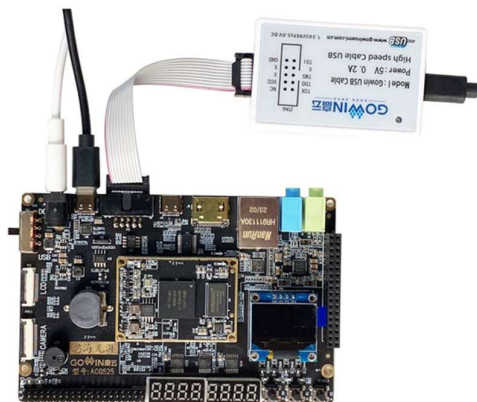


Figure 6-15 Hardware Connection Diagram

6.5.4 Configuring Serial Port Debugging Assistant

Open a serial port software on the PC. Here we use Serial Hunter. Set the serial port number corresponding to the development board on the serial port software. You can open the Device Manager to view the port number (the development board needs to be powered on), as shown in Figure 6-16 below.

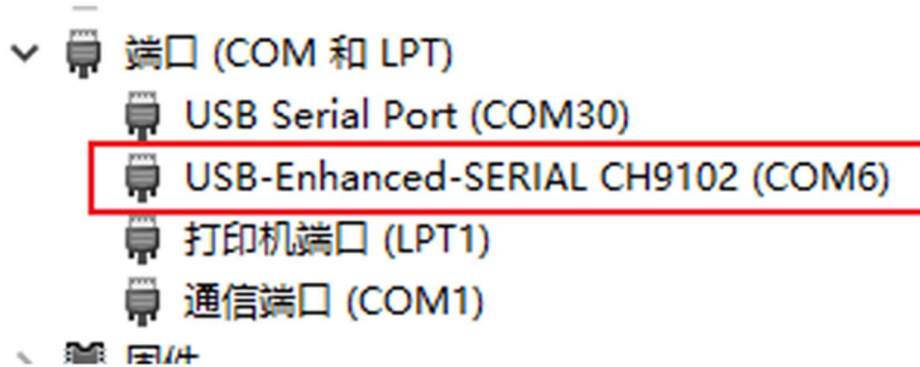


Figure 6-16 Corresponding Port Number in Device Manager

Then set the baud rate to 9600, data bits to 8bit, stop bit to 1bit, click Start Serial Port. You can see the specific connection status in the print information window below the serial port software, as shown in Figure 6-17 below.

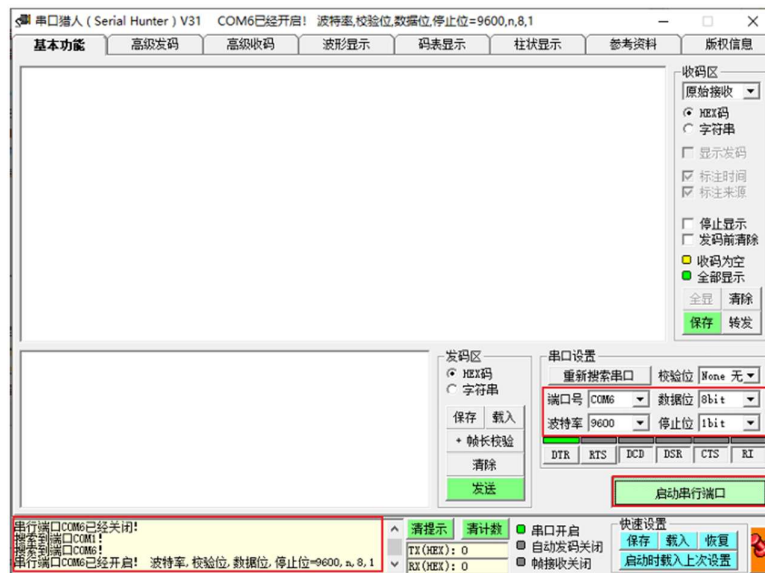


Figure 6-17 Serial Port Receive Settings of Serial Hunter Tool

6.5.5 Downloading fs File

Click Program Device to enter the download interface. Then the Cable Setting interface pops up. Click Save, then click the icon to download the fs file. After the download is complete, it is shown in Figure 6-18 below.

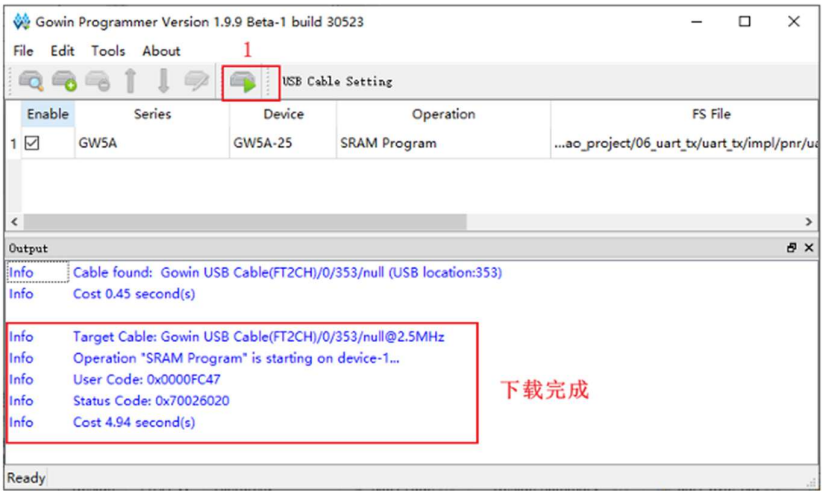


Figure 6-18 Schematic of Successful Download

After Program is complete, run it for a while. You can see the serial assistant receiving data incrementing from 00, as shown in Figure 6-19 below. It can be seen that it meets the design expectations, and the received and sent data are consistent.



Figure 6-19 Serial Hunter Receiving Data

6.6 Reflection and Summary

In this chapter, we learned the classification and principles of UART, introduced the data format of the UART protocol and the meanings of related parameters, and designed the hardware circuit of the serial port. During board-level debugging, by writing test files to control the data and control signals sent by the serial port, the function of FPGA sending data to display on the serial host computer was realized.

Chương 7 Serial Port Receive Module Design and Verification

Project Source Code, |----02_Design Examples, |----ch7_uart_rx

Chapter Overview

This chapter will learn the design of the UART data reception part. Addressing the potential impact of strong electromagnetic interference on data in actual use, this section proposes an improved serial port receive module design method.

7.1 Serial Port Receive Principle Analysis

The previous section covered the design and implementation of the serial port transmit module. The timing diagram for the UART transmitter sending one byte of data is shown in Figure 7-1 below.

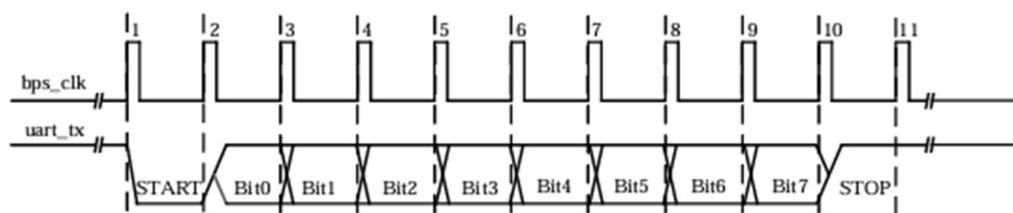


Figure 7-1 Timing Diagram of Sending One Byte of Data

This lecture introduces the design and implementation of the serial port receive module. When sampling each bit on the data line, generally, the middle point of each data bit is considered the most stable. Therefore, in general applications, the level collected at the middle moment is considered the level of this data bit, as shown in Figure 7-2 below.

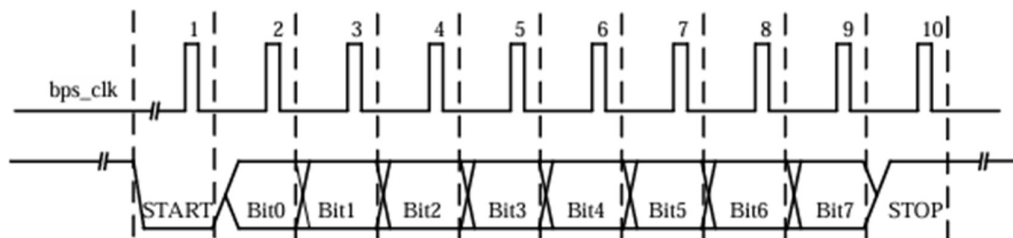


Figure 7-2 Serial Port Receive Timing Diagram

However, in practical industrial applications, there is often very strong electromagnetic interference on site. It is unreliable to sample only once and take it as the level state of the data. It is very likely to coincidentally sample the interfered signal, leading to erroneous results. Therefore, the following improved single-bit data receiving method is proposed here, using a multiple sampling method to calculate probability for state determination, as shown in Figure 7-3 below.

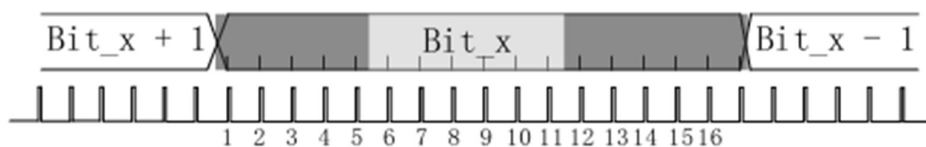


Figure 7-3 Schematic of Improved Serial Port Receiving Method

In the figure above, each data bit is further divided into 16 small segments. For the Bit_x data bit, considering that the data is highly likely to be unstable during the period just after a change and just before a change (the two segments marked in dark gray), sampling data in these two time periods is very likely to yield incorrect results. Therefore, the levels in these two time periods are judged as invalid and directly ignored during sampling. The middle time period (marked in light gray), the data itself is relatively stable and generally represents the correct result. This is the middle measurement method mentioned earlier, but it does not rule out the possibility of erroneous level pulses in this segment due to strong electromagnetic interference. Therefore, multiple samples are taken for this segment of level, and the probability of high and low levels is calculated. Among the 6 sampling results, the level that appears more times is taken as the sampling result. For example, if the 6 sampling results are 1/1/1/1/0/1, the level result is taken as 1. If it is 0/0/1/0/0/0, the level result is taken as 0. When 1s and 0s each account for half of the 6 sampling results (3 times each), it can be judged that the current communication line environment is very harsh, the data is unreliable, and no processing is performed.

7.2 UART Asynchronous Serial Communication Receive Module Design and Implementation

7.2.1 Serial Port Receive Module Interface Design

Based on the above principles, the overall block diagram of the serial port receive module is shown in Figure 7-4 below, and its interface list is shown in the table below.



Figure 7-4 Serial Port Receive Module Overall Block Diagram

Table 7-1 Module Interface List

Signal Name	I/O	Function Description
clk	I	Module system clock 50 MHz
reset_n	I	Module asynchronous reset signal
uart_rx	I	Serial data input
baud_set	I	Baud rate selection signal

Signal Name	I/O	Function Description
data_byte	O	Parallel data output
rx_done	O	Reception completion signal

7.2.2 Single Bit Asynchronous Signal Synchronization Design

Here, the serial port receive signal `uart_rx` is an asynchronous signal relative to the FPGA internal signals. If used directly as input without processing, it is easy to cause timing violations leading to metastability. Therefore, it is necessary to first synchronize the signal to the FPGA clock domain before it can be used by subsequent modules. A common synchronization method is to use two-stage flip-flops, that is, using flip-flops to register the signal for two beats to synchronize with the system clock. The reference circuit is as shown in Figure 7-5 below. Where `uart_rx` is the asynchronous serial input signal, and `uart_rx_sync2` is the synchronized signal.

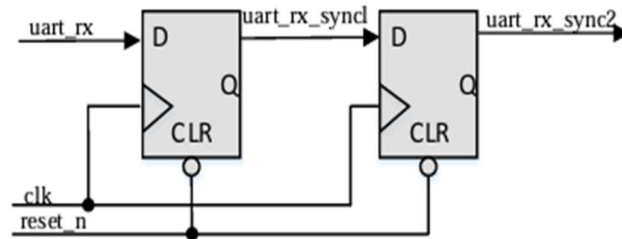


Figure 7-5 Single Bit Signal Synchronization Schematic

Describing the above circuit principle in Verilog language:

```

always@(posedge clk or posedge reset_n)
if(reset_n)begin
uart_rx_sync1 <= 1'b1;
uart_rx_sync2 <= 1'b1;
end
else begin
uart_rx_sync1 <= uart_rx;
uart_rx_sync2 <= uart_rx_sync1;

end

```

In some high-speed designs, there are also designs using three-stage flip-flops for single-bit signal synchronization. This method is only to improve the Mean Time Between Failure (MTBF). Its circuit connects another flip-flop after the two-stage flip-flops.

7.2.3 Edge Detection Design

From the serial port reception timing diagram, it is known that the start signal for data reception is the serial data changing from an idle high level to a low level, that is, a falling edge where the level changes from high to low. This requires detecting the falling edge. The principle is to use the characteristic that the input state of a register is the output state of the next moment under the control of a clock signal for comparison and judgment, as shown in Figure 7-6 below.

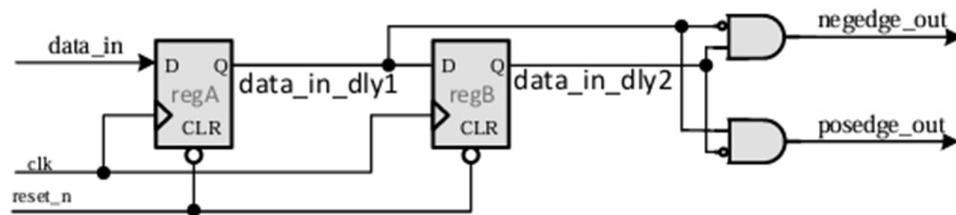


Figure 7-6 Edge Detection Principle

For the detection process, assume data_in changes from 0 to 1, i.e., a rising edge occurs. The specific timing diagram is shown in Figure 7-7 below.

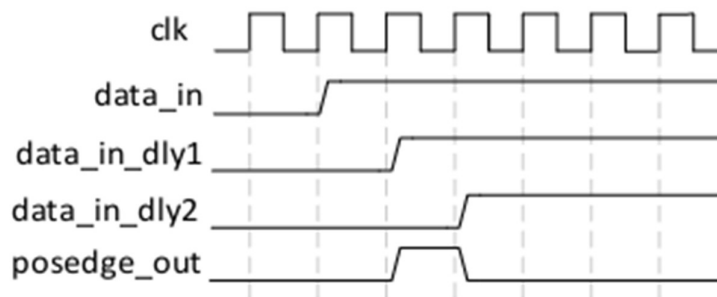


Figure 7-7 Rising Edge Detection Principle Waveform

The output signal of the first register regA is data_in_dly1, which is delayed by one clock cycle relative to data_in;

The output signal of the second register regB is data_in_dly2, which is delayed by one clock cycle relative to data_in_dly1; Performing relevant combinational logic operations on the outputs of the two registers (data_in_dly1 & !data_in_dly2) can detect a rising edge;

When data_in changes from 1 to 0, i.e., a falling edge occurs, the specific timing diagram is shown in Figure 7-8 below.

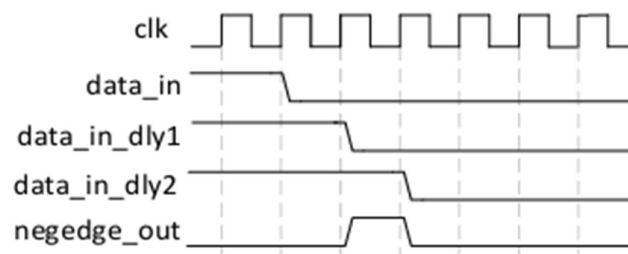


Figure 7-8 Falling Edge Detection Waveform

Similarly, performing relevant combinational logic operations on the outputs of the two registers (!data_in_dly1 & data_in_dly2) can detect a falling edge. Through the above method, when data_in has a rising edge, the signal pedge_out will generate a high level for one clock cycle. When there is a falling edge, the signal negedge_out will generate a high level for one clock cycle. When there is no rising edge or falling edge change, pedge_out and negedge_out remain in a low-level state. "!" used above is logical NOT operation.

Logical NOT operation on 4'b0110 results in 4'b0000; while "~" is bitwise NOT, bitwise NOT on 4'b0110

results in 4'b1001. For single bit data, the results after using "!" and "~" are the same.

The serial port receive module needs to detect the falling edge to identify the start bit of the serial port receive signal. Detect the falling edge of the synchronized serial input data. The specific code is as follows.

```
always@(posedge clk or posedge reset)
if(reset_n)begin
    uart_rx_reg1 <= 1'b0;
    uart_rx_reg2 <= 1'b0;
end
else begin
    uart_rx_reg1 <= uart_rx_sync2;
    uart_rx_reg2 <= uart_rx_reg1;
end
assign
    uart_rx_nedge = !uart_rx_reg1 & uart_rx_reg2;
    uart_rx_posedge=uart_rx1&uart_rx_reg2;
```

7.2.4 Sampling Clock Generation Module Design

One of the main components of the serial port receive module is the baud rate clock generation module. Here, according to the oversampling method mentioned in the principle part of this chapter, the actual sampling frequency is 16 times the baud rate. The relationship between the count value and the baud rate is shown in Table 7-2 Sampling Clock Calculation, where the system clock period is System_clk_period, which is 20ns here.

Table 7-2 Sampling Clock Calculation

Baud rate	Baud period	Clock count calculation	System_clk_period = 20ns
9600	104167ns	104167/System_clk_period	325-1
19200	52083ns	52083/System_clk_period	163-1
38400	26041ns	26041/ System_clk_period	81-1
57600	17361ns	17361/ System_clk_period	54-1
115200	8680ns	8680/ System_clk_period	27-1

Here, a selector is still used to implement the correspondence between different baud rates and sampling clock division count values. The design code is as follows.

```

always@(posedge clk or posedge reset_n)
if(reset_n)
    bps_DR <= 16'd324;
else begin
    case(baud_set)
        0:bps_DR <= 16'd324;
        1:bps_DR <= 16'd162;
        2:bps_DR <= 16'd80;
        3:bps_DR <= 16'd53;
        4:bps_DR <= 16'd26;
        default:bps_DR <= 16'd324;
    endcase
end
end

```

Now generate the sampling clock, which is 16 times the baud rate clock.

```

//counter
always@(posedge clk or posedge reset)
if(reset)
    div_cnt <= 16'd0;
else if(uart_state)begin
    if(div_cnt == bps_DR)
        div_cnt <= 16'd0;
    else
        div_cnt <= div_cnt + 1'b1;
end
else
    div_cnt <= 16'd0;

// bps_clk gen
always@(posedge clk or posedge reset)
if(reset)
    bps_clk <= 1'b0;
else if(div_cnt == 16'd1)
    bps_clk <= 1'b1;
else
    bps_clk <= 1'b0;

```

Sampling clock counter. One of the counter clearing conditions `bps_cnt == 8'd159` represents that the reception of one byte is completed.

`(bps_cnt == 8'd12 && (START_BIT > 2))` is to implement start bit detection error checking, which will be explained in detail later.

```

//bps counter
always@(posedge clk or posedge reset)
if(reset)
    bps_cnt <= 8'd0;
else if(bps_cnt == 8'd159 | (bps_cnt == 8'd12 && (START_BIT > 2)))

```

```

    bps_cnt <= 8'd0;
else if(bps_clk)
    bps_cnt <= bps_cnt + 1'b1;
else
    bps_cnt <= bps_cnt;

always@(posedge clk or posedge reset)
if(reset)
    rx_done <= 1'b0;
else if(bps_cnt == 8'd159)
    rx_done <= 1'b1;
else

```

7.2.5 Sampled Data Receive Module Design

Taking the start bit in Figure 7-3 as an example, the bps_cnt values corresponding to the middle sampling time period are 6, 7, 8, 9, 10, 11 respectively. At these moments, the data of this bit is directly accumulated. Then, the first bps_cnt value of the sampling time period for the next data bit is the first bps_cnt value of the sampling time period for the previous data bit plus 16. Therefore, the bps_cnt values for the sampling time period of the first data bit Bit0 following the start bit are 22, 23, 24, 25, 26, 27. Similarly, at these moments, the data of this bit is directly accumulated. By analogy, the bps_cnt values corresponding to the sampling time periods of other bits can be obtained.

Now explain why one of the clearing conditions above is (bps_cnt == 8'd12 && (START_BIT > 2)). In the ideal case (true start bit), when the bps_cnt count value is 12, the calculated value of START_BIT should be 0. However, in reality, there may be an interference signal instead of a real start signal that also causes a falling edge. At this time, if no judgment is added and it is directly regarded as a start signal to enter the receiving state, it will inevitably lead to receiving incorrect data, and may also lead to missing the correct data reception. To increase the anti-interference ability, the following idea is adopted here: when a falling edge appears on the data line, first assume it is a start signal, and then sample its middle segment. If the accumulated result of these 6 samples is greater than 2, that is, at least half of the states in the 6 samples are high levels, then this obviously does not meet the requirement of the continuous low level of a true start signal. At this time, the falling edge that just arrived is regarded as an interference signal and not as a start signal.

```

always@(posedge clk or posedge reset_n)
if(reset_n)begin
    START_BIT <= 3'd0;
    data_byte_pre[0] <= 3'd0;
    data_byte_pre[1] <= 3'd0;
    data_byte_pre[2] <= 3'd0;
    data_byte_pre[3] <= 3'd0;
    data_byte_pre[4] <= 3'd0;
    data_byte_pre[5] <= 3'd0;
    data_byte_pre[6] <= 3'd0;
    data_byte_pre[7] <= 3'd0;

```

```

STOP_BIT <= 3'd0;
end
else if(bps_clk)begin
case(bps_cnt)
0:begin // Reset các thanh ghi khi bắt đầu một frame mới
START_BIT <= 3'd0;
data_byte_pre[0] <= 3'd0;
data_byte_pre[1] <= 3'd0;
data_byte_pre[2] <= 3'd0;
data_byte_pre[3] <= 3'd0;
data_byte_pre[4] <= 3'd0;
data_byte_pre[5] <= 3'd0;
data_byte_pre[6] <= 3'd0;
data_byte_pre[7] <= 3'd0;
STOP_BIT <= 3'd0;
end

```

//Mỗi bit UART được lấy mẫu **6 lần** (ví dụ: bit start được lấy mẫu ở các giá trị bps_cnt = 6,7,8,9,10,11). Giá trị uart_rx_sync2 (tín hiệu UART đã đồng bộ hóa) được cộng dồn vào thanh ghi tương ứng. Sau 6 lần lấy mẫu, giá trị của thanh ghi sẽ nằm trong khoảng 0-6. Nếu giá trị >= 4, bit đó được coi là '1'; ngược lại là '0'

```

6,7,8,9,10,11:START_BIT <= START_BIT + uart_rx_sync2;
22,23,24,25,26,27:data_byte_pre[0]<=data_byte_pre[0]+uart_rx_sync2;
38,39,40,41,42,43:data_byte_pre[1] <= data_byte_pre[1] + uart_rx_sync2;
54,55,56,57,58,59:data_byte_pre[2] <= data_byte_pre[2] + uart_rx_sync2;
70,71,72,73,74,75:data_byte_pre[3] <= data_byte_pre[3] + uart_rx_sync2;
86,87,88,89,90,91:data_byte_pre[4] <= data_byte_pre[4] + uart_rx_sync2;
102,103,104,105,106,107:data_byte_pre[5]<=data_byte_pre[5]+uart_rx_sync2;
118,119,120,121,122,123:data_byte_pre[6]<=data_byte_pre[6]+uart_rx_sync2;
134,135,136,137,138,139:data_byte_pre[7]<=data_byte_pre[7]+uart_rx_sync2;
150,151,152,153,154,155:STOP_BIT <= STOP_BIT + uart_rx_sync2;

```

- **default:** //Khi bps_cnt không rơi vào các khoảng lấy mẫu, các thanh ghi giữ nguyên giá trị

```

begin
START_BIT <= START_BIT;
data_byte_pre[0] <= data_byte_pre[0];
data_byte_pre[1] <= data_byte_pre[1];
data_byte_pre[2] <= data_byte_pre[2];
data_byte_pre[3] <= data_byte_pre[3];
data_byte_pre[4] <= data_byte_pre[4];
data_byte_pre[5] <= data_byte_pre[5];
data_byte_pre[6] <= data_byte_pre[6];
data_byte_pre[7] <= data_byte_pre[7];
STOP_BIT <= STOP_BIT;
end
endcase
end

```

7.2.6 Data State Determination Module

As introduced in the principle section, a single data bit needs to be sampled 6 times, and the data that appears more frequently is taken as the sampling result. That is, only data that appears more than 3 times in 6 samples can be used as valid data.

For this, a numerical comparator can be used in combination with the received data `data_byte_pre[n]` for judgment, or it can be directly set equal to the highest bit data of the current bit. Explained with the following example: When `data_byte_pre[n]` are binary 011B/010B/100B/101B respectively, the decimal format of these data are 3d/2d/4d/5d respectively. It can be found that 100B/101B are greater than or equal to 4d. When the highest bit is 1, i.e., the accumulated data value is greater than or equal to 4d, it indicates the true data value is 1; when the highest bit is 0, i.e., the accumulated data value is less than or equal to 3d, it indicates the true data value is 0. Therefore, only the highest bit needs to be judged.

```
always@(posedge clk or posedge reset_n)
if(reset_n)
    data_byte <= 8'd0; //reset về 0
else if(bps_cnt == 8'd159)begin //chốt dữ liệu khi kết thúc 1 frame
    data_byte[0] <= data_byte_pre[0][2]; //lấy bit MSB
    data_byte[1] <= data_byte_pre[1][2];
    data_byte[2] <= data_byte_pre[2][2];
    data_byte[3] <= data_byte_pre[3][2];
    data_byte[4] <= data_byte_pre[4][2];
    data_byte[5] <= data_byte_pre[5][2];
    data_byte[6] <= data_byte_pre[6][2];
    data_byte[7] <= data_byte_pre[7][2];
end
```

7.3 Simulation Testing

After completing the design, perform functional simulation on it. In the following testbench file, the stimulus input generating

data uses the output of the data sending module from the previous chapter. Therefore, the stimulus file only needs to modify port information in the previous chapter's stimulus file, instantiate this module, and connect the `uart_tx` output by the sending module to the `uart_rx` on the receiving module. The modified partial stimulus file is as follows:

```
wire uart_tx_rx;

uart_byte_tx uart_byte_tx(
    .clk(clk),
    .reset_n(reset_n),

    .data_byte(data_byte_tx),
    .send_en(send_en),
    .baud_set(baud_set),

    .uart_tx(uart_tx_rx),
    .tx_done(tx_done),
```

```

        .uart_state(uart_state)
    );

    uart_byte_rx uart_byte_rx(
        .clk(clk),
        .reset_n(reset_n),

        .baud_set(baud_set),
        .uart_rx(uart_tx_rx),

        .data_byte(data_byte_rx),
        .rx_done(rx_done)
    );

```

After designing the simulation testbench file, click to enter the Modelsim software, create a new project for functional simulation, and you can see the waveform file shown in Figure 7-9 below. The serial port transmit module `uart_tx` sent 0xaa and 0x55 respectively. The serial port receive module `uart_rx` received the serial data and converted it to parallel output. The output data are 0xaa and 0x55, which are completely consistent with the data sent by the serial port transmit module.

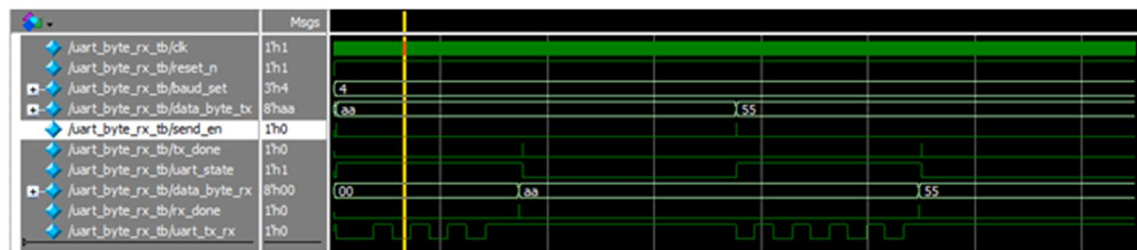


Figure 7-9 Single byte Data Reception Simulation Waveform Diagram

7.4 Serial Port Receive Test Module

Create a new `uart_rx_test.v` and set it as the top-level design. Instantiate `uart_byte_rx` and `uart_byte_tx` into this module. Every time the serial port receives a byte, the `rx_done` signal is generated. At this time, the `rx_done` signal is assigned to `send_en`, and the data received by the serial port receive module is sent out again through the serial port transmit module. The code for the serial port receive test module is as follows:¹²³

```

module uart_rx_test(
    input clk,
    input reset_n,
    input uart_rx,
    output uart_tx
);
    reg [7:0] data_byte_tx;
    wire [7:0] data_byte_rx;

```

```

reg send_en;
wire tx_done;
wire rx_done;
always@(posedge clk or negedge reset_n)
if(!reset_n) begin
    data_byte_tx <= 8'd0;
    send_en <= 1'd0;
end
else if(rx_done) begin
    data_byte_tx <= data_byte_rx;
    send_en <= rx_done;
end
else begin
    data_byte_tx <= data_byte_tx;
    send_en <= 1'd0;
end
end
uart_byte_tx uart_byte_tx(
    .clk(clk),
    .reset_n(reset_n),

    .data_byte(data_byte_tx),
    .send_en(send_en),
    .baud_set(3'd0),

    .uart_tx(uart_tx),
    .tx_done(tx_done),
    .uart_state( )
);
uart_byte_rx uart_byte_rx(
    .clk(clk),
    .reset_n(reset_n),

    .baud_set(3'd0),
    .uart_rx(uart_rx),

    .data_byte(data_byte_rx),
    .rx_done(rx_done)
);

```

Endmodule

7.5 Board-Level Verification

The board-level verification part of this experiment mainly verifies the following goals:

1. Whether the generated fs file can be correctly downloaded to the development board.
2. Open the serial port debugging assistant on the computer, send a byte of data, and observe whether the serial port assistant can receive the sent data back.

System Hardware Required:

1. ACG525 Development Board.
2. Gowin downloader and download cable.
3. 12V power supply.
4. One Type-C cable.
5. A PC with complete development functions that meets the experiment requirements.

7.5.1 Adding I/O Constraints

Click FloorPlanner on the Process interface on the right side of the Gowin software to create a new .cst file to store I/O constraints, as shown in Figure 7-10 below.

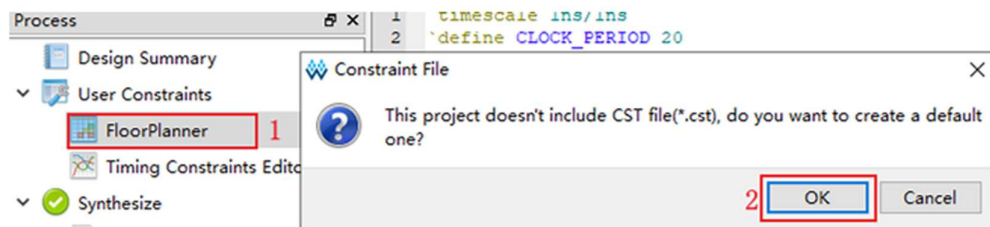


Figure 7-10 Creating New I/O Constraints

Then perform pin constraints, as shown in Figure 7-11 below.

Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
clk	input		T9	4	False	LVC MOS33
reset_n	input		N9	5	False	LVC MOS33
uart_rx	input		U8	5	False	LVC MOS33
uart_tx	output		V8	5	False	LVC MOS33

Figure 7-11 Pin Constraints

7.5.2 Place and Route

Click Place & Route in Process to perform place and route. After successful place and route, the fs file will be generated. The Console prompt is as shown in Figure 7-12.

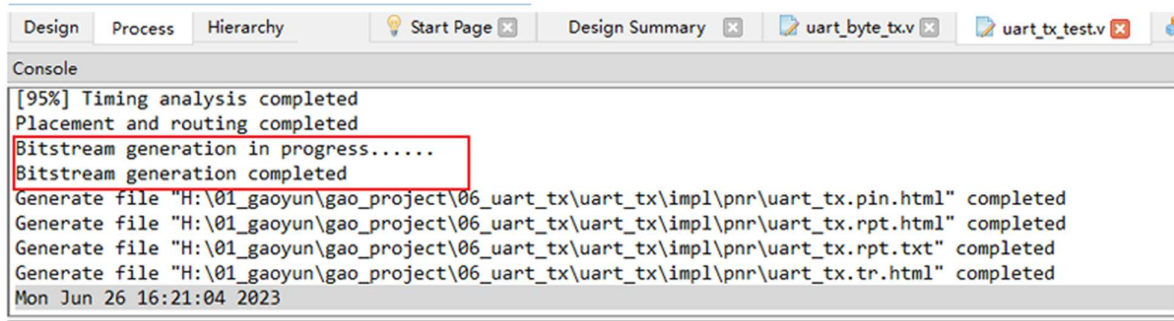


Figure 7-12 fs File Generation Success Schematic

7.5.3 Hardware Connection

Connect the downloader, power supply, and serial cable to the development board in sequence. The overall hardware connection is shown in Figure 7-13 below.

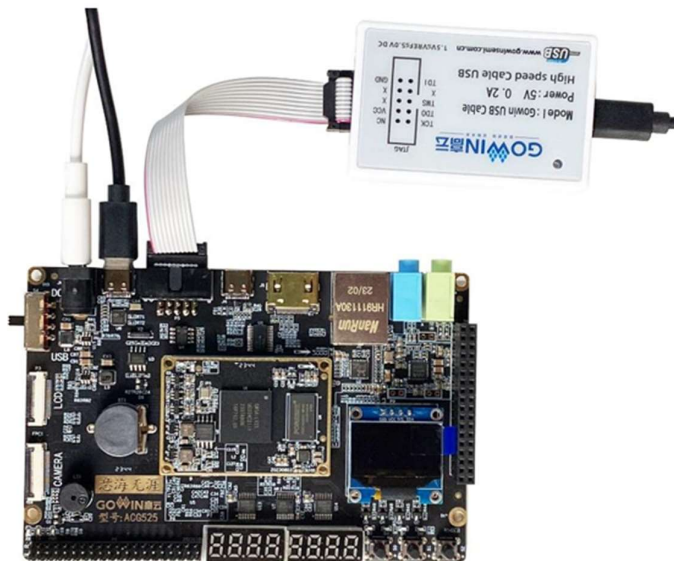


Figure 7-13 Hardware Connection Diagram

7.5.4 Configuring Serial Port Debugging Assistant

Open a serial port software on the PC. Here we use Serial Hunter. Set the serial port number corresponding to the development board on the serial port software. Baud rate 9600, data bits 8bit, stop bit 1bit. Then click Start Serial Port. You can see the specific connection status in the print information window below the serial port software, as shown in Figure 7-14 below.

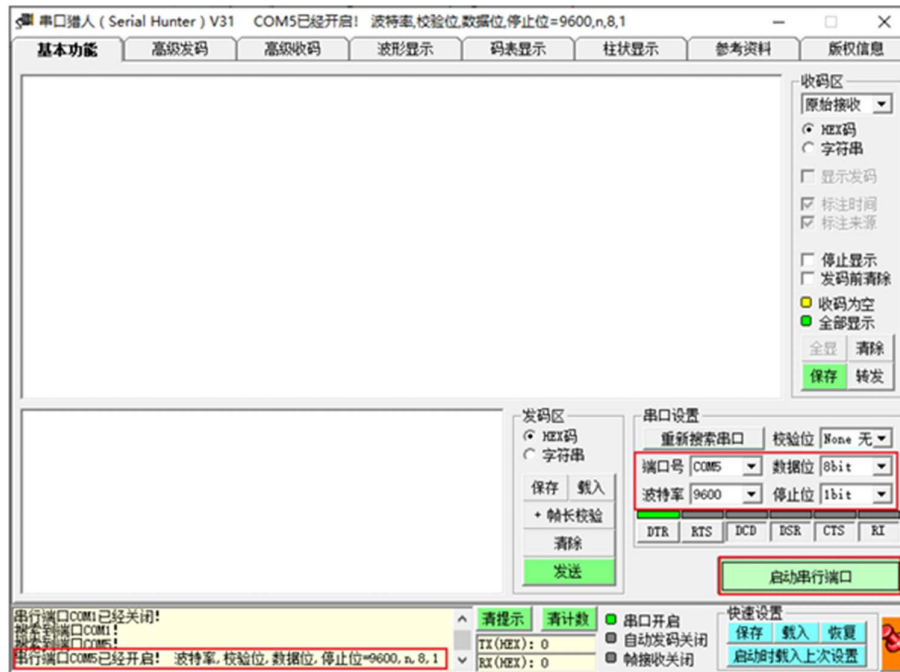


Figure 7-14 Serial Port Receive Settings of Serial Hunter Tool

7.5.5 Downloading fs File

Click Program Device to enter the download interface. Cable Setting interface pops up. Click Save, then click the icon to download the file. After the download is complete, shown in Figure 7-15

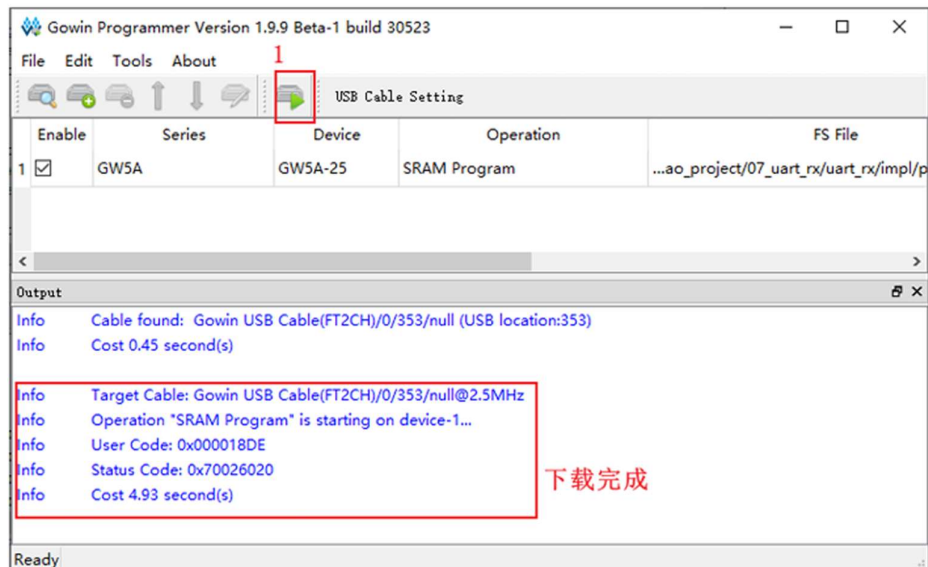


Figure 7-15 Schematic of Successful Download

After Program, enter AA in the code sending area, then click send. We can receive the AA we sent in the code receiving area, as shown below. From this, we can see that the serial port single-byte receive module is working properly.



Figure 7-16 Serial Port Sending and Receiving Data Consistent

7.6 Reflection and Summary

This chapter learned the relevant principles of serial port reception. In the design process, aiming at problems such as strong electromagnetic interference at the industrial site, an improved data receiving method based on weights was proposed. And in the board-level test, combined with the serial port transmit module, the serial port loopback function test was implemented to verify that the serial port receive module works properly. It should be noted that the test of this chapter's experiment only supports single-byte serial port loopback experiments. If multiple bytes of data are sent at once, it will cause the serial port transmit module to have not finished sending, while the serial port receive module has already received new data, resulting in inconsistency between the data finally received by the serial port assistant and the data sent, causing data loss.

Chapter 8: Serial Port Controlling LED Light

Project Source Code, |----02_Design Examples, |----ch8_uart_rx_ctrl_led

Chapter Overview

In FPGA design, we often use serial ports to complete some data transmission and reception. Depending on the type of data sent, the serial port can be used for image transmission, control, read/write access, and other scenarios. This section will lead everyone to implement the design of serial port controlling LED lights based on the serial port receive module designed in previous chapters.

The design allows users to specify the LED's on/off pattern and the time value for each state change via the serial port. The on/off pattern consists of 8 on/off states. The LED will continuously repeat the on/off pattern with these 8 on/off states as a cycle, thereby achieving the purpose of serial port controlling LED on/off.

8.1 Design Analysis

From the design purpose, we can know that the design needs to control the LED on/off and state change time via the serial port.

These two are different functions corresponding to two different control words. For convenience, we call them the status control word and the period control word respectively.

In the design, the LED continuously cycles through 8 states, so we can set the status control word to 8 bits, each bit corresponding to one of the 8 LED states.

The period control word controls the LED change time. If the LED change time is too small and the change speed is too fast, the human eye cannot observe the change process. Therefore, we can use a slightly larger, 32-bit data to represent the period control word.

Therefore, in this design, to control the LED to implement the required function, the serial port needs to send a 5-byte control word. For example, if we want to control the LED to change every 5ms according to the rule of on-off-on-off-on-off-on-off, as shown in Figure 8-1:

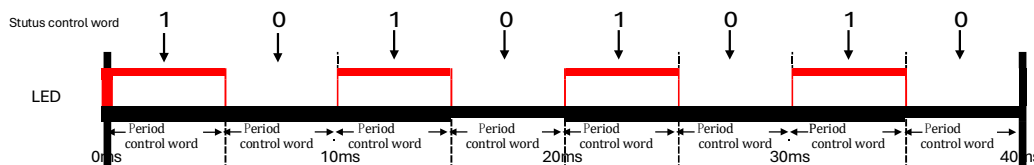


Figure 8-1 LED On/Off Control (5ms)

Then we only need to set the value of the status control word to 0xAA (8'b10101010), and then generate a 5ms period flag signal through the period control word. Every time a flag signal is generated, take one bit of the status control word, and control the specified IO to output high/low level according to the value of the corresponding bit to drive the LED on/off. Repeat this to implement the LED on-off-on-off-on-off-on-off cycle changing every 5ms.

Similarly, if we want to control the LED to change every 1s according to the rule of on-on-on-on-off-on-off-on, as shown in Figure 8-2:

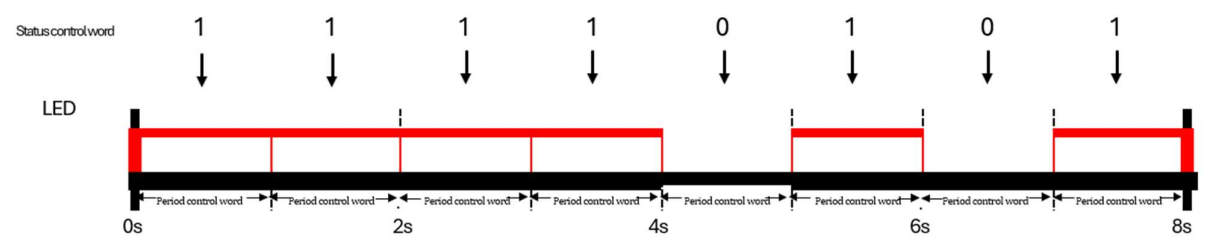


Figure 8-2 LED On/Off Control (1S)

Then we just need to set the value of the status control word to 0xF5 and generate a 1s period flag signal through the period control word.

8.2 Defining Serial Port Transmission Protocol Frame

After determining that the size of the control word data is 5 bytes, to ensure that the received data is valid data, and also to facilitate locating the function of each byte in the received data, we can first define a serial port transmission protocol frame, as shown in Figure 8-3:

55	A5	time_set [31:24]	time_set [23:16]	time_set [15:8]	time_set [7:0]	ctrl [7:0]	F0
----	----	---------------------	---------------------	--------------------	-------------------	---------------	----

Figure 8-3 Custom Serial Port Transmission Protocol Frame

This protocol frame is customized by us, consisting of frame header (0x55 0xA5) + 5 bytes of data + frame tail (0xF0). The user sends data on the host according to this transmission protocol frame, and the FPGA judges and unpacks the data based on this protocol after receiving it. Only when the continuous multi-byte data received by the serial port meets this protocol is it considered valid data. Subsequently, time_set and ctrl are used as the period control word and status control word respectively to control the LED.

Through this method, we can effectively avoid the situation where control words are still effective as valid commands when extra transmission, missed transmission, or even erroneous transmission occurs due to accidents during transmission.

8.3 Principle and Idea of Serial Port Controlling LED

After determining the serial port protocol frame for design transmission, the design work can officially begin.

For the design, to control the LED on/off pattern and state change time value via the serial port, first, there must be the ability to receive serial data. In the previous course, we have detailed the design and implementation of the serial port reception logic, and we can directly use that design in this design.

The serial port receive module can only transmit single-byte data at a time. According to our defined serial port protocol frame, a valid command contains at least 8 bytes. Therefore, a serial

port command conversion module is needed to buffer the single-byte data output by the serial port receive module into 8 bytes, judge whether the current command is correct based on the buffered data, and extract and output the status control word and period control word in the correct command.

The output control words are ultimately used to control the LED on/off state and period. To achieve this function, we can define a counter and a flag signal with a bit width of 3. Use the period control word as the maximum value of the counter. When the counter reaches the maximum count value, let the flag signal increment by 1, and the counter restarts counting. In this way, we can control the change frequency of the flag signal through the period control word. Since the bit width of the flag signal is 3, as the count value accumulates, the flag signal will cyclically change between 0 and 7 at a fixed frequency. We only need to drive the specified IO to output high/low level according to the value of the corresponding bit of the status control word when the flag signal is at different values, to realize the LED changing according to the specified state at a fixed frequency.

Based on the above analysis, to implement serial port control of LED on/off and its frequency, we can divide it into three parts. Namely:

1. Receive serial data from the computer and convert it into multiple single-byte data outputs.
2. Buffer single-byte data into 8 bytes, judge whether the data is a valid frame, and extract the correct control word for output.
3. Control the LED to cycle through 8 states at a certain frequency according to the control word via the counter and status signal.

Therefore, the system block diagram of this design is shown in Figure 8-4 below.

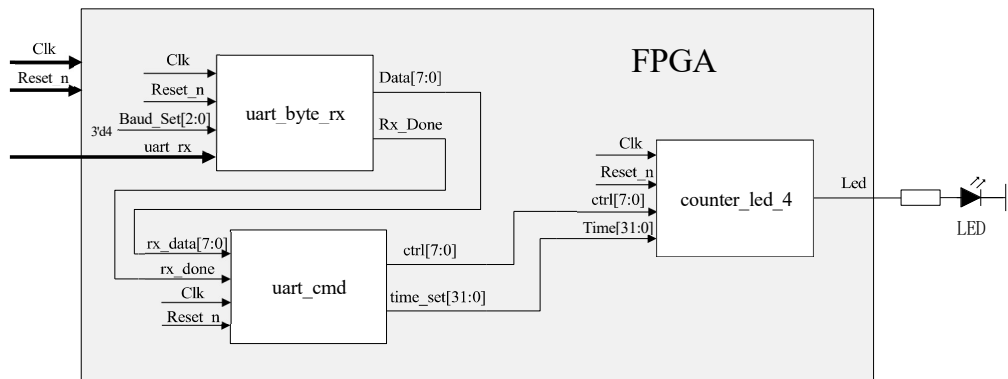


Figure 8-4 Design Structure Block Diagram

Among them, uart_byte_rx is the single-byte serial port receive module designed in the previous chapter; uart_cmd is the serial port command conversion module. This module buffers single-byte data to obtain continuous 8-byte data content. Then, every time a new byte of data is received, the values of these continuous 8-byte contents are checked for frame structure. If the data content matches a command frame structure, the period control word (time_set[31:0]) and status control word (ctrl[7:0]) are extracted; counter_led_4 is the counter driver module, which drives the LED light on and off at a certain frequency and pattern according to the input control words.

Based on this design structure, the module design can begin.

8.3.1 Serial Port Command Conversion Module Design (uart_cmd)

The serial port receive module transmits one byte of data at a time. According to our custom serial port transmission protocol frame, determining whether the data is for controlling the LED requires at least 8 bytes. Therefore, we need to buffer the data. The code is as follows:

```
input Clk;
input [7:0]rx_data;

reg [7:0] data_str [7:0];
always@(posedge Clk) if(rx_done)begin
    data_str[7] <= #1 rx_data; data_str[6] <= #1
    data_str[7]; data_str[5] <= #1 data_str[6];
    data_str[4] <= #1 data_str[5]; data_str[3] <= #1
    data_str[4]; data_str[2] <= #1 data_str[3];
    data_str[1] <= #1 data_str[2]; data_str[0] <= #1
    data_str[1];
end
```

Here we use shift buffering. The biggest advantage of this method is that the data buffered in this way is continuously changing and consistent with the length of the serial port transmission protocol frame. In this way, we only need to perform a frame structure detection on the 8 bytes of data in data_str once when new byte data is received, without checking all received data.

Once the data in data_str meets our defined serial port transmission protocol frame, the corresponding control word is extracted and output. The code is as follows:

```
input Reset_n; output
reg[7:0]ctrl;
output reg[31:0]time_set;
reg r_rx_done;
always@(posedge Clk)
    r_rx_done <= rx_done;
always@(posedge Clk or negedge Reset_n) if(!Reset_n) begin
    ctrl <= #1 0;
    time_set <= #1 0;
end else if(r_rx_done)begin
    if((data_str[0]==8'h55)&&(data_str[1]==8'hA5)&&(data_str[7]==8'hF0))begin
        time_set[31:24] <= #1 data_str[2];
        time_set[23:16] <= #1 data_str[3]; time_set[15:8]
        <= #1 data_str[4]; time_set[7:0] <= #1
        data_str[5]; ctrl <= #1 data_str[6];
    end
end
```

Here, because there is a one clock cycle delay between the highest bit of buffered data and the input data, it is necessary to register the rx_done signal for one beat before using it as a condition to judge the data in the buffer.

When the r_rx_done signal is valid, it represents that the module has received a new byte of data. At this time, frame structure judgment is performed on the data buffered in data_str. When the data conforms to the frame structure, it is considered a valid command. The period control word

and status control word are extracted and combined from the corresponding bits, and then output to the counter driver module.

By buffering received data into 8 bytes and judging frame structure, the design has strong fault tolerance for commands issued by the user. Even if the user sends extra, missed, or wrong partial data during command issuance, the design can analyze and judge the received data according to the frame structure and find the correct command from it.

8.3.2 Counter Driver Module Design (counter_led_4)

The counter driver module needs to count according to the period control word and generate a flag signal, and control the specified IO to generate high/low level according to the status control word to drive the LED on/off.

The generation of the flag signal is very simple. Just define a counter. When the value of the counter reaches the period control word, let the flag signal increment once. The code is as follows:

```
input Clk; input Reset_n;
input [31:0]Time; reg
[31:0]counter;

always@(posedge Clk or negedge Reset_n) if(!Reset_n)
    counter <= #1 0;
else if(counter >= Time - 1) counter <= #1 0;
else
    counter <= #1 counter + 1'b1;

reg [2:0]counter2;
always@(posedge Clk or negedge Reset_n) if(!Reset_n)
    counter2 <= #1 0;
else if(counter >= Time - 1) counter2 <= #1
    counter2 + 1'b1;
```

Here, counter2 as the flag signal has a width of 3, which can represent 0~7. When it exceeds 7, it will automatically overflow and clear to zero. The control word used to control the LED on/off state also happens to have 8 bits. Therefore, we can drive the LED on/off according to the value of counter2 through a case statement. The code is as follows:

```
always@(posedge Clk or negedge Reset_n) if(!Reset_n)
    Led <= #1 0;
else case(counter2) 0:Led <= #1
    Ctrl[0];
    1:Led <= #1 Ctrl[1];
    2:Led <= #1 Ctrl[2];
    3:Led <= #1 Ctrl[3];
    4:Led <= #1 Ctrl[4];
    5:Led <= #1 Ctrl[5];
    6:Led <= #1 Ctrl[6];
    7:Led <= #1 Ctrl[7];
```

```
default:Led <= #1 Led; endcase
```

Whenever the counter reaches the count value, the value of counter2 increments once, causing the LED drive to change. When the value of counter2 exceeds 7, it overflows and clears to zero, becoming 0 again. Thus, the LED can cycle continuously among the input 8 states at a certain frequency.

8.3.3 Creating Top-Level Encapsulation

After completing the design of each module, we also need a top-level module to connect these modules together to achieve the expected function. This design only requires top-level instantiation and port connection of each module. The complete code is as follows:

```
module uart_rx_ctrl_led( Clk,
    Reset_n, Led,
    uart_rx
);
    input Clk; input Reset_n;
    output Led; input uart_rx;

    wire [7:0]ctrl;
    wire [31:0]time_set; wire
    [7:0]rx_data; wire rx_done;

    parameter Baud_Set = 3'd4; counter_led_4

    counter_led(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .Ctrl(ctrl),
        .Time(time_set),
        .Led(Led)
    );

    uart_cmd uart_cmd(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .rx_data(rx_data),
        .rx_done(rx_done),
        .ctrl(ctrl),
        .time_set(time_set)
    );

    uart_byte_rx uart_byte_rx(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .Baud_Set(Baud_Set),
        .uart_rx(uart_rx),
        .Data(rx_data),
        .Rx_Done(rx_done)
```

```
);
```

```
Endmodule
```

At this point, we have completed the design of serial port controlling LED light on/off. To verify whether the design can achieve our expected function, we need to perform simulation verification to observe whether the waveform timing is correct.

8.3.4 Stimulus Creation and Simulation Testing

In the stimulus file, we need to simulate the serial port to provide input data for the design. To test whether the module can correctly identify the control word data, some interference data needs to be input in the simulation. The complete code of the stimulus file is as follows:

```
`timescale 1ns / 1ps
module uart_rx_ctrl_led_tb();

    reg Clk;
    reg Reset_n; wire Led;
    reg uart_rx;
    wire [31:0] delay_time;

    uart_rx_ctrl_led uart_rx_ctrl_led(
        .Clk(Clk),
        .Reset_n(Reset_n),
        .Led(Led),
        .uart_rx(uart_rx)
    );
    parameter Baud_Set = 3'd4;

    assign delay_time = (Baud_Set == 3'd0) ? 20'd104166:
                        (Baud_Set == 3'd1) ? 20'd52083: (Baud_Set ==
                        3'd2) ? 20'd26041: (Baud_Set == 3'd3) ?
                        20'd17361:
                        20'd8680;

    initial Clk = 1; always#10 Clk =
    ~Clk;
    initial begin Reset_n = 0;
        uart_rx = 1;
        #201;
        Reset_n = 1;
        #200;
        uart_tx_byte(8'h55); #(delay_time*10);
        uart_tx_byte(8'ha5); #(delay_time*10);
        uart_tx_byte(8'h55);
        #(delay_time*10);
        uart_tx_byte(8'ha5);
        #(delays_time*10);
        uart_tx_byte(8'h00);
```

```

        #(delay_time*10);
        uart_tx_byte(8'h00);
        #(delay_time*10);
        uart_tx_byte(8'hc3);
        #(delay_time*10);
        uart_tx_byte(8'h50);
        #(delay_time*10);
        uart_tx_byte(8'hAA);
        #(delay_time*10);
        uart_tx_byte(8'hf0);
        #(delay_time*10);
        uart_tx_byte(8'h55);
        #(delay_time*10);
        uart_tx_byte(8'ha5);
        #(delay_time*10);
        uart_tx_byte(8'h9a);
        #(delay_time*10);
        uart_tx_byte(8'h78);
        #(delay_time*10);
        uart_tx_byte(8'h56);
        #(delay_time*10);
        uart_tx_byte(8'h34);
        #(delay_time*10);
        uart_tx_byte(8'h12);
        #(delay_time*10);
        uart_tx_byte(8'hf1);
        #(delay_time*10); #15000000;
        $stop;
end
task uart_tx_byte; input
    [7:0]tx_data; begin
        uart_rx = 1;
        #20;
        uart_rx = 0;
        #delay_time;
        uart_rx = tx_data[0];
        #delay_time;
        uart_rx = tx_data[1];
        #delay_time;
        uart_rx = tx_data[2];
        #delay_time;
        uart_rx = tx_data[3];
        #delay_time;
        uart_rx = tx_data[4];
        #delay_time;
        uart_rx = tx_data[5];
        #delay_time;
        uart_rx = tx_data[6]; #delay_time;
    end
end

```

```

        uart_rx = tx_data[7]; #delay_time;
        uart_rx = 1; #delay_time;
    end endtask
endmodule

```

Considering that the interval between bits and between bytes is different when simulating serial output at different baud rates, the design uses assign to automatically define the delay value based on the baud_set value set by the user. The baud rate set in the stimulus is 115200, so the time for serial transmission of each bit is $1/115200 \times 10^9 = 8680\text{ns}$. To reduce code duplication, we use a task to implement the process of simulating serial output. By calling the task, the work of issuing instructions to the module under test by simulating the serial port can be completed.

It can be seen that in addition to sending valid commands, the stimulus file also sent some interference data. Among them, the valid command is 0x55_a5_00_00_c3_50_aa_f0, which means the counter switches the LED driving level every 50000 counts, which is 1ms. The LED changes in the order of on-off-on-off-on-off-on-off (10101010,

0xAA). After writing the stimulus file, simulation can begin to observe if the waveform timing is correct. The simulation waveform for this design is shown in Figure 8-5:

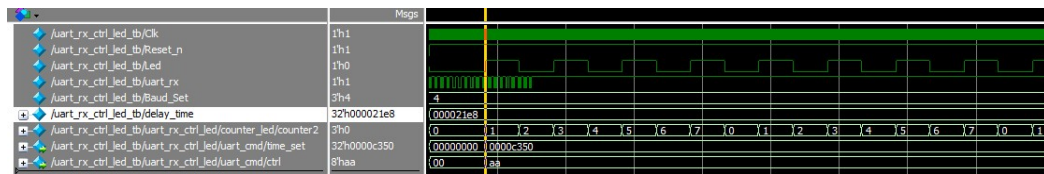


Figure 8-5 Simulation Waveform

For convenience of observation, the time_set signal and ctrl signal of the uart_cmd module and the counter2 signal of the counter_led_4 module are added here. It can be seen that after the data is received by the serial port, the uart_cmd module successfully identifies the correct command and extracts the correct data: time_set is 0xc350, ctrl is 0xaa. The level driving the LED changes according to the rule of high-low-high-low-high-low-high-low (on-off-on-off-on-off-on-off) when the counter2 signal changes.

After counter2 count exceeds 7, it starts to overflow and clear to zero, then starts to accumulate from 0, and the LED continues to flash according to the previous rule. It can be seen that the simulation waveform is consistent with our expectations, indicating that the design is normal and can fully realize the preset function. At this point, we have one last step left, which is to verify the actual effect of the design on the hardware through board-level verification.

8.4 Board-Level Verification

This section will assign pins to the design, and based on the fs file generated by the assigned project, perform programming verification on the ACG525 development board. Through the serial port debugging software on the computer side, issue commands to control the LED on/off status and flashing frequency. Verify the design by observing whether the final board-level verification result is consistent with expectations.

8.4.1 Pin Constraints

Although we have completed the top-level encapsulation of the design, to test on the ACG525 development board, we also need to synthesize the design, confirm it is correct, perform pin assignment, and constrain levels. The pin assignment for this design is shown in Table 8-1:

Table 8-1 Pin Constraint Table

Signal Name	Pin NO.	Signal Name	Pin NO.
CLK	T9	uart_rx	U8
Reset_n	B16	led	D14

After configuration is complete, constrain the corresponding pins to LVCMOS33. Next, generate the fs file. After the fs file generation is complete, connect the hardware to prepare for programming verification.

8.4.2 System Hardware Required

1. ACG525 Development Board x1
2. Gowin Downloader x1
3. Serial Cable x1
4. Power Cable x1

8.4.3 Hardware Connection

Connect the development board's download cable and power cable. Use the serial cable to connect one end to the UART interface on the development board and the other end to the USB port of the computer, as shown in Figure 8-6 below.

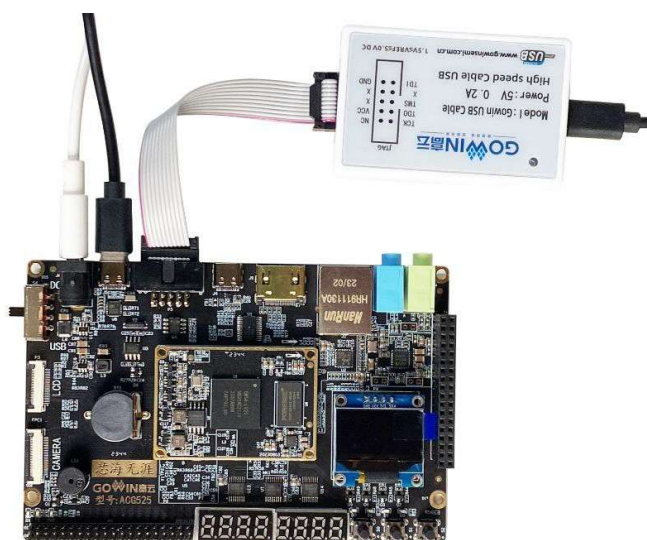


Figure 8-6 Hardware Connection

After connection is complete, ensure the development board power switch is toggled to the ON side. Next, program the generated fs file into the development board. After programming is

complete, open the Device Manager to query the serial port number. The name of the serial port in the Device Manager is shown in Figure 8-7:

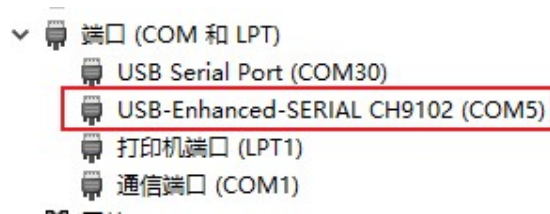


Figure 8-7 Query Port Number

Here, my computer's serial port number is COM5. Users should refer to the port number in their own computers when querying. Then open Serial Hunter, connect to the port just queried in the software, and set the baud rate to 115200. Then issue commands via Serial Hunter. The issuing method is shown in Figure 8-8:

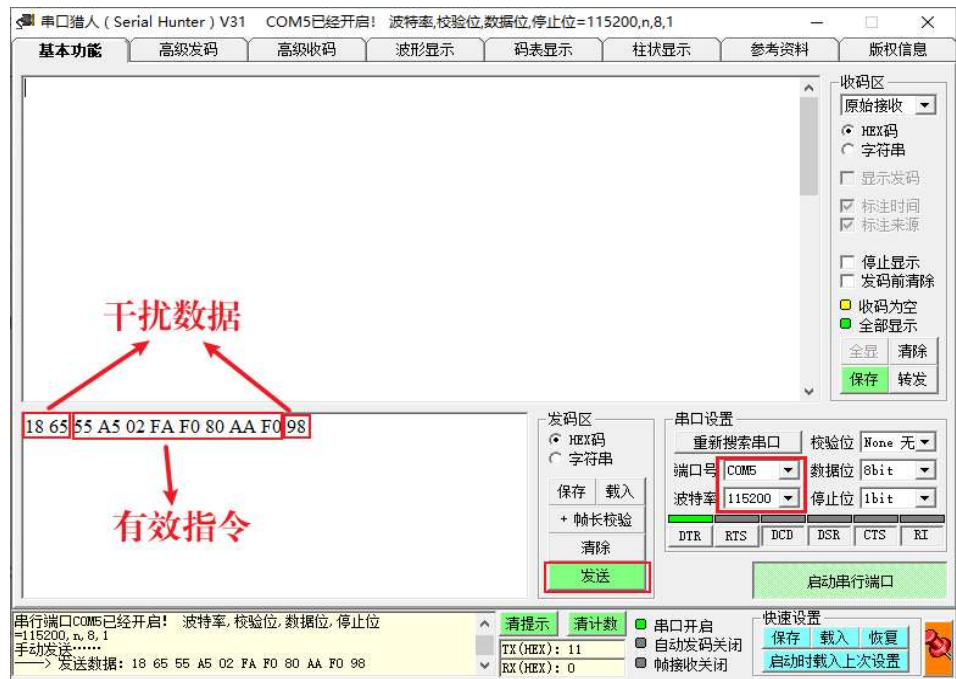


Figure 8-8 Serial Hunter Sending Codes

It can be seen that the valid command is 55 A5 02 FA F0 80 AA F0. Therefore, the period control word is 0x02FAF080 (50_000_000), and the status control word is 0xAA (10101010). That is, the LED changes every 1s according to the rule of on-off-on-off-on-off-on-off, cycling continuously every 8 changes. After sending the command and interference data together to the FPGA, LED0 starts to flash as expected.

Subsequently, I tried sending commands multiple times and intentionally sent extra, missed, or wrong parts. To prevent the phenomenon from being affected by the previous command, the program was reset before each command was sent. The relevant commands and board-level phenomena are as follows:

1. Send command 18 65 55 A5 02 FA F0 80 F0 F0 23 12, LED0 on the development board is on for 4s then off for 4s, cycling continuously.
2. Send command 55 A5 02 FA 80 A7 F0, LED on the development board has no reaction.
3. Send command 55 A5 02 FA F0 80 F0 A0, LED on the development board has no reaction.
In operation 2, the command was missed. Although the correct frame header 55 A5 was detected, because the corresponding bit of the frame tail was empty, the command was judged as an invalid command, and the LED would not be driven. Consistent with the corresponding board-level phenomenon.
In operation 3, the command was wrong. Although the correct frame header 55 A5 was detected, the value of the frame tail was inconsistent with the value we defined, so the command was judged as an invalid command, and the LED would not be driven. Consistent with the corresponding board-level phenomenon.
In the four command issuance tests, the LED was driven well according to our expectations. At this point, the board-level verification is successful, and the design can successfully implement the expected function on the ACG525 development board. The design is correct.

8.5 Summary

This chapter led everyone to complete the design of serial port controlling LED lights. The design uses the serial port to issue commands, ensures the correctness of commands through a custom serial port transmission protocol frame, and locates the corresponding control words. Finally, the control words are passed to the LED counter driver module to complete the control of the LED on/off state and frequency.

The function of the module designed in this chapter is not complex. The purpose is only to provide readers with an idea of using the serial port to implement control functions. The application scenarios of serial ports are very extensive. Users can broaden their thinking to consider and try more complex and interesting designs.

Chapter 9: Multi-Byte Serial Port Transceiver Module Design and Verification

Project Source Code, |----02_Design Examples, |----ch9_uart_loopback

Chapter Overview

In previous chapters, we guided everyone to complete the design of single-byte serial port transceiver modules, but the application scenarios for these two modules are relatively small. In most scenarios, serial ports are used for continuous multi-byte transmission and reception. Therefore, this chapter will be based on the existing single-byte serial port transceiver modules, leading everyone to complete the design of the multi-byte serial port transmit module and the multi-byte serial port receive module, and verify the design through simulation and board-level testing.

9.1 Multi-Byte Serial Port Transmission Principle and Design

To implement multi-byte serial port transmission, taking the commonly used 8N1 configuration as an example (i.e., eight data bits, no parity, one stop bit), many students have a misunderstanding, thinking that it is only necessary to fill the data between the start signal and the stop signal. For example, if I want to use the serial port to send the 16-bit data 0x1234, according to the above idea, I should simulate the serial port output 0_0010_1100_0100_1000_1, as shown in Figure 9-1:

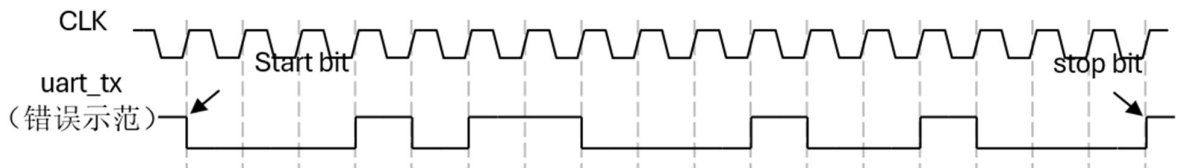


Figure 9-1 Serial Port Sending 0x1234 (Incorrect Example)

After being received by the destination serial port, these data will be parsed according to 8N1 to extract the data bits. One parsing requires receiving 10 bits of data. If parsed correctly, the middle 8 bits of valid data will be extracted. To parse and obtain the 16-bit 0x1234, it needs to be parsed twice, which means 20 bits of data are needed. However, when we simulated the serial port, we only sent 18 bits of data, so no matter what is finally parsed, it will not be the 0x1234 we expect to receive.

The correct transmission should be divided into two times. The first transmission is 0_0100_1000_1, and the second transmission is 0_0010_1100_1, as shown in Figure 9-2:

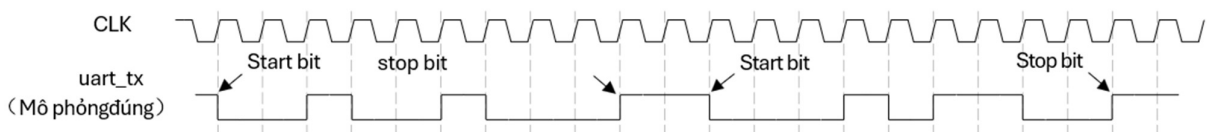


Figure 9-2 Serial Port Sending 0x1234

After receiving the data, the destination serial port will parse the data according to 8N1, obtaining 8'b0001_0010 (0x12) and 8'b0011_0100 (0x34) in two separate times. Splicing the two 8-bit data together yields 0x1234.

Therefore, multi-byte serial port transmission can be seen as a serial transmission process of consecutive single-byte data. In the previous chapters, we have led everyone to learn and complete the design of the single-byte serial port transmit module. In this chapter, we only need to split the data to be sent into multiple single bytes, and then call the single-byte serial port transmit module to send them one by one to achieve serial transmission of multi-byte data.

9.1.1 Multi-Byte Serial Port Transmission Principle

To ensure data continuity, we can use the single-byte transmission completion signal (byte_tx_done) of the single-byte serial port transmit module as a condition signal. When this signal is valid, it indicates that one byte of data transmission is complete, and at this time, the next byte of data needs to be output to the single-byte serial port transmit module.

For a multi-byte serial port transmission, the amount of data to be sent is determined; the serial port cannot send indefinitely. Therefore, after each byte of data is sent, we also need to judge whether the transmission is complete. Calculate the currently sent data based on the byte_tx_done signal, and then compare it with the received data bit width. If the transmission is not completed, continue sending the next byte. Once the transmission is judged to be complete, a serial port transmission completion signal is generated, and this transmission ends.

Based on this, we can design a state machine to generate single-byte data, judge whether a single-byte transmission is complete, and judge whether the entire transmission process is complete. Therefore, the structure of the serial port transmit module designed is shown in Figure 9-3:

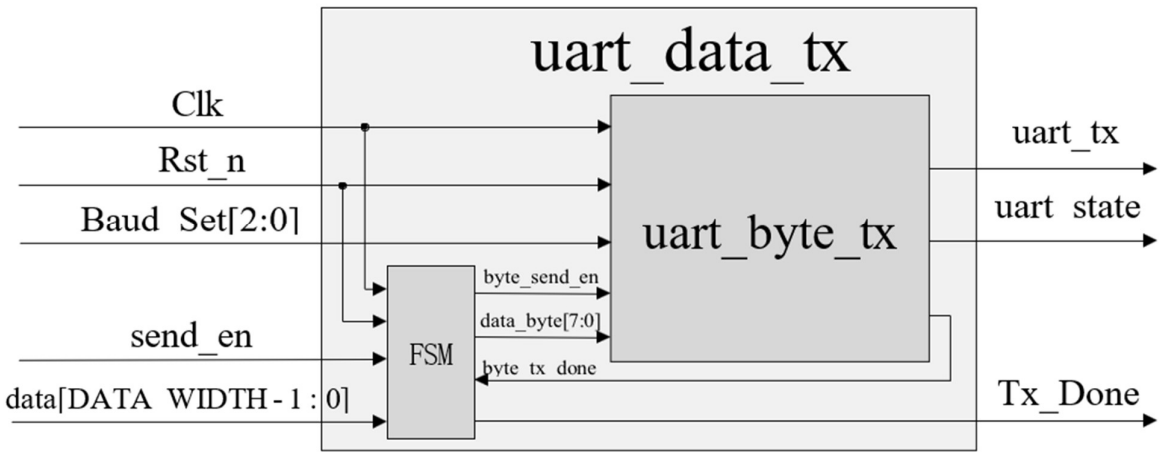


Figure 9-3 Multi-Byte Serial Port Transmit Module

9.1.2 Multi-Byte Serial Port Transmit Module Design

After determining the overall function and structure of the design, the next step is the code implementation of the design. First is the instantiation of the single-byte serial port transmit module. The code is as follows:

```
input Clk; input Rst_n;
input [2:0]Baud_Set; output
uart_tx; output uart_state;

reg [7:0] data_byte; reg
byte_send_en; wire byte_tx_done;

uart_byte_tx uart_byte_tx(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .data_byte(data_byte),
    .send_en(byte_send_en),
    .Baud_Set(Baud_Set),
    .uart_tx(uart_tx),
    .Tx_Done(byte_tx_done),
    .uart_state(uart_state)
);
```

To enable the design to adapt to more application scenarios, we can use parameter to define the input data bit width of the multi-byte serial port transmit module. parameter is used to define constants inside or outside the module. The biggest advantage of defining constants in this way is that parameters can be passed (overridden) when the module is instantiated. In this way, we can use parameter to set the input data bit width of the multi-byte serial port transmit module to 8. The code is as follows:

```
parameter DATA_WIDTH = 8;
```

In this way, by default, the module will only implement single-byte serial port transmission. If we want the module to send multiple bytes at once, for example, 8 bytes, we only need to define the input data bit width as 64 through parameter in the top layer, and the parameter can be passed to the sub-module to override the constant value.

For the multi-byte serial port transmit module, when the input send_en is valid, it indicates the start of transmission. The state machine generates one byte of transmission data and a transmission enable signal to the single-byte serial port transmit module. Subsequently, the state machine needs to wait for the byte_tx_done signal to confirm that the current byte data has been sent. At the same time, the state machine needs to count based on this signal to judge whether the transmission is complete. If the transmission is not complete, continue sending the next byte of data and the enable signal. If the transmission is complete, the state machine needs to generate a transmission completion signal tx_done, marking the end of this multi-byte transmission.

In summary, the state machine requires at least four states to implement the above functions. The four states are respectively used to wait for transmission enable, initiate single-byte data transmission, wait for single-byte data transmission completion, and check if all data transmission is complete. The corresponding four state codes are defined as follows:

```
localparam S0 = 0; // Wait for send request
```

```
localparam S1 = 1; // Initiate single-byte data transmission
```

```
localparam S2 = 2; // Wait for single-byte data transmission completion
```

```
localparam S3 = 3; // Check if all data transmission is complete
```

Next, implement the functions of the four states. First is state S0. This state needs to wait for the send request signal send_en. When send_en is valid, receive the incoming data and jump the state to S1. The corresponding code is as follows:

```
S0:
begin
    data_byte <= 0;
    cnt <= 0;
    Tx_Done <= 0; if(send_en)begin
        state <= S1;
        data_r <= data_r; //data_r chuỗi dữ liệu cần gửi
    end
    else begin
        state <= S0; data_r <=
        data_r;
    end
end
```

Next is state S1. This state needs to generate a single-byte transmission enable signal and single-byte data, and jump the state to S2. Considering that in different application scenarios, there will be different requirements for the byte order of data, we can use parameter to define a constant MSB_FIRST to control the big-endian or little-endian byte order of the data. The code is as follows:

```
parameter MSB_FIRST = 1; // 1: big-endian, 0 little-endian
```

S1 state code is as follows:

```
S1:// tạo tín hiệu cho phép gửi và gửi, data_byte dữ liệu hiện tại
begin
    byte_send_en <= 1;
    if(MSB_FIRST == 1)//chọn byte cao trước
    begin
        data_byte <= data_r[DATA_WIDTH-1:DATA_WIDTH - 8];
        data_r <= data_r << 8; //dịch trái data_r 8 bit
    end
    else begin
        data_byte <= data_r[7:0]; //gán 8 bit thấp cho data_byte data_r <=
        data_r >> 8; //dịch phải data_r 8 bit
    end
    state <= S2;
end
```

Subsequently is state S2. This state needs to wait for the single-byte data transmission to complete, that is, wait for the byte_tx_done signal of the single-byte serial port transmit module. When this

signal is valid, it represents the completion of single-byte data transmission. We can count the currently sent data volume based on this signal and jump the state to S3. The code is as follows:

```
S2:
    begin
        byte_send_en <= 0; // hạ xuống inactive sau khi active S1
        if(byte_tx_done)begin
            state <= S3;
            cnt <= cnt + 9'd8;
        end else
            state <= S2;
    end
```

Finally is state S3. We can generate the Tx_Done signal by comparing the amount of data sent and the input bit width amount. When the amount of data sent is greater than or equal to the input bit width, it means all data has been sent. At this time, the Tx_Done signal can be generated and the state jumps to S0, waiting for the next transmission to start. When the amount of data sent is less than the input bit width, it means the transmission has not ended yet. At this time, jump the state to S1 to prepare for the data transmission of the next byte. The code is as follows:

```
S3:
    if(cnt >= DATA_WIDTH)begin state <= S0;
        cnt <= 0;
        Tx_Done <= 1;
    end
    else begin
        state <= S1;
        Tx_Done <= 0;
    end
```

At this point, we have completed the design of the multi-byte serial port transmit module. As to whether the module can work properly, further verification is needed. Therefore, next we need to create a simulation stimulus file to perform simulation verification on the module.

9.1.3 Stimulus Creation and Simulation Testing

The creation of simulation stimulus is relatively simple. Just set the input bit width and the value of big/little endian when instantiating the multi-byte serial port transmit module, and generate input data and send enable signals for the design in the stimulus. The complete code of this simulation stimulus file is as follows:

```
module uart_data_tx_tb();

    parameter DATA_WIDTH = 32; parameter
    MSB_FIRST = 0;

    reg Clk; reg Rst_n;
    reg [DATA_WIDTH - 1 : 0]data; reg send_en;
    wire uart_tx; wire Tx_Done; wire uart_state;
```

```

uart_data_tx #(
    .DATA_WIDTH(DATA_WIDTH),
    .MSB_FIRST(MSB_FIRST)
)
uart_data_tx(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .data(data),
    .send_en(send_en),
    .Baud_Set(3'd4),
    .uart_tx(uart_tx),
    .Tx_Done(Tx_Done),
    .uart_state(uart_state)
);
initial Clk = 1;
always #10 Clk = !Clk;
initial begin
    Rst_n = 0;
    data = 0;
    send_en = 0;
    #201;
    Rst_n = 1;

    #2000;
    data = 32'h01234567;
    send_en = 1;
    #20;
    send_en = 0;
    #20;
    @(posedge Tx_Done); #1;
    data = 32'h12345678;
    send_en = 1;
    #20;
    send_en = 0;
    #20;
    @(posedge Tx_Done); #1;
    data = 32'h23456789;
    send_en = 1; #20;
    send_en = 0;
    #20;
    @(posedge Tx_Done); #1;
    #2000;
    $stop;
end endmodule

```

In the stimulus file, the input data bit width is set to 32, the output is little-endian, and three known data are input. Therefore, taking the first input data as an example, the output data of the first multi-byte serial port transmission should be 0x67, 0x45, 0x23, 0x01 sequentially, totaling 32 bytes of data. The subsequent two transmission results are similar. By comparing whether the simulation waveform is consistent with the deduction, we can know whether the design is correct.

The simulation waveform of this design is shown in Figure 9-4:

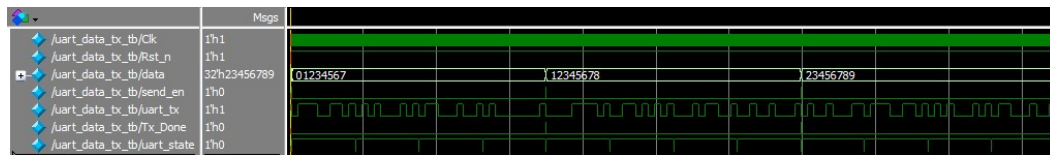


Figure 9-4 Multi-Byte Serial Port Transmit Module Simulation Waveform

It can be seen that the three data were successfully received, and the serial port completed three multi-byte transmission tasks. To facilitate viewing the specific data sent by the serial port, the data_byte and Tx_Done signals of the uart_byte_tx module are added here for observation. The complete waveform timing is shown in Figure 9-5:

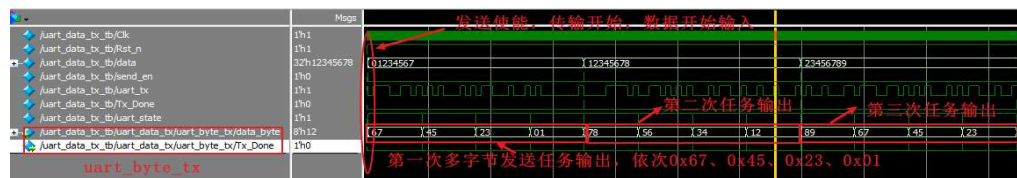


Figure 9-5 First Multi-Byte Serial Port Transmission

It can be seen that after the send_en signal is valid, the multi-byte serial port transmit module starts to work, and the output value is consistent with the expected result of the first multi-byte serial port transmission. The results of the subsequent two transmissions are also consistent with expectations. The detailed timing is not listed here. This design is successful.

9.2 Multi-Byte Serial Port Reception Principle and Design

The serial port receive module receives continuous serial data when working, and outputs valid data in bytes. The so-called multi-byte serial port reception means that after receiving data, the serial port receive module no longer outputs single-byte data, but splices the data, combines them into a multi-byte data, and then outputs it.

Combined with the multi-byte serial port transmit module we designed earlier, to implement the multi-byte serial port receive module design, we can instantiate the single-byte serial port receive module, and then design a state machine to splice the data output by the single-byte serial port receive module. At the same time, based on the reception completion signal generated by the single-byte serial port receive module, count the amount of data currently received (spliced). When the spliced data reaches the specified bit width, the data is output.

9.2.1 Multi-Byte Serial Port Reception Principle

Of course, this alone is not enough. If during the reception process, due to some external interference, part of the data changes and cannot be identified, then the subsequently successfully identified data will replace the positions of these data, eventually splicing into incorrect data.

For example, suppose that during a certain serial port reception, the serial port should have correctly received 0x98, 0x76, 0x54, 0x32. However, during transmission, due to external

interference, partial bit data of 0x76 changed and was not identified by the serial port. The data finally received was 0x98, 0x54, 0x32. If these data are directly given to the state machine for splicing, the final output will inevitably be incorrect data.

To prevent this situation, we need to time each reception. Once the reception times out, a reception timeout flag signal is generated, and the current serial port reception is terminated through the state machine. Through the above analysis, we can derive the system structure of this design as shown in Figure 9-6:

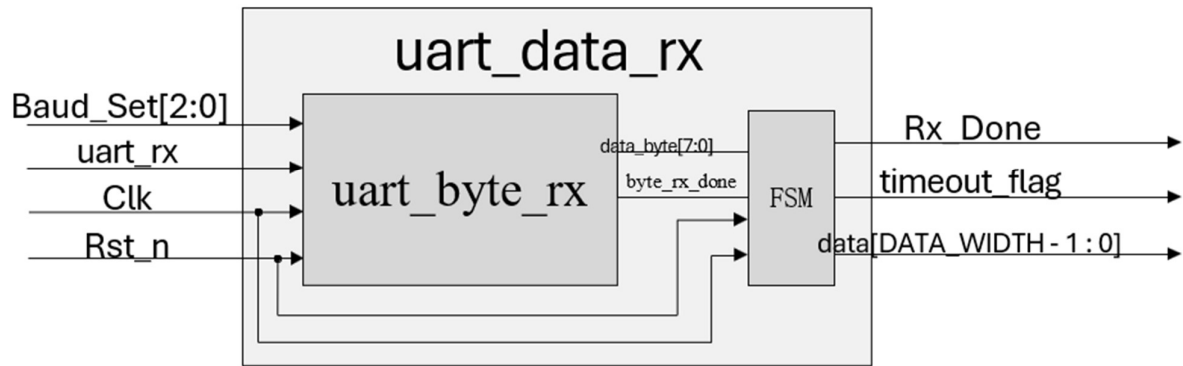


Figure 9-6 Multi-Byte Serial Port Receive Module System Block Diagram

9.2.2 Multi-Byte Serial Port Receive Module Design

After determining the overall structure of the design, the next step is the code implementation of the design. First is the instantiation of the single-byte serial port receive module. The code is as follows:

```

input Clk; input Rst_n;
input uart_rx;
wire [7:0] data_byte; wire
byte_rx_done;
uart_byte_rx uart_byte_rx(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .baud_set(Baud_Set),
    .uart_rx(uart_rx),
    .data_byte(data_byte), // byte vừa được nhận từ mô-đun UART 1 byte
    .Rx_Done(byte_rx_done) // cờ báo hiệu hoàn tất nhận dữ liệu
);

```

To enable the design to adapt to more application scenarios, we can use parameter to define the output bit width of the multi-byte serial port receive module as 16. In this way, we can modify the module output bit width through the top layer at any time when calling this module. The code is as follows:

```

parameter DATA_WIDTH = 16;

```

Next, we should consider how to generate the reception timeout signal. For serial port reception, in different application scenarios, different baud rates will be used, and the frequency of serial port data transmission is also different under each baud rate. So how do we judge whether the reception has timed out?

Here, we refer to the definition of serial port transmission timeout in Modbus. In Modbus, it is stipulated that the message interval must be greater than 3.5 character times, that is, transmission exceeding 3.5 character times is regarded as a new message. Based on this, in this design, we consider that during serial port data reception, once the interval between data exceeds 3.5 character times, it is a reception timeout.

Taking the commonly used 115200 baud rate as an example, we can calculate that the time required for the serial port to normally receive one byte of data is $1/115200 \times 10^9 \times (1+8+1) = 86.805\mu s$. Then if the serial port does not receive data for more than $86.805 \times 3.5 = 303.8175\mu s$, it can be considered a reception timeout.

In this design, we use a 50MHz clock, that is, each clock cycle is 20ns. The time spent by the serial port receiving one byte of data can be determined by counting the clock rising edges between two serial port reception completion signals. Therefore, we can calculate that when the count value is greater than $303.8175 \times 10^3 / 20 - 1 = 15190$, reception times out. Once reception times out, we let it generate a timeout flag signal.

As for when the counter should count, we can let the counter work at the beginning of each byte reception and be cleared when the byte reception is completed. Specifically, it uses the low-level start signal during serial transmission (!uart_rx) and the byte_rx_done signal generated by the single-byte serial port receive module after receiving a byte. Therefore, the design code for judging and generating the serial port reception timeout flag signal is as follows:

```
parameter TIMEOUT = 15190; // 115200 baud rate is 15190, 9600 baud rate is 182292

reg [31:0] timeout_cnt; always@(posedge Clk or
negedge Rst_n) if(!Rst_n)
    timeout_flag <= 1'd0;
else if(timeout_cnt >= TIMEOUT)
    timeout_flag <= 1'd1; else
if(state == S0)
    timeout_flag <= 1'd0;

reg to_state; // dùng để theo dõi trạng thái của việc nhận dữ liệu
always@(posedge Clk or negedge Rst_n) if(!Rst_n)
    to_state <= 0; else
if(!uart_rx)
    to_state <= 1; else
if(byte_rx_done)
    to_state <= 0;
always@(posedge Clk or negedge Rst_n) if(!Rst_n)
    timeout_cnt <= 32'd0; else
if(to_state) begin
```

```

        if(byte_rx_done) timeout_cnt <= 32'd0;
        else if(timeout_cnt >= TIMEOUT)
            timeout_cnt <= TIMEOUT;//nếu timeout_cnt=TIMEOUT giữ nguyên=TIMEOUT
        else
            timeout_cnt <= timeout_cnt + 1'd1;
    end
end

```

After designing the reception timeout signal, next is the state machine. Combined with the previous principle analysis and structure diagram, we can know the workflow of the state machine in this design is as follows:

1. Wait for the reception completion signal of the single-byte receive module, buffer the first byte of data, and count the buffered data.
2. Start timeout counting. If reception times out before the next transmission completion signal arrives, generate the Rx_Done signal, and the transmission ends. If not timed out, splice the new byte of data and count the buffered data.
3. Compare the buffered data with the output data bit width to check if all data reception is complete. If complete, generate the Rx_Done signal, and transmission ends. If not complete, continue step 2.

Therefore, the entire state requires only three states to implement the function. State definition and initialization are as follows:

```

localparam S0 = 0; // Wait for single-byte reception completion signal
localparam S1 = 1; // Judge if reception times out
localparam S2 = 2; // Check if all data reception is complete

```

Next is the implementation of state S0. Considering that in different application scenarios, there may be different byte order requirements for data, we can use parameter to define a constant MSB_FIRST to control the big-endian or little-endian byte order of the data. The code for this state is as follows:

```

S0, S1
begin
    Rx_Done <= 0;
    data_r <= 0;
    if(DATA_WIDTH == 8)begin
        data <= data_byte;
        Rx_Done <= byte_rx_done;
    end
    else if(byte_rx_done)begin state <=
        S1;
        cnt <= cnt + 9'd8; if(MSB_FIRST == 1)
            data_r <= {data_r[DATA_WIDTH - 1 - 8 : 0], data_byte};
        else
            data_r <= {data_byte, data_r[DATA_WIDTH - 1 : 8]};
        end
    end
end
end

```

```

        if(timeout_flag)begin state <= 0;
            Rx_Done <= 1;
        end
        else if(byte_rx_done)begin state <= S2;
            cnt <= cnt + 9'd8;
            if(MSB_FIRST == 1)
                data_r <= {data_r[DATA_WIDTH - 1 - 8 : 0], data_byte};
            else
                data_r <= {data_byte, data_r[DATA_WIDTH - 1 : 8]};
        end
S2:
        if(cnt >= DATA_WIDTH)begin state <= S0;
            cnt <= 0;
            data <= data_r; Rx_Done <=
                1;
        end
        else begin
            state <= S1; Rx_Done <= 0;
        end
    end
end

```

At this point, we have completed the design of the multi-byte serial port receive module. As to whether the module can work properly, further verification is needed. Therefore, next we need to create a simulation stimulus file to perform simulation verification on the module.

9.2.3 Stimulus Creation and Simulation Testing

To facilitate testing, in the stimulus file, we can instantiate the multi-byte serial port transmit module designed earlier to generate the serial input signals needed to test the multi-byte serial port receive module. The complete code of this simulation stimulus file is as follows:

```

module uart_data_rx_tb();

    parameter DATA_WIDTH = 32; parameter
    MSB_FIRST = 0;
    reg Clk; reg Rst_n;

    reg [DATA_WIDTH - 1 : 0]data; wire
    [DATA_WIDTH - 1 : 0]rx_data; reg send_en;
    wire uart_tx; wire uart_rx;
    wire Tx_Done; wire
    uart_state; wire Rx_Done;
    wire timeout_flag; assign uart_rx =

    uart_tx;

    uart_data_tx #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )
    uart_data_tx(

```

```

        .Clk(Clk),
        .Rst_n(Rst_n),
        .data(data),
        .send_en(send_en),
        .Baud_Set(3'd4),
        .uart_tx(uart_tx),
        .Tx_Done(Tx_Done),
        .uart_state(uart_state)
    );
    uart_data_rx #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )
    uart_data_rx(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .uart_rx(uart_rx),
        .data(rx_data),
        .Rx_Done(Rx_Done),
        .timeout_flag(timeout_flag),
        .Baud_Set(3'd4)
    );

```

```

initial Clk = 1; always #10 Clk
= !Clk;

```

```

initial begin
    Rst_n = 0;
    data = 0;
    send_en = 0;
    #201;
    Rst_n = 1;
    #2000;
    data = 32'h12345678;
    send_en = 1;
    #20;
    send_en = 0;
    #20;
    @(posedge Tx_Done); #1;
    #1000000;
    data = 32'h87654321;
    send_en = 1;
    #20;
    send_en = 0;
    #20;
    @(posedge Tx_Done); #1;
    #1000000;
    data = 32'h24680135;
    send_en = 1;

```

```

        #20;
        send_en = 0;
        #20;
        @(posedge Tx_Done);
        #1;
        #400000;
        $stop;
    end

endmodule

```

In the stimulus file, the baud rate is set to 115200. To verify whether the module can correctly judge whether the transmission times out, the interval between every two data inputs is set to 1000.001us.

At the same time, in the stimulus file, DATA_WIDTH is set to 32, and the byte order is set to little-endian. Since it is little-endian, the multi-byte serial port transmit module will send the low-order byte first and then the high-order byte. Taking the first data transmission as an example, the multi-byte serial port transmit module will send data 0x78, 0x56, 0x34, 0x12 sequentially. Since the multi-byte serial port receive module also uses little-endian byte order, when these byte data are received and spliced by the multi-byte serial port receive module, the byte received first will be treated as the lowest bits. Therefore, the data output by the multi-byte serial port receive module is 32'h12345678.

Based on the theoretical results, observe whether the simulation waveform is consistent with the deduction, we can know whether the design is correct. The overall simulation waveform of this design is shown in Figure 9-7.

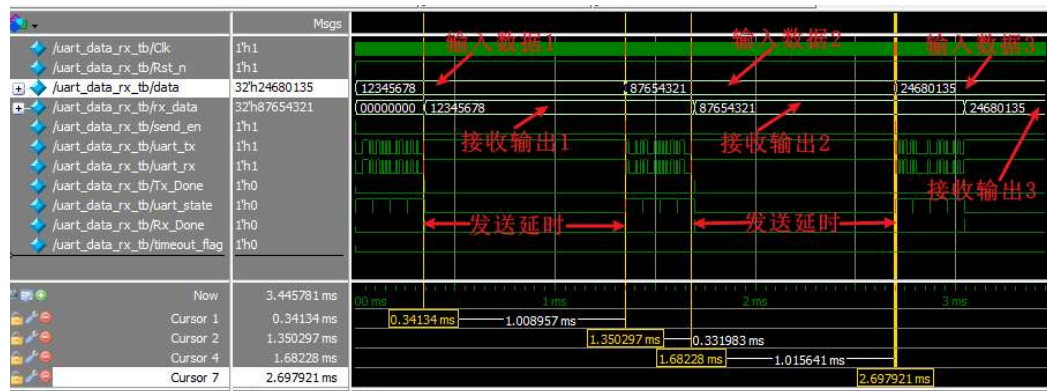


Figure 9-7 Simulation Overall Waveform

It can be seen that even with a delay of 1000.001us, the timeout_flag signal is not set high, and the output data of the multi-byte serial port receive module is consistent with our expectations.

Next, let's observe the intermediate process of data transmission to see if the byte order function is working properly. For convenience of observation, the data_byte (output byte data) signal of uart_byte_tx and the data_byte (input byte data) signal of uart_byte_rx are added to the simulation. The waveform timing is shown in Figure 9-8:

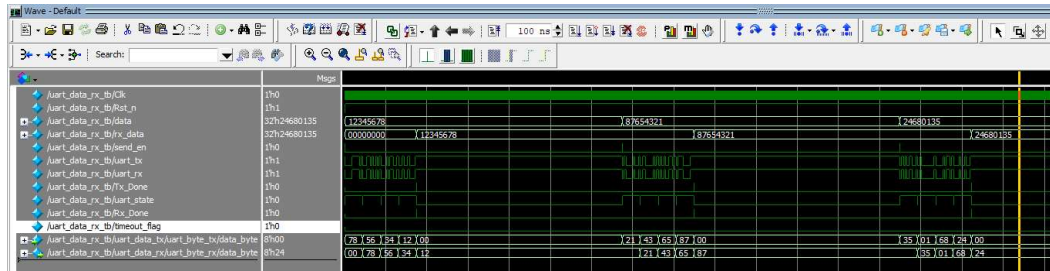


Figure 9-8 Multi-Byte Serial Port Reception

It can be seen that the data waveform is consistent with our derivation, indicating that the byte order function is also working normally. At this point, the design of the multi-byte serial port receive module is complete.

9.3 Creating Top-Level Encapsulation for Testing

Although we have verified the feasibility of the two modules' functions through simulation testing, considering that simulation is ultimately an ideal condition and will always differ from actual application situations, the design still needs board-level testing to verify the integrity of its functions.

For convenience of testing, here we create a new top layer, instantiate the multi-byte serial port transceiver module, and connect the output terminal of the receive module to the input terminal of the transmit module. In this way, the data received by the receive module will be sent out through the transmit module. Then through the serial port debugging software on the computer side, a simple serial port loopback can be implemented. By judging whether the loopback data is consistent, we can know whether the module function is implemented.

The top-level code is as follows:

```
module uart_loopback( Clk,
    Rst_n, uart_rx,
    led, uart_tx
);
    parameter DATA_WIDTH = 32; parameter
    MSB_FIRST = 0;
    input Clk; input Rst_n;
    input uart_rx;
    output [2:0]led; output
    uart_tx;
    wire [DATA_WIDTH-1:0]data; wire
    Rx_Done;
    wire [7:0]data_byte;
    uart_data_rx #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )
    uart_data_rx(
        .Clk(Clk),
        .Rst_n(Rst_n),
```

```

        .uart_rx(uart_rx),

        .data(data),
        .Rx_Done(Rx_Done),
        .timeout_flag(led[0]), .Baud_Set(3'd4)
    );

    uart_data_tx #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )uart_data_tx(
        .Clk(Clk),
        .Rst_n(Rst_n),

        .data(data),
        .send_en(Rx_Done),
        .Baud_Set(3'd4),

        .uart_tx(uart_tx),
        .Tx_Done(led[1]),
        .uart_state(led[2])
    );

endmodule

```

In the top layer, DATA_WIDTH is set to 32, the baud rate is 115200, and the timeout signal, transmission completion signal, and serial port status signal are connected to LED lights for display.

After adding the test top layer and confirming correct synthesis, we need to add pin assignments and level constraints for the design. The pin assignment table for this design is shown in Table 9-1:

Table 9-1 Pin Assignment Table

Signal Name	Pin NO.	Signal Name	Pin NO.
CLK	T9	led[2]	B9
Rst_n	B16	led[1]	C14
uart_rx	U8	led[0]	D14
uart_tx	V8		

The pin level constraint is LVCMOS 33. After setting is complete, compile the design and generate the fs file. Next, prepare for board-level verification of the design.

9.4 Board-Level Verification

This section will use the designed multi-byte serial port transceiver module on the ACG525 development board to loop back data from the computer via serial port loopback. Verify the design module by comparing whether the loopback data is consistent with the sent data.

9.4.1 System Hardware Required

1. ACG525 Development Board x1
2. Gowin Downloader x1
3. Serial Cable x1

4. Power Cable x1

9.4.2 Hardware Connection

Connect the development board's download cable and power cable. Use the serial cable to connect one end to the UART interface on the development board and the other end to the USB port of the computer, as shown in Figure 8-6 below.

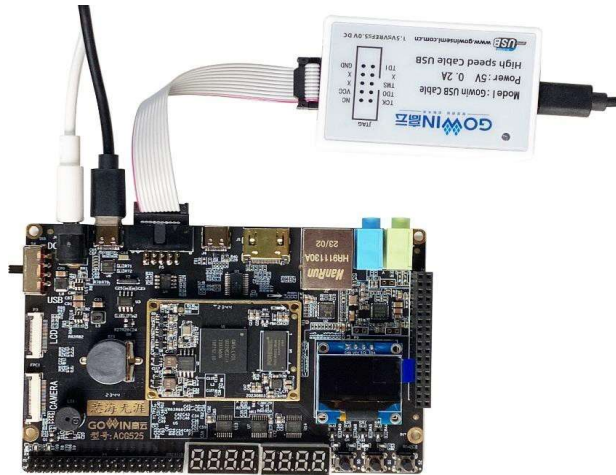


Figure 9-9 Hardware Connection

After connection is complete, ensure the development board power switch is toggled to the ON side. Next, program the generated fs file into the development board. After programming is complete, open the Device Manager to query the serial port number. The name of the serial port in the Device Manager is shown in Figure 9-10:

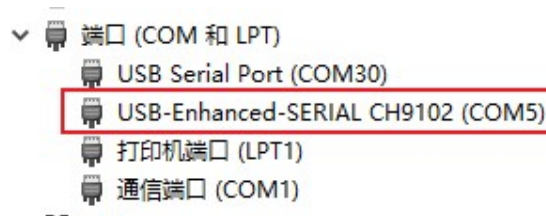


Figure 9-10 Query Port Number

Here, my computer's serial port number is COM5. Users should refer to the port number in their own computers when querying. Then open Serial Hunter, connect to the port just queried in the software, and set the baud rate to 115200. Then input 4 bytes of data, click send to start loopback, as shown in the experimental result in Figure 9-11:

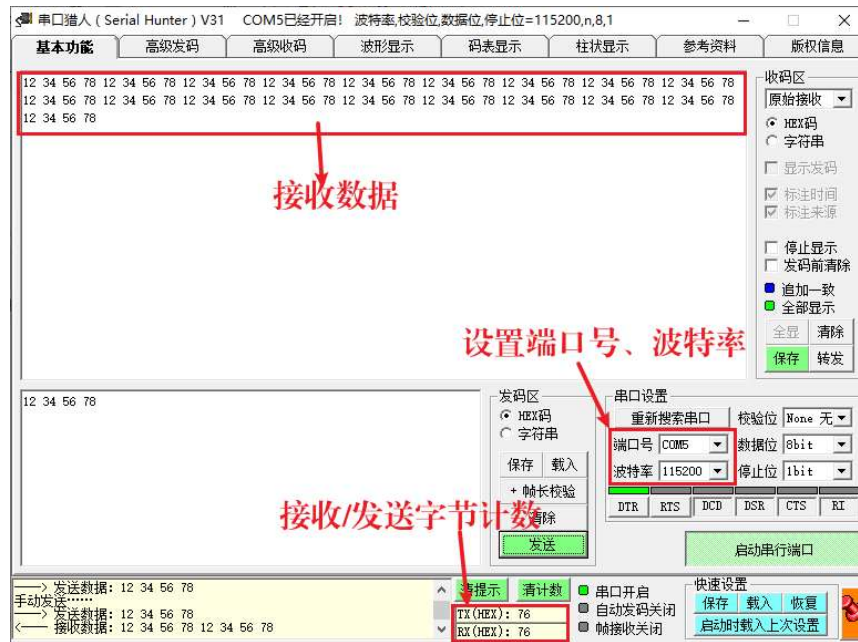


Figure 9-11 Experimental Result

Here I sent multiple times, totaling 76 bytes of data. It can be seen that the software received 76 bytes of loopback data, and the received data is consistent with the sent data. This proves that the function is normal, and the design of the multi-byte serial port transceiver module is successful.

9.5 Summary

This chapter guided everyone to complete the design of the multi-byte serial port receive module and the multi-byte serial port transmit module, and verified the feasibility of the design through simulation testing and board-level verification. For ease of use, some signals in the module use parameter definitions. Users can modify these parameter values according to their own design needs through top-level parameter passing.

Chapter 10: State Machine Design Example

Project Source Code, |----02_Design Examples, |----ch10_fsm_hello

Chapter Overview

State Machine, or Finite State Machine (FSM), is a mathematical model representing a finite number of states and the transitions and actions between these states. This section will learn concepts related to state machines and use a state machine to implement the detection of a specific string. Understand the differences, advantages, and disadvantages of one-segment, two-segment, and three-segment state machines.

10.1 State Machine Working Principle

State machines are divided into Moore type finite state machines and Mealy type finite state machines. The output of a Moore state machine is determined only by the current state (does not directly depend on input). The output of a Mealy finite state machine depends not only on its current state but also on its input, which is also the difference from the Moore finite state machine. Their schematic diagrams are shown in Figure 10-1 Moore State Machine and Figure 10-2 Mealy State Machine respectively.

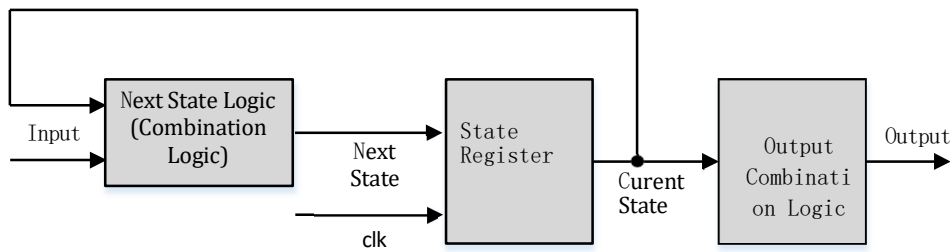


Figure 10-1 Moore State Machine

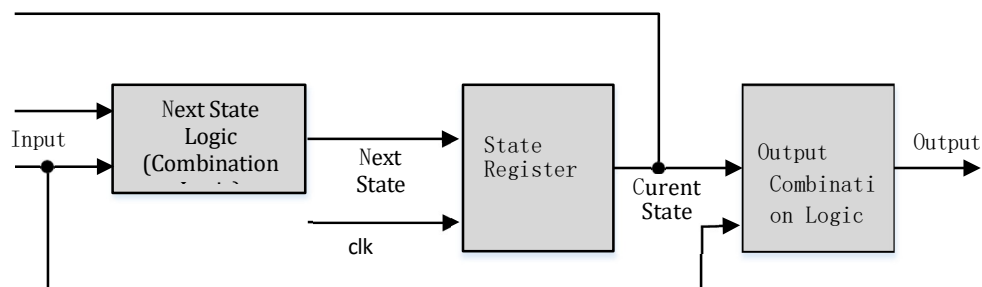


Figure 10-2 Mealy State Machine

State machines can perform state transitions according to preset states based on control signals, which brings up the issue of how to effectively encode states. The simplest encoding method is to directly use binary encoding representation. In addition, there are Gray code and One-hot code. Assuming there are eight states from A to H, the different encoding formats are shown in Table 10-1 below.

Table 10-1 Different State Encoding Formats

Encoding Format State	Binary	one-hot	Gray
A	3' b000	8' b0000_0000	4' b0000
B	3' b001	8' b0000_0010	4' b0001
C	3' b010	8' b0000_0100	4' b0011
D	3' b011	8' b0000_1000	4' b0010
E	3' b100	8' b0001_0000	4' b0110
F	3' b101	8' b0010_0000	4' b0111
G	3' b110	8' b0100_0000	4' b0101
H	3' b111	8' b1000_0000	4' b0100

From the table above, it can be found that: One-hot code uses one register for each state, and only needs to compare one bit when comparing states, which is simpler than other decoding circuits; Gray code requires the same number of registers as binary code, decoding is complex, but adjacent bits only change by one bit, generally used for multi-bit conversion in asynchronous multi-clock domains, such as asynchronous FIFO; Binary code is the most common encoding method, uses fewer registers, but decoding is more complex.

According to suggestions from FPGA manufacturers, which encoding format to choose depends on the complexity of the state machine, device type, and requirements for recovering from illegal states. It can be seen from the RTL view generated using different encoding formats that binary uses fewer registers than one-hot code. Binary can implement a 100-state state machine with 7 registers, but one-hot code requires 100 registers. However, on the other hand, although one-hot code uses more registers, its combinational logic is relatively simple. Generally in CPLDs, binary is recommended due to providing more combinational logic resources, while in FPGAs, one-hot code is recommended due to providing more sequential logic.

State machine description methods can be divided into one-segment, two-segment, and three-segment.

One-segment: The entire state machine is written into one always block. This module describes both state transitions and state inputs and outputs.

Two-segment: Uses two always blocks to describe the state machine. One always block uses synchronous sequential logic to describe state transitions, and the other module uses combinational logic to judge state transition conditions and describe state transition rules and their outputs.

Three-segment: Based on the two always block description method, uses three always blocks. One always block uses synchronous sequential logic to describe state transitions, one always uses combinational logic to judge state transition conditions and describe state transition rules, and another always module describes state outputs (can use combinational circuit output or sequential circuit output).

It can be seen that the difference between a two-segment finite state machine and a one-segment finite state machine is that the sequential part (state transition) and the combinational part (judging state transition conditions and generating outputs) are separated and written as two

always statements, which is a two-segment finite state machine. Separating the judgment of state transition conditions and generating outputs in the combinational part further results in a three-segment finite state machine. In the two-segment style, when the combinational logic is particularly complex, attention should be paid to adding a flip-flop at the back to eliminate the influence of glitches generated by combinational logic on the output. The three-segment style does not have this problem because the third always block generates flip-flops. In fact, modern devices do not care about this little resource consumption. It is recommended to use the two-segment or three-segment style and output registered state machine outputs to describe finite state machines.

Matters that should be mainly noted when writing state machines are: to avoid generating unnecessary latches, it is necessary to exhaustively list output actions corresponding to all states, or use default to define undefined state actions; when defining states, it is recommended to use parameter definitions localparam or parameter, so that states are clearer and less prone to errors during writing, and also convenient to modify. To return to the initial state or a predetermined state upon reset or runaway, asynchronous or synchronous reset should be included in the design to ensure the state machine has an initial state upon power-up.

10.2 String Detection State Machine Implementation

To implement the detection of the string "hello", first draw its state transition diagram, as shown in Figure 10-3 below.

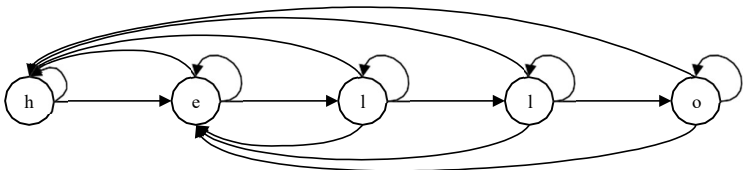


Figure 10-3 String "hello" Detection State Transition Diagram

From the figure above, it can be seen that if the transition condition is not met in a state, the state machine either returns to the initial state h or returns to state e (does not meet the backward transition condition, but the input character is 'h'), and each state has a specific direction, its output is only determined by the current state, so this is a Moore type state machine.

Create a new project named hello, add and create a new source file in this project, and save it as hello.v. The schematic of the entire string "hello" detection module is shown in Figure 10-4 below.

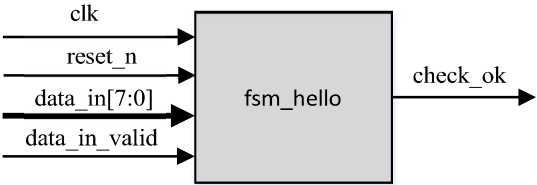


Figure 10-4 String Detection Module Diagram

Table 10-2 State Machine Module Port Function Description

Port Name	I/O	Function Description
clk	I	Module clock signal, 50 MHz
reset_n	I	Active-low reset signal for the module
data_in	I	8-bit input data
data_in_valid	I	Input valid flag; high level indicates valid input data, low level indicates no valid data
check_ok	O	Asserted high for one clock cycle when the string "hello" is detected successfully

First, use One-hot code to define each state.

```
localparam
    CHECK_h  = 5'b0_0001, CHECK_e  =
    5'b0_0010, CHECK_l1 = 5'b0_0100,
    CHECK_l2 = 5'b0_1000,
    CHECK_o  = 5'b1_0000;
```

Then, set the initial state.

```
always@(posedge clk or posedge reset)
if(reset)begin
    check_ok <= 1'b0; state <=
    CHECK_h;
end
else begin
    case(state)
        //phần này của mã bị bỏ qua và sẽ được mô tả sau
        default:state <= CHECK_h; endcase
    end
```

Start writing the state transition part according to Figure 10-3. When in the initial state CHECK_h, if the character 'h' input is detected, jump to the CHECK_e state to start detecting the character 'e', otherwise remain in the CHECK_h state waiting for the character 'h' input.

```
CHECK_h:
begin
    check_ok <= 1'b0; if(data_valid &&
    data_in == "h")
        state <= CHECK_e; else
        state <= CHECK_h;
    end
```

When in state CHECK_e, if character 'e' input is detected, enter CHECK_l1 state to start detecting the first character 'l'; if character 'h' input is detected, remain in CHECK_e state waiting for

character 'e' input; if any other character input is detected, jump back to the initial state to wait for a new round of detection; otherwise stay in state CHECK_e waiting for character input.

```
CHECK_e:
    if(data_valid && data_in == "e") state <=
        CHECK_l1;
    else if(data_valid && data_in == "h") state <=
        CHECK_e;
    else if(data_valid) state <=
        CHECK_h;
    else
        state <= CHECK_e;
```

When in state CHECK_l1, if character 'l' input is detected, enter CHECK_l2 state to start detecting the second character 'l'; if character 'h' input is detected, jump to CHECK_e state waiting for character 'e' input; if any other character input is detected, jump back to the initial state to wait for a new round of detection; otherwise stay in state CHECK_l1 waiting for character input.

```
CHECK_l1:
    if(data_valid && data_in == "l") state <=
        CHECK_l2;
    else if(data_valid && data_in == "h") state <=
        CHECK_e;
    else if(data_valid) state <=
        CHECK_h;
    else
        state <= CHECK_l1;
state <= CHECK_l1;
```

When in state CHECK_l2, if character 'l' input is detected, enter CHECK_o state to start detecting character 'o'; if character 'h' input is detected, jump to CHECK_e state waiting for character 'e' input; if any other character input is detected, jump back to the initial state to wait for a new round of detection; otherwise stay in state CHECK_l2 waiting for character input.

```
CHECK_l2:
    if(data_valid && data_in == "l") state <=
        CHECK_o;
    else if(data_valid && data_in == "h") state <=
        CHECK_e;
    else if(data_valid)
        state <= CHECK_h;
    else
        state <= CHECK_l2;
```

When in state CHECK_o, if character 'o' input is detected, it means the detection of "hello" string is completed, return to the initial state to wait for a new round of detection, and control check_ok to generate a clock pulse; if character 'h' input is detected, jump to CHECK_e state waiting for character 'e' input; if any other character input is detected, jump back to the initial state to wait for a new round of detection; otherwise stay in state CHECK_o waiting for character input.

```
CHECK_o:
```

```

if(data_valid && data_in == "h") state <=
CHECK_e;
else if(data_valid)begin state <=
CHECK_h; if(data_in == "o")
check_ok <= 1'b1; else
check_ok <= 1'b0;
end else
state <= CHECK_o;

```

Perform analysis and synthesis on the designed code until there are no errors or critical warnings.

10.3 Stimulus Creation and Simulation Testing

To test the simulation, write a test stimulus file. Create a new Verilog File under the project and name it fsm_hello_tb.v. This stimulus file generates normal clock and reset signals, and also simulates different character input situations. The Testbench design designs the generation of a single character in the form of a task. By calling the task to generate different strings, the correctness of the design is verified. The specific code is as follows.

```

`timescale 1ns/1ns
`define CLK_PERIOD 20 module

fsm_hello_tb;

reg clk;
reg reset_n;

reg data_valid; reg
[7:0]data_in;

wire check_ok; fsm_hello

fsm_hello(
    .clk(clk),
    .reset_n(reset_n),
    .data_valid(data_valid),
    .data_in(data_in),
    .check_ok(check_ok)
);

initial clk = 1; always#(`CLK_PERIOD/2) clk =
~clk;

initial begin
    reset_n = 0;
    data_valid = 0;
    data_in = 0;
    #(`CLK_PERIOD*20);
    reset_n = 1;
    #(`CLK_PERIOD*20 + 1);
    repeat(2) begin gen_char("I");

```

```

                                #(`CLK_PERIOD);
gen_ch

                                gen_char("h");
                                #(`CLK_PERIOD);
                                gen_char("e");
                                #(`CLK_PERIOD);
                                gen_char("l");
                                #(`CLK_PERIOD);
                                gen_char("l");
                                #(`CLK_PERIOD);
                                gen_char("l");
                                #(`CLK_PERIOD);
                                gen_char("h");
                                #(`CLK_PERIOD);
                                gen_char("e");
                                #(`CLK_PERIOD);
                                gen_char("l");
                                #(`CLK_PERIOD);
                                gen_char("l");
                                #(`CLK_PERIOD);
                                gen_char("o");
                                #(`CLK_PERIOD);
                                gen_char("e");
                                #(`CLK_PERIOD);
                                gen_char("h");
                                #(`CLK_PERIOD);
                                gen_char("h");

                                #(`CLK_PERIOD);
                                gen_char("o"); #(`CLK_PERIOD);
end

                                #200;
                                $stop;
end

task gen_char;
    input [7:0]char; begin
        data_in = char; data_valid = 1'b1;
        #(`CLK_PERIOD);
        data_valid = 1'b0;
    end endtask
endmodule

```

Open Modelsim, create a new project named "fsm_hello", and add files "fsm_hello.v" and "fsm_hello_tb.v". Then perform simulation configuration. After configuration, run the simulation. You can see the waveform file as shown in Figure 10-5 below. It can be seen that state transitions do not occur before the reset signal is set high. After the reset is valid, only when a character is input (data_valid is high), the state will transition according to the design, and no transition errors

occur. When the string "hello" is correctly detected, check_ok generates a one-cycle pulse indicating a successful detection.

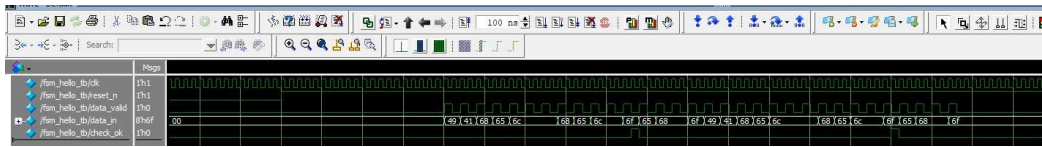


Figure 10-5 Simulation Waveform File

At this point, a simple state machine design is completed. In subsequent chapters, the state machine design idea will be used frequently. The two-segment and three-segment state machines will not be elaborated here, but will be explained in detail in specific examples.

10.4 String Detection Board-Level Testing

Through the learning of the string detection state machine above, combined with the serial port receive module learned in previous chapters, design a serial port receive string detection system. The block diagram of the entire system is shown in Figure 10-6 below. The system function is that the FPGA receives the string data stream sent by the PC through the serial port via the serial port receive module, and then detects the string through the string detection module. Every time the "hello" string is completely received, the LED light is toggled.

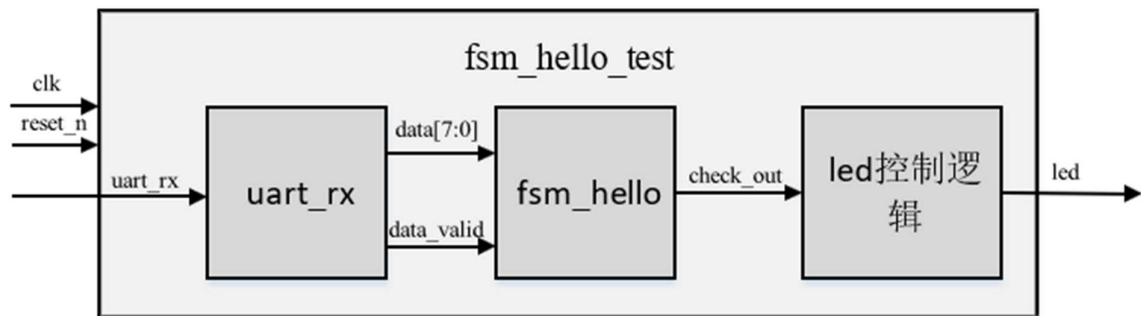


Figure 10-6 Serial Port Reception String Detection System Block Diagram

The top layer of the system design is fsm_hello_test. The specific design code is as follows.

```
module fsm_hello_test(
    clk, reset_n,

    uart_rx, led
);
    assign reset=~reset_n;
    input clk;    //system clk 50Mhz
    input reset_n;    //chân reset

    input uart_rx;    //tín hiệu nối tiếp

    output led;    //LED hiển thị
```

```

    reg led;
    wire [7:0]data;// Serial received data, used as character data input
    wire data_valid; //1byte nhận xong lên 1

    wire check_ok; //1 chuỗi đúng lên 1

//nhận dữ liệu cổng nối tiếp làm đầu vào để phát hiện chuỗi
uart_byte_rx uart_byte_rx(
    .clk(clk),
    .reset_n(reset_n),

    .baud_set(3'd0),
    .uart_rx(uart_rx),

    .data_byte(data),
    .rx_done(data_valid)
);

fsm_hello fsm_hello(
    .clk(clk),
    .reset_n(reset_n),
    .data_valid(data_valid),
    .data_in(data),
    .check_ok(check_ok)
);
always@(posedge clk or posedge reset) if(reset)
    led <= 1'b0; else
    if(check_ok)
        led <= ~led; else
        led <= led;
endmodule

```

The simulation of the top-level module can be modeled after the serial port receive module. The design of the top-level simulation code will not be explained here additionally. Readers can verify the correctness of the top-level design function by simulation themselves. Under the condition that the top-level design function is correct determined by simulation, constrain the top-level input/output pins. led is bound to the pin of LED0 on the board. The specific pin constraints are shown in Figure 10-7 below.

	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive
1	clk	input		T9	4	False	LVC MOS33	OFF
2	led	output		D14	1	False	LVC MOS33	8
3	reset_n	input		B16	1	False	LVC MOS33	OFF
4	uart_rx	input		U8	5	False	LVC MOS33	OFF

Figure 10-7 Pin Constraints

After pin constraints are completed, compile the entire project and generate the fs file. Download the fs file to the development board. After the fs file is downloaded to the FPGA, you can see that LED0 on the bound pin is off. Connect the computer and the FPGA serial port with a data cable.

Open a serial port software on the computer (taking Serial Hunter as an example), select the serial port number connected to the FPGA, set the serial port baud rate to 9600, and set the send data format to String. Send several sets of string data on the serial port software and observe the changes of LED0 on the board. Below are the serial port software windows sending strings 3 times in sequence.



Figure 10-8 Sending Test String 1



Figure 10-9 Sending Test String 2



Figure 10-10 Sending Test String 3

- (1) When the string "hello" is sent, LED0 on the board changes from off to on, test result is correct.
- (2) When the string "helllo" is sent, LED0 on the board does not change, test result is correct.
- (3) When the string "lhoolehelo" is sent, LED0 on the board changes from on to off, test result is correct.

10.5 Summary

This chapter's experiment led readers to understand the working principle of state machines and the implementation of string detection state machines. Readers can test by sending other different string data via computer, observe the change of LED0 on the board. When the sent string contains

the complete "hello" string, LED0 will toggle once (on to off or off to on). Board-level testing can verify the correctness of the design, and the entire system design achieves the expected effect.

Chapter 11: Independent Key Debounce Design and Verification

Project Source Code, |----02_Design Examples, |----ch11_key_filter

Chapter Overview

As a basic human-machine interaction component, keys (buttons) are widely used in electronic design, from system resets to control settings. Currently, there are many types of keys, such as multi-directional keys, self-locking keys, membrane keys, etc.

This section will introduce the method of single-bit data asynchronous signal synchronization and edge detection into the key debounce design commonly used in FPGA. Random number generation functions and stimulus models are introduced in the simulation test stimulus file. Finally, a practical example is used to demonstrate the modular design method.

11.1 Key Physical Structure and Circuit Design

The hardware schematic of a common key is shown in Figure 11-1 below.

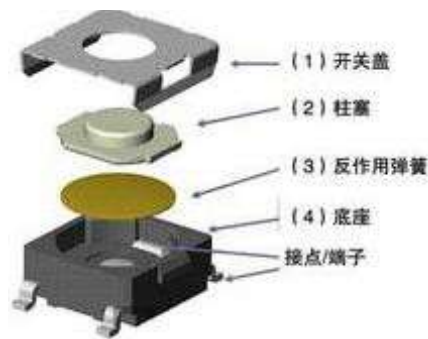


Figure 11-1 Key Hardware Structure Schematic

The ACG525 development board carries two-pin SMD keys, located at the bottom right corner of the front of the learning board respectively. The schematic is shown in the figure below. From the schematic, it can be seen that the IO port is high level when the key is not pressed, and becomes low level when the key is pressed. Therefore, the system can judge the state of the key by detecting the level of the IO.

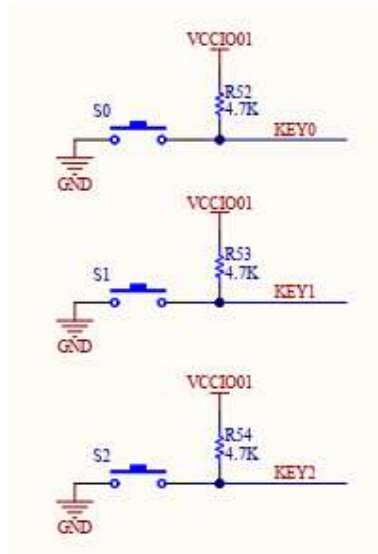


Figure 11-2 Key Schematic

It can be seen from the key structure schematic that there is a reaction spring in the key, so additional physical jitter will be generated when pressed or released. Physical jitter will generate level jitter. In the process of the key being pressed and then released, its level change is shown in Figure 11-3 below. The top is the ideal waveform output, and the bottom is the actual waveform output.

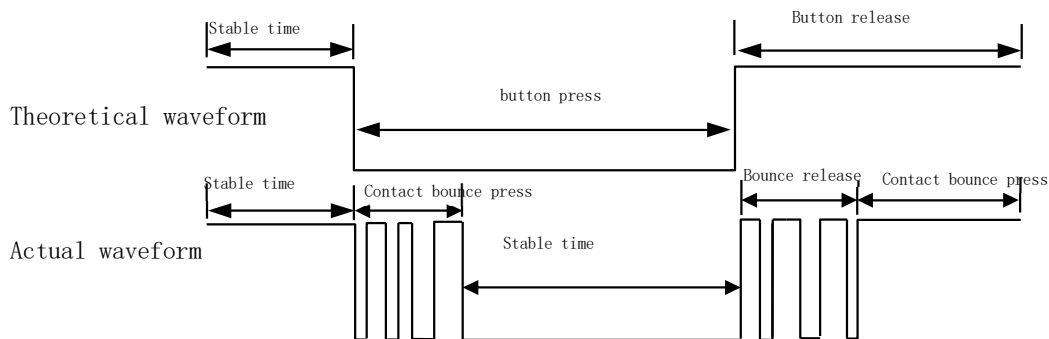


Figure 11-3 Level Change of Key from Pressed to Released

In the figure above, the number of jitters generated and the interval time are unpredictable. This requires filtering to eliminate the impact that jitter may cause on other external devices. Generally, the total time of jitter will last within 20ms. This jitter can be eliminated through hardware circuits or logic design, or through software. Hardware circuit debouncing is suitable for occasions with a small number of keys.

11.2 Hardware Circuit Implementation of Key Debounce

The circuit diagram using an RS flip-flop based on NAND gates to eliminate jitter is shown in Figure 11-4 below. Assuming the initial state switch is connected to B, at this time $S=1$, $R=0$, the flip-flop is set to 0, i.e., $Q=0$; in the process of B switching to A, there is a state where the switch contacts neither A nor B, at this time $S=1$, $R=1$, the flip-flop holds, i.e., $Q=0$; at the moment of switching to A,

$S=1$, $R=0$ (Note: typically $S=0$, $R=1$ for NAND latch set state, verify logic), the flip-flop is set to 1, i.e., $Q=1$. At this time, even if A jitters, i.e., $S=1$, the state of the flip-flop can still maintain the current state, i.e., $Q=1$ unchanged. The waveform diagram of the whole process is shown in the figure. Similarly, the state change of the flip-flop when the switch switches from A to B can be analyzed.

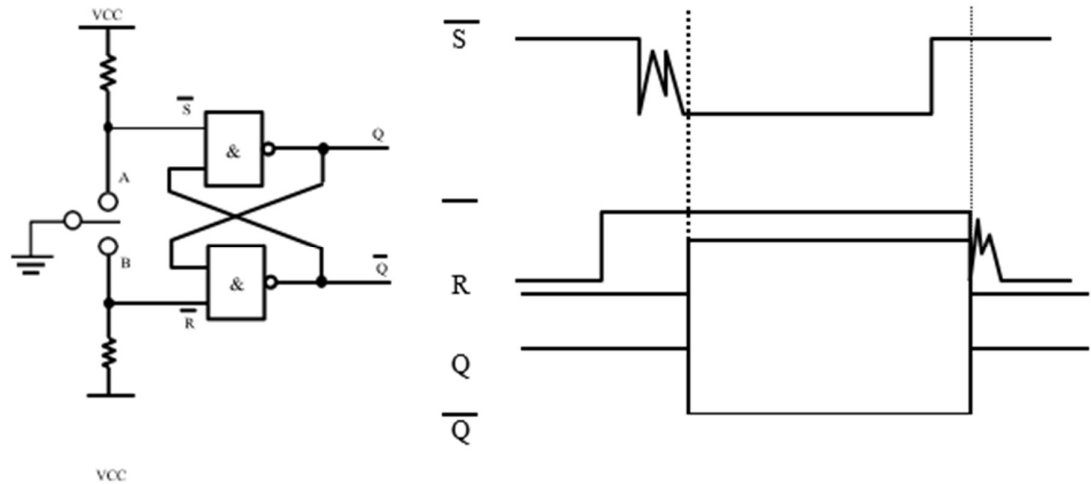


Figure 11-4 RS Flip-flop Debounce Circuit Based on NAND and Waveform Analysis

An RS flip-flop composed of NOR gates can also be used to implement the function. Its circuit is shown in Figure 11-5. You can analyze the working state of the circuit yourself based on the above analysis process.

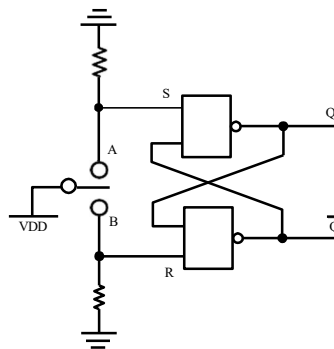


Figure 11-5 RS Flip-flop Debounce Circuit Based on NOR

It can be seen that the above two connection methods are only suitable for single-pole double-throw switches and are not suitable for the currently common two-pin or four-pin keys. For such keys, RC circuits shown in Figure 11-6 and Figure 11-7 below are commonly used.

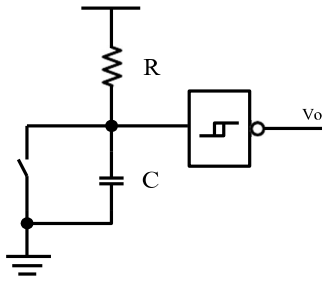


Figure 11-6 High Level Output

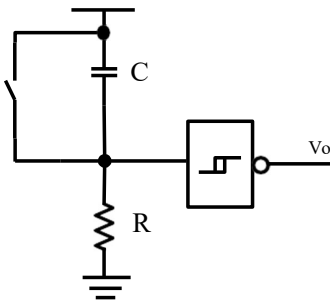


Figure 11-7 Low Level Output

The principle is to use capacitors and resistors to integrate the waveform, and then use a Schmitt trigger (comparator) to shape the waveform to obtain an ideal output waveform. For the high-level output circuit, when the switch is open, the capacitor charges to $V_c=5V$, and V_o output is 0; when the switch is closed, the capacitor discharges to $V_c=0$, and V_o output is 5V; when the key jitters, if the charging time constant RC is small, the capacitor will continue the charging and discharging process, charging to 5V and discharging to 0V. If the charging time constant RC is set large enough, when the switch jitters from the open state to the closed state, the capacitor does not charge above the threshold voltage of the Schmitt trigger, so the output V_o remains 5V. When the switch is open, the capacitor recharges to 5V, exceeding the threshold voltage, and the output changes to a low level. The whole process is shown in Figure 11-8. Jitter occurs when closing or releasing, but once truly stable, the output terminal will only change once.

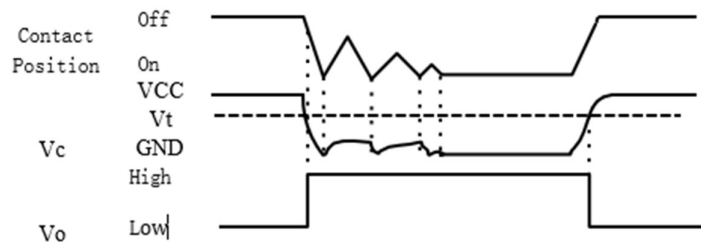


Figure 11-8 RC Circuit with High Level Output

A monostable trigger composed of a 555 timer can also be used to eliminate key jitter. The output terminal V_o is normally low level. When the falling edge of the pulse V_i arrives, it reverses to high level. At this time, the power supply charges C through R . When the voltage on capacitor C reaches $V_{CC} \cdot 2/3$, the discharge tube inside the 555 timer conducts, and C discharges quickly through it, and the output terminal flips to low level. In the process of the capacitor voltage rising from 0 to $2/3V_{CC}$, the output terminal is always high. At this time, even if there is jitter, the output will not change. By reasonably selecting the transient stability time T_d , the effect of anti-jitter can be achieved.

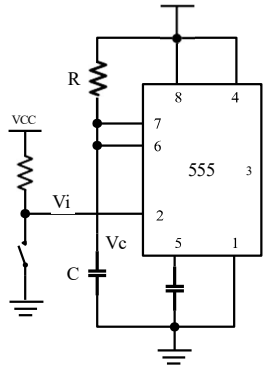


Figure 11-9 Debounce Circuit Based on Monostable Trigger

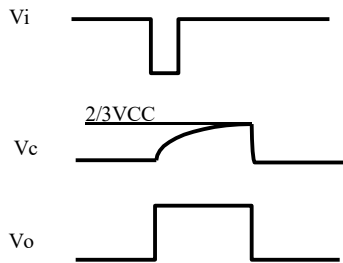


Figure 11-10 Waveform Analysis of Debounce Circuit Based on Monostable Trigger

11.3 State Machine Implementation of Key Debounce

11.3.1 Module Interface Design

For a single key debounce module, its interface is shown in Figure 11-11 below,



Figure 11-11 Key Debounce Module Interface

Table 11-1 Key Debounce Module Interface Function Description

Interface Name	I/O	Function Description
clk	I	Module working clock, 50 MHz
reset_n	I	Module reset signal (active low)
key_in	I	Key input signal
key_flag	O	Key state transition flag (asserted when key state changes)
key_state	O	Current key state flag

11.3.2 State Machine Transition States and Condition Design

For FPGA design, state machines are usually used for key debounce processing. As seen in the figure below, if state encoding analysis is performed according to the concept of state machines, the following states are obtained: idle state IDLE when not pressed, press jitter filtering state FILTER0, press stable state DOWN, and release jitter filtering state FILTER1. Their one-hot encodings are 4'b0001, 4'b0010, 4'b0100, and 4'b1000 respectively. Their state transition diagram is shown in Figure 11-12 below. It can be seen that it is a Mealy state machine.

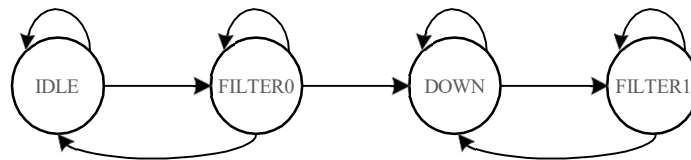


Figure 11-12 State Transition Diagram

The state transition conditions are shown in Table 11-2 below.

Table 11-2 Key Debounce State Transition Conditions

Current State	Next State	Transition Condition
IDLE	FILTER0	A falling edge on the key input is detected; start press debounce.
FILTER0	IDLE	A rising edge is detected during press debounce and the 20 ms timer has not expired; the key is considered still bouncing and unstable, return to IDLE.
FILTER0	DOWN	A falling edge is detected and the debounce timer exceeds 20 ms; the key is confirmed as pressed.

Current State	Next State	Transition Condition
DOWN	FILTER1	After the pressed state is stable, a rising edge is detected; start release debounce.
FILTER1	DOWN	During release debounce, if a falling edge is detected before the 20 ms timer expires, the key is considered still bouncing; continue debounce and return to DOWN.
FILTER1	IDLE	A rising edge is detected and the debounce timer exceeds 20 ms; the key is confirmed as released and the process is completed.

11.3.3 Key Input Signal Edge Detection

From the design of the state machine, it is necessary to detect the edge of the key input signal `key_in`. The theoretical knowledge of edge detection can be referred to the explanation in the serial port receive module, which will not be repeated here. The specific key data signal edge detection code is as follows.

```

always@(posedge clk or posedge reset) if(reset)begin
    key_in_sync1 <= 1'b0; key_in_sync2
    <= 1'b0;
end
else begin
    key_in_sync1 <= key_in; key_in_sync2 <=
    key_in_sync1;
end
// Use D flip-flop to store the level state of the external input signal (synchronized to the system
clock domain) at two adjacent clock rising edges

always@(posedge clk or posedge reset)

always@(posedge clk or posedge reset) if(reset)begin
    key_in_reg1 <= 1'b0; key_in_reg2
    <= 1'b0;
end
else begin
    key_in_reg1 <= key_in_sync2; key_in_reg2 <=
    key_in_reg1;
end
// Generate edge signals

assign key_in_nedge = !key_in_reg1 & key_in_reg2;
assign key_in_pedge = key_in_reg1 & (!key_in_reg2);

```

11.3.4 Counter Design

There should also be a 20ms counter module and a counter enable module in the state transition table. These can also be merged into one always module. Generally, it is recommended that one always block only operates on one signal.

```
always@(posedge clk or posedge reset) if(reset)
    cnt <= 20'd0; else
if(en_cnt)
    cnt <= cnt + 1'b1; else
    cnt <= 20'd0;

always@(posedge clk or posedge reset) if(reset)
    cnt_full <= 1'b0;
else if(cnt == 20'd999_999) cnt_full <= 1'b1;
else
    cnt_full <= 1'b0;
```

11.3.5 State Machine Design

Now start the state machine design. First, use local parameters to define the states of the state machine.

localparam

```
IDLE= 4'b0001, FILTER0=
4'b0010, DOWN= 4'b0100,
FILTER1= 4'b1000;
```

Since there are few states and judgment conditions, a one-segment state machine is used here for description first. When resetting, clear the counter, return the state to IDLE, and key_flag and key_state also return to the initial state.

```
always@(posedge clk or posedge reset) if(reset)begin
    en_cnt <= 1'b0; state <= IDLE;
    key_flag <= 1'b0; key_state <=
    1'b1;
end
else begin
    case(state)
        default:begin
            state <= IDLE; en_cnt <= 1'b0;
            key_flag <= 1'b0; key_state <=
            1'b1;
        end endcase
    end
end
```

When in the IDLE state (not pressed), if a falling edge is detected, the state enters the press jitter filtering state FILTER0, and the counter is enabled. Otherwise, the IDLE state is maintained.

```

IDLE :begin
    key_flag <= 1'b0; if(key_in_nedge)begin
        state <= FILTER0; en_cnt <=
            1'b1;
    end else
        state <= IDLE;
end

```

When in the FILTER0 state, if a rising edge arrives before the 20ms timing ends, it is considered that the key is still in the press jitter process, the state returns to IDLE and the counter is cleared to 0. After the last jitter in the pressing process, there will be no rising edge, and the counter can count continuously. After the count is full, key_flag is set to 1, key_state is set to 0, the state enters the press stable state DOWN, and the counter is cleared to 0. In this way, the state of the key can be determined by judging key_flag && !key_state. If it is 1, it is pressed.

FILTER0:begin

```

FILTER0:begin
    if(cnt_full)begin
        key_flag <= 1'b1; key_state <=
            1'b0;
        en_cnt <= 1'b0; state <=
            DOWN;
    end
    else if(key_in_pedge)begin state <= IDLE;
        en_cnt <= 1'b0;
    end else
        state <= FILTER0;
end

```

After entering the key stable state DOWN, clear key_flag to 0. If a rising edge is detected, enter the release jitter filtering state FILTER1, otherwise maintain the current state.

```

DOWN:begin
    key_flag <= 1'b0; if(key_in_pedge)begin
        state <= FILTER1; en_cnt <= 1'b1;
    end else
        state <= DOWN;
end

```

After entering the FILTER1 state, if a falling edge is detected before the 20ms count ends, it is considered that the key is still in the release jitter process, the state returns to DOWN and the counter is cleared to 0. During the release process, after the last jitter, there will be no falling edge, and the counter can count continuously. After the count is full, both key_flag and key_state are set to 1, the state enters IDLE, and the counter is cleared to 0, waiting for the next key press.

```

FILTER1:begin
    if(cnt_full)begin key_flag <=
        1'b1; key_state <= 1'b1; state
        <= IDLE; en_cnt <= 1'b0;
    end
    else if(key_in_nedge)begin en_cnt <= 1'b0;

```

```

        state <= DOWN;
    end else
        state <= FILTER1;
    end

```

If rewritten as a two-segment state machine, it is as follows (state transition part omitted).

```

always@(posedge clk or posedge reset) if(reset)
    curr_state <= IDLE;
else
    curr_state <= next_state;

always@(*) begin
    case(curr_state)
// This part is the same as one-segment state machine

    endcase
end

```

11.4 Testbench Design and Simulation Testing

In addition to generating normal clock and reset signals, the stimulus in the testbench file also simulates the process of key pressing and releasing. This simulation stimulus uses a separate design simulation model for the key. Thus, the internal structure of the entire stimulus file is shown in Figure 11-13 below:

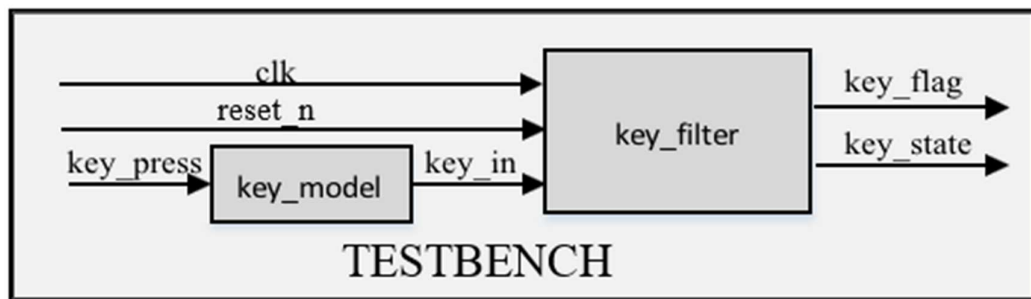


Figure 11-13 Internal Structure of Stimulus File

11.4.1 Key Model key_model Design

The function of the key model is to simulate the output of the signal during the real key process, including the jitter during the key process. Considering that the jitter time in the real key process is uncertain, the random number generation function \$random in Verilog is introduced here to generate jitter.

The \$random system function can generate a signed 32-bit random integer. The general usage is \$random%b, where b>0. This will generate a random number in the range of (-b+1):(b-1). If only positive random numbers are obtained, {\$random}%b can be used. In this section, it is necessary to generate press jitter and release jitter within 20ms. Actually, jitter within random numbers of 25'd20_000_000 should be generated. To save simulation time here, only a 16-bit random number is generated, that is, 0 to 16'd65535.

11.4.2 Use of Tasks

Task is a commonly used syntax in testbench. Its syntax is as follows:

```
task <task name>;  
  
<port and data type declaration statements>  
  
<statement 1>  
  
<statement 2>  
  
<statement n>  
  
endtask
```

The syntax for calling a task is as follows:

```
<task name> (port 1, port 2, ... port n);
```

For a key press and release process, it is designed in the form of a task. Here, 50 times of 0~65535ns press jitter are implemented, then key_in is assigned a fixed value of 0 and delayed for 25ms (greater than 20ms means stable). Also, after release jitter, key_in is assigned a fixed value of 1 and delayed for 25ms.

```
task key_gen; begin  
    key_out = 1'b1; repeat(50)begin  
        myrand = {$random}%65536;//0~65535; #myrand key_out  
        = ~key_out;  
    end  
    key_out = 0;  
    #25000000;  
  
    repeat(50)begin  
        myrand = {$random}%65536;//0~65535; #myrand key_out  
        = ~key_out;  
    end  
    key_out = 1;  
    #25000000;  
end endtask
```

Based on the above, to enable the key model key_model module to be controlled by an external key_press, i.e., giving key_press a pulse, key_model outputs a key signal once. Thus, in the key_model module, the arrival of the rising edge of key_press is used as a condition to call the above task. The specific code is as follows.

```
initial begin key_out = 1'b1;  
while(1)  
    begin  
        @(posedge key_press); key_gen;
```

```
end
```

```
end
```

After designing the key model `key_model` module, the design of the entire simulation testbench is relatively simple. Add and create a `key_filter_tb.v` file, instantiate the key debounce module and key model in it, and generate clock and reset stimulus signals. The specific code is as follows. The simulation simulates two key presses and releases.

```
`timescale 1ns/1ns
`define CLK_PERIOD 20 module

key_filter_tb;

    reg clk;
    reg reset_n; reg
    key_press; wire key;

    wire key_flag; wire key_state;

    key_model key_model_inst(
        .key_press(key_press),
        .key_out(key)
    );

    key_filter key_filter_inst(
        .clk(clk),
        .reset_n(reset_n),

        .key_in(key),
        .key_flag(key_flag),
        .key_state(key_state)
    );

    initial clk= 1; always#(`CLK_PERIOD/2) clk =
    ~clk;
    initial begin
        reset_n = 1'b0; key_press = 1'b0;
        #(`CLK_PERIOD*10);
        reset_n = 1'b1; #(`CLK_PERIOD*10 + 1);

        key_press = 1'b1; #(`CLK_PERIOD);
        key_press = 1'b0; #60000000;
        key_press = 1'b1; #(`CLK_PERIOD);
        key_press = 1'b0;
        #60000000;
        $stop;
    end
endmodule
```

After designing the simulation testbench, open Modelsim to create a new project for simulation. You can obtain the simulation waveforms shown in Figure 11-14 and Figure 11-15 below. Both press stability and release stability will generate a one-clock-cycle `key_flag` high-level signal, and

key_state will also change normally. Zooming in on the generated jitter process shows that each jitter time is different, thus successfully simulating the random jitter process.

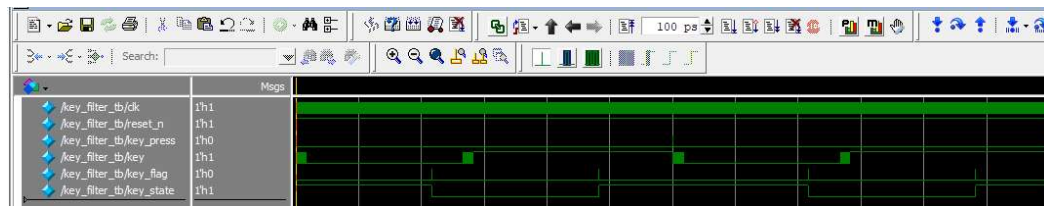


Figure 11-14 Key Debounce Model Simulation Waveform

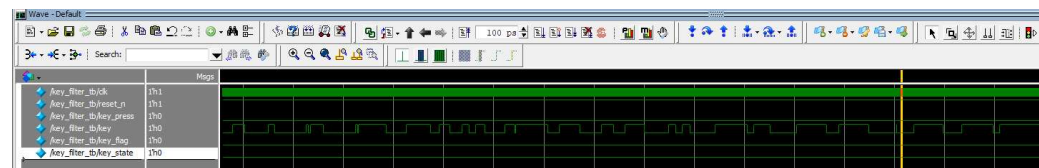


Figure 11-15 Random Jitter Process

From the waveform analysis above, it can be seen that the independent key debounce function designed in this chapter simulates normally.

11.5 Summary

This program plays a connecting role in all learning chapters. Since subsequent chapters mostly use direct referencing for this program, no pin constraints were performed for this program here. It is normal that synthesis cannot be completed. This program can complete the design task of this chapter if the correct waveform can be obtained using simulation tools.

In the design process of this chapter, single-bit asynchronous signal synchronization, edge detection, and state machine related designs were reviewed. Tasks and random functions (\$random) were introduced in the simulation process, and simulation models were used to simplify the writing of stimulus files. Specific board-level verification will be introduced in detail in the next chapter.

Chapter 12: Basics of Modular Design: Adder/Subtractor Counter

Project Source Code, |----02_Design Examples, |----ch12_key_led

Chapter Overview

In the design process of relatively larger projects, design content is usually not written in a single design file. Instead, different sub-files are designed for different functions, and finally instantiated and called in the top-level file.

In the previous chapter, the debounce of independent keys was designed and verified. Here, based on the key debounce module of the previous lecture, an adder/subtractor counter is implemented. Every time key 0 is pressed, the display state of 4 LEDs increases by 1 in binary addition format. Every time key 1 is pressed, the display state of 4 LEDs decreases by 1 in binary addition format. This is used to learn modular design methods. In Chapter 5, independent modules written in Chapter 4 will be combined to design some projects with strong applicability, further understanding the advantages of modular design methods.

12. 1Module Function Partitioning

To realize that two keys control the LED lights to add and subtract in binary counting mode, the top-level module key_led_top can be divided into two key debounce modules and one LED control module. The connection between modules is shown in Figure 12-1.

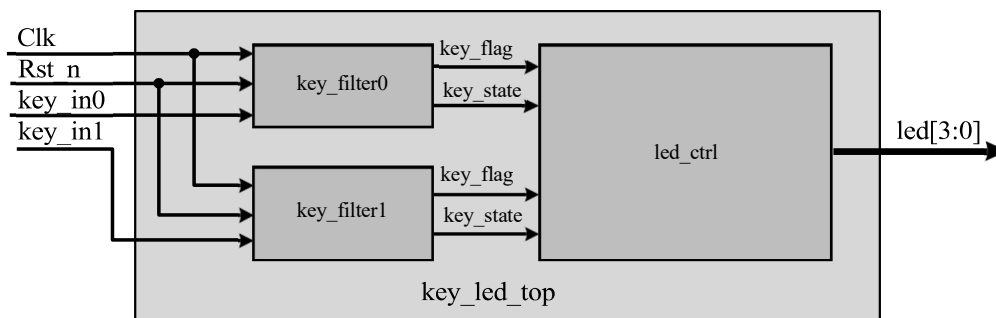


Figure 12-1 Top-Level Module Port Diagram

12.1.1 Module Function Design

The key_filter here has been written in the previous chapter, just call it in key_led_top. Now write led_ctrl. From the figure below, its port list can be derived as follows.

```
input Clk; input Rst_n;
input key_flag0, key_flag1; input
key_state0, key_state1;
output [3:0]led;
```

This module needs to increment or decrement the counter according to the state of the two keys. From the previous section, it can be seen that when key_flag && key_state is 1 (Note: Logic in

previous chapter says key_flag=1 and key_state=0 for PRESS. Please verify logic consistency with source code logic: key_flag0 && !key_state0), it means the key is pressed.

```
reg [3:0]led_r;
always@(posedge Clk or negedge Rst_n) if(!Rst_n)
    led_r <= 4'b0000;
else if(key_flag0 && !key_state0) led_r <= led_r
    + 1'b1;
else if(key_flag1 && !key_state1) led_r <= led_r
    - 1'b1;
else
    led_r <= led_r;
```

The initial value of the counter led_r is 4b0000. Here, when key 0 is pressed, the counter increments by one, and the counter becomes 4b0001. From the led light circuit diagram on the development board, it is known that the led light is lit by a high level. Directly output the value of led_r to led, and you can see the on/off effect of the led lights as off-off-off-on.

```
assign led = led_r;
```

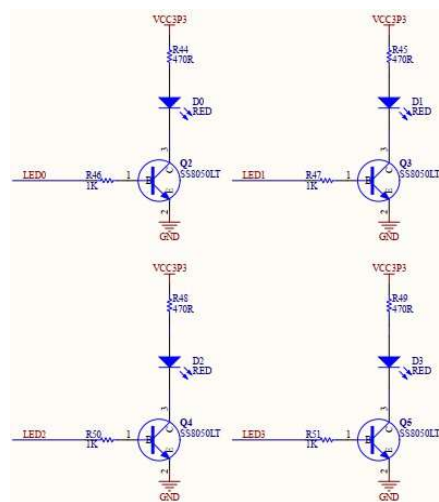


Figure 12-2 LED Light Circuit Diagram

In this way, each independent module is written. Now start the design of the top-level file. In the newly created key_led_top, instantiate three modules and set the internal signal type to wire.

```
module key_led_top(Clk,Rst_n,key_in0,key_in1,led);
    input Clk; input Rst_n;
    input key_in0; input
    key_in1;

    output [3:0]led;

    wire key_flag0,key_flag1; wire
    key_state0,key_state1;
```

```

key_filter key_filter0(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .key_in(key_in0),
    .key_flag(key_flag0),
    .key_state(key_state0)
);

key_filter key_filter1(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .key_in(key_in1),
    .key_flag(key_flag1),
    .key_state(key_state1)
);

led_ctrl led_ctrl0(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .key_flag0(key_flag0),
    .key_flag1(key_flag1),
    .key_state0(key_state0),
    .key_state1(key_state1),
    .led(led)
);

```

endmodule

Perform analysis and synthesis until there are no errors or warnings.

12.2 Stimulus Creation and Simulation Testing

To test the simulation, write a test stimulus file. Here, since two keys are called for simulation, the key simulation model designed in the previous lecture needs to be rewritten to add an enable signal press. That is, execute output key once when press rises. The task press_key part remains unchanged. If the called two simulation models are not rewritten, they will execute at the same time, leading to errors.

Create a new key_led_tb.v file, enter the following content, save it to the testbench folder, and perform analysis and synthesis again until there are no errors or warnings. The following content generates clock and reset signals, and also simulates the process of key 0 being pressed and released twice, and key 1 being pressed and released twice. The overall code is as follows.

```

`timescale 1ns/1ns

`define clk_period 20 module

key_led_tb;

reg Clk; reg Rst_n;
wire key_in0; wire

```

```

key_in1;
reg press0,press1; wire

[3:0]led;

key_led_top key_led_top(
    .Clk(Clk),
    .Rst_n(Rst_n),
    .key_in0(key_in0),
    .key_in1(key_in1),
    .led(led)
);

key_model key_model0(
    .press(press0),
    .key(key_in0)
);
key_model key_model1(
    .press(press1),
    .key(key_in1)
);
initial Clk= 1; always#(`clk_period/2) Clk =
~Clk;
initial begin
    Rst_n = 1'b0; press0 = 0;
    press1 = 0;
    #(`clk_period*10) Rst_n = 1'b1;
    #(`clk_period*10 + 1);

    press0 = 1; #(`clk_period*3)
    press0 = 0;

    #80_000_000;

    press0 = 1; #(`clk_period*3)
    press0 = 0;

    #80_000_000;

    press1 = 1; #(`clk_period*3)
    press1 = 0;

    #80_000_000;

    press1 = 1; #(`clk_period*3)
    press1 = 0;

    #80_000_000;
    $stop;
end

```

endmodule

After setting up the simulation script, perform functional simulation. You can see the waveform file shown in Figure 12-3. Whenever key 0 is pressed, the counter led_r increments by 1. After key 1 is pressed, the counter led_r decrements by 1.

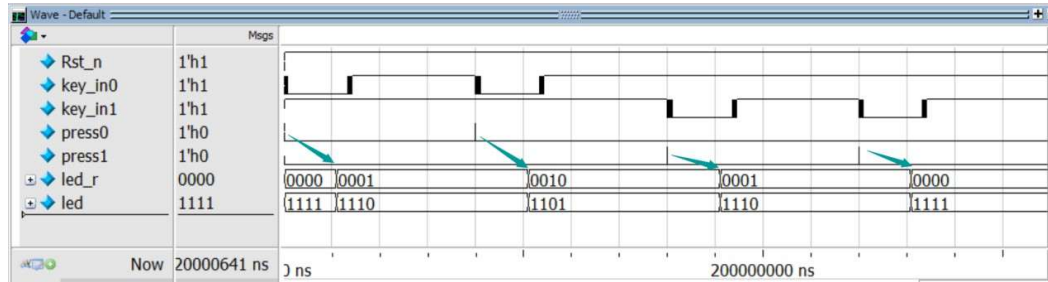


Figure 12-3 Functional Simulation Waveform

At this point, the functional simulation verification work is completed.

12.3 Board-Level Verification

The board-level verification part of this experiment mainly verifies the following goals:

1. Whether the program can be correctly programmed and downloaded.
2. On the ACG525 development board, after the corresponding key is pressed, can the led light up according to the design intent.

System Hardware Required:

3. ACG525 Development Board.
4. One power cable
5. A PC with complete development functions that meets the experiment requirements

12.3.1 I/O Constraints

We need to click FloorPlanner to perform I/O constraints. The specific constraints are as follows.

Table 12-1 I/O Constraint Table

Category	Functional Signal	ACG525 Development Board Pin
Basic Pin	clk	T9
Basic Pin	reset_n	C15
Basic Pin	key_in0	B16
Basic Pin	key_in1	A15

Category	Functional Signal	ACG525 Development Board Pin
Basic Pin	led[0]	D14
Basic Pin	led[1]	C14
Basic Pin	led[2]	B9
Basic Pin	led[3]	A9

After pin binding is completed, it is shown in Figure 12-4 below.

I/O Constraints								
	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type	Drive
1	Clk	input		T9	4	False	LVC MOS33	OFF
2	Rst_n	input		C15	1	False	LVC MOS33	OFF
3	key_in0	input		B16	1	False	LVC MOS33	OFF
4	key_in1	input		A15	1	False	LVC MOS33	OFF
5	led[0]	output		D14	1	False	LVC MOS33	8
6	led[1]	output		C14	1	False	LVC MOS33	8
7	led[2]	output		B9	0	False	LVC MOS33	8
8	led[3]	output		A9	0	False	LVC MOS33	8

Figure 12-4 Effect Diagram After Pin Binding

12.3.2 Place and Route

Click "Place & Route" again. Errors disappear, and the fs file is successfully generated, as shown in Figure 12-5 below.

```

Console
[70%] Routing Phase 2 completed
[80%] Routing Phase 2 completed
[90%] Routing Phase 3 completed
Running timing analysis.....
[95%] Timing analysis completed
Placement and routing completed
Bitstream generation in progress.....
Bitstream generation completed
Generate file "H:\01_gaoyun\gao_project\12_key_led\key_led\impl\pnr\key_led.pin.html" completed
Generate file "H:\01_gaoyun\gao_project\12_key_led\key_led\impl\pnr\key_led.rpt.html" completed
Generate file "H:\01_gaoyun\gao_project\12_key_led\key_led\impl\pnr\key_led.rpt.txt" completed
Generate file "H:\01_gaoyun\gao_project\12_key_led\key_led\impl\pnr\key_led.tr.html" completed
Wed Jun 28 10:30:02 2023
%
Console Message

```

Figure 12-5 Successfully Generating fs File

12.3.3 Downloading fs File and Observing Phenomena

Click "Program Device" to enter the download interface. The "Cable Setting" interface pops up, as shown in Figure 12-6 below. Click Save.

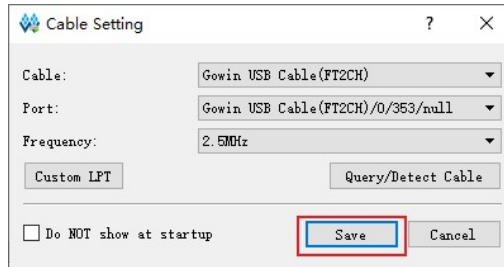


Figure 12-6 Schematic of Device Detection

Then click the icon  to download the fs file. After Programming, perform the key test shown in Figure 12-7. If it matches the design idea, the design is correct.

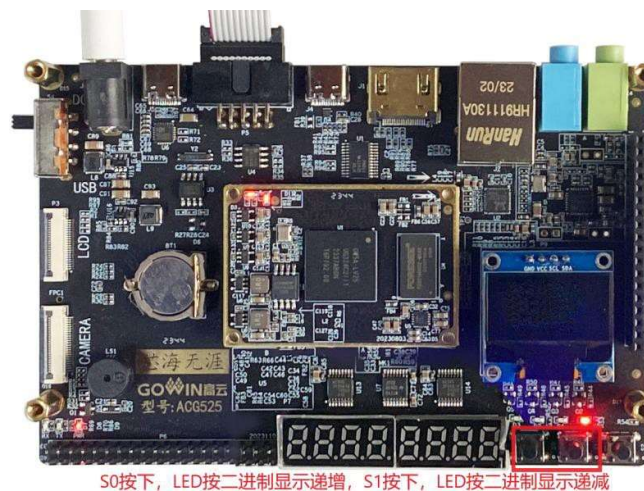


Figure 12-7 Board-Level Verification of Adder/Subtractor Counter

Seeing the phenomenon corresponding to the previous description means the design is correct. Thus, through simulation and board-level verification, it is shown that the adder/subtractor counter design is correct.

12.4 Common Problem Explanation

Key pressing, counter counting increment/decrement must be processed by key debounce. Otherwise, due to the mechanical characteristics of the key, an uncertain number of level transitions will be generated.

12.5 Summary

This chapter implemented the design of addition and subtraction through modular design, and performed behavioral simulation and board-level verification. Modular design methods are widely used in subsequent chapters, especially in comprehensive experiments, where the advantages of this design method can be better appreciated. It is recommended that readers follow the content of this experiment to complete the entire experiment.

Chapter 13: 8-Bit 7-Segment Digital Tube Driver Design and Verification

Project Source Code, |----02_Design Examples, |----ch13_hex8_hc595

Chapter Overview

Common display devices in electronic systems include digital tubes (7-segment displays), LCDs, and VGA monitors. Digital tubes can be divided into segment displays (7-segment, starburst, etc.) and dot matrix displays (88, 1616, etc.). LCD applications can be divided into character LCDs (1602, 12864, etc.) and true color LCD screens. VGA monitors are generally current computer monitors. The ACG525 development board provides hardware interfaces for all three types of devices.

This chapter will implement FPGA driving digital tubes for dynamic display, extract the implemented circuit structure, write code starting from the circuit structure, and verify the design through simulation. The core of this lesson is no longer the code, but the circuit structure. Once the circuit structure is determined, writing the code is just a process of construction according to the plan. This is also the hardware thinking that is getting closer to the essence of FPGA design.

13.1 Digital Tube Driver Principle

The structure diagram of an 8-segment digital tube is shown in Figure 13-1 below.

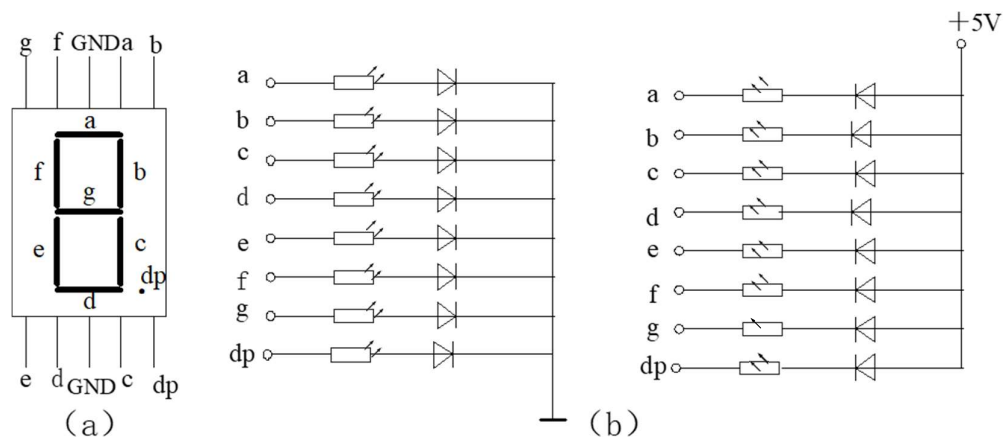


Figure 13-1 8-Segment Digital Tube Structure Diagram

As can be seen from the figure above, there are two structures of digital tubes: Common Cathode and Common Anode. The difference between the two is whether the common terminal is connected to ground or high level. For common cathode digital tubes, a high level needs to be given to the corresponding segment to light it up, while for common anode digital tubes, a low level needs to be given to light it up. The digital tube on the development board is a common anode digital tube.

At the same time, in order to display numbers or characters, the numbers or characters must be encoded and decoded. Here, without considering the decimal point first, it is simplified to a 7-segment digital tube. Its encoding and decoding format is shown in the following table:

Table 13-1 Digital Tube Encoding and Decoding Table

Data_disp	Segment Code (Binary Format)								Segment Code (Hexadecimal Format)
	a	b	c	d	e	f	g	h	
0	0	0	0	0	0	0	1	1	8'hc0
1	1	0	0	1	1	1	1	1	8'hf9
2	0	0	1	0	0	1	0	1	8'ha4
3	0	0	0	0	1	1	0	1	8'hb0
4	1	0	0	1	1	0	0	1	8'h99
5	0	1	0	0	1	0	0	1	8'h92
6	0	1	0	0	0	0	0	1	8'h82
7	0	0	0	1	1	1	1	1	8'hf8
8	0	0	0	0	0	0	0	1	8'h80
9	0	0	0	0	1	0	0	1	8'h90
a	0	0	0	1	0	0	0	1	8'h88
b	1	1	0	0	0	0	0	1	8'h83
c	0	1	1	0	0	0	1	1	8'hc6
d	1	0	0	0	0	1	0	1	8'ha1
e	0	1	1	0	0	0	0	1	8'h86
f	0	1	1	1	0	0	0	1	8'h8e

There are two working modes for segment digital tubes: static display mode and dynamic display mode. The characteristic of static display is that the segment selection of each digital tube must be connected to an 8-bit data line to maintain the displayed character code. Once the character code is sent, the displayed character can be maintained until a new character code is sent. This method requires independent data lines for each digital tube, so the hardware circuit is relatively complex, the cost is high, and it is rarely used.

To save IOs and costs, the circuit structure shown in Figure 13-2 below is generally used. In this way, connecting 3 digital tubes together saves 7*2 I/Os compared to the static method.

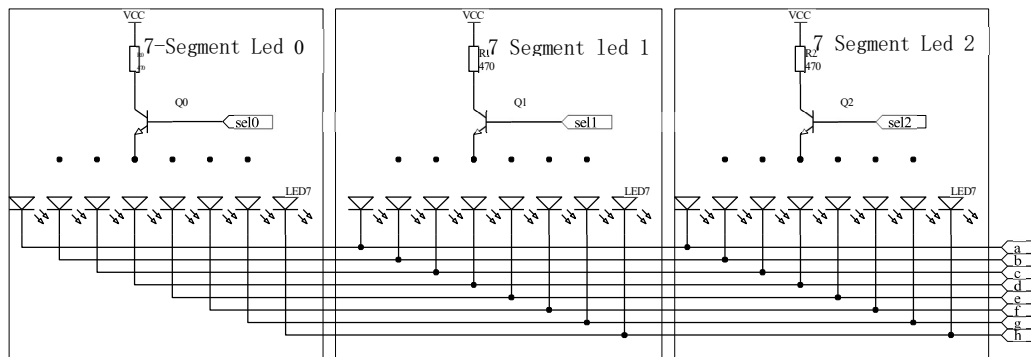


Figure 13-2 Equivalent Circuit Diagram of Three-Digit Digital Tube

This realizes another display mode, dynamic display. The characteristic of dynamic display is that the segment selection lines of all digital tubes are connected in parallel, and the bit selection line controls which digital tube is valid. The digital tube to be lit is displayed by dynamic scanning. The so-called dynamic scanning display means sending out the character code and the corresponding bit selection to each digital tube in turn. Utilizing the afterglow of the light-emitting diode and the visual persistence effect of the human eye, it feels as if all digital tubes are displaying at the same time.

Now for example, assume the scanning time is set to 1S. These three digital tubes are divided into 3s. In the 1st second, the sel data line is b100, at this time digital tube 0 is selected. If a=0, LED0 of digital tube 0 can be lit; In the 2nd second, the sel data line is b010, at this time digital tube 1 is selected. If b=0, LED1 of digital tube 1 can be lit; In the 3rd second, the sel data line is `b001, at this time digital tube 2 is selected. If c=0, LED2 of digital tube 2 can be lit. The effect at this time will be that LED0 of digital tube 0 lights up for one second, then LED1 of digital tube 1

lights up for one second, and finally LED2 of digital tube 2 lights up for one second, cycling like this again.

In this way, if a 1ms refresh time is used, due to the afterglow effect of the digital tube and the visual persistence of humans, LED0 of digital tube 0, LED1 of digital tube 1, and LED2 of digital tube 2 will appear to light up "simultaneously" without any flickering sensation.

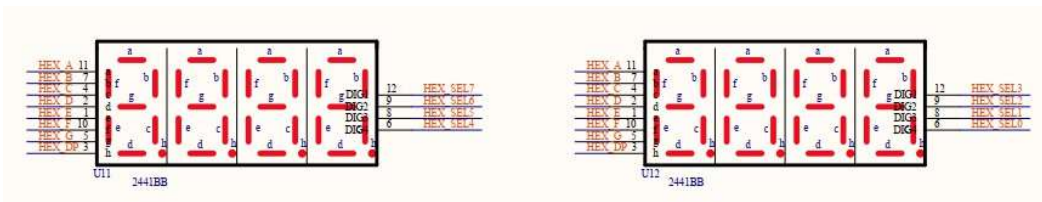


Figure 13-3 Schematic of Two 7-Segment 4-Digit Digital Tubes Cascaded

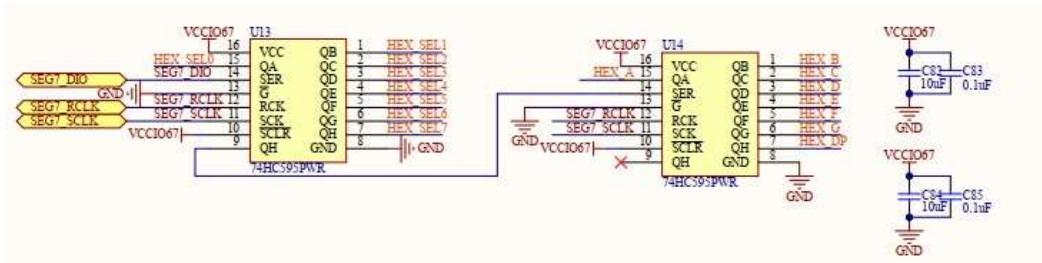


Figure 13-4 Schematic of 7-Segment 8-Digit Digital Tube Using 74HC595 Driver Scheme

13.2 Digital Tube Dynamic Scanning Driver Design

13.2.1 Module Interface Design and Internal Function Partitioning

From the above analysis, the input/output block diagram shown in Figure 13-5 below can be derived. Its interface list is shown in the table below:

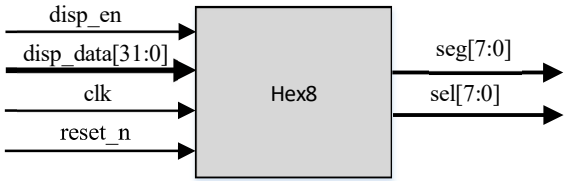


Figure 13-5 Digital Tube Module Block Diagram

Table 13-2 Module Interface List

Signal Name	I/O	Function Description
clk	I	Module working clock, 50 MHz
reset_n	I	Module reset signal
disp_en	I	Seven-segment display enable signal; 1 = enable, 0 = disable
disp_data[31:0]	I	Display data for 8 seven-segment digits; each 4-bit nibble represents one BCD code
sel[7:0]	O	Digit select signals; selects the currently active seven-segment digit
seg[7:0]	O	Segment control signals; determines the segments to be illuminated for the current digit

According to the above analysis, first there must be a driving clock with a period of 1ms, so a frequency division circuit is needed; when performing digital tube bit selection, a circular shift is needed; after selecting the bit, a selector is needed to gate the data input bit; to implement the function of the digital tube encoding/decoding table, a decoder is needed.

The logic circuit diagram of the digital tube driver module can be simplified as shown in Figure 13-6 below. The function of each part is shown in Table 13-3 below.

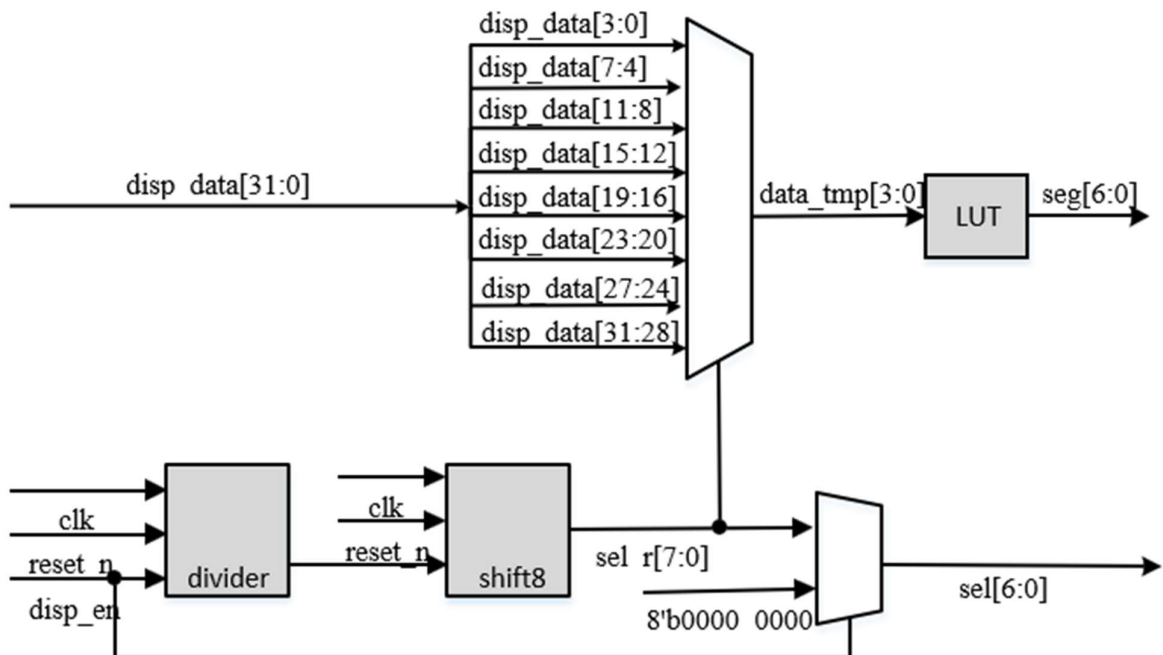


Figure 13-6 Digital Tube Driver Module Logic Circuit Diagram

Table 13-3 Sub-module Function Description

0	1	0	0	0	0	0	0	Led7 Segment 1
0	0	1	0	0	0	0	0	Led7 Segment 2
.....								
0	0	0	0	0	0	0	1	Led7 Segment 7

The code is as follows:

```
always@(posedge clk_1K or posedge reset)
if(reset)
    sel_r <= 8'b0000_0001; else if(sel_r
== 8'b1000_0000)
    sel_r <= 8'b0000_0001; else
    sel_r <= sel_r << 1;
```

13.2.4 Digital Tube Data Display Design

Use an 8-to-1 multiplexer. The selection terminal is the currently scanned digital tube, i.e., the output terminal of the circular shift register. Use the multiplexer to send the data to be displayed to the corresponding digital tube.

```
reg [3:0]data_tmp;//lưu trữ data always@(*)
case(sel_r)
    8'b0000_0001:data_tmp = disp_data[3:0]; 8'b0000_0010:data_tmp =
disp_data[7:4]; 8'b0000_0100:data_tmp = disp_data[11:8];
    8'b0000_1000:data_tmp = disp_data[15:12]; 8'b0001_0000:data_tmp =
disp_data[19:16]; 8'b0010_0000:data_tmp = disp_data[23:20];
    8'b0100_0000:data_tmp = disp_data[27:24]; 8'b1000_0000:data_tmp =
disp_data[31:28]; default:data_tmp = 4'b0000;
Endcase
```

13.2.5 Display Data Decoding Design

As mentioned earlier, to make the digital tube display numbers or characters, the numbers or characters must be encoded and decoded. Here, a 4-input look-up table is used to implement 8-bit output display decoding.

```
always@(*)
case(data_tmp)
    4'h0:seg = 7'b1000000;
    4'h1:seg = 7'b1111001;
    4'h2:seg = 7'b0100100;
    4'h3:seg = 7'b0110000;
    4'h4:seg = 7'b0011001;
    4'h5:seg = 7'b0010010;
    4'h6:seg = 7'b0000010;
    4'h7:seg = 7'b1111000;
    4'h8:seg = 7'b0000000;
    4'h9:seg = 7'b0010000; 4'ha:seg =
7'b0001000; 4'hb:seg = 7'b0000011;
    4'hc:seg = 7'b1000110; 4'hd:seg =
7'b0100001;
```

```

4'he:seg = 7'b0000110;
4'hf:seg = 7'b0001110;
endcase

```

13.2.6 Module Enable Design

The concept of modular design is to make each module independent, and its port design should facilitate future calling and control. Based on this concept, an enable signal needs to be added here. Regarding the enable sub-module, it can be implemented directly using a 2-to-1 multiplexer.

```
assign sel = (en)?sel_r:8'b0000_0000;
```

13.3 74HC595 Driver Module Design

The digital tube design on the development board uses the chip 74HC595. The function of this chip is a shift register, which saves FPGA pins through shifting. The FPGA only needs to output 3 pins to achieve the purpose of sending digital tube data. Compared with traditional segment selection and bit selection methods, it greatly saves IO design resources.

The specific usage method and technical parameters of 74HC595 are as follows: It can be seen in the data sheet that the chip operating frequency values of 74HC595 are different under different operating temperatures and operating voltages, as shown in Table 13-5. Since the chip is powered by 3.3V on the learning board, when designing its operating frequency, the 50M crystal oscillator clock divided by four is directly used as its operating clock, allowing it to work at 12.5M frequency under 3.3V status.

Table 13-5 74HC595 Chip Design Frequency Parameter Lower Limit

SYMBOL	PARAMETER	Vcc(V)	MIN	TYP	MAX	UNIT
f_{max}	maximum clock frequency SH_CP and ST_CP	2.0	4.8	-	-	MHz
		4.5	24	-	-	MHz
		6.0	28	-	-	MHz
T _{amb} = -40to + 125°C						

The driver module ports of 74HC595 are shown in Figure 13-7, and its interface list is shown in Table 13-6.

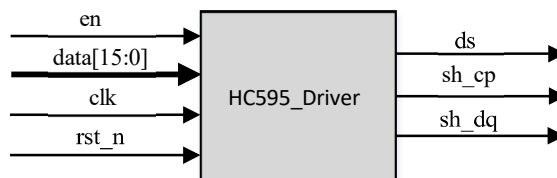


Figure 13-7 74HC595 Driver Input and Output Ports

Table 13-6 74HC595 Driver Input and Output Port Function Description

Signal Name	I/O	Function Description
clk	I	50 MHz system clock

Signal Name	I/O	Function Description
rst_n	I	Active-low reset signal
en	I	Seven-segment display enable signal; 1 = enable, 0 = disable
data[15:0]	I	SEG and SEL control data for 8 seven-segment digits
ds	O	Serial data output
sh_cp	O	Shift register clock output
st_cp	O	Storage (latch) register

In the 74HC595 driver module design, first design the operating clock generation, dividing the 50M clock by 2.

```
parameter CNT_MAX = 2;

reg [15:0]r_data; always@(posedge
clk) if(s_en)
    r_data <= data;

reg [7:0]divider_cnt;//分频计数器;

always@(posedge clk or posedge reset) if(reset)
    divider_cnt <= 0;
else if(divider_cnt == CNT_MAX - 1'b1) divider_cnt <=
    0;
else
    divider_cnt <= divider_cnt + 1'b1;

wire sck_plus;
assign sck_plus = (divider_cnt == CNT_MAX - 1'b1);
```

Count sck_plus to be used in the look-up table to implement serial data input and the generation of shift clock sh_cp and storage clock st_cp.

```
reg [5:0]SHCP_EDGE_CNT;

always@(posedge clk or posedge reset) if(reset)
    SHCP_EDGE_CNT <= 0;
else if(sck_plus)begin if(SHCP_EDGE_CNT ==
    6'd32)
    SHCP_EDGE_CNT <= 0;
else
    SHCP_EDGE_CNT <= SHCP_EDGE_CNT + 1'b1;
```

```

end else
    SHCP_EDGE_CNT <= SHCP_EDGE_CNT;

```

Look-up table implements state output.

```

always@(posedge clk or posedge reset)
if(reset)begin
    st_cp <= 1'b0; ds <=
    1'b0; sh_cp <= 1'd0;
else begin
    case(SHCP_EDGE_CNT)
        0: begin sh_cp <= 0; st_cp <= 1'd0; ds <= r_data[15];end
        1: begin sh_cp <= 1; st_cp <= 1'd0;end
        2: begin sh_cp <= 0; ds <= r_data[14];end
        3: begin sh_cp <= 1; end
        4: begin sh_cp <= 0; ds <= r_data[13];end
        5: begin sh_cp <= 1; end
        6: begin sh_cp <= 0; ds <= r_data[12];end
        7: begin sh_cp <= 1; end
        8: begin sh_cp <= 0; ds <= r_data[11];end
        9: begin sh_cp <= 1; end
        10: begin sh_cp <= 0; ds <= r_data[10];end
        11: begin sh_cp <= 1; end
        12: begin sh_cp <= 0; ds <= r_data[9];end
        13: begin sh_cp <= 1; end
        14: begin sh_cp <= 0; ds <= r_data[8];end
        15: begin sh_cp <= 1; end
        16: begin sh_cp <= 0; ds <= r_data[7];end
        17: begin sh_cp <= 1; end
        18: begin sh_cp <= 0; ds <= r_data[6];end
        19: begin sh_cp <= 1; end
        20: begin sh_cp <= 0; ds <= r_data[5];end
        21: begin sh_cp <= 1; end
        22: begin sh_cp <= 0; ds <= r_data[4];end
        23: begin sh_cp <= 1; end
        24: begin sh_cp <= 0; ds <= r_data[3];end
        25: begin sh_cp <= 1; end
        26: begin sh_cp <= 0; ds <= r_data[2];end
        27: begin sh_cp <= 1; end
        28: begin sh_cp <= 0; ds <= r_data[1];end
        29: begin sh_cp <= 1; end
        30: begin sh_cp <= 0; ds <= r_data[0];end
        31: begin sh_cp <= 1; end
        32: st_cp <= 1'd1;
        default:
            begin
                st_cp <= 1'b0;
                ds <= 1'b0;
                sh_cp <= 1'd0;
            end
    endcase
end

```

```

end
endcase end

```

13.4 Digital Tube Display Module Simulation Testing

The following generates reset signal, enable signal, initialization and switching of data to be displayed, displaying "12345678", "87654321", and "89abcdef" on the digital tube respectively.

```

initial begin
    reset_n = 1'b0;
    //en = 1;
    disp_data = 32'h12345678;
    #(`clk_period*20); reset_n = 1;
    #(`clk_period*20); #20000000;
    disp_data = 32'h87654321; #20000000;
    disp_data = 32'h89abcdef; #20000000;
    $stop;
end

```

After designing the digital tube driver and the corresponding simulation file, open Modelsim to create a new simulation project for functional simulation. You can see the global view of the simulation test waveform shown in Figure 13-8 below. The partial waveform is zoomed in as shown in Figure 13-9 below (switched to binary display for better observation). It can be seen that before the reset signal is set high, the digital tube displays 0. After the reset ends, the digital tube starts to display normally. And when the data to be displayed is 89ABCDEFh (MSB), the digital tubes from 1 to 8 are sequentially gated and display FEDCBA98h (LSB) respectively. Simulation passed.

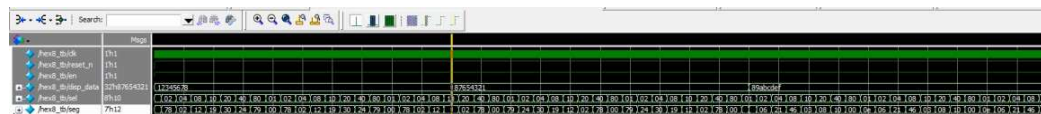


Figure 13-8 Simulation Test Waveform Global View



Figure 13-9 Partial Waveform View

13.5 Board-Level Debugging and Verification

Add and create a new top-level file hex_data.v, set it as the top-level design, and instantiate the written digital tube driver module into the top-level file to make the digital tube display "01234567". The code is as follows:

```

module hex_top(
    clk, reset_n, sh_cp,

```

```

    st_cp, ds
);

input clk; //50M input reset_n;

output sh_cp; output
st_cp; output ds;

wire [31:0]disp_data;
wire [7:0] sel;//chọn vị trí của Led7đoạn)
wire [6:0] seg;//Chọn đoạn trên Led để hiển thị nội dung)

assign disp_data = 32'h01234567; hc595_driver

hc595_driver(
    .clk(clk),
    .reset_n(reset_n),
    .data({1'd0,seg,sel}),
    .s_en(1'b1),
    .sh_cp(sh_cp),
    .st_cp(st_cp),
    .ds(ds)
);

hex8 hex8(
    .clk(clk),
    .reset_n(reset_n),
    .en(1'b1),
    .disp_data(disp_data),
    .sel(sel),
    .seg(seg)
);

endmodule

```

Analyze and synthesize the top layer until there are no errors and warnings. Then click FloorPlanner for I/O constraints. The specific I/O constraints are shown in Figure 13-10 below.

I/O Constraints							
	Port	Direction	Diff Pair	Location	Bank	Exclusive	IO Type
1	clk	input		T9	4	False	LVC MOS33
2	ds	output		H4	7	False	LVC MOS33
3	reset_n	input		B16	1	False	LVC MOS33
4	sh_cp	output		F4	7	False	LVC MOS33
5	st_cp	output		F3	7	False	LVC MOS33

Figure 13-10 I/O Constraint Diagram

Then click "Place & Route" to layout and route. If there are no errors, the fs file for this experiment will be generated. Click Program Device to download. After the download is complete, the digital tube will display as shown in Figure 13-11.



Figure 13-11 Digital Tube Displaying Data

As can be seen from the above figure, the "." after each number is lit. This is because when instantiating the "hc595_driver" module in the top layer, the highest bit of data is 0. If you want it not to be lit, just set this bit to 1. At this time, the digital tube will display as shown in Figure 13-12.



Figure 13-12 Digital Tube Displaying Data ("." Not Lit)

At this point, the dynamic display of the digital tube is completed.

13.6 Common Problem Explanation

1. In this experiment, the development board uses the onboard 74HC595 three-wire driver mode. Using this mode can greatly save IO port resource usage.
2. In this case, in the HEX8 module, a counter is used to divide the frequency, dividing the 50MHZ clock to generate a 1KHz low-frequency clock, and then this divided clock signal is used directly to drive the D flip-flop. This clock signal does not go through the global clock path by default, so there will be large clock jitter and skew. Another disadvantage of using this clock is that the delay to each D flip-flop varies greatly, which will affect the stability of the system. This clock signal is called a gated clock. Later in the serial port transmit module, we will use the serial port baud rate clock signal as an example to demonstrate the design and usage of high-quality clock signals in FPGA systems (spoiler alert: this clock is called an enable clock).

13.7 Summary

This chapter introduced the following driving methods for digital tubes: static scanning, dynamic scanning, and SPI driving mode. Benefiting from the advantages of modular design, if you need to realize FPGA directly driving the digital tube, you only need to call the hex8 and drive hc595 files directly. This achieves the effect of saving pins through shift registers while retaining the architecture that can use segment selection and bit selection parallel output at any time.

It is recommended that readers follow the content of this experiment to complete the entire experiment

