



FPGA Made Easy

Start with Kiwi board and Gowin EDA tool

Advisors: Duong Van Thanh
Truong Ngoc Phu
David Trieu
Hoang Duc Hieu
Authors: Dinh Xuan Quang

HO CHI MINH CITY, JULY 2025

Contents

1	Preface	5
2	Creating project led blink with Gowin EDA tool	6
2.1	Counters for sequential logic circuit design	8
2.2	Creating verilog file	9
2.2.1	Synthesize	11
2.2.2	Hierarchy	12
2.3	Physic constraints	13
2.3.1	FloorPanner	13
2.4	Place and Route	15
2.5	Board-level validation	16
3	Gowin and Modelsim simulation experiments	18
3.1	Modelsim compiles the Gowin device library	18
3.1.1	Create a new Modelsim library	18
3.1.2	Compile Gowin's device library files	19
3.1.3	Add a library to the Modelsim default library list	21
3.2	Modelsim simulation validation process	22
3.2.1	Write Verilog simulation design code	22
3.2.2	Create a Modelsim project and add simulation files	23
3.2.3	Compile the simulation file	24
3.2.4	Configure the simulation environment	25
3.2.5	Simulation interface waveform observation	27
3.2.6	Save the waveform file	28
3.2.7	Reopen the simulation project	29
4	PWM (Pulse Width Modulation) to control LED brightness	31
4.1	Principle of PWM	31
4.2	PWM Generator Module Design	32
4.3	Simulation	34
4.4	Board-Level Debugging and Verification	36
5	Key filtering	39
5.1	Key physical structure and circuit design	39
5.2	Key physical structure and circuit design	39
5.3	The state machine implements key debounce	41
5.3.1	Module interface design	41

5.3.2	State machine transfer state as well as conditional design	41
5.3.3	Key input signal edge detection	42
5.3.4	Counter design	43
5.3.5	State machine design	43
5.4	Testbench design and simulation testing	46
5.4.1	The design of the key model key_model	46
5.4.2	Use of tasks	47
5.5	Design key control counter led	50
6	UART_TX Design and verification of serial port transmitter module	53
6.1	UART key parameters and time series diagrams	53
6.2	Design and implementation of UART asynchronous serial communication transmis- sion module	54
6.3	Baurate clock module design	55
6.4	Data output module design	57
6.5	TESTBENCH code for modelsim	58
6.6	Coding test for real device	59
6.7	Simulation	60
6.8	Testing with board	61
6.8.1	GWU2U Virtual Serial Port setting	61
6.8.2	Using hercules software	62
7	UART_RX Design and verification of serial port receiver module	64
7.1	Analysis of the principle of serial port reception	64
7.2	Design and implementation of UART asynchronous serial communication receiving module	64
7.2.1	Serial port receiving module interface design	64
7.2.2	Single-bit asynchronous signal synchronization design	65
7.3	Edge detection	66
7.4	Design of the sample clock generation module	66
7.5	Design of sampling data receiving module	68
7.6	Data status determination module	70
7.7	TESTBENCHfor modelsim	70
7.8	Code for testing real device	72
7.9	Simulation	73
7.10	Testing with board	74

8	UART_loopback Design and verification of multi-byte serial transceiver module	75
8.1	Principle and design of multi-byte serial port transmission	75
8.1.1	The principle of multi-byte serial port transmission	75
8.2	Design of multi-byte serial port transmitting module	76
8.3	Principle and design of multi-byte serial port reception	80
8.3.1	Multi-byte serial port receiving principle	80
8.4	Create a top-level package for testing	85
8.5	Testing with board	86
9	References	88

1 Preface

This book is designed as a practical and educational guide for students, engineers, and makers who are beginning their journey into digital circuit design and FPGA development. Centered around the **Onekiwi 1P5 FPGA development board**, the content is tailored for hands-on exploration of hardware modules using Verilog HDL and the Gowin EDA tools.

As embedded systems and programmable logic continue to shape the future of electronics, it is essential to equip learners with both theoretical foundations and real-world implementation skills. Through structured chapters, each focused on a specific module—such as counters, UART communication, video interfaces, memory access, and display drivers—this book bridges the gap between learning and doing.

This is the **first edition** of the book, and while it lays the groundwork with a range of fundamental and advanced projects, it is also a work that will evolve based on feedback, technological changes, and the growing needs of the FPGA education community.

The FPGA 1P5 board was chosen for its accessibility, affordability, and robust support for industry-standard design workflows. All designs in this book can be simulated, tested, and deployed using the Gowin IDE and supporting tools like ModelSim, making it ideal for both classroom and self-guided learning.

I sincerely hope this book serves as a useful companion on your FPGA development journey, and I welcome all comments and contributions for future improvements.

Ho Chi Minh City, July2025

Author,

Dinh Xuan Quang

FPGA engineer at Onekiwi semiconductor

2 Creating project led blink with Gowin EDA tool

- Double-click the Gowin icon to launch the Gowin software.



Figure 1: Gowin EDA tool

- The startup screen in Gowin is divided into 4 sections, as follows:
 - (1) Recent projects: Processes that have been opened recently.
 - (2) Quick Start: Quick Start, including New Project, Open Project, Open Example Project
 - (3) Tools: Tools, including FloorPlanner, Timing Constrains Editor, Programmer (data flow configuration or download)
 - (4) User Manuals: User manual, click to enter the product introduction of the official website of Gowin.

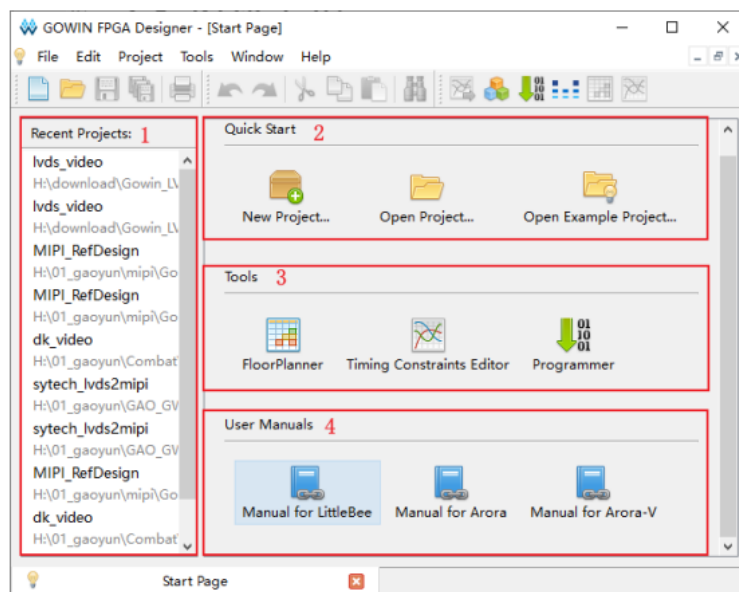


Figure 2: Gowin EDA tool

- Click the New Project,the New dialog box will appear, as shown in Figure 3 below, click the OK button.

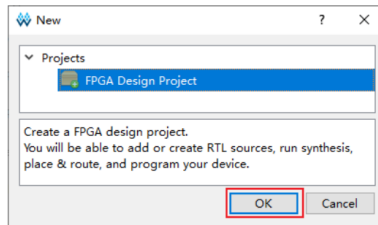


Figure 3: *Creating project interface*

- The Gowin wizard will appear,fill the project name and project path,then click to next.

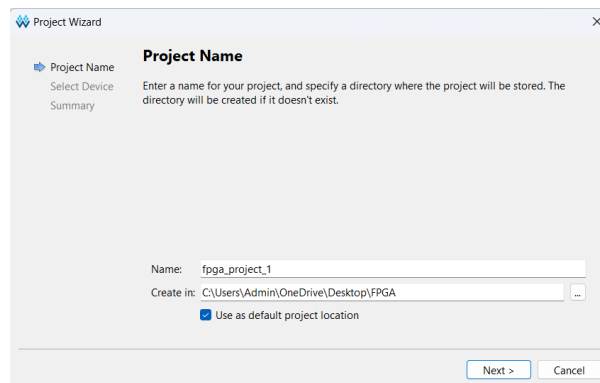


Figure 4: *Filling name and path project*

- The Select Device box will appear,you can find your specific FPGA chip model used in design base on series,package,device,speed,device version.Then choose your chip and click OK.Our board is :GW1N-UV1P5QN48XFC7/I6.

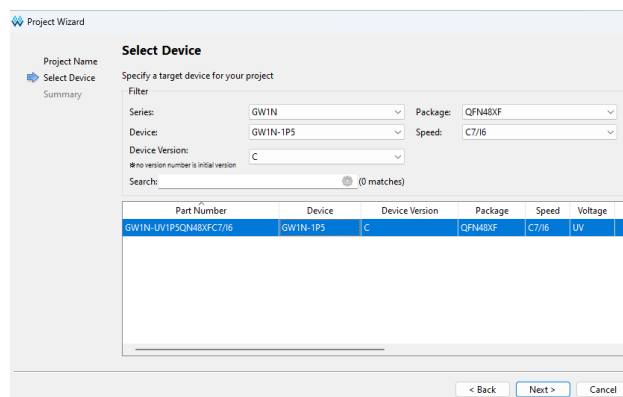


Figure 5: *Select your device*

- The summary of your chip information will appear for you to check it, then click back if you want to change your chip or finish.

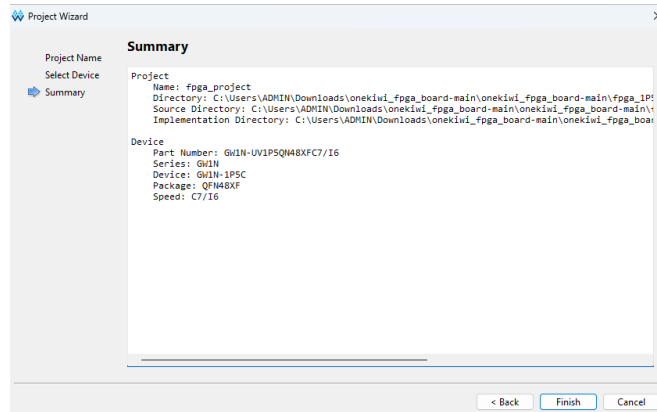


Figure 6: Summary information

- Next, you will see the Gowin design interface.

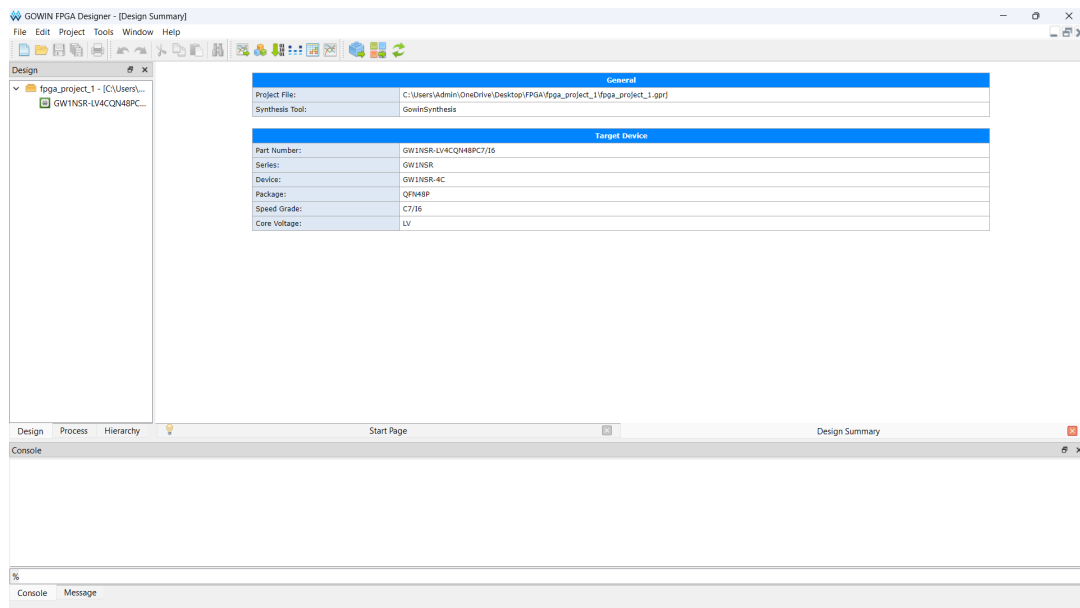


Figure 7: Gowin design interface

2.1 Counters for sequential logic circuit design

A counter is a digital circuit whose core component is the flip-flop, used to count input pulses. The maximum number of pulses it can store is called its modulus. Counters are essential in applications like frequency division and timing, and they come in various types based on their counting method (binary, decimal, or arbitrary base), clock source (synchronous or asynchronous), and

counting direction (up, down, or reversible). In Verilog HDL, which is a hardware description language used to design actual circuits rather than write software code, counters are implemented using elements like adders, comparators, registers, and selectors. An example application is using a counter to control LED flashing.

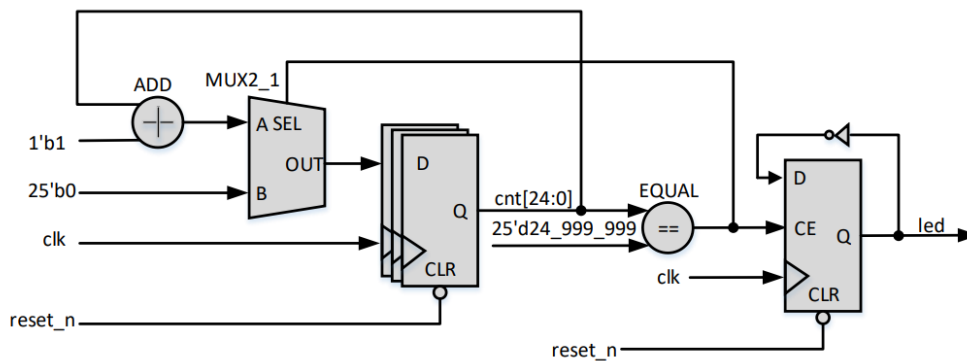


Figure 8: The counter controls the LED logic circuit diagram

From the above analysis, the schematic diagram of the module interface of this design can be obtained, as shown in Figure below:

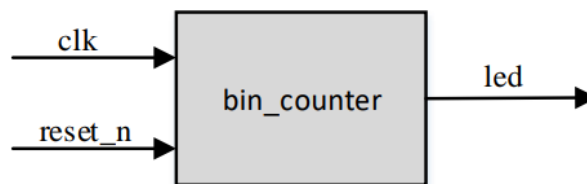


Figure 9: Schematic diagram of the counter module

The functions of each port are described in Table below:

The name of the port	I/O	Description of the port function
clk	I	The module's operating clock is 27MHz,
reset_n	I	Module reset, low level reset
led	O	LED control terminal, the LED light is on when the output is low

Table 1: Table of binary counter description module

2.2 Creating verilog file

Then you will create verilog file by clicking on the toolbar or click File->New,choose Verilog File,then click OK.

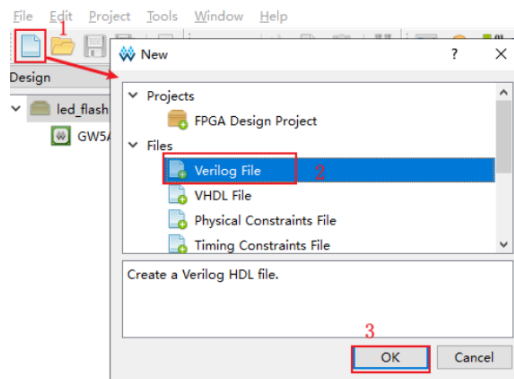


Figure 10: Add the Verilog source file

Fill the name and path of the verilog file then click OK.

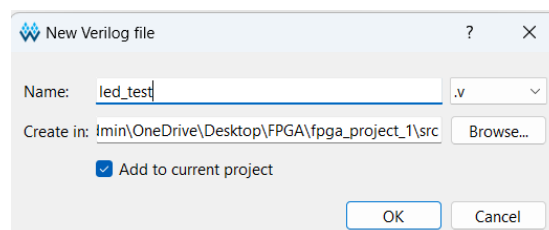


Figure 11: Fill name and path of verilog file

After that, we can see the new folder Verilog Files include file verilog we just create.

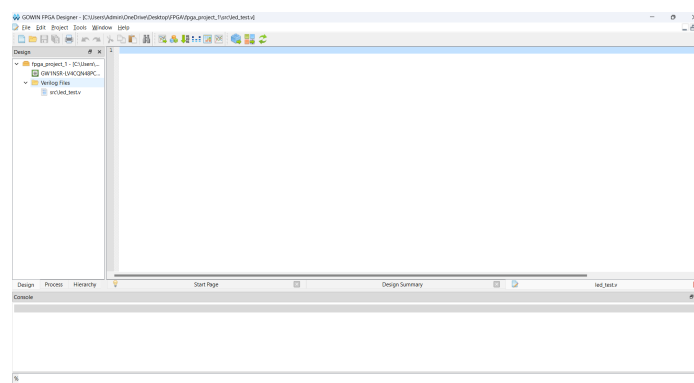


Figure 12: Verilog design interfaces

Coding example:

Listing 1: LED blinking after 1s delay example

```
module led_flash(  
    input clk,  
    input rst_n,
```

```
output reg led,
output reg led1
);

reg [25:0] cnt;

always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        cnt <= 26'd0;
    else if (cnt < 26'd26_999_999) // Assuming a 27 MHz clock, this counts ~1 second
        cnt <= cnt + 1'b1;
    else
        cnt <= 26'd0;
end

always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n) begin
        led <= 1'b0;
        led1 <= 1'b1; // LED1 starts OFF, LED2 starts ON
    end

    else if (cnt == 26'd26_999_999) begin
        led <= ~led;
        led1 <= ~led1; // Toggle LED1 opposite to LED
    end

end

endmodule
```

2.2.1 Synthesize

Using ctrl+S or click file on taskbar and choose save your verilog file. Then click to process like the figure below. Next double-click to Synthesize or click icon run synthesis on the taskbar. If your code has errors there will be icon red on the synthesis and system will show your message error below, else there will be green icon in the synthesis.

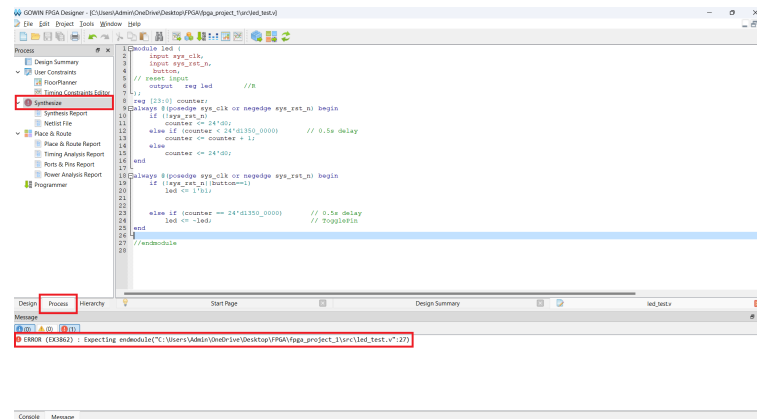


Figure 13: Synthesis process

Consulting:the tool just check syntax errors, if your design have logical errors,system can not show message errors, for example: incorrect connections between modules.However,you are provided provided with the function of viewing RTL design analysis, click Tools->Schematic Viewer->RTL Design Viewer in turn, and then you can see the RTL description and analysis stage.

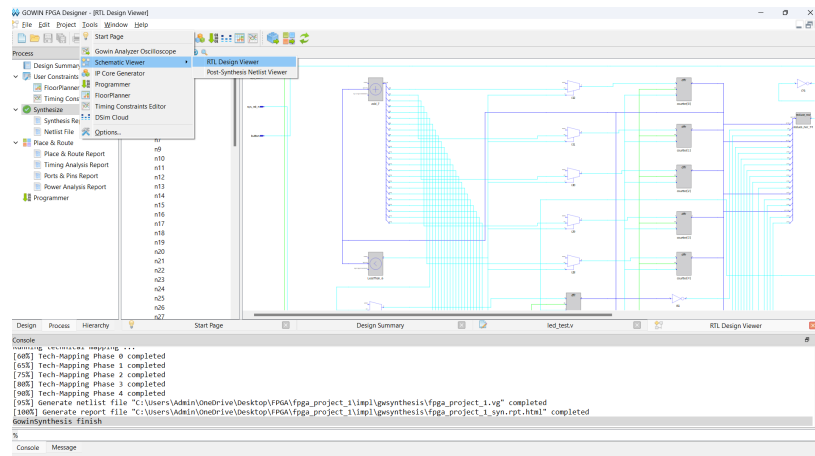


Figure 14: RTL Design Viewer

2.2.2 Hierarchy

You can choose Hierarchy near process to see information about your module include: Registers,LUT,ALU,BSRAM,SSRAM

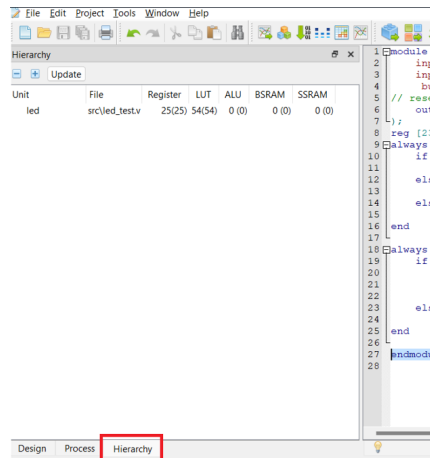


Figure 15: *Hierarchy information*

In the hierarchy window, we place the mouse over the verilog file, right-click on the file, and set the file as the top-level file, as shown in below Figure. Once this is set, the file name will be bolded, and the synthesis will be re-analyzed

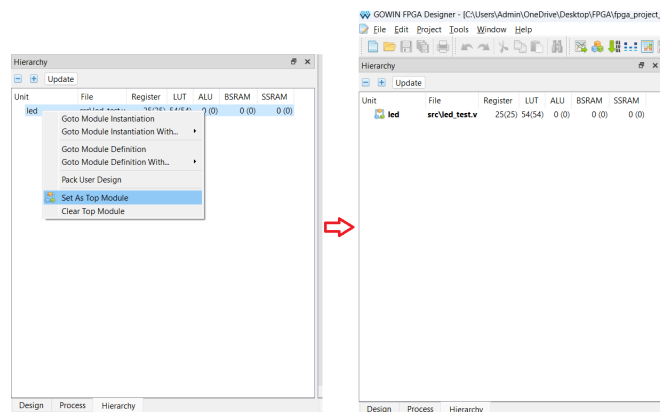


Figure 16: *Hierarchy set*

2.3 Physic constraints

2.3.1 FloorPanner

Clicking on Process->FloorPanner for physical constraints or clicking directly on toolbar, as shown in figure below

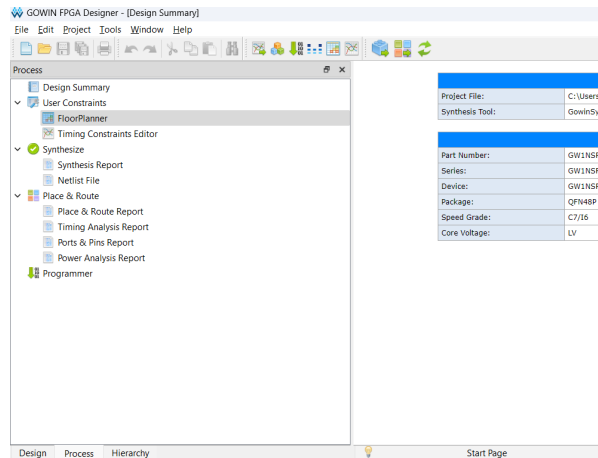


Figure 17: *Choosing floorPanner*

Then the dialog box will appear, t since we don't create a new .cst file to store the physics constraints, a dialog box will pop up asking if we need to create a .cst file, here we can directly click OK.

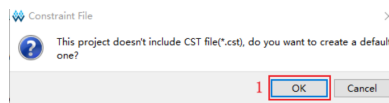


Figure 18: *The Create Physics Constraint File dialog box is displayed*

Next,go to the FloorPlanner page, click I/O Constraints, perform I/O constraints, and then assign pins and level standards according to your board

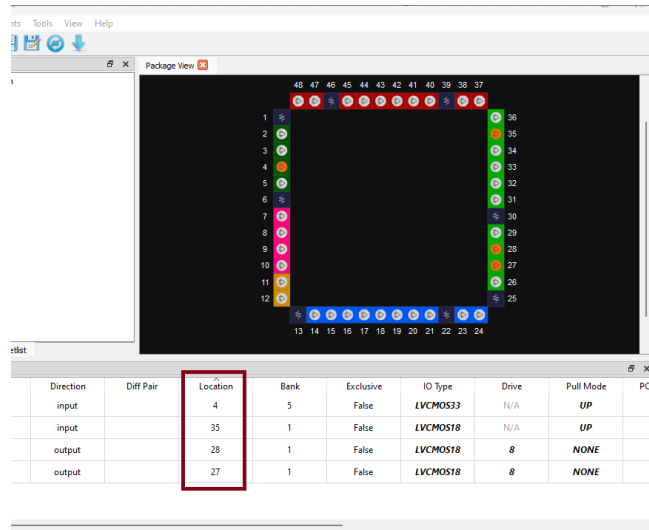


Figure 19: *I/O constraint interface*

After you fill the location, click Save icon on the taskbar, then a file .cst will be created in Physical Constraint Files folder in design part.

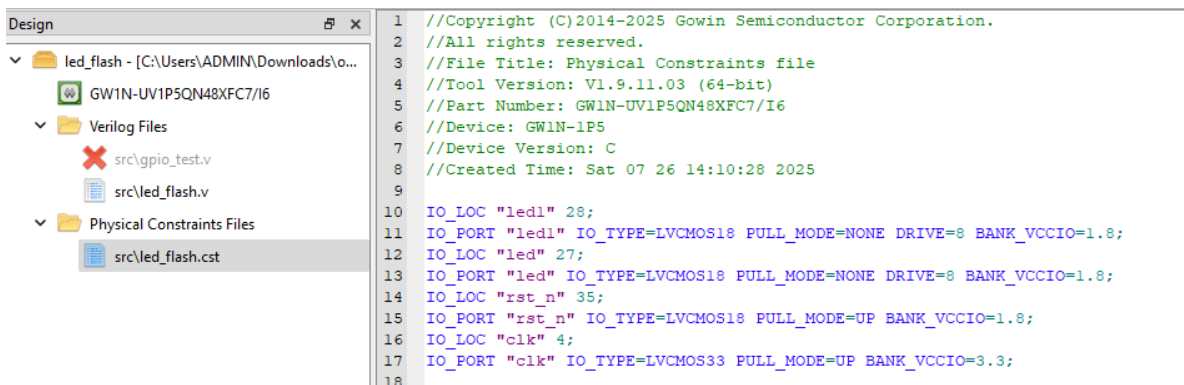


Figure 20: *View the contents of the physical constraint file*

2.4 Place and Route

We click Process->Place and Route to place and route, after the place-route is successful, we can see "Bitstream generation completed", if it fails, please modify it according to the prompt information given in the Console

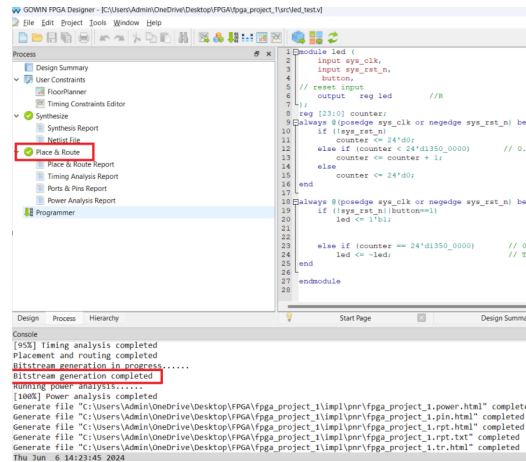


Figure 21: Place-and-route successful

2.5 Board-level validation

Connect your device to your computer, then click on the Programmer in process.

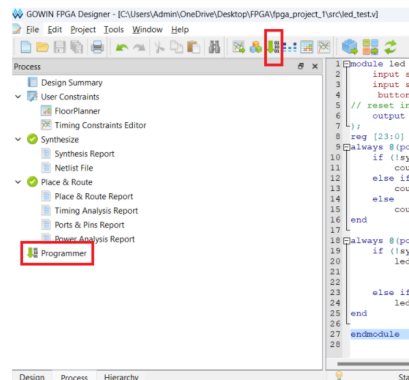


Figure 22: Programmer image

After that, the following figure interface will pop up indicating that the device has been detected, and then click save. If you see No USB Cable Connection, check your hardware connection.

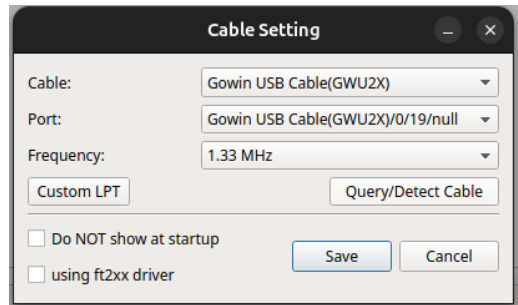


Figure 23: A configuration prompt has been detected

Clicking on Programmer/configure to download the fs file to the FPGA chip. After the download is successful, the message will be printed.

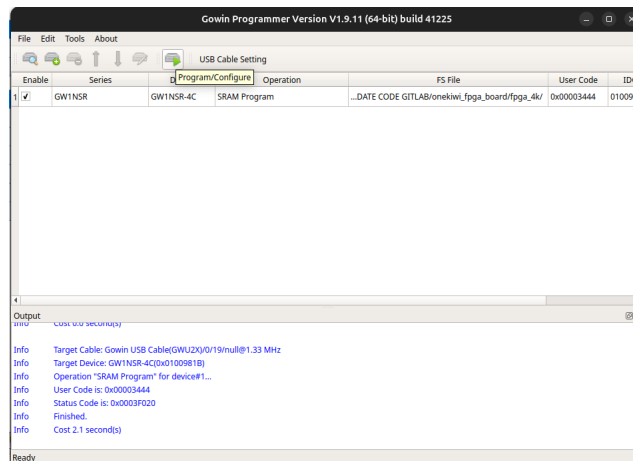


Figure 24: Download the fs file and a message indicating that the file is successfully downloaded

Observation phenomenon: At this time, we can see that the corresponding LED0 light on the development board is flashing, which means that our experiment is successful. At this time, we can see that after the program is downloaded, the Led is blinking after 0.5s.

3 Gowin and Modelsim simulation experiments

3.1 Modelsim compiles the Gowin device library

Gowin's IP CORE or primitives are only used in the corresponding development platform, and the third-party tool Modelsim cannot obtain the internal running logic results of Gowin's IP CORE, so it cannot be directly simulated, and the Gowin library needs to be compiled. This chapter begins with a tutorial on how to compile a Gowin library, which needs to be done the first time you use the Gowin FPGA, and the compiled library can be used at any time without having to compile it before each simulation.

3.1.1 Create a new Modelsim library

- Open the ModelSim software and click the [File]-> [Change Directory] option in the menu bar to enter the path switching page.

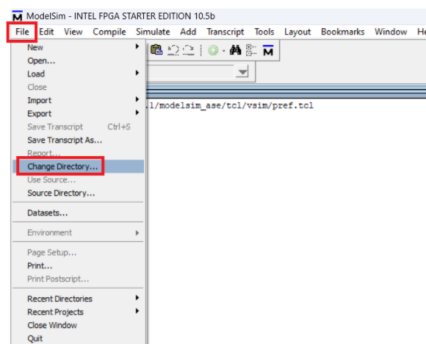


Figure 25: Go to the Path Switching page

- On the Select Folder page that opens, navigate to the Modelsim software installation directory, then select folder



Figure 26: Path of modelsim folder

- Next step, choose FILE->new library. In the interface that opens, select the first item "a new library", "library Physical Name" and enter the library name, click OK, as shown below.

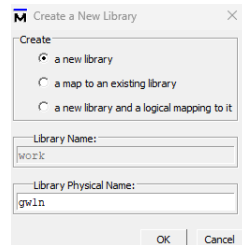


Figure 27: *Create a new library*

- If a gw1n(empty) library appears in the library list, and the path is the location set earlier (in the root directory of the project), the new library is complete.

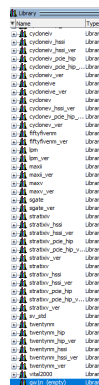


Figure 28: *The new library is completed*

3.1.2 Compile Gowin's device library files

- Click Compile -> Compile in the menu bar to open the compilation library file interface.

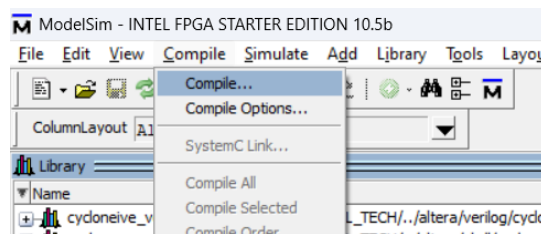


Figure 29: *Open the Compile Library file interface*

- You must select library as the gw1n we just created, and then modify the library path of the gw5a series device with the file path to Gowin EDA software, specifically:
`"...\GowinV1.9.9.11x64\IDE\simlib",`



Figure 30: *Locate the Gowin series library file*

- Under this path, find the prim_sim.v file and click the Compile button to start compiling, as shown in Figure below. At this time, the transcript window of Modelsim will quickly refresh the compilation process information, and after the compilation is completed, click the Done button to complete the compilation. (Remember to check name of the library)

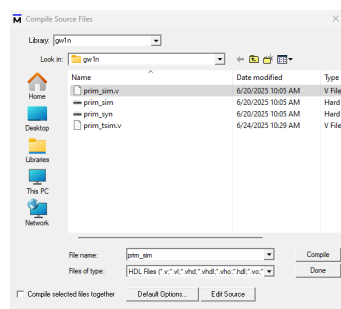


Figure 31: *Locate the library file to compile*

- Once compiled, the Transcript interface will display 0 errors, 0 warnings, and a variety of components under the gw5a library, as shown in Figure below.

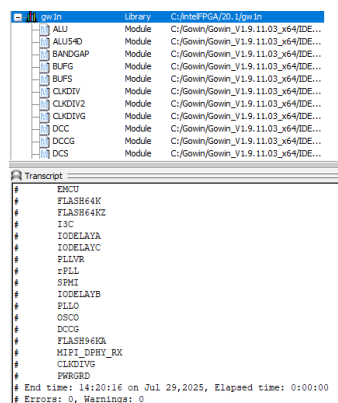


Figure 32: *The series device library has been compiled*

3.1.3 Add a library to the Modelsim default library list

- Find the installation location of Modelsim, this is a read-only file, right-click to select the Modelsim.ini file, click Properties, and check the read-only attribute in the property settings interface.

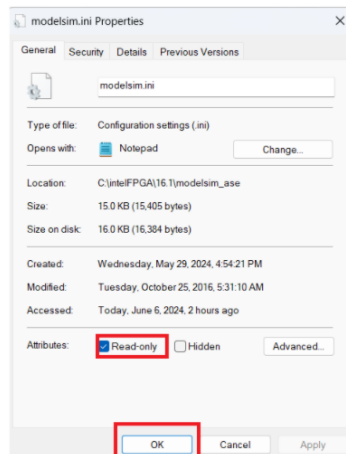


Figure 33: Removed the Moderssim. Ini's read-only attribute

- Then open the file with Notepad or Notepad++, and then about 17 lines of the file, add the specified statement of the gw1n library:

```
gw1n = $MODEL_TECH/../../gw1n
```

and save as shown below.

```
[Library]
std = $MODEL_TECH/../../std
ieee = $MODEL_TECH/../../ieee
verilog = $MODEL_TECH/../../verilog
vital2000 = $MODEL_TECH/../../vital2000
std_developerskit = $MODEL_TECH/../../std_developerskit
synopsys = $MODEL_TECH/../../synopsys
modelsim_lib = $MODEL_TECH/../../modelsim_lib
sv_std = $MODEL_TECH/../../sv_std
gw1n= $MODEL_TECH/../../gw1n
```

Figure 34: Add the GW5a library file path

- Restore the read-only properties of modelsim.ini. Now that we've covered how to compile the library for Gowin 5A series FPGAs using Modelsim, and the 1N and 2A series devices can also be compiled in the same way, and then we can use Modelsim to create simulation projects and simulate design projects.

3.2 Modelsim simulation validation process

3.2.1 Write Verilog simulation design code

- Create a verilog file name led_flash_tb in gowin,

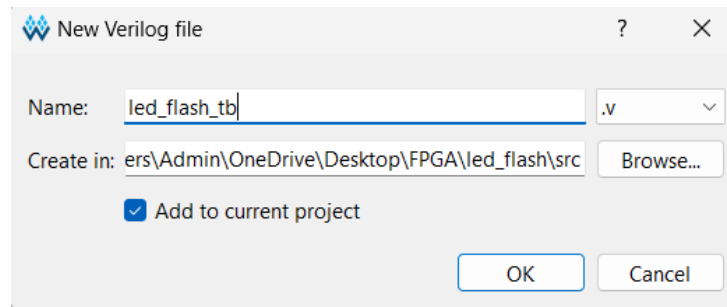


Figure 35: *Create simulation file name*

- Coding example:

Listing 2: *Testbench for led_flash*

```
`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#18.519 Clk = ~Clk;
    initial begin
        Rst_n = 0;
        #201;
        Rst_n = 1;
        #200;
        #100000000;
    end
endmodule
```

- Then remember to run synthesis to check error.

3.2.2 Create a Modelsim project and add simulation files

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

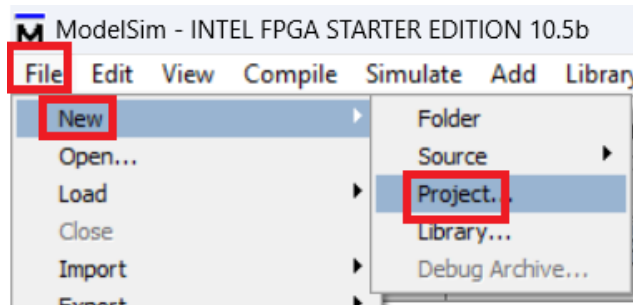


Figure 36: Establish a Modelsim project

- We name the project as the corresponding project name "led_flash", "Project Location" is the project path, click "Browse" to set, according to the needs of us save the simulation project to our design project, create a new sim folder and select, as shown in Figure below

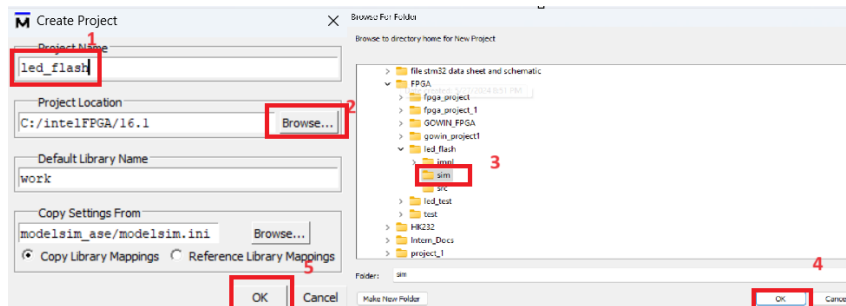


Figure 37: Modelsim project settings

- After that choose "Add Existing File"

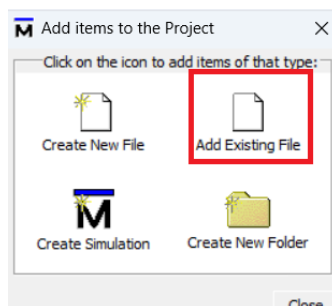


Figure 38: Add File to Project dialog box

- Selecting the Browse to select the project design file "led_flash.v" and "led_flash_tb.v", to add .

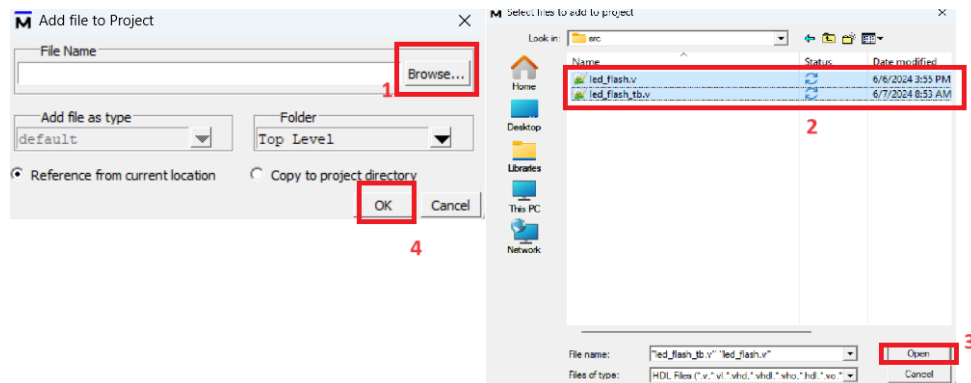


Figure 39: *Select the file to add*

- After the addition is complete, you can see the project files we added in the Modelsim.

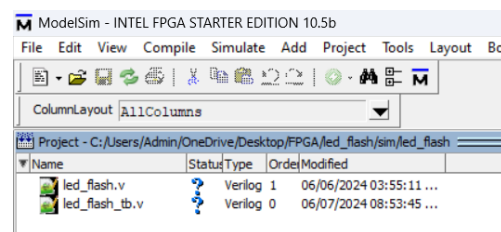


Figure 40: *After the file is added, the diagram is displayed*

3.2.3 Compile the simulation file

- There are two ways of compilation: the first is Compile Selected, the function of compiling selected files requires first selecting one or several files, and the translation of the selected can be completed after executing the command; The second is Compile All, which compiles all the files in the project in the order in which they are compiled. Here, we choose to compile all files and select Compile->Compile All, as shown in Figure below:

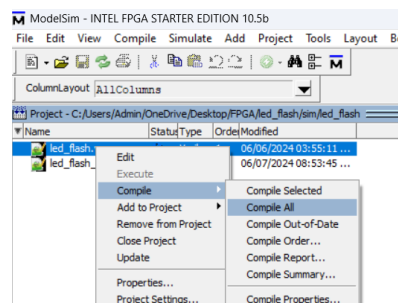


Figure 41: *Compile all project files*

- After compilation, the status at the end of the file changes indicates that the compilation has

passed. The "Transcript" below also indicates that the file has been compiled successfully.

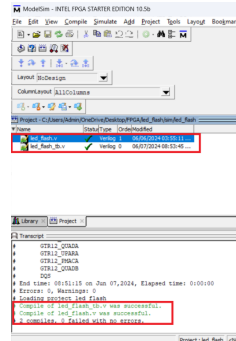


Figure 42: *Compile all project files*

3.2.4 Configure the simulation environment

- We right-click in the project status bar, then select "Add to Project"-> "Simulation Configuration..." and click on it

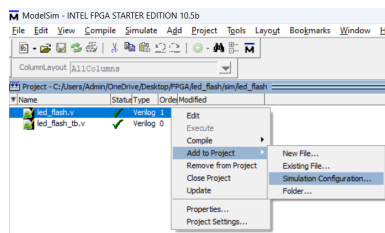


Figure 43: *Enter the operation diagram of configuring the simulation environment*

- In Add Simulation Configuration page, select the "led_flash_tb" module in the work library as the top level of the design in the Design tab page, and click Copy Module Name as the name of the simulation configuration "Simulation Configuration Name" to ensure that the naming is consistent, as shown in Figure below. In complex engineering design, we can design multiple different simulation configurations at the top level to simulate and test the project

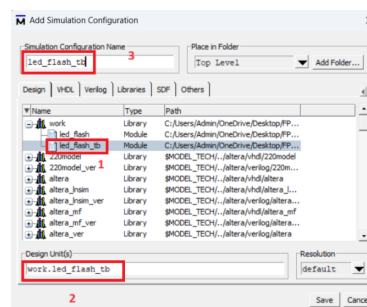


Figure 44: *Simulation environment interface configuration 1*

- Click on "Optimization Options...", (If your version has) and in "Optimization Options.." Select "Apply full visibility to all modules(full debug mode)" in the settings bar and click "OK".

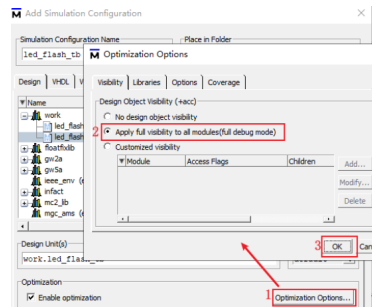


Figure 45: *Simulation environment interface configuration 2*

- Click on the "libraries" tab and click on "Add..." in the "Search libraries(-L)" section. Add the new library file "gw1n", select the library file "gw1n" in "Search Libraries First(-Lf)" as well, and finally click "Save" to save the settings, as shown in Figure below.

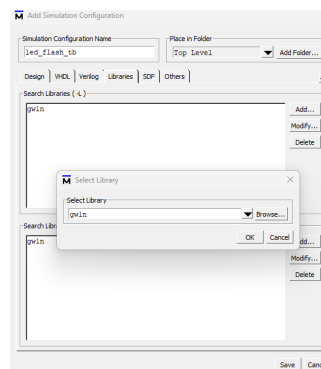


Figure 46: *Simulation environment interface configuration 3*

- After the above steps have been completed, and after saving the configuration, we can see that the simulation configuration file "led_flash_tb" is generated in the "project" column

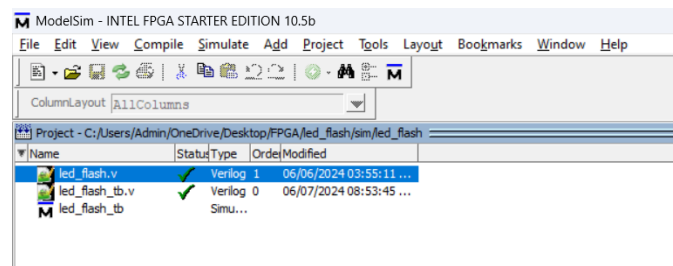


Figure 47: *Schematic diagram of successful generation of simulation file*

3.2.5 Simulation interface waveform observation

- We first need to double-click on the newly generated simulation file "led_flash_tb" to enter the "sim" field

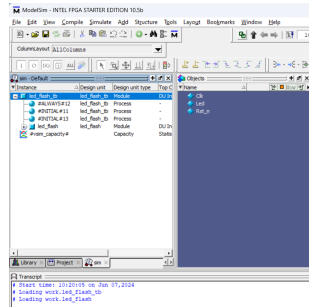


Figure 48: Enter the SIM interface

- In the "sim" interface, we can add the waveform of the module we want to observe, select the module, right-click and select "Add Wave"

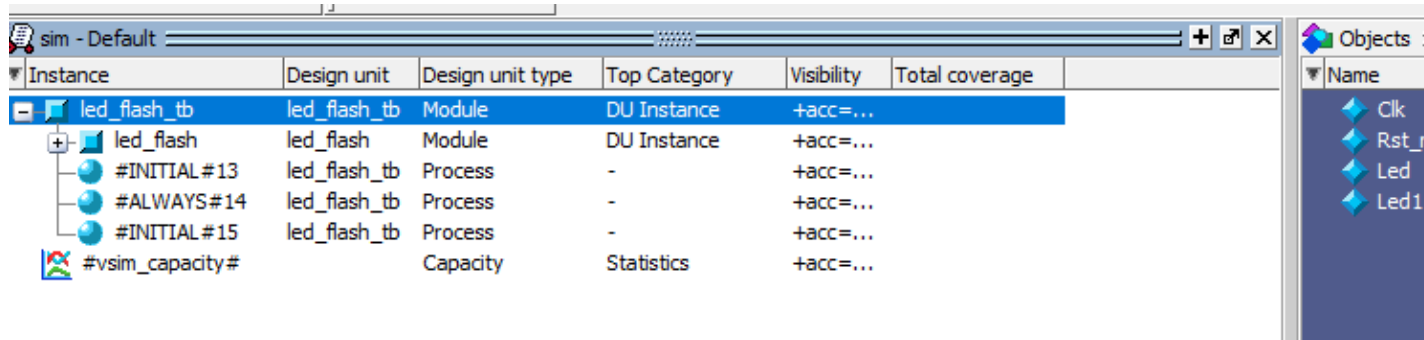


Figure 49: Add the module waveform that needs to be observed

- After adding the waveform, go back to the "Library" column, right-click on "Walker" and click Updat to update the led_flash_tb" file in the "Work" bar, as shown below.

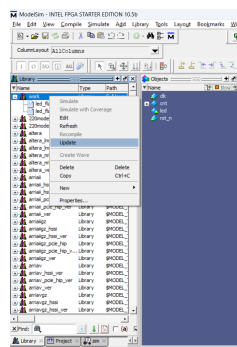


Figure 50: Update the work bar

- Click Run all to start running the simulation

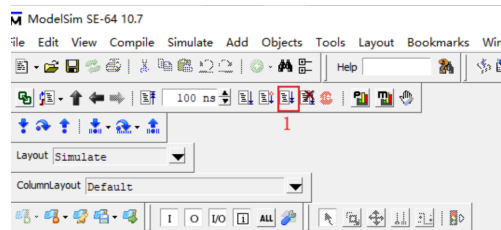


Figure 51: *Start the simulation*

- We will then jump to the Wave window, wait for the program to run for a while, and then we can see the waveform change.

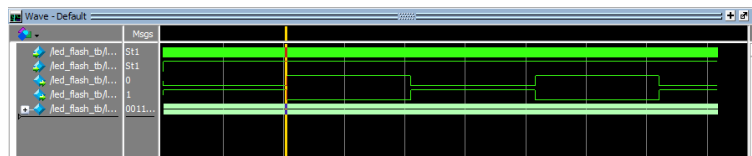


Figure 52: *Observe the waveform*

- If you want to stop running, click on the toolbar at the top of the Wave window, and start again, at this time, click "Zoom Full" to quickly observe the waveform, if you want to understand the function of each button in the toolbar, move the mouse over the button, you can see the explanation, such as continue to run the simulation button.

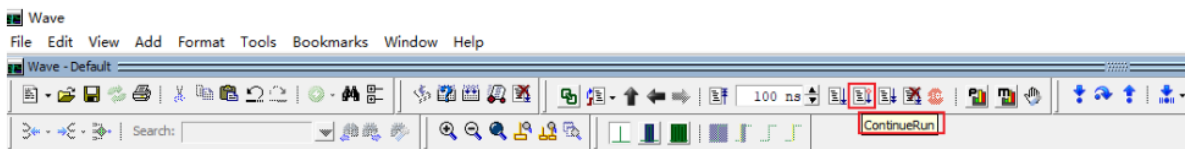


Figure 53: *Wave interface tool button function view*

- During development, after making changes to the design file, click Save and check that the syntax is correct before re-running the program. In Modelsim, we enter the "Library" interface, re-"Recompile" the "led_flash_tb" and "led_flash" in the "Work" directory, and reload the simulated waveform

3.2.6 Save the waveform file

- First of all, the first step is to save the dataset sim, open the sim window, and click file->save dataset sim or Click Save .wlf file on the SIM page and name the file as follows.

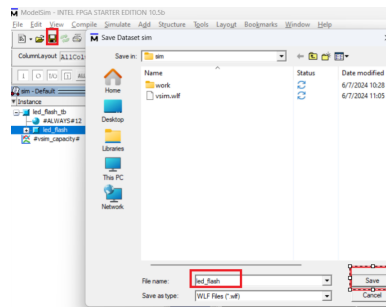


Figure 54: Saving the .wlf file

- In the second step, the Wave interface saves the waveform. After clicking the save icon, the waveform file "wave.do" will be saved in the project's simulation file sim by default.

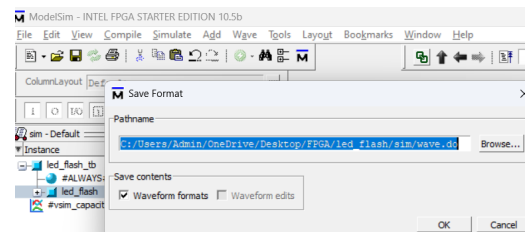


Figure 55: Save the waveform file

3.2.7 Reopen the simulation project

- After closing the simulation, the user needs to reopen the simulation by opening the .mpf file

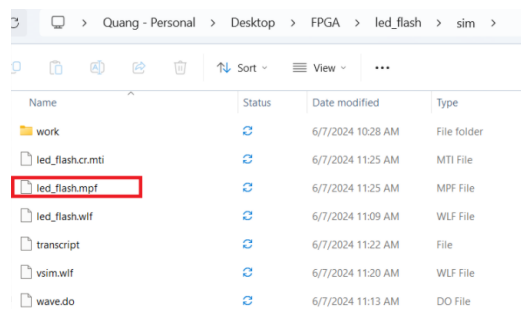


Figure 56: Reopen the simulation project

- When opening ".mpf" for the first time, the system will not choose to open it with Modelsim by default, readers need to manually set the way to open the ".mpf" file, and readers who install multiple versions of Modelsim also need to pay attention to setting the corresponding version of Modelsim software. If you have saved the waveform file before, after opening the Modelsim software, click file->all file to open the saved .wlf file, as shown in Figure 5 below

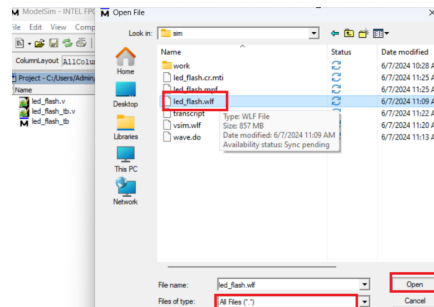


Figure 57: Open the saved .wlf file

- Then click file->load->Macro file to open the saved .do file, as shown in Figure below, and then you can see the waveform file we saved earlier.

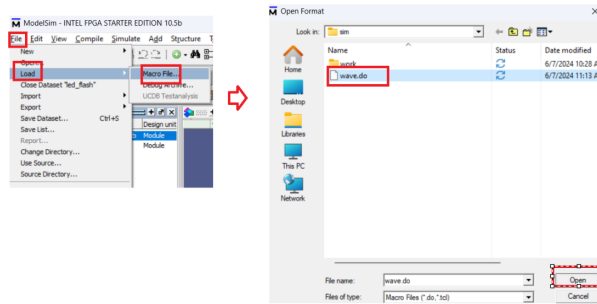


Figure 58: Open the saved waveform file

4 PWM (Pulse Width Modulation) to control LED brightness

A PWM signal has:

- A fixed frequency (e.g., 5 kHz)
- A duty cycle that controls how long the signal stays high in each cycle

4.1 Principle of PWM

PWM, or Pulse Width Modulation, is a control technique widely used in fields such as measurement, communication, and power electronics. Most inverter circuits today use PWM control.

From the figure, we can see that when the signal period is fixed, varying the duration of the high level changes the duty cycle. In inverter circuits, such PWM signals control the conduction time of MOSFETs, which in turn adjusts the average power delivered.

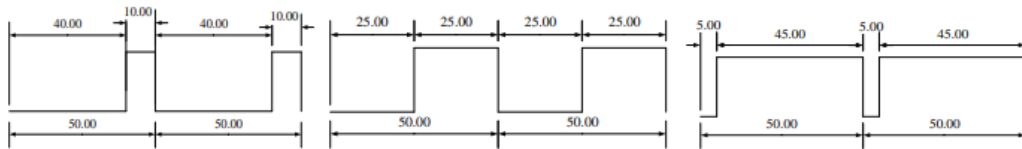


Figure 59: *PWM waveforms with different duty cycles*

Through the above analysis, it can be seen that to design a PWM generation circuit, it is necessary to achieve the adjustment of the signal's frequency and duty cycle. In microcontrollers or DSPs, the method to generate a PWM wave is to use the on-chip timer for cyclic counting. By setting a timing period of the timer, the frequency of the corresponding output PWM signal can be determined. Additionally, there is a comparator that compares the real-time count value of the timer with a user-defined comparison value. Based on the comparison result, it controls the output signal's level. By setting different comparison values, PWM signals with different duty cycles can be output.

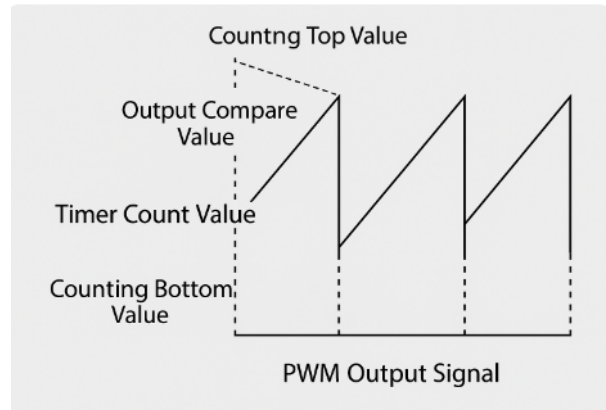


Figure 60: Principle of generating PWM using a timer

4.2 PWM Generator Module Design

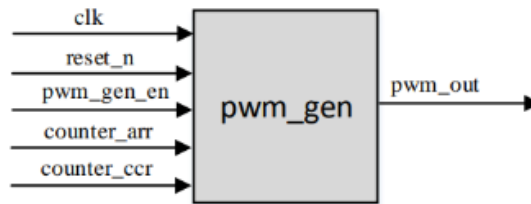


Figure 61: PWM module design

The signal	I/O	Description of the function
clk	I	The module global clock is 27MHz
reset_n	I	Module global reset signal
pwm_gen_en	I	PWM generates the module enable signal
counter_arr	I	Enter a 32-bit pre-reload value to determine the frequency
counter_ccr	I	Enter a 32-bit output comparison value to determine the duty cycle
pwm_out	O	Receives an end signal

Table 2: Table of module PWM description

The final output frequency of the PWM wave is calculated by:

$$f_{\text{pwm}} = \frac{f_{\text{clk}}}{\text{counter}_{\text{arr}}}$$

where $\text{counter}_{\text{arr}}$ is the preload value for the decrement counter. The counter counts down from $\text{counter}_{\text{arr}}$ to 1. Therefore, when the desired output frequency is determined, the preload value can be calculated as:

$$\text{counter}_{\text{arr}} = \frac{f_{\text{clk}}}{f_{\text{pwm}}} - 1$$

For example, if we wish to set the output signal frequency to 5 kHz:

$$\text{counter}_{\text{arr}} = \frac{f_{\text{clk}}}{f_{\text{pwm}}} - 1 = \frac{50000000}{5000} - 1 = 10000$$

Thus, setting the $\text{counter}_{\text{arr}}$ value to 10000 will achieve an output signal frequency of 5 kHz.

After the PWM output frequency is determined, the output duty cycle is calculated as the ratio of the output compare value to the preload value. The formula is:

$$\text{PW} = \frac{\text{counter}_{\text{ccr}}}{\text{counter}_{\text{arr}}}$$

Thus, when the output duty cycle is determined, the output compare value can be calculated as:

$$\text{counter}_{\text{ccr}} = \text{PW} \times \text{counter}_{\text{arr}}$$

For example, if the output frequency is 5 kHz and the output duty cycle is 70%:

$$\text{counter}_{\text{ccr}} = \text{PW} \times \text{counter}_{\text{arr}} = 0.7 \times 10000 = 7000$$

During operation, modifying the pre-reload value sets the frequency of the output PWM signal and will affect the output duty cycle at the same time, while modifying the output comparison value in the case of a determined pre-reload value allows the output duty cycle to be set.

Here we write a 32-bit counter to implement frequency control.

```
always @(posedge clk or posedge reset)
if (reset)
    pwm_gen_cnt <= 32'd1;
else if (pwm_gen_en)
begin
    if (pwm_gen_cnt <= 32'd1)
        pwm_gen_cnt <= counter_arr;    // When count reaches 1, load auto-reload register
        value
    else
        pwm_gen_cnt <= pwm_gen_cnt - 1'b1; // Decrement counter by 1
end
else
    pwm_gen_cnt <= counter_arr;        // When not enabled, counter equals auto-reload
    register value
```

Write another comparator, where the output of the comparator is the output of the PWM, to achieve duty cycle control.

```
always @(posedge clk or posedge reset)
if (reset)                                // On reset, PWM output is low
    pwm_out <= 1'b0;
else if (pwm_gen_cnt <= counter_ccr) // When count is less than compare value, PWM
    output is high
    pwm_out <= 1'b1;
else
    pwm_out <= 1'b0;                    // When count is greater than compare value, PWM
    output is low
```

4.3 Simulation

Complete the design above and analyze it until there are no errors. Then add and create a simulation file named pwm_ge_tb.v. In the simulation file, you only need to generate a 50MHz(or 27MHZ) reference clock source, then provide pre-load values and output comparison values, and finally enable counting to simulate PWM output. Some of the stimulus files are as follows:

```
initial begin
    reset_n = 0;
    pwm_gen_en = 0;
    counter_arr = 0;
    counter_ccr = 0;
    #(`CLK_PERIOD*20 +1);
    reset_n = 1;
    #(`CLK_PERIOD*10 +1);
    counter_arr = 1000;    // Set output signal frequency to 50KHz
    counter_ccr = 400;     // Set PWM output duty cycle to 40%
    #(`CLK_PERIOD*10);
    pwm_gen_en = 1;       // Start counting to generate PWM output
    #100050;
    counter_ccr = 700;     // Set PWM output duty cycle to 70%
    #100050;
    pwm_gen_en = 0;       // Stop counting to disable PWM output
    counter_arr = 500;     // Set output signal frequency to 100KHz
    counter_ccr = 250;     // Set PWM output duty cycle to 50%
    #(`CLK_PERIOD*10);
    pwm_gen_en = 1;       // Start counting to generate PWM output
    #50050;
    counter_ccr = 100;     // Set PWM output duty cycle to 20%
    #50050;
```

```
pwm_gen_en = 0;           // Stop counting to disable PWM output
#200;
$stop;
end
```

After designing the simulation files, open the Modelsim project, add the required files, and then add the port signals of pwm_gen_tb to the Wave window for observation. Click to observe the global waveform as shown in Figure below.

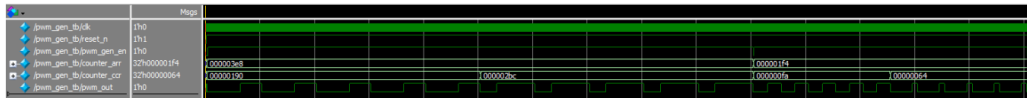


Figure 62: *pwm simulation*

Zoom in on the local waveform, as shown in the figure below. At this point, the output PWM frequency is set to 50KHz (counter_arr is 1000) and the duty cycle is 40percent (counter_ccr is 400), resulting in the simulated waveform. Standard lines can be added using tools in the window to align with the edges of one cycle of the PWM output. From the diagram, it can be seen that the low level duration is 12us, the high level duration is 8us, and the total signal period is 20us, which corresponds to a frequency of 50KHz. The duty cycle is $8/20 = 0.4$. This meets the established design. Similarly, other waveforms can be analyzed.

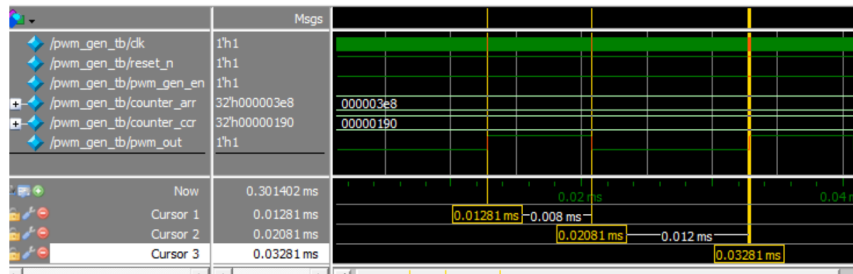


Figure 63: *50kHz duty cycle 0.4 simulation waveform chart*

The following figure shows the simulation waveform when the PWM output frequency is set to 50KHz (counter_arr is 1000) and the duty cycle is 70percent (counter_ccr is 700). From the figure, it can be seen that the low level period is 6us, the high level period is 14us, and the total signal period is 20us, which corresponds to a frequency of 50KHz. The duty cycle is $14/20 = 0.7$.

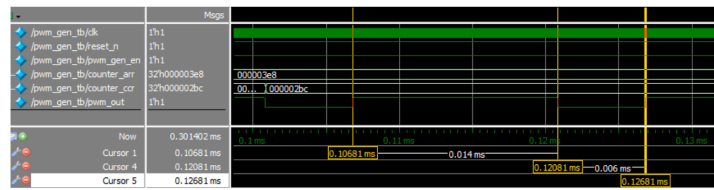


Figure 64: 50kHz duty cycle 0.7 simulation waveform chart

The figure below shows the simulation waveform when the PWM wave output frequency is set to 100KHz (counter_arr is 500) and the duty cycle is 50percent (counter_ccr is 250). From the figure, it can be seen that the low level period is 5us, the high level period is 5us, and the total signal period is 10us, which means the frequency is 100KHz. The duty cycle is $5/10 = 0.5$.

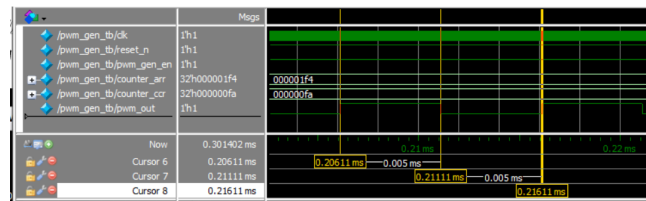


Figure 65: 100kHz duty cycle 0.5 simulation waveform chart

The figure below shows the simulation waveform when the PWM wave frequency is set to 100KHz (counter_arr is 500) and the duty cycle is 20percent (counter_ccr is 100). From the figure, it can be seen that the low level period is 8us, the high level period is 2us, and the total signal period is 10us, which means the frequency is 100KHz. The duty cycle is $2/10 = 0.2$.

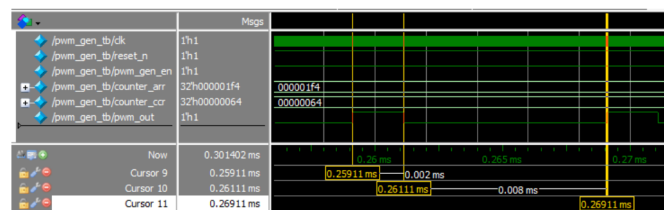


Figure 66: 100kHz duty cycle 0.2 simulation waveform chart

4.4 Board-Level Debugging and Verification

This part of the circuit is relatively simple, so this board-level verification section will not elaborate on the detailed design reasoning of the code, but will only provide the specific code. The code part for the buzzer playing tones is shown below.

```
module pwm_gen(  
    clk,  
    reset_n,  
  
    pwm_gen_en,  
    counter_arr,  
    counter_ccr,  
    pwm_out  
);  
    wire reset;  
    input clk;      // System clock input, 50 MHz  
    input reset_n;  // Reset signal input, active low  
    input pwm_gen_en; // PWM generation enable signal  
    input [31:0]counter_arr; // 32-bit preload value input  
    input [31:0]counter_ccr; // 32-bit output compare value input  
    output pwm_out;    // PWM output signal  
  
    reg pwm_out;  
    reg [31:0]pwm_gen_cnt; // Define a 32-bit counter for PWM generation  
  
    assign reset = ~reset_n;  
    always@(posedge clk or posedge reset)  
    if(reset)  
        pwm_gen_cnt <= 32'd1;  
    else if(pwm_gen_en)  
    begin  
        if(pwm_gen_cnt <= 32'd1)  
            pwm_gen_cnt <= counter_arr;    // When the counter decreases to 1, load the  
                                            // preload register value  
        else  
            pwm_gen_cnt <= pwm_gen_cnt - 1'b1; // Counter decrements by 1  
    end  
    else  
        pwm_gen_cnt <= counter_arr;    // When not enabled, counter value equals  
                                        // preload register value  
  
    always@(posedge clk or posedge reset)  
    if(reset)    // On reset, PWM output is low  
        pwm_out <= 1'b0;  
    else if(pwm_gen_cnt <= counter_ccr) // When counter value is less than compare value,  
                                        // PWM output is high  
        pwm_out <= 1'b1;
```

```
else
    pwm_out <= 1'b0;           // When counter value is greater than compare
                               value, PWM output is low
endmodule
```

5 Key filtering

5.1 Key physical structure and circuit design

- As a basic human-computer interaction element, the button is widely used in electronic design, from system reset to control settings. At present, there are many types of buttons, such as multi-directional buttons, self-locking buttons, membrane buttons, etc
- In this section, the methods of asynchronous signal synchronization and edge detection of single-bit data are introduced into the key debounce design commonly used in FPGAs. The random number generation function and the stimulus model are introduced into the simulation test stimulus file. Finally, a practical example will be given to demonstrate the way modular design works.

5.2 Key physical structure and circuit design

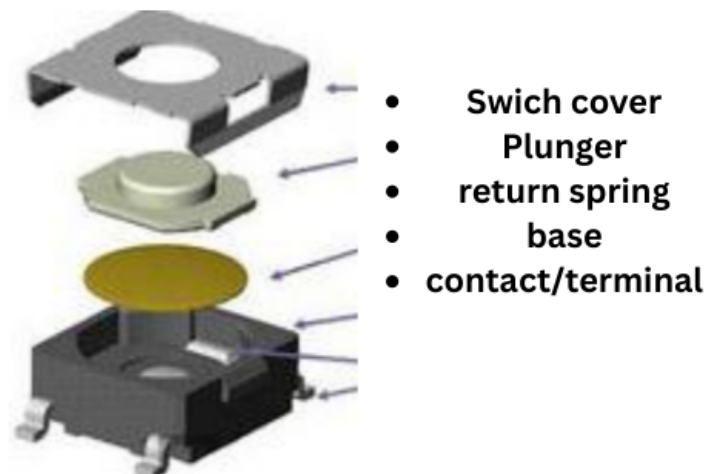


Figure 67: 1 Schematic diagram of the hardware structure of the button

- The development board contains two-pronged chip buttons, which are located in the lower right corner of the front of the learning board, as shown in the schematic diagram below. As can be seen from the schematic diagram, the IO port is high when the key is not pressed, and it becomes low when the key is pressed, so the system can judge the status of the key by detecting the level of the IO.
- Schematic of button:

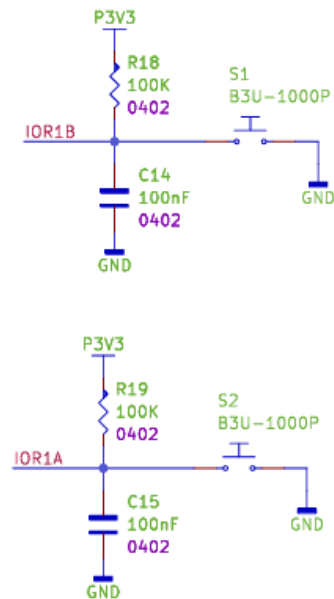


Figure 68: The change in the level of the key from pressing to release

- In the schematic diagram of the key structure, it can be seen that there is a reaction spring in the key, so when it is pressed or released, additional physical jitter will be generated, and the physical jitter will produce level jitter. As shown in Figure 11-3 below, the level of the key changes from pressing to release, with the ideal waveform output at the top and the actual waveform output at the bottom.

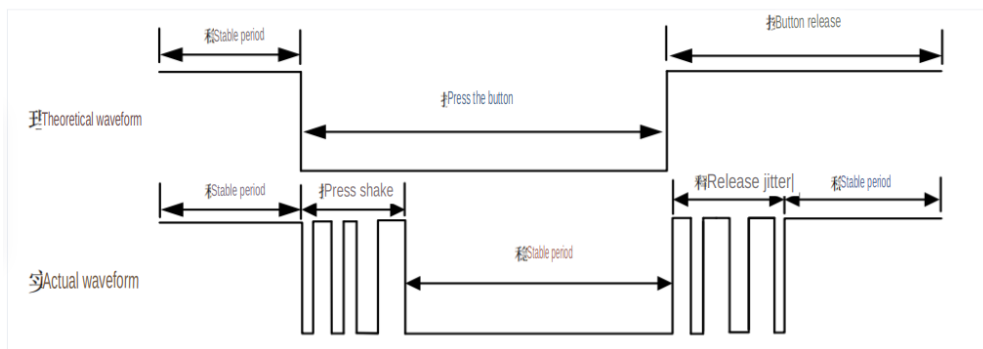


Figure 69: 3 The change in the level of the button from pressing to releasing it

- In the diagram above, the number and interval of jitter generated are unpredictable, so filtering is required to eliminate the impact that jitter may have on other external devices. In general, the total time of jitter lasts less than 20ms. This jitter can be eliminated by hardware circuitry or logic design, or it can be done by software. Among them, the hardware circuit eliminates jitter, which is suitable for occasions with a small number of buttons.

5.3 The state machine implements key debounce

5.3.1 Module interface design

4For a single-button debouncer module, the interface is shown in Figure 11-11 below, and the interface declaration function is described in the following table



Figure 70: *Key debounce module interface*

The name of the interface	I/O	Description of the function
clk	I	Module working clock, 27M clock
reset_n	I	Module reset signal
key_in	I	Key input
key_flag	O	Flag for key status switching
key_state	O	Key status flags

Table 3: *Description of the interface function of the key debounce module*

5.3.2 State machine transfer state as well as conditional design

For FPGA designs, state machines are often used for key debounce. As can be seen in the figure below, the following states are obtained by analyzing the state code according to the concept of a state machine: IDLE IDLE is not pressed, Jitter Filter State FILTER0 is pressed, DOWN is pressed, and Jitter Filter State is released FILTER1. The unique hotcode codes are 4'B0001, 4'B 0010, 4'B 0100 and 4'B 1000. The state transition diagram is shown in Figure 11-12 below, which shows that it is a Milli state machine

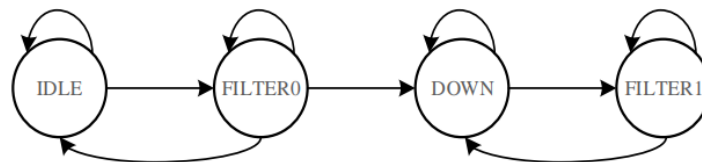


Figure 71: *State Transition Diagram*

The following table describes the conditions for the transition of its status:

Current status	Next state	Transfer conditions
IDLE	FILTER0	When the falling edge of the key input is detected, press debounce is initiated
FILTER0	IDLE	If the rising edge is detected and the 20ms time is not exceeded in the press debiting, the key is considered to be still shaking and is still not stable, so it returns to the initial state
FILTER0	DOWN	The timer starts when the falling edge is detected, and if the timer exceeds 20ms, it is considered to be pressed indeed
DOWN	FILTER1	After the press is stable, if the upper and lower edges are detected, the release debiting is initiated
FILTER1	DOWN	In the process of resuming debiting when the button is released, the timing starts when the rising edge is detected, and when the falling edge is detected and does not exceed 20ms timing, it is considered to be still shaking and continues to deshake
FILTER1	IDLE	The timing starts when the rising edge is detected, and if the timing exceeds 20ms, it is considered that the release is indeed complete

Table 4: State transitions for key debounce FSM

5.3.3 Key input signal edge detection

From the point of view of the design of the state machine, the edge of the key input signal `key_in` needs to be detected, and the theoretical knowledge about edge detection can refer to the serial port receiving module to explain, and the explanation is not repeated here, and the edge detection code of the specific key data signal 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;//1st stage
    key_in_sync2 <= key_in_sync1; //2nd stage: now synchronized!
end
//Use the D flip-flop to store the level status of an external input signal (already
    synchronized into the system clock domain) at the rising edge of two adjacent clocks
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; // current state
    key_in_reg2 <= key_in_reg1; //previous state
end
//Generate a jump edge signal
assign key_in_nedge = !key_in_reg1 & key_in_reg2; //1->0
assign key_in_pedge = key_in_reg1 & (!key_in_reg2); //0->1
```

5.3.4 Counter design

There should also be a 20ms counter module and a counter enable module in the state transition table, which can also be combined into a single Always module. It is generally recommended that an Always block operate on only 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'd539_999)
    cnt_full <= 1'b1;
else
    cnt_full <= 1'b0;
```

5.3.5 State machine design

Now let's start designing a state machine, first defining the state of the state machine with local parameterization

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

Due to the small number of states and judgment conditions, a one-stage state machine is used

to describe it first. When reset the counter is cleared to zero, the state returns to the IDLE state, and key_flag and key_state 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 the state is IDLE when it is not pressed, if a falling edge is detected, the state enters the FILTER0 of the Jitter Filter state and the counter is enabled, otherwise it continues to remain in the IDLE state.

```
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 the rising edge arrives before the timer ends for 20ms, it is considered that the key is still pressed and the jitter process is pressed, and the state returns to IDLE and clears the 0 counter. When the last jitter is pressed during the pressing process, there will be no rising edge, and the counter can be counted all the time, and when the count is full, the key_flag will be set to 1 and the key_state will be set to 0, and the state will enter the steady state of pressing DOWN and the counter will be cleared to 0. In this way, the state of the key can be determined by judging key_flag !key_state, and pressing it if it is 1.

```
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
end
```

Once you enter the key steady state DOWM, clear the key_flag to 0. If a rising edge is detected, it enters the Release Jitter Filtering State FILTER1, otherwise it remains in 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
end
```

After entering the FILTER1 state, if the falling edge is detected before the 20ms count is over, it is considered that the key release jitter process is still the process, and the status returns to DOWN and clears the 0 counter. When the last jitter is shaken during the release process, there will be no falling edge, and the counter can be counted all the time, and when the count is full, the key_flag and key_state will be set to 1, and the state will enter IDLE and the counter will be cleared to 0, waiting for the next key to be pressed.

```
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 you rewrite it as a two-stage state machine, it will be as follows, and the state transition part will be 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 like a segmented state machine
endcase
end
```

5.4 Testbench design and simulation testing

In addition to generating a normal clock and a reset signal, the testbench file excitation also simulates the process of pressing the key from pressing to letting go, and this simulation excitation designs the key in the way of a separate design simulation model, so that the internal structure of the whole excitation file is as shown in Figure below:

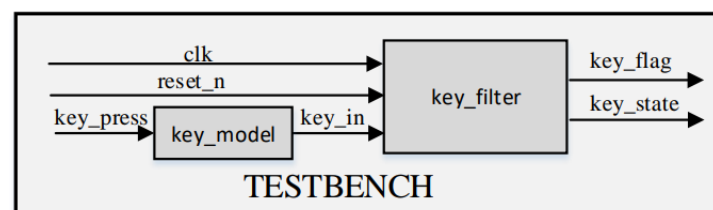


Figure 72: *Filtering testbench*

5.4.1 The design of the key model key_model

- The function of the key model is to simulate the output of the signal during the real key press, including the jitter during the key press. Considering that the jitter time during the

real key pressing process is uncertain, the random number generation function random in verilog is introduced here to generate jitter.

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

5.4.2 Use of tasks

Task is a common syntax in Testbench, and its syntax is as follows:

```
task <task name>;
    <Port and Data Type Declaration Statement>
    < statement 1>
    < statement 2>
    < statement n>
endtask
```

The syntax for the call of the task is as follows:

```
<taskname>(Port 1, Port 2,... port n);
```

For a key press and let go of the whole process in the form of a task design, here to achieve fifty times of 0 65535ns press jitter, and then key_in assigned a fixed value of 0 and a delay of 25ms (greater than 20ms is stable), but also to achieve the release of jitter, key_in assigned a fixed value of 1 and a delay of 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
```

On the basis of the above, in order to realize the key model, the key_model module can be controlled by an external key_press, that is, a pulse is given to the key_press, and a key signal is key_model output, so that the above task is called in the key_model module by the arrival of the key_press rising edge as the condition, and the specific code is as follows.

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

Once the Key Model key_model module has been designed, the design of the entire simulation Teshbench is relatively simple. Add and create a new key_filter_tb.v file to instantiate the key debounce module and key model in and generate a clock and reset stimulus signal, the code is as follows. The simulation resulted in two key presses

```
`timescale 1ns/1ns
`define CLK_PERIOD 37.037
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 the Modelsim new project for simulation, and you can get the simulation waveform shown in Figures below and Figures below key_state key_flag. Here you can see that the jitter time is different for each jitter, so that the random jitter process is successfully simulated.

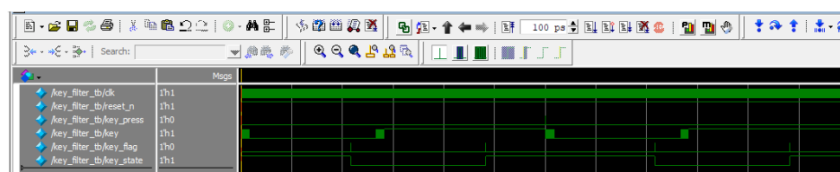


Figure 73: *Key debounce model simulation waveform diagram*

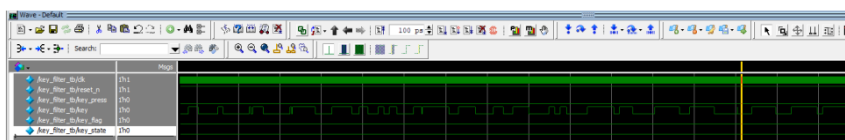


Figure 74: *Random jitter process*

5.5 Design key control counter led

In this module we will test 2 leds is display as binary number, 1 button is decrease counter, 1 button is increase counter.

Module Led control

```
module led_ctrl(
    Clk,          // 50M clock input
    Rst_n,        // Module reset
    key_flag0,    // Key S0 flag signal
    key_flag1,    // Key S1 flag signal
    key_state0,   // Key S0 state signal
    key_state1,   // Key S1 state signal
    led
);

input Clk;
input Rst_n;
input key_flag0, key_flag1;

input key_state0, key_state1;

output [1:0]led;

reg [1:0]led_r;

always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    led_r <= 2'b00;
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;

assign led = led_r;

endmodule
```

Module Key_led_top

```
module key_led_top(
    Clk,      //27MHZ
    Rst_n,    //RESET
    key_in0,
    key_in1,
    led
);

    input Clk;
    input Rst_n;
    input key_in0;
    input key_in1;

    output [1:0]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
```

6 UART_TX Design and verification of serial port transmitter module

6.1 UART key parameters and time series diagrams

- UART communication requires a number of settings before it can be used, the most common of which include the number of data bits, baud rate size, parity type, and stop bits.
- Data bits: This parameter defines the number of data bits sent from start to stop for a single UART data transfer. You can choose 5, 6, 7, or 8 (default).
- 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.
- 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. If the user chooses the data length to be 8 bits, it is called "Non" because there are no extra bits to use as parity bits.
- Stop bits: After each byte of data bits are sent, a stop bit is sent to mark the completion of a data transfer and is used to help the receiving party resynchronize the hardware. You can choose from 1 (default), 1.5, or 2 digits
- In the RS-232 standard, the most common configuration is 8N1 (i.e., eight data bits, no parity, one stop bit), which sends a one-byte timing diagram as shown in Figure below. According to a complete byte including a start bit, 8 bits of data, a stop bit that is a total of 10 bits of data, in order to completely realize the transmission of these 10 bits of data, 11 baud rate clock pulses are required, the first pulse marks the beginning of a transmission, and the 11th pulse marks the end of a transmission.

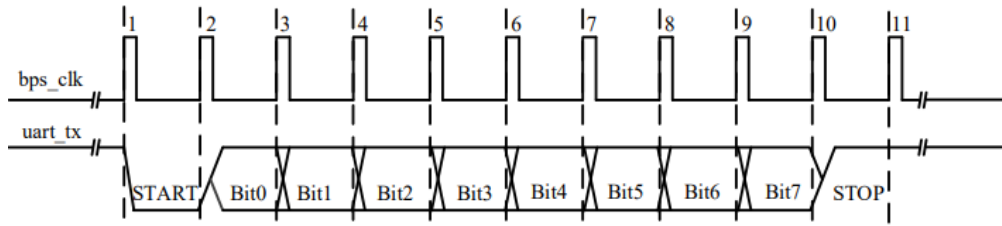


Figure 75: *process of uart transfer*

- When the first rising edge of the bps_clk signal arrives, the byte transmitting module starts to send the start bit, and the next 2 to 9 rising edges send 8 data bits, and the 10th rising edge to the 11th rising edge is the transmission of the stop bit.

6.2 Design and implementation of UART asynchronous serial communication transmission module

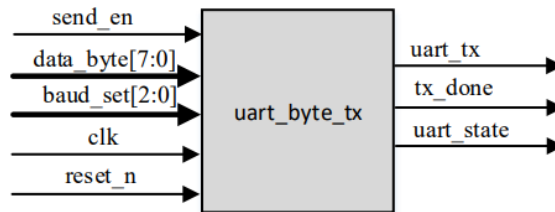


Figure 76: *Module of uart transfer*

The signal	I/O	Description of the function
clk	I	The module global clock is 27MHz
reset_n	I	Module global reset signal
data_byte	I	8 bits to be transferred
send_en	I	Send an enable signal
baud_set	I	Baud rate to set the signal
uart_tx	O	Serial port sends signal output
tx_done	O	Sends an end signal, one clock cycle high
uart_state	O	Send status, 1 when in the send state

Table 5: *Table of module description*

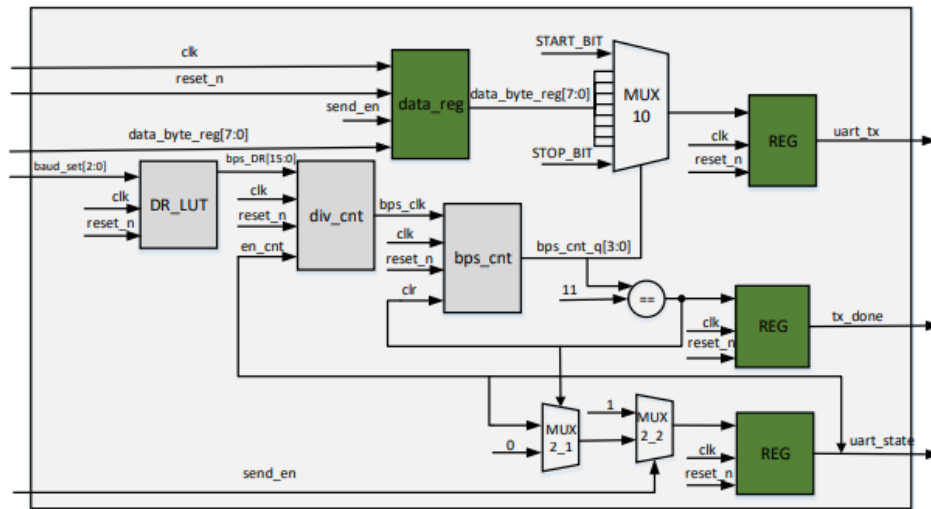


Figure 77: Module of uart transfer specifickly

- send_en:enable sending
- baud_set[2:0]:set value baudrate

6.3 Baurate clock module design

- Our system have clock frequency 27MHZ,formula:= $\frac{27 \times 10^6}{baudrate} - 1$ (change the value base on the frequency)
- baudrate:9600->2811
- 19200->1405
- 38400->702
- 57600->467
- 115200->233

```
always@(posedge clk or posedge reset)
  if(reset)
    bps_DR <= 16'd2811; // Default baud rate setting
  else begin
    case(baud_set)
      0:bps_DR <= 16'd2811; // Baud rate option 0(27MHZ, so baurate
        9600:27*10^6/9600 -1=2811)
      1:bps_DR <= 16'd1405; // Baud rate option 1:19200
      2:bps_DR <= 16'd702; // Baud rate option 2:38400
```

```
3:bps_DR <= 16'd467; // Baud rate option 3:57600
4:bps_DR <= 16'd233; // Baud rate option 4:115200
default:bps_DR <= 16'd2811; // Default value
endcase
end
```

Checking the uart_state

```
always@(posedge clk or posedge reset)
if(reset)
    uart_state <= 1'b0;
else if(send_en)
    uart_state <= 1'b1;
else if(bps_cnt == 4'd11)
    uart_state <= 1'b0;
else
    uart_state <= uart_state;
```

Checking reset signal:

```
always@(posedge clk or posedge reset)
if(reset)
    data_byte_reg <= 8'd0;
else if(send_en)
    data_byte_reg <= data_byte;
else
    data_byte_reg <= data_byte_reg;
```

Here we use a counter to generate a 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;
```

6.4 Data output module design

Determine the state of the data sending cycle by counting the baud rate clock

```
always@(posedge clk or posedge reset)
if(reset)
    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 in order for the module to be able to control or call other modules, a byte is generated to signal the end of the transfer. After a data bit is transmitted, the tx_done signal outputs a high level of the clock.

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

Data transmission status control module

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

6.5 TESTBENCH code for modelsim

```
`timescale 1ns/1ns
`define CLK_PERIOD 37.037
module uart_byte_tx_tb;
    reg clk;
    reg reset_n;
    reg [7:0]data_byte;
    reg send_en;
    reg [2:0]baud_set;
    wire uart_tx;
    wire tx_done;
    wire uart_state;
    uart_byte_tx uart_byte_tx(
        .clk(clk),
        .reset_n(reset_n),
        .data_byte(data_byte),
        .send_en(send_en),
        .baud_set(baud_set),
        .uart_tx(uart_tx),
        .tx_done(tx_done),
        .uart_state(uart_state)
    );
    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
endmodule
```

6.6 Coding test for real device

```
module uart_tx_test(
    input clk,
    input reset_n,
    output uart_tx,
    output led
);
    parameter MCNT = 26_999_999; //
    reg [7:0] data_byte;
    reg send_en;
    reg [25:0] cnt; //
    wire tx_done;
    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),
    .uart_state(led)
);
endmodule
```

6.7 Simulation

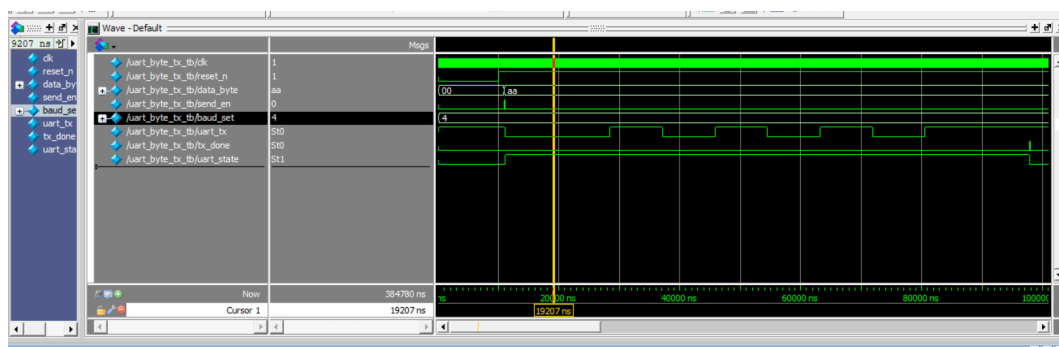


Figure 78: *Simulation with modelsim*

6.8 Testing with board

6.8.1 GWU2U Virtual Serial Port setting

For install driver for GWU2U we use Zadig software (on Window)

Install Zadig at :<https://zadig.akeo.ie/>

Connect GWU2U device to computer with USB interface, double-click to open Zadig (administrator privileges required), click Options, check the "List All Device" option, and all USB devices connected to the computer will be listed

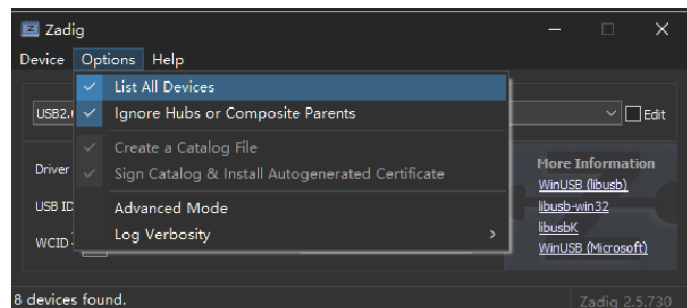


Figure 79: Zadig interface 1

Select Gowin U2U, the device that requires driver installation

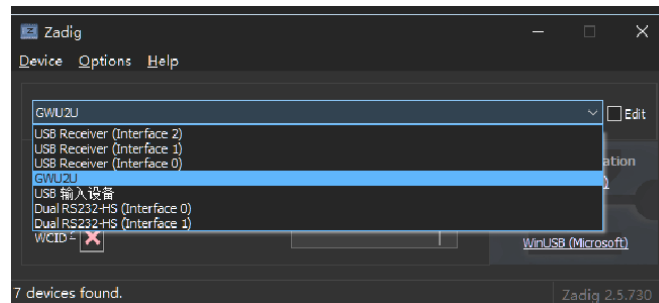


Figure 80: Zadig interface 1

Select the driver to be installed, use virtual serial port driver (VCP), and select USB Serial (CDC).

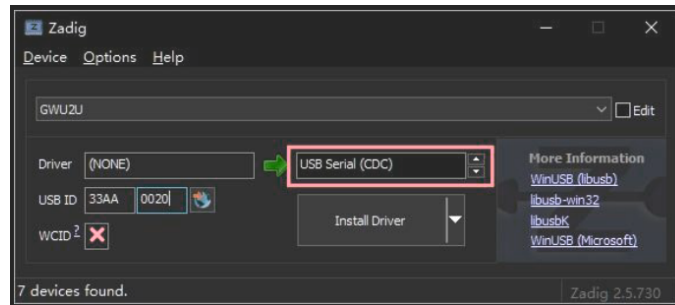


Figure 81: *Zadig interface 2*

Click “Install Driver”. The driver will be installed after a few moments.

6.8.2 Using hercules software

Link for installing: <https://www.hw-group.com/software/hercules-setup-utility>

Connect the GWU2U(UART) USB to the computer, then open device manager find the COM port:

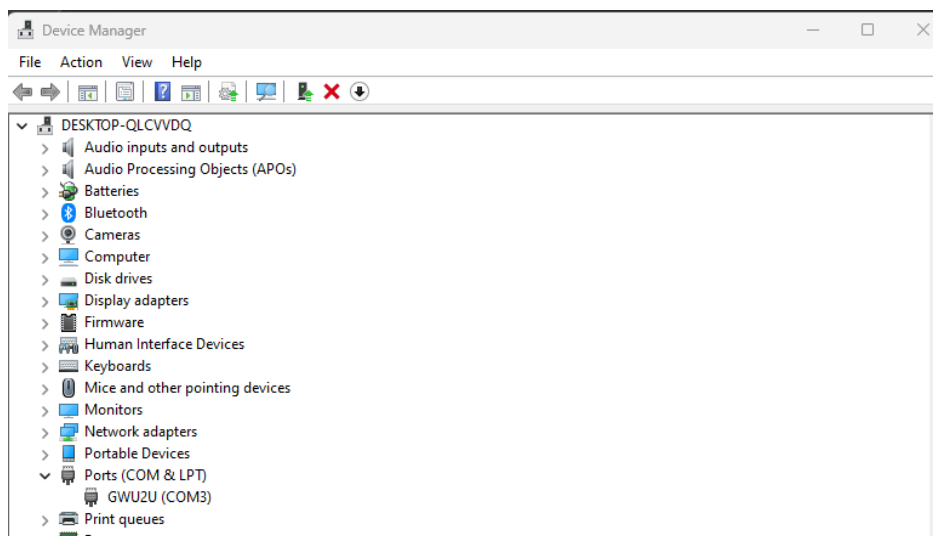


Figure 82: *Find the COM port*

We using hercules software

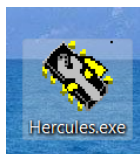


Figure 83: *hercules software*

Setting up 8bit,choose the COM port connect to UART,configure baud,parity

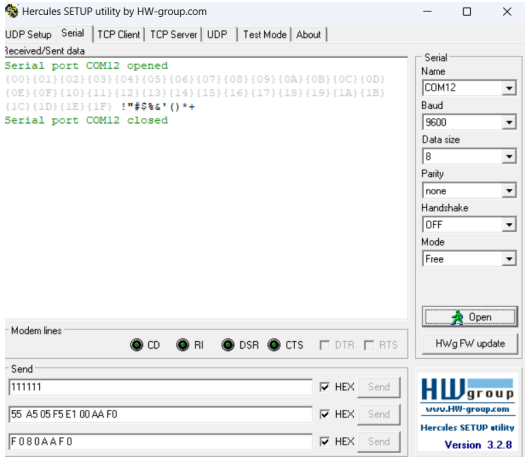


Figure 84: *process of uart transfer*

7 UART_RX Design and verification of serial port receiver module

7.1 Analysis of the principle of serial port reception

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

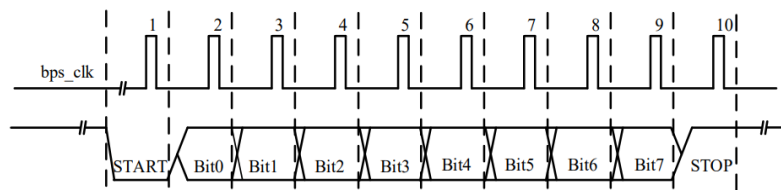


Figure 85: *Process of uart transfer*

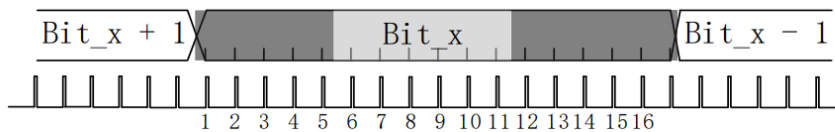


Figure 86: *Taking sample of 1 bit receive*

Because data can be unstable when it change, specially in the dark gray part. The middle of period is relatively stable and represent the correct result. Taking 6 sample and find the probability of high and low. For example 1/1/1/0/1/1 result 1 and 0/1/0/0/0/0 is 0, when 0/0/0/1/1/1 enviroment is bad, data is no reliable , not processed.

7.2 Design and implementation of UART asynchronous serial communication receiving module

7.2.1 Serial port receiving module interface design



Figure 87: *UART_RX receive module*

The signal	I/O	Description of the function
clk	I	The module global clock is 27MHz
reset_n	I	Module global reset signal
data_byte	O	8 bits to be received
baud_set	I	Baud rate to set the signal
uart_rx	I	Serial port sends signal output
rx_done	O	Receives an end signal

Table 6: Table of module receive description

7.2.2 Single-bit asynchronous signal synchronization design

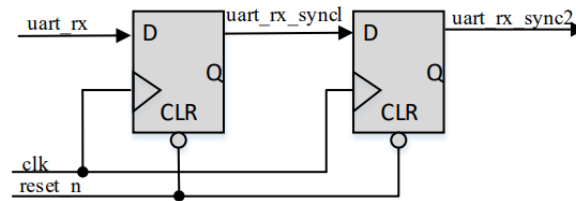


Figure 88: Schematic logic design

```

always@(posedge clk or posedge reset)
if(reset)begin
    uart_rx_sync1 <= 1'b0;
    uart_rx_sync2 <= 1'b0;
end
else begin
    uart_rx_sync1 <= uart_rx;
    uart_rx_sync2 <= uart_rx_sync1;
end
end

```

- Because asynchronous signal, signal is not stable (0 or 1).
- First Synchronization Register (uart_rx_sync1): This register captures the asynchronous UART input signal uart_rx on the rising edge of the clock. If uart_rx changes state close to the clock edge, the output of this register might be metastable.
- Second Synchronization Register (uart_rx_sync2): This register captures the output of the first synchronization register on the next clock edge. By this time, the signal from uart_rx_sync1 is likely to have settled to a stable state, thus reducing the chance of metastability propagating further.

- The double synchronization ensures that any metastable state in `uart_rx_sync1` has time to resolve before it affects the rest of the logic.

7.3 Edge detection

- Using for detect for falling edge to start uart state.

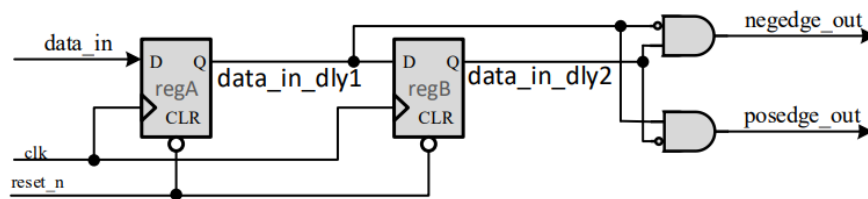


Figure 89: *principle of edge detection*

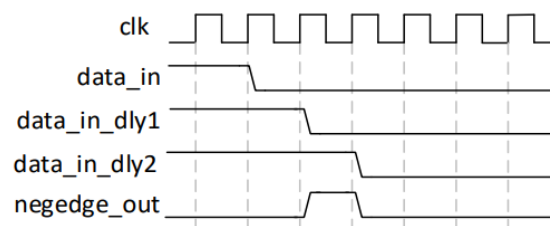


Figure 90: *Falling edge detection waveform*

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

- For find rising edge , `uart_edge=uart_rx_reg1` and `!uart_rx_reg2`

7.4 Design of the sample clock generation module

- If clock frequency = 27MHz, number of baudrate clock bit is 16. (Formula: $(27000000 / (16 \times \text{baudrate})) - 1$)
 - baudrate: 9600 > 324

- 19200->162
- 38400->80
- 57600->53
- 115200->26

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

- Counter for Baud Rate Clock Generation

```
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;
```

- take enough clock for read one sample of bit
- Baud Rate Clock Generation (bps_clk)

```
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;
```

identify when start read a new sample bit.

- Baud Rate Counter (bps_cnt)

```
//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)//(16x10-1)
    rx_done <= 1'b1;
else
    rx_done <= 1'b0;
```

- identify get enough sample of bits.
- WHY CHOOSING CASE =8'd12 and START_BIT>2:
 - because at 6->11 the sample of bit add to "START_BIT", the START_BIT We want to be low to start,position 12 we want to check that START_BIT,if >2 min there are more 3 sample with high mean START_BIT can be high not low.Therefore we stop becasue START_BIT is invalid.

7.5 Design of sampling data receiving module

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

- data_byte_pre is array with 8 register with 3bits ,store value .

7.6 Data status determination module

```
always@(posedge clk or posedge reset)
if(reset)
    data_byte <= 8'd0;
else if(bps_cnt == 8'd159)begin
    data_byte[0] <= data_byte_pre[0][2];
    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
```

- Because the data take 6 samples, so it should occur more than 3 so we take the value at position 3(if =1 mean 4,5,6 samples value of 1.)

7.7 TESTBENCHfor modelsim

```
`timescale 1ns/1ns
`define CLK_PERIOD 37.037
module uart_byte_rx_tb;
    reg clk;
    reg reset_n;
    reg [2:0] baud_set;
    reg [7:0] data_byte_tx;
    reg send_en;
    wire tx_done;
    wire uart_state;
    wire [7:0]data_byte_rx;
    wire rx_done;
    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)
);
initial clk = 1;
always#(`CLK_PERIOD/2)clk = ~clk;
initial begin
    reset_n = 1'b0;
    data_byte_tx = 8'd0;
    send_en = 1'd0;
    baud_set = 3'd0;
    #(`CLK_PERIOD*20 + 1 );
    reset_n = 1'b1;
    #(`CLK_PERIOD*50);
    //send first byte
    data_byte_tx = 8'haa;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;

    @(posedge tx_done)
    #(`CLK_PERIOD*5000);
    //send second byte
    data_byte_tx = 8'h55;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;
    @(posedge tx_done)
    #(`CLK_PERIOD*5000);
    $stop;
```

```
end  
endmodule
```

7.8 Code for testing real device

```
`timescale 1ns/1ns  
`define CLK_PERIOD 20  
module uart_byte_rx_tb;  
    reg clk;  
    reg reset_n;  
    reg [2:0] baud_set;  
    reg [7:0] data_byte_tx;  
    reg send_en;  
    wire tx_done;  
    wire uart_state;  
    wire [7:0] data_byte_rx;  
    wire rx_done;  
    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)  
    );  
    initial clk = 1;  
    always#(`CLK_PERIOD/2)clk = ~clk;  
    initial begin  
        reset_n = 1'b0;  
        data_byte_tx = 8'd0;  
        send_en = 1'd0;
```

```

    baud_set = 3'd0;
    #(`CLK_PERIOD*20 + 1 );
    reset_n = 1'b1;
    #(`CLK_PERIOD*50);
    //send first byte
    data_byte_tx = 8'haa;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;
    @(posedge tx_done)
    #(`CLK_PERIOD*5000);
    //send second byte
    data_byte_tx = 8'h55;
    send_en = 1'd1;
    #`CLK_PERIOD;
    send_en = 1'd0;
    @(posedge tx_done)
    #(`CLK_PERIOD*5000);
    $stop;
end
endmodule

```

7.9 Simulation

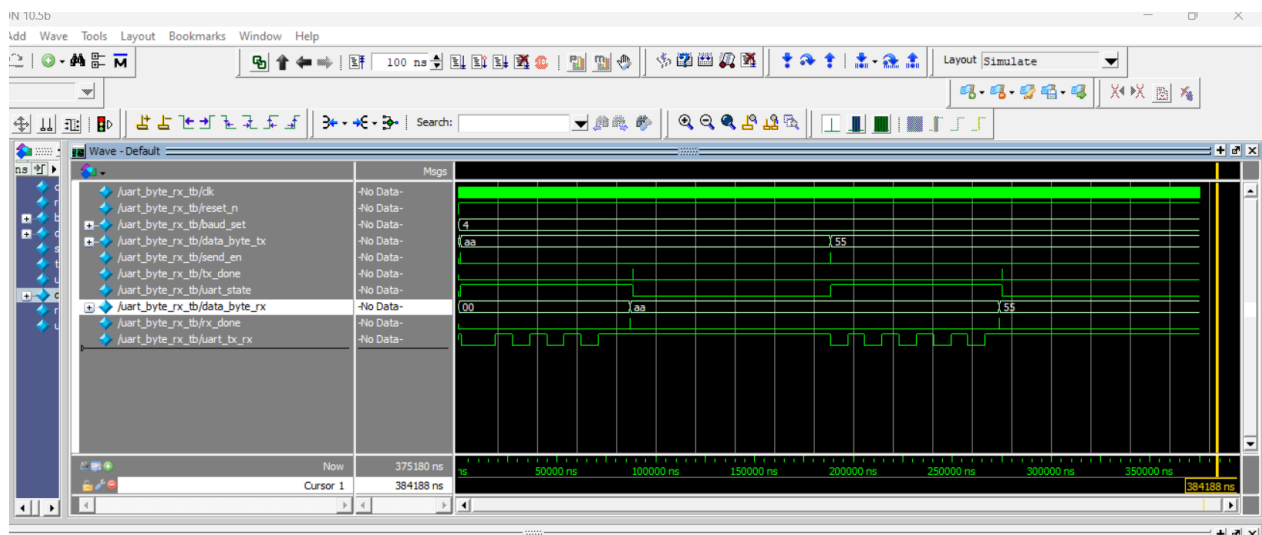


Figure 91: *UART receiver simulation with modelsim*

7.10 Testing with board

- Using the hercules software.

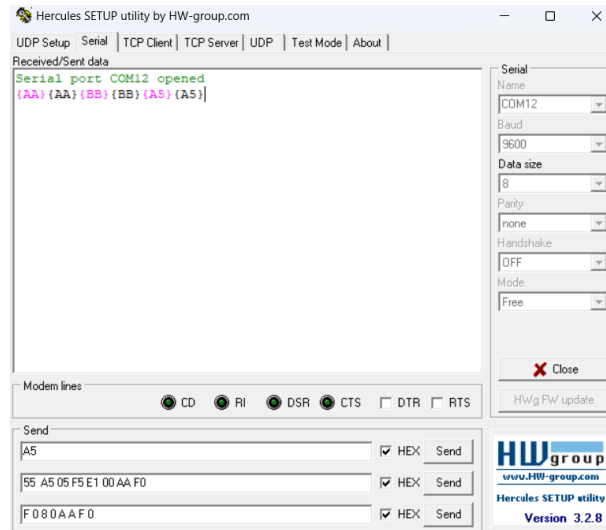


Figure 92: *UART receiver testing with board*

8 UART_loopback Design and verification of multi-byte serial transceiver module

8.1 Principle and design of multi-byte serial port transmission

- In order to realize multi-byte serial port transmission, taking the commonly used 8N1 configuration as an example (i.e., eight data bits, no parity, and one stop bit),
- For example, if I want to use the serial port to send 16-bit data 0x1234, according to the above idea, the correct transfer should be split into two parts, the first 0_0100_1000_1 and the second 0_0010_1100_1. After receiving the data, the destination serial port will parse the data according to 8N1 to obtain 8'b0001_0010 (0x12) and 8'b0011_0100 (0x34) twice. Stitch together two 8-bits of data to get the 0x1234

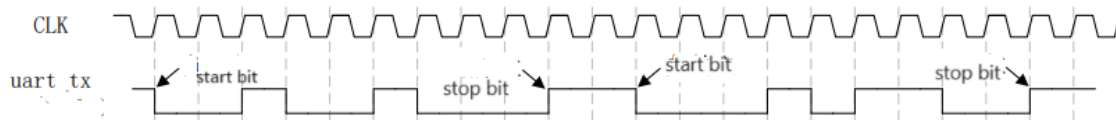


Figure 93: Serial port sends 0x1234

- Therefore, multi-byte serial port transmission can be regarded as the serial port transmission process of multiple consecutive single-byte data
- 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 transmitter module to send them one by one, and the serial port sending of multi-byte data can be realized

8.1.1 The principle of multi-byte serial port transmission

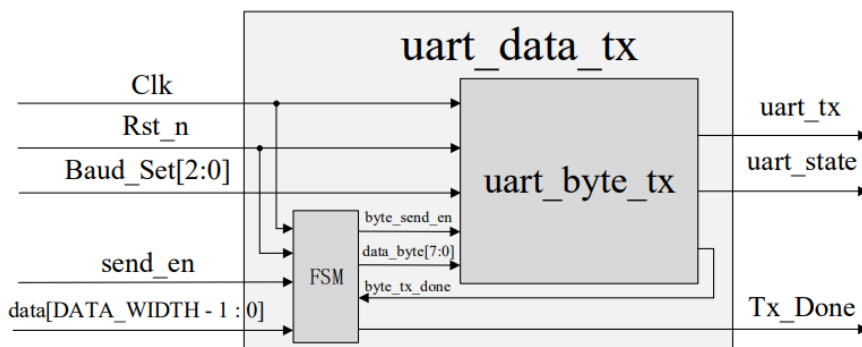


Figure 94: Multi-byte serial port transmitting module

- `uart_data_tx`
 - The top-level module responsible for managing the transmission of multi-byte data.
 - integrates a lower-level module, `uart_byte_tx`, which handles the transmission of single-byte data
 - includes a Finite State Machine (FSM) to manage the control flow and ensure the proper sequencing of data bytes
- `uart_byte_tx`
 - Responsible for transmitting a single byte of data.
 - provides status signals such as `byte_tx_done` (indicating the completion of a single-byte transmission) and `uart_state` (indicating the current state of the UART transmitter).
- FSM(Finite State Machine)
 - The FSM coordinates the flow of data between `uart_data_tx` and `uart_byte_tx`.
 - generates control signals (`byte_send_en`, `data_byte[7:0]`) to manage the transmission of each byte.
 - monitors the `byte_tx_done` signal to determine when to send the next byte and when the entire data transmission is complete.
- Declaration
 - `Clk`: Clock signal to synchronize the operations.
 - `Rst_n`: Reset signal (active low) to initialize the module.
 - `Baud_Set[2:0]`: Baud rate setting for configuring the transmission speed. `send_en`: Enable signal to start the transmission process.
 - `data[DATA_WIDTH-1:0]`: Input data to be transmitted.
 - `uart_tx`: Output signal carrying the transmitted serial data.
 - `uart_state`: Status signal indicating the state of the UART transmitter.
 - `Tx_Done`: Signal indicating the completion of the entire data transmission.

8.2 Design of multi-byte serial port transmitting module

- After determining the overall function and structure of the design, the next step is the code implementation of the design. The first is the instantiation of the single-byte serial port transmitting module, and 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)
);
```

- Setting the bit width of the input data of the multi-byte serial port sending module to 8.

```
parameter DATA_WIDTH = 8;
```

- If we want the module to send multiple bytes at once, for example, 8 bytes, we only need to define the input data at the top level with a bit width of 64, and the parameter can be passed to the submodule to override the constant value.
- For a multi-byte serial transmitter module, when the input send_en is valid, the transmission begins, and the state machine generates one byte of transmit data and sends an enable signal to the single-byte serial transmitter. The state machine then waits 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 judge whether the transmission is complete based on the signal count. If the transmission is not complete, the next byte of data and the enable signal are transmitted. If the transfer is complete, the state machine needs to generate a send completion signal tx_done to mark the end of the multi-byte send
- A state machine needs at least four states to achieve the above functionality. The four states are used to wait for the send to be enabled, to initiate a single-byte data send, to wait for the single-byte data to be sent, and to check whether all data is transmitted. The corresponding four status codes are defined as follows

```
localparam S0 = 0; //Wait for the request to be sent
localparam S1 = 1; //initiate a single-byte data send
localparam S2 = 2; //Wait for the single-byte data to be sent
localparam S3 = 3; // Check that all data has been sent
```

- The next step is to implement the functionality of the four states, the first is the state S0, which needs to wait for the send request signal send_en to receive the incoming data and make the state jump to S1 when the send_en is valid. The corresponding codes are as follows:

```
S0:
begin
data_byte <= 0;
cnt <= 0;
Tx_Done <= 0;
if(send_en)begin
state <= S1;
data_r <= data;
end
else begin
state <= S0;
data_r <= data_r;
end
end
```

- Next is the S1 state, which needs to generate a single-byte transmit enable signal along with a single-byte data, and make the state jump to S2. Considering that the endianness of data will have different requirements in different application scenarios, we can define a constant by using a parameter MSB_FIRST control the size-side end-of-side Indianness of the data. Here's the code:

```
parameter MSB_FIRST = 1;
S1:
begin
byte_send_en <= 1;
if(MSB_FIRST == 1)begin
data_byte <= data_r[DATA_WIDTH-1:DATA_WIDTH - 8];
data_r <= data_r << 8;
end
else begin
data_byte <= data_r[7:0];
```

```
data_r <= data_r >> 8;
end
state <= S2;
end
```

- This is followed by the S2 state, which waits for the single-byte data to be sent, i.e., the byte_tx_done signal from the single-byte serial port sending module. When the signal is valid, it means that the single-byte data has been sent, and we can count the amount of data that has been sent based on the signal, and jump the status to S3. code

```
S2:
begin
byte_send_en <= 0;
if(byte_tx_done)begin
state <= S3;
cnt <= cnt + 9'd8;
end
else
state <= S2;
end
```

- Finally, there is the S3 state, which can be Tx_Done generated by comparing the amount of data sent with the amount of input bit width. When the amount of data sent is greater than or equal to the input bit width, it means that all the data has been sent, and a Tx_Done signal can be generated and the status can jump to S0 to wait for the next transmission to start. When the amount of sent data is less than the input bit width, the transmission has not yet ended, and the status is jumped to S1 to prepare for the next byte of data transmission. 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 transmitter module. As for whether the module can work properly, further verification is needed, so we need to create a simulation stimulus file to simulate and verify the module.

8.3 Principle and design of multi-byte serial port reception

- When the serial port receiving module is working, it will receive continuous serial data, and output the effective data in bytes. And the so-called multi-byte serial port reception, is that after the serial port receiving module receives the data, it no longer outputs with single-byte data, but splices the data and combines it into a multi-byte data post output.
- Combined with the multi-byte serial port transmitter module we designed earlier, to realize the design of the multi-byte serial port receiving module, we can instantiate the single-byte serial port receiving module, and then design a state machine to splice the data output by the single-byte serial port receiving module. At the same time, based on the received completion signal generated by the single-byte serial port receiving module, count the amount of data that has been received (spliced) at present. When the spliced data reaches the specified bit width, the data will be output.

8.3.1 Multi-byte serial port receiving principle

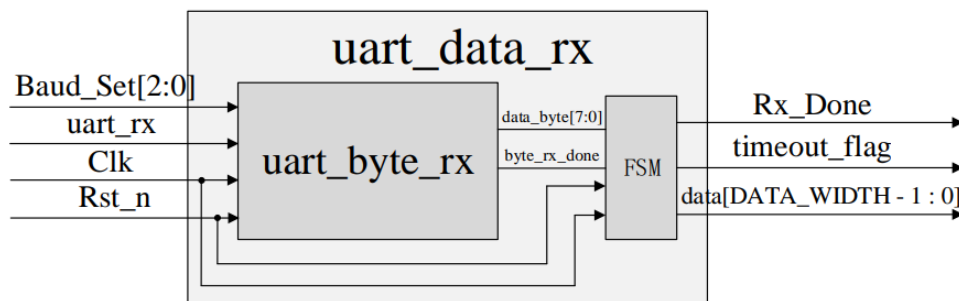


Figure 95: Block diagram of a multi-byte serial port receiving module

- Once the overall structure of the design has been determined, the next step is the code implementation of the design. The first is the instantiation of the single-word serial port receiving module, and 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),
.Rx_Done(byte_rx_done)
);
```

- In order to adapt the design to more application scenarios, we can use parameter to define the output bit width of the multi-byte serial receiver module to 16. In this way, when we call the module, we can modify the output bit width of the module at any time through the top level. Here's the code:

```
parameter DATA_WIDTH = 16
```

- Next, it's time to think about how to generate a receive timeout signal. For serial port reception, different baud rates are used in different application scenarios, and the frequency of data transmitted by the serial port is also different under each baud rate. So how can we tell if the receive is a timeout?
- Here, we refer to the modbus definition of serial transmission timeout. In modbus, the interval between messages is specified to be greater than 3.5 characters, i.e. the transmission is considered new if it exceeds 3.5 characters. Based on this, in this design, we believe that when receiving data through the serial port, once the data exceeds 3.5 characters, 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 receive one byte of data normally is $1/115200 * 10^9 * (1 + 8 + 1) = 86.805us$. Then, if the serial port does not receive data for more than $86.805 * 3.5 = 300.8175us$, it can be regarded as a timeout.
- In this design, we are using a 50MHz clock, which is 20ns per clock cycle. The time it takes for a serial port to receive one byte of data can be determined by counting the rising edge of the clock between the two serial ports receiving the completed signal. Therefore, we can find the receive timeout when the count value is greater than $300.8175 * 10^3 / 20 - 1 = 15190$. As soon as we receive a timeout, we let it generate a timeout flag.
- In this design, we are using a 50MHz clock, which is 20ns per clock cycle. The time it takes for a serial port to receive one byte of data can be determined by counting the rising edge

of the clock between the two serial ports receiving the completed signal. Therefore, we can find the receive timeout when the count value is greater than $300.8175 * 10^3 / 20 - 1 = 15190$. As soon as we receive a timeout, we let it generate a timeout flag.

- As for when the counter should be counted, we can make the counter work at the beginning of each byte receive and be cleared to zero when the byte receive is complete. Specifically, the low-level start signal (!) when transmitting through the serial port is used. `uart_rx`), and the `byte_rx_done` signal generated by a single-byte serial receiving module after receiving a byte. Therefore, the design code for judging and generating the serial port receiving timeout flag signal is as follows:

```
parameter TIMEOUT = 15190; //115200 is 15190, 9600 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;
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;
else
timeout_cnt <= timeout_cnt + 1'd1;
end
```

- After designing the receive timeout signal, the next step is the state machine. Combined with

the previous principle analysis and structural diagram, we can know that the workflow of the state machine in this design is as follows:

1. Wait for the receipt completion signal of the single-byte receiving module, cache the first byte of data, count the cached data
 2. Start timeout counting, if the next transmission completes the signal before the reception times out, a Rx_Done signal is generated, and the transmission ends. If it does not time out, the new byte of data is concatenated and the cached data is counted.
 3. Compare the bit width of the cached data with the output data and check whether all data has been received. If completed, a Rx_Done signal is generated and the transmission ends. If you don't finish, proceed to step 2
- Therefore, the whole state only needs three states to implement the function, and the state is defined and initialized as follows:

```
localparam S0 = 0; // Wait for a single-byte received completion signal
localparam S1 = 1; // Determine whether the receive timed out
localparam S2 = 2; // Check if all data has been received
```

- The next step is the implementation of the S0 state, considering that in different application scenarios, there may be different endianness requirements for the data, we can use parameter to define a constant MSB_FIRST control the size-side endianness of the data. The status codes are as follows:

```
S0:
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
```

- This is followed by an implementation of the S1 state, with the following code:

```
S1:  
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
```

- Finally, there is the implementation of the S2 state, which is as follows:

```
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
```

- At this point, we have completed the design of the multi-byte serial port receiver module. As for whether the module can work, further validation is required, so the next step is to create a simulation stimulus file to simulate and verify the module.

8.4 Create a top-level package for testing

- Although we have verified the feasibility of the functions of the two modules through simulation tests, there will always be discrepancies between simulation and the actual application situation, considering that simulation is the ideal condition after all. Therefore, the design also needs to undergo board-level testing to verify its functional integrity.
- In order to facilitate the test, here we create a new top-level, instantiation multi-byte serial transceiver module, and connect the output of the receiving module with the input of the sending module. In this way, the data received by the receiving module will be sent out through the sending module, and then through the serial port debugging software on the computer side, a simple serial port loopback can be realized. By judging whether the data in the loop is consistent, you can know whether the module function is realized.
- 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 [1:0] 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'd0)
);
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'd0),
.uart_tx(uart_tx),
.Tx_Done(led[1]),
.uart_state()
);
endmodule
```

- In the top layer, the DATA_WIDTH is set to 32, the baud rate is 9600, and the timeout signal, send completion signal, and serial port status signal are connected to the LED light for display.

8.5 Testing with board

Base on the number of DATA_WIDTH you are setting,when fpga receive enough data it will send you again the data in hercules.In my example I setting 32bit data.The result of example is like figure below:

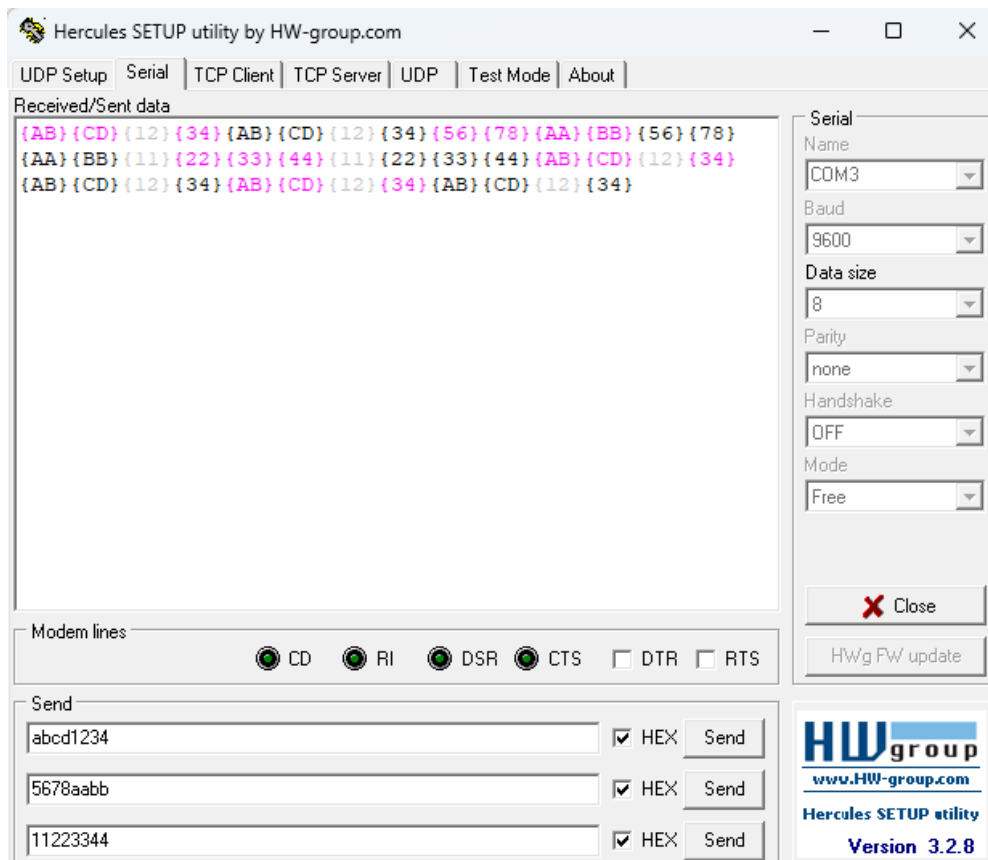


Figure 96: *Testing UART multiple byte resut*

9 References

1. Gowin Semiconductor Corp., *Gowin HyperRAM Memory Interface IP User Guide*, Version 2.1E.
2. Gowin Semiconductor Corp., *Gowin Video Frame Buffer IP User Guide*, Version 2.1E.
3. Gowin Semiconductor Corp., *Gowin DVI TX IP User Guide*, available with IDE package.
4. Gowin Semiconductor Corp., *Clock Resource and PLL Guide*, documentation for GW_PLLVR, TMDS_PLLVR, and CLKDIV modules.
5. CoreCourse.cn — FPGA learning portal with practical Verilog tutorials and IP integration: <http://www.corecourse.cn>
6. Gowin EDA Tools Help System — Built-in IDE documentation for IP configuration, timing constraints, and simulation setup.
7. Gowin Semiconductor Corp., *Gowin HDMI/DVI RX and TX IP User Guide*, describing HDMI/DVI Rx and Tx functionality and integration in Gowin FPGAs.
8. Gowin Semiconductor Corp., *Gowin MIPI D-PHY RX/TX Advance IP User Guide*, IPUG948-2.1.1E, covering MIPI CSI/DSI physical layer.
9. Gowin Semiconductor Corp., *Gowin MIPI D-PHY RX/TX IP User Guide*, IPUG112-2.02E, earlier version of MIPI D-PHY documentation.
10. Gowin Semiconductor Corp., *Arora V Hardened MIPI D-PHY User Guide*, UG296-1.1.1E, MIPI for Arora-V devices
11. Gowin Semiconductor Corp., *White Paper: Gowin MIPI Solution Capability Overview*, WP1153-1.1E, a summary of MIPI IP offerings.