



FPGA Made Easy

Start with Kiwi board and Gowin EDA tool

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

HO CHI MINH CITY, JULY 2025

Contents

1	Preface	8
2	Creating project with Gowin EDA tool	9
2.1	Creating verilog file	11
2.1.1	Synthesize	13
2.1.2	Hierarchy	14
2.2	Physic constraints	15
2.2.1	FloorPanner	15
2.3	Place and Route	16
2.4	Board-level validation	17
3	Gowin and Modelsim simulation experiments	19
3.1	Modelsim compiles the Gowin device library	19
3.1.1	Create a new Modelsim library	19
3.1.2	Compile Gowin's device library files	20
3.1.3	Add a library to the Modelsim default library list	22
3.2	Modelsim simulation validation process	23
3.2.1	Write Verilog simulation design code	23
3.2.2	Create a Modelsim project and add simulation files	24
3.2.3	Compile the simulation file	25
3.2.4	Configure the simulation environment	26
3.2.5	Simulation interface waveform observation	28
3.2.6	Save the waveform file	29
3.2.7	Reopen the simulation project	30
4	Counters for sequential logic circuit design	32
4.1	How the counter works	32
4.1.1	Project establishment	33
4.1.2	Code design	34
4.2	Functional simulation verification	35
4.2.1	Build a Modelsim simulation project	36
4.2.2	Configure the simulation environment	37
4.2.3	Simulation interface waveform observation	38
4.3	Board-level debugging and validation	38
4.4	Reflection and summary	39

5	Passive Buzzer Drive Design and Verification	40
5.1	Passive Buzzer Circuit Design	40
5.2	Principle of Passive Buzzer Driving	41
5.3	PWM Generator Module Design	42
5.4	Simulation	44
5.5	Board-Level Debugging and Verification	46
5.6	Selection of pre-reload values and scale download verification	48
6	Key filtering	51
6.1	Key physical structure and circuit design	51
6.2	Key physical structure and circuit design	51
6.3	The state machine implements key debounce	53
6.3.1	Module interface design	53
6.3.2	State machine transfer state as well as conditional design	53
6.3.3	Key input signal edge detection	54
6.3.4	Counter design	55
6.3.5	State machine design	55
6.4	Testbench design and simulation testing	58
6.4.1	The design of the key model key_model	58
6.4.2	Use of tasks	59
7	UART_TX Design and verification of serial port transmitter module	62
7.1	UART key parameters and time series diagrams	62
7.2	Design and implementation of UART asynchronous serial communication transmiss- ion module	63
7.3	Baurate clock module design	64
7.4	Data output module design	66
7.5	TESTBENCH code for modelsim	67
7.6	Coding test for real device	68
7.7	Simulation	69
7.8	Testing with board	70
8	UART_RX Design and verification of serial port receiver module	71
8.1	Analysis of the principle of serial port reception	71
8.2	Design and implementation of UART asynchronous serial communication receiving module	71
8.2.1	Serial port receiving module interface design	71
8.2.2	Single-bit asynchronous signal synchronization design	72
8.3	Edge detection	73

8.4	Design of the sample clock generation module	73
8.5	Design of sampling data receiving module	75
8.6	Data status determination module	77
8.7	TESTBENCHfor modelsim	77
8.8	Code for testing real device	79
8.9	Simulation	80
8.10	Testing with board	81
9	UART_loopback Design and verification of multi-byte serial transceiver module	82
9.1	Principle and design of multi-byte serial port transmission	82
9.2	The principle of multi-byte serial port transmission	82
9.3	Design of multi-byte serial port transmitting module	83
9.4	Principle and design of multi-byte serial port reception	87
9.4.1	Multi-byte serial port receiving principle	87
9.5	Create a top-level package for testing	92
10	I2C	94
10.1	I2C protocol parsing	94
10.2	I2C device ground	95
10.3	I2C Memory Address	95
10.4	I2C Write Timing	96
10.4.1	Single-byte Write Timing	96
10.4.2	Continuous Write Timing (Page Write)	98
10.5	I2C Read Timing	99
10.5.1	Single-Byte Read Timing	99
10.5.2	Continuous Read Timing (Page Read)	100
10.6	i2c_bit_shift Module Design	101
10.6.1	Bit Transfer Unit Interface Design	101
10.6.2	Design of Position Transmission Unit State Machine	102
10.6.3	Bit Transfer Unit State Machine Design	103
10.6.4	Key Logic Design of Bit Transfer Unit	104
10.7	Design and Explanation of i2c_bit_shift Module	105
10.8	Design and Explanation of i2c_bit_shift Module	110
10.8.1	I2C control	114
10.9	Example	115

11 Usage of the PLL IP Core	116
11.1 Principle of the PLL Circuit	116
11.2 Introduction to Phase-Locked Loops (PLLs)	116
11.3 PLLs in FPGA Architectures	117
11.4 Impact of PLLs on HDMI Interfaces	117
11.5 Impact of PLLs on Camera Interfaces	118
11.6 Considerations and Best Practices	118
11.7 PLL IP module implementation	119
11.8 Conclusion	120
12 VGA	121
12.1 Overview of VGA Development	121
12.1.1 Chapter Objective	121
12.1.2 Introduction to VGA	121
12.1.3 VGA Signal Pins	121
12.1.4 Generating Analog Color Signals from FPGA	121
12.1.5 Controlling VGA Timing with FPGA	122
12.1.6 Why Learn VGA Before HDMI?	122
12.1.7 Conclusion	123
12.2 VGA Display Imaging Principle	123
12.2.1 Raster Scan Principle	123
12.2.2 Frame Structure and Timing Intervals	124
12.2.3 Synchronization Signals	126
12.2.4 Active Display Region	126
12.2.5 Counter-Based Implementation in FPGA	126
12.2.6 Timing Accuracy	126
12.2.7 Conclusion	127
12.3 VGA Timing Detail	127
12.3.1 VGA Timing Overview	127
12.3.2 Total Timing Calculation	127
12.3.3 Timing Parameters Table	128
12.3.4 Resolution vs. Total Frame Size	128
12.3.5 Importance of Timing Accuracy	129
12.3.6 640*480 resolution VGA controller timing analysis	129
12.4 Module design	130
12.4.1 Line Scan Counter Design	131
12.4.2 Field scanning counter design	131
12.5 Synchronization signal design for operations	131

12.5.1	Output data	132
12.5.2	Output row and column scanning position	133
12.5.3	Output data latch clock signal	133
12.5.4	VGA Controller Design Example	133
12.5.5	Simulation	136
13	Multi-resolution adaptive VGA controller design	139
13.1	The design purpose of a multi-resolution adaptable VGA controller	139
13.2	VGA timing parameters based on conditional compilation	141
13.3	Design of VGA controller with multi-resolution support based on conditional compilation	145
14	FPGA-based HDMI/DVI display	148
14.1	FPGA-Based DVI Circuit Design	148
14.2	The differences and connections between HDMI and DVI	149
14.3	Introduction to DVI Data Link	150
14.4	Principles and Implementation of TMDS	151
14.5	Principle of TMDS Minimal Transmission Coding	153
14.6	Principle of Minimizing Transmission Implementation	153
14.7	TMDS encoding implementation	156
14.8	Serial Transmission Module	159
14.8.1	Principle of Serial Transmission	159
14.8.2	FPGA Implementation of DDR Interface	160
14.8.3	TMDS data bits correspond to the DDR interface	162
14.8.4	Implementation of serial transmission module coding	163
14.9	DVI transmitter implementation	167
14.10	Display Color Bar Experiment Based on DVI Interface	170
15	Camera OV2640 and HDMI Implementation	172
15.1	DVI Transmit IP implementation	173
15.2	HyperRAM Memory Interface IP Structure	174
15.3	Video Frame Buffer IP	176
15.4	Phase locked loop(PLL) and clock divide setting	178
15.4.1	Clock_Div for pixel clock HDMI	179
15.5	Camera OV2640 Module	180
15.5.1	Module OV2640_Registers Description	181
15.5.2	Module OV2640_Register	187
15.5.3	Module OV2640_Controller	188
15.5.4	Syn_gen Module	191

15.5.5	Top module	194
15.6	Physical constraint setting	197
15.6.1	Result	198
16	References	199

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 4K 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 4K 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 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.

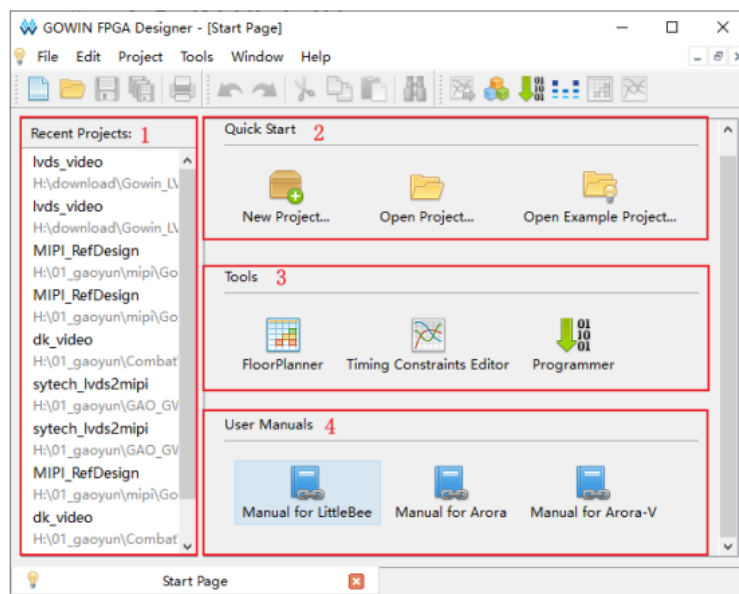


Figure 2: Gowin EDA tool

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

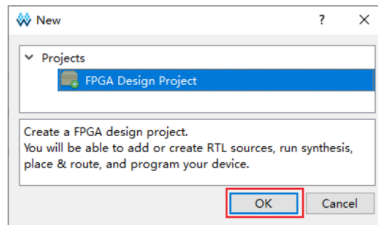


Figure 3: *Creating project interface*

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

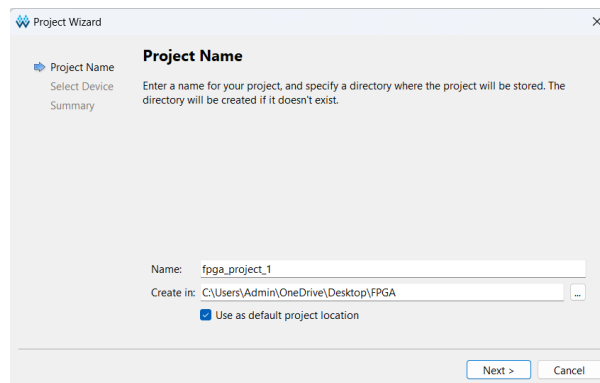


Figure 4: *Filling name and path project*

- 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.

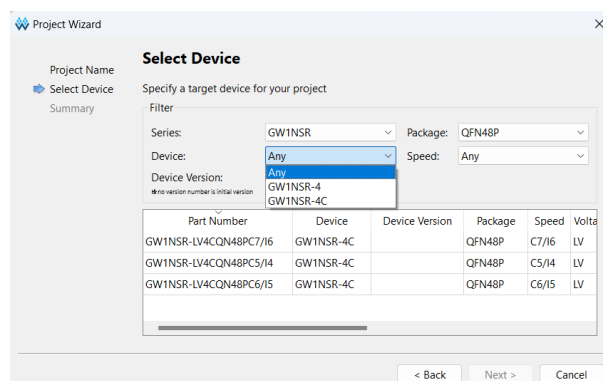


Figure 5: *Select your device*

- 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.

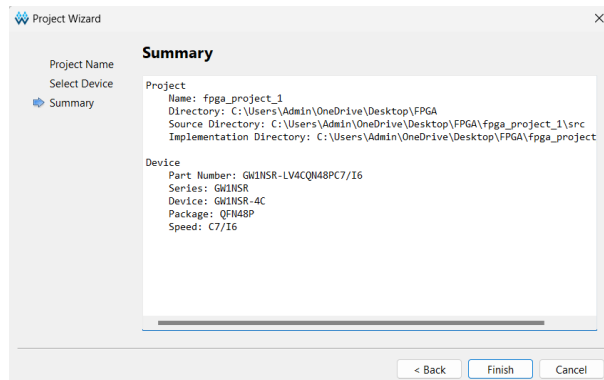


Figure 6: Summary information

- Next, you will see the Gowin design interface.

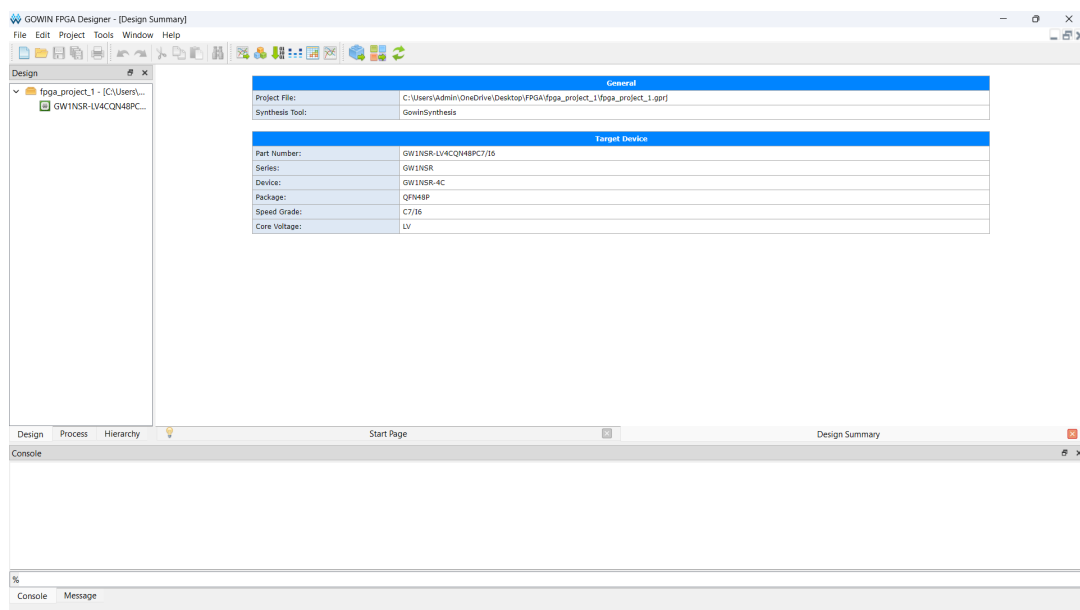


Figure 7: Gowin design interface

2.1 Creating verilog file

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

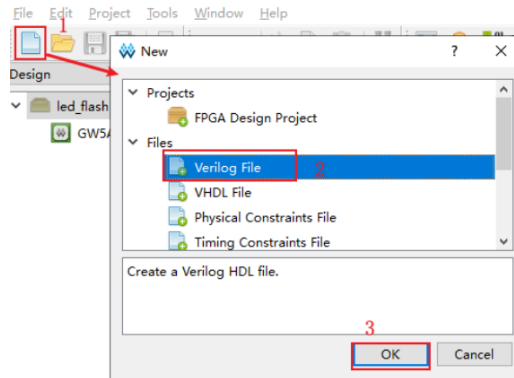


Figure 8: Add the Verilog source file

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

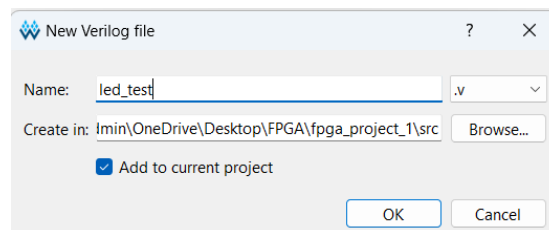


Figure 9: Fill name and path of verilog file

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

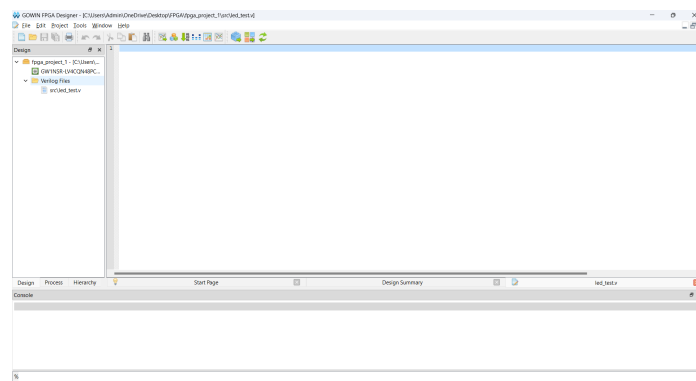


Figure 10: Verilog design interfaces

Coding example:

Listing 1: LED blinking after 1s delay example

```
module led_flash(  
    input clk, //27MHZ  
    input rst_n, output reg led
```

```

);
reg [25:0] cnt;
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        cnt <= 26'd0;
    else if (cnt < 26'd26_999_999)
        cnt <= cnt + 1'b1;
    else
        cnt <= 26'd0;
end
always @ (posedge clk or negedge rst_n)
begin
    if (!rst_n)
        led <= 1'b0;
    else if (cnt == 26'd26_999_999)
        led <= ~led;
    else
        led <= led;
end
endmodule

```

2.1.1 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.

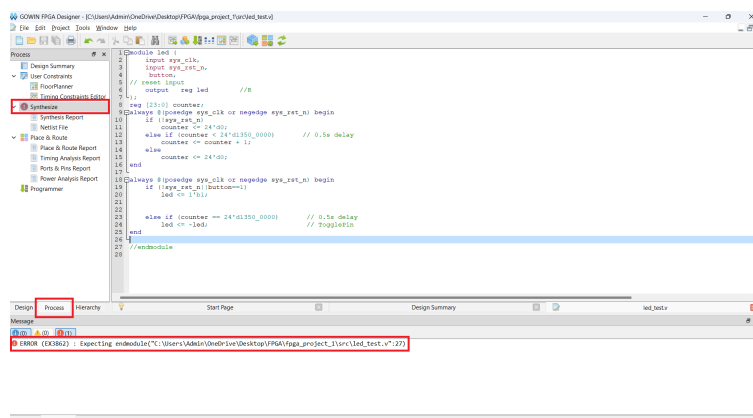


Figure 11: Synthesis process

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.

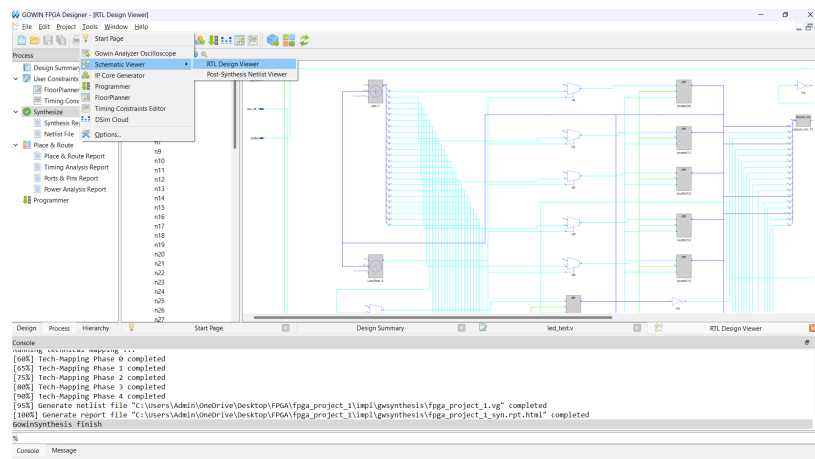


Figure 12: RTL Design Viewer

2.1.2 Hierarchy

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

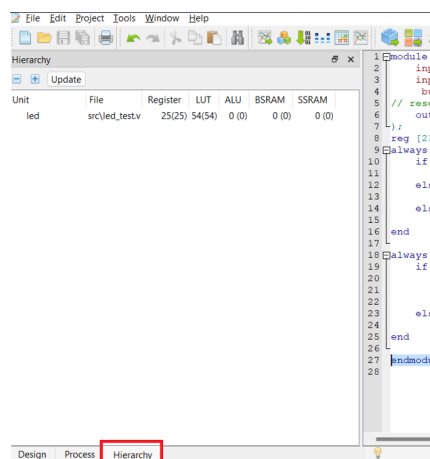


Figure 13: Hierarchy information

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

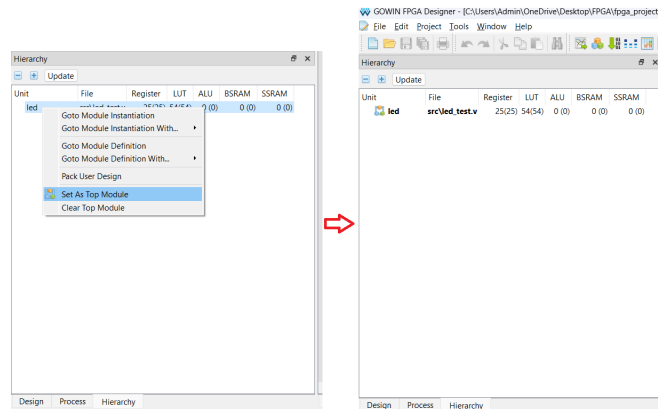


Figure 14: *Hierarchy set*

2.2 Physic constraints

2.2.1 FloorPanner

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

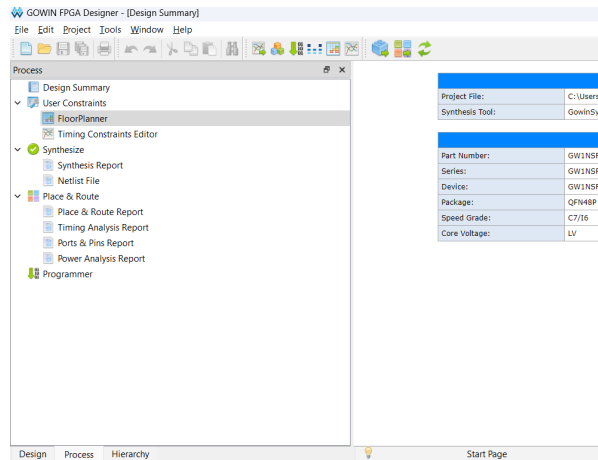


Figure 15: *Choosing floorPanner*

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.

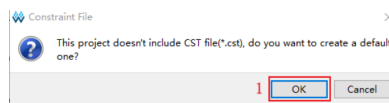


Figure 16: *The Create Physics Constraint File dialog box is displayed*

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

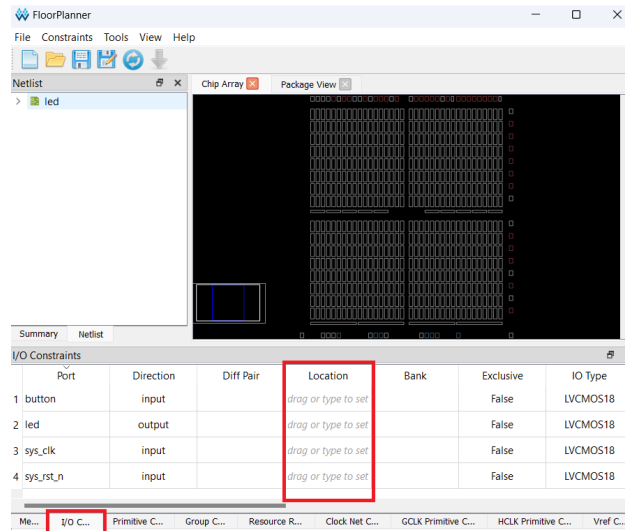


Figure 17: I/O constraint interface

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.

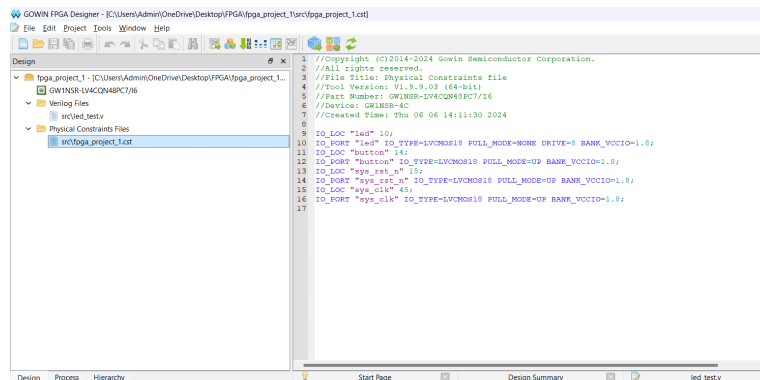


Figure 18: View the contents of the physical constraint file

2.3 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

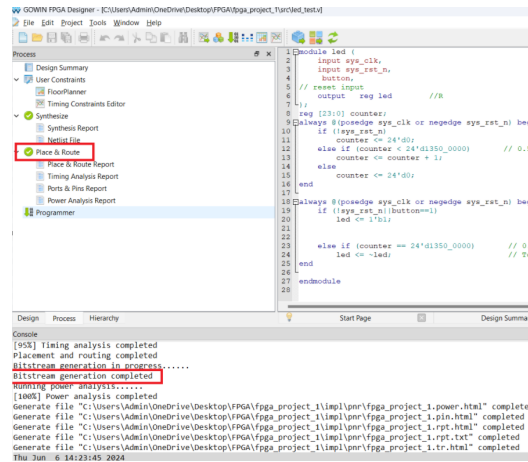


Figure 19: Place-and-route successful

2.4 Board-level validation

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

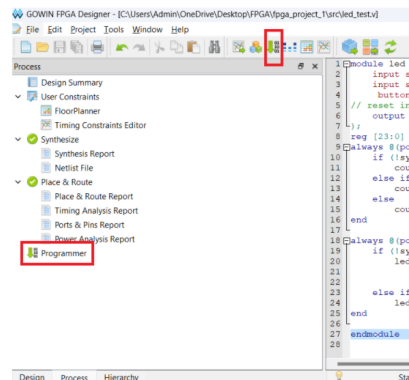


Figure 20: Programmer image

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.

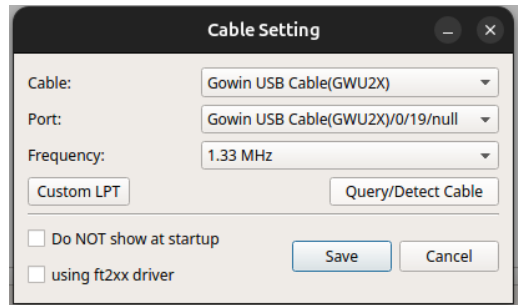


Figure 21: A configuration prompt has been detected

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

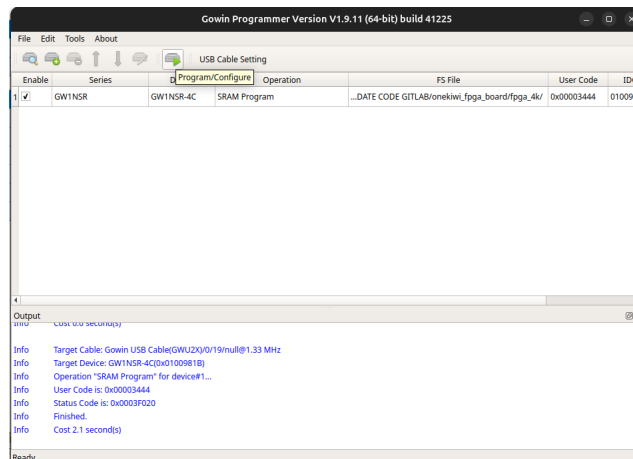


Figure 22: Download the fs file and a message indicating that the file is successfully downloaded

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.
- For the board kiwi4k so we replace gw5a by gw1n in all next steps.

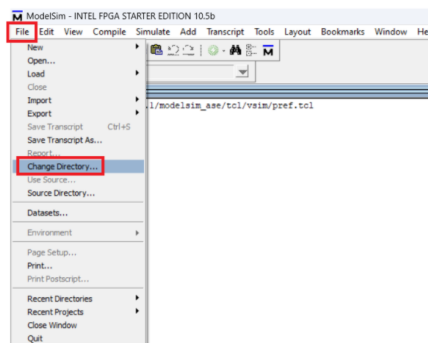


Figure 23: Go to the Path Switching page

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



Figure 24: 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, for example, we are compiling a device of the Gowin 5a series, name it "gw5a", Gowin1n name gw1n and then click OK, as shown below.

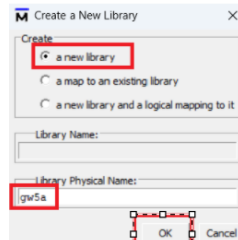


Figure 25: *Create a new library*

- If a gw5a(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 26: *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.

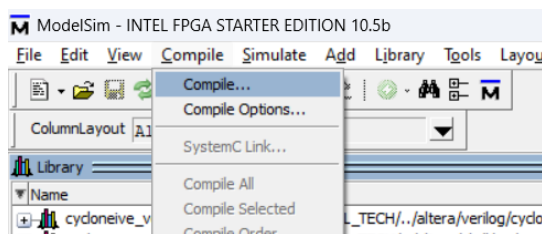


Figure 27: *Open the Compile Library file interface*

- You must select library as the gw5a we just created, and then modify the library path of the gw5a series device with the file path to Gowin software, specifically:
`"...\GowinV1.9.9.03x64\IDE\simlib",`

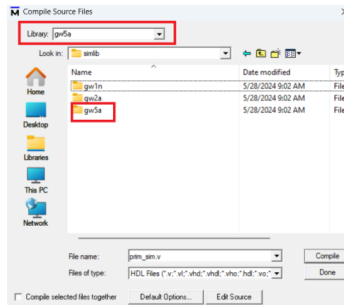


Figure 28: *Locate the Gowin series library file*

- 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.

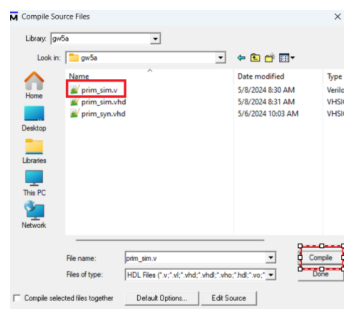


Figure 29: *Locate the library file to compile*

- 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.

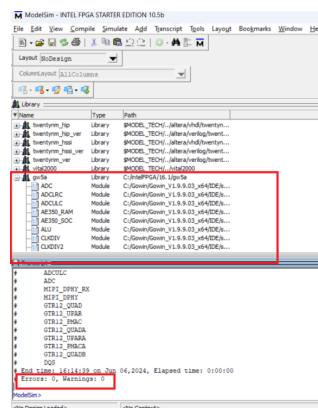


Figure 30: *The series device library has been compiled*

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.

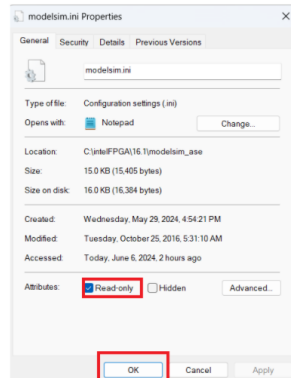


Figure 31: Removed the Moderssim. Ini's read-only attribute

- Then open the file with Notepad or Notepad++, and then about 17 lines of the file, add the specified statement of the gw5a library:

```
gw5a = $MODEL_TECH/../../gw5a
```

and save as shown below.



Figure 32: 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,

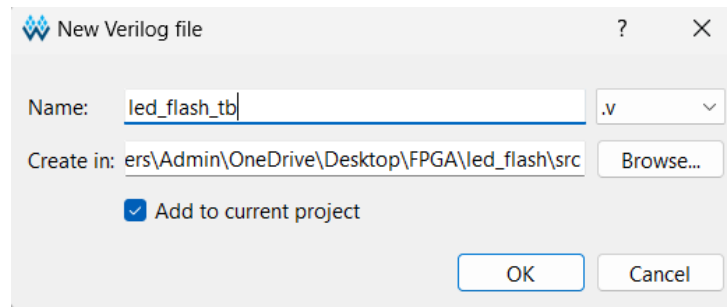


Figure 33: *Create simulation file name*

- 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.

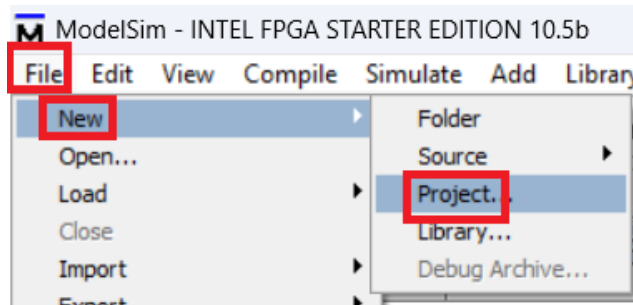


Figure 34: Establish a Modelsim project

- 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

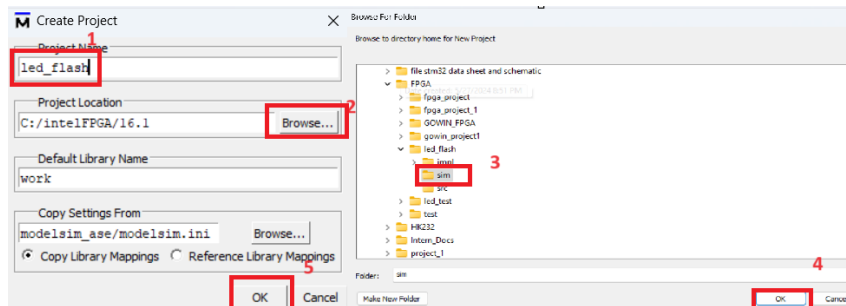


Figure 35: Modelsim project settings

- After that choose "Add Existing File"

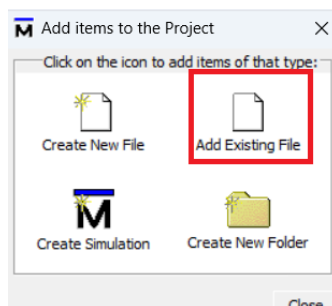


Figure 36: Add File to Project dialog box

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

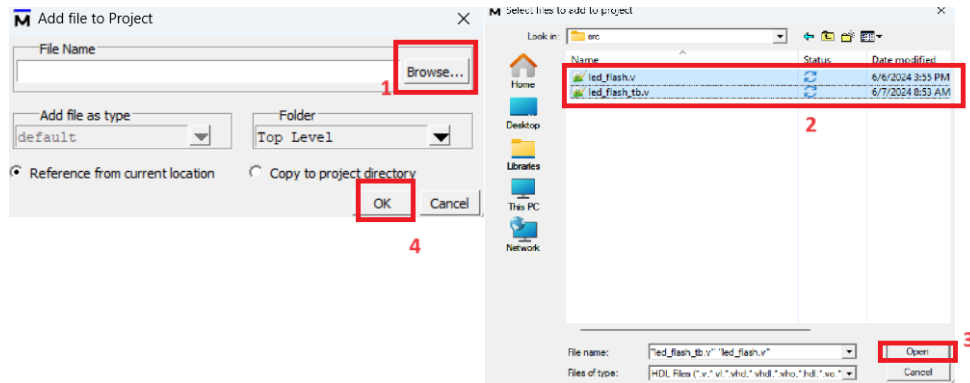


Figure 37: Select the file to add

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

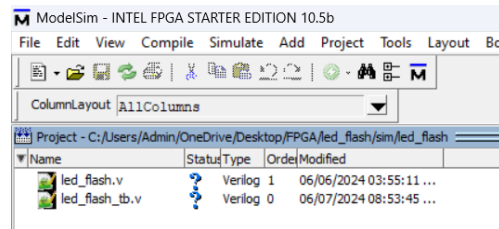


Figure 38: After the file is added, the diagram is displayed

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:

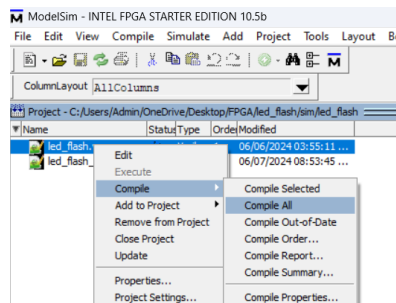
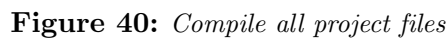


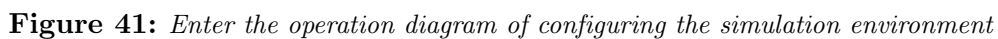
Figure 39: Compile all project files

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

Page 26 / 199



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



- 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



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

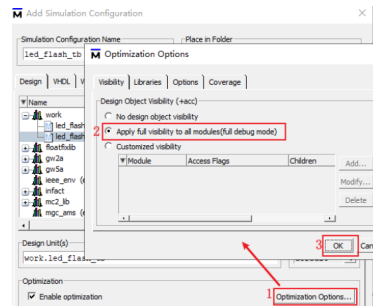


Figure 43: *Simulation environment interface configuration 2*

- Click on the "libraries" tab and click on "Add..." in the "Search libraries(-L)" section. Add the new Gaoyun library file "gw5a", select the library file "gw5a" in "Search Libraries First(-Lf)" as well, and finally click "Save" to save the settings, as shown in Figure 3-24 below.

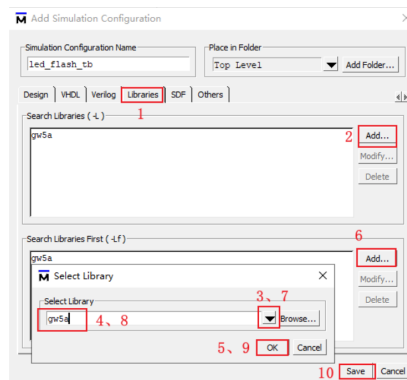


Figure 44: *Simulation environment interface configuration 3*

- 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

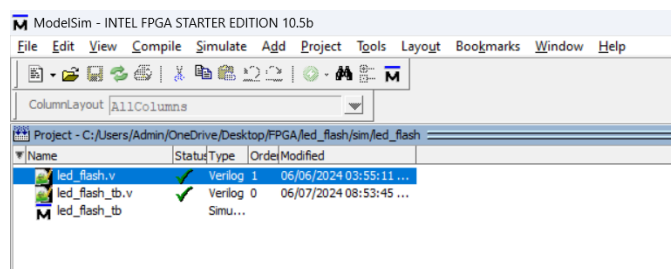


Figure 45: *Schematic diagram of successful generation of simulation file*

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

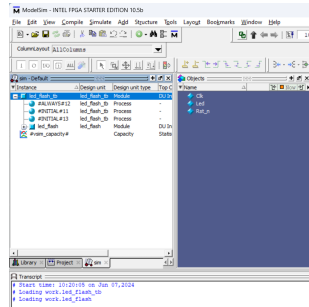


Figure 46: Enter the SIM interface

- 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"

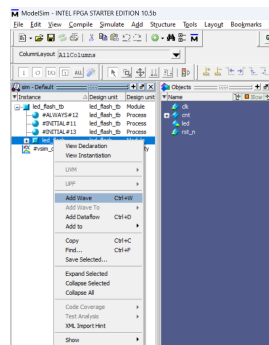


Figure 47: Add the module waveform that needs to be observed

- 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.

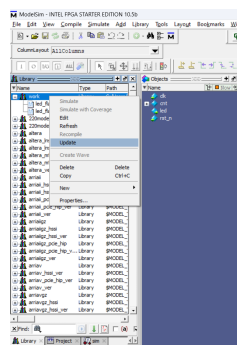


Figure 48: Update the work bar

- Click Run all to start running the simulation

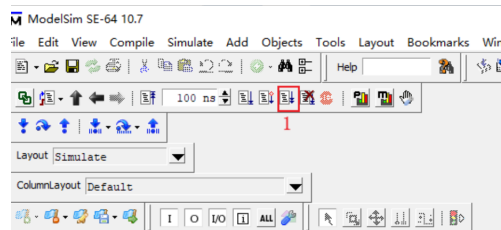


Figure 49: *Start the simulation*

- 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.

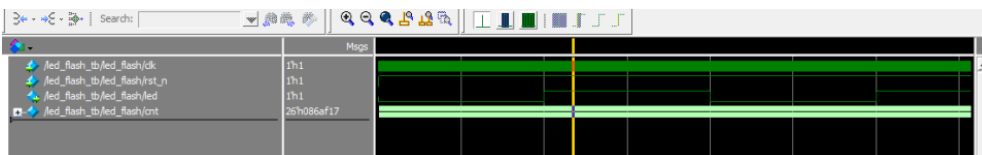


Figure 50: *Observe the waveform*

- 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.

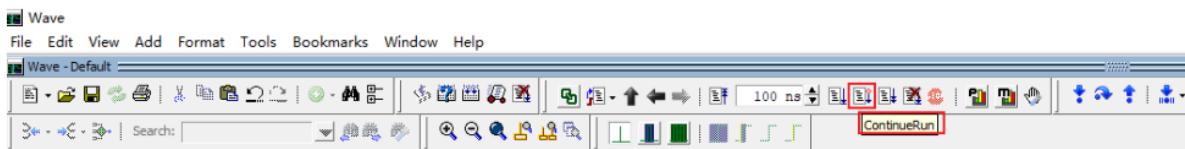


Figure 51: *Wave interface tool button function view*

- 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.

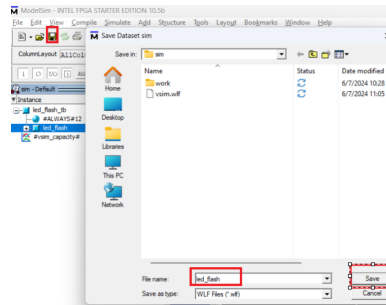


Figure 52: *Saving the .wlf file*

- 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.

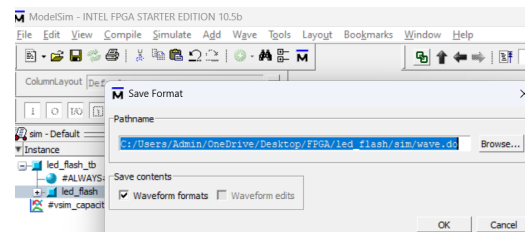


Figure 53: *Save the waveform file*

3.2.7 Reopen the simulation project

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

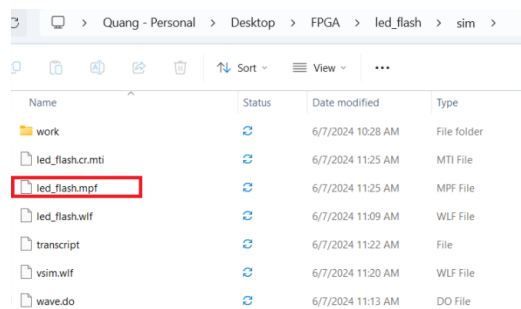


Figure 54: *Reopen the simulation project*

- 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

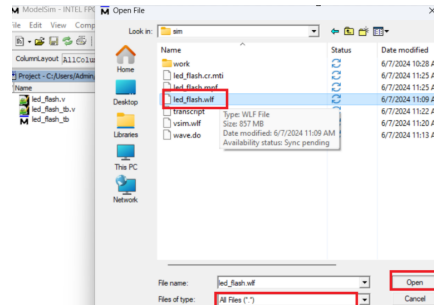


Figure 55: *Open the saved .wlf file*

- 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.

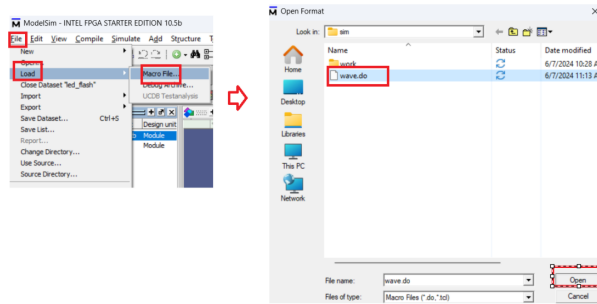


Figure 56: *Open the saved waveform file*

4 Counters for sequential logic circuit design

CHAPTER GUILD

A sequential logic circuit is a steady-state output of a circuit at any time that depends not only on the current input, but also on the state formed by the input at the previous moment. This is the opposite of a combinatorial logic circuit, where the output of the combinatorial logic is only a function of the current input. In other words, sequential logic has storage elements to store information, while combinatorial logic does not.

There are many types of sequential logic circuits, and in this section, we will use the most commonly used counters as examples to learn simple sequential logic circuit design, and compare the differences with combinatorial logic circuits. Here a counter is designed so that the LED status on the ONEKIWI 4K board is flipped every 500ms. The crystal oscillator on the ONEKIWI4K development board is 27MHz, which means that the clock cycle is 37,037ns, so that $500\text{ms} = 500_000_000\text{ns} / 37.037\text{ns}$ 13_500_000 can be calculated, that is, a counter count of 13_500_000 times is required, that is, a counter of at least 25 bits is required. And every time the count number reaches zero, it needs to be cleared and recounted.

4.1 How the counter works

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.

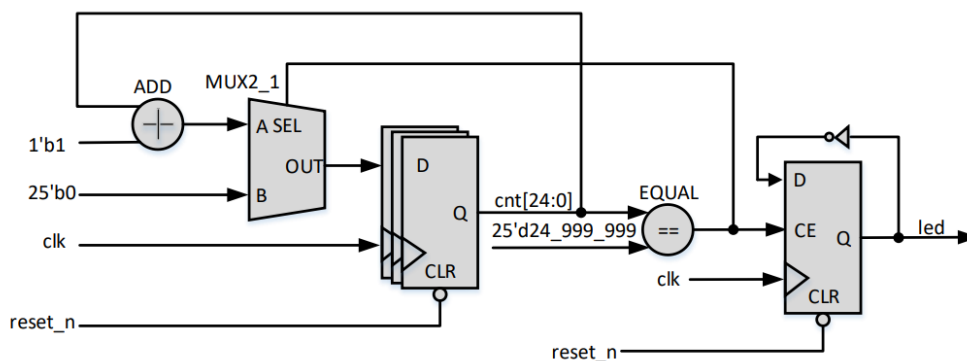


Figure 57: The counter controls the LED logic circuit diagram

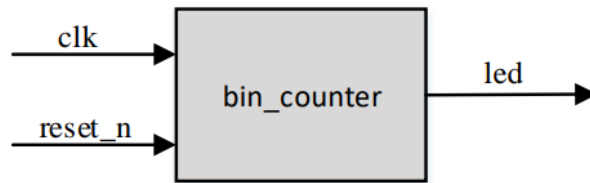


Figure 58: Schematic diagram of the counter module

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

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

4.1.1 Project establishment

As described in the previous section, create a new project named bin_counter. Double-click the icon of the desktop Gowin software to open the Gowin development environment, and follow the steps below:

1. Start page – Click on New Project.
2. New page – click OK.
3. Project Name interface - bin_counter the project name, set the project path, and click Next.
4. Select Device interface - select the chip model, select according to the FPGA chip on your board (there is a model number on the chip), and then click Next.
5. Summary screen - click Finish to complete the creation of the project.

General	
Project File:	/home/quang1806/Desktop/USING FOR UPDATE CODE GITLAB/onekiwi_fpga_board/fpga_4k/Example code/Bincounter/chap_bincounter/chap_bincounter.gprj
Synthesis Tool:	GowinSynthesis
Target Device	
Part Number:	GW1NSR-LV4CQN48PC6/I5
Series:	GW1NSR
Device:	GW1NSR-4C
Package:	QFN48P
Speed Grade:	C6/I5
Core Voltage:	LV

Figure 59: Schematic diagram of the completion of the project

4.1.2 Code design

The list of ports can be derived from the block diagram of the counter module and the analysis of the port function table of the counter module.

```
module bin_counter
(
    clk,
    reset_n,
    led
);
    input clk;
    input reset_n;
    output led;
    reg led;
endmodule
```

- From the experimental principle, it can be seen that a counter is needed, because the counter is counted from 0 instead of 1, so it is cleared when the count value is counted to 25'd13_499_999 instead of zero when it is counted to 25'd13_500_000, here the maximum value of the counter is expressed parametrically defined using the parameter, and the advantage of using the parameterized definition is that the parameter used in the design can be completely modified by modifying the value of the maximum value defined in the parameter. The specific code is as follows.

```
parameter MCNT = 13_499_999;
reg [24:0] cnt; // Define Counter Registers

//Counter Count Process
always@(posedge clk or negedge reset_n)begin
    if (!reset_n)begin
        cnt <= 25'd0;
    end
    else if (cnt < 25'd13_499_999)begin // Assuming a 27 MHz clock, this counts ~1
        second
        cnt <= cnt + 1'b1;
    end
    else begin
        cnt <= 25'd0;
    end
end
```

- When the counter counts to the preset value, let the LED be reversed once to achieve the purpose of flipping on and off

```
//The LED output controls the processs
always@(posedge clk or negedge reset_n)
  if (!reset_n) begin
    led <= 1'b0;
  end
  else if (cnt == 25'd13_499_999) begin
    led <= ~led;
  end
endmodule
```

4.2 Functional simulation verification

The above design content is analyzed and synthesized until there are no errors or warnings. Create a new counter replica file. The specific steps are as follows.

1. Click New Simulation File in the toolbar.
2. In the New pop-up window, select Verilog File and click OK.
3. In the Name column of the New Verilog file pop-up window, name the file bin_counter_tb, and click OK to create the simulation file successfully.

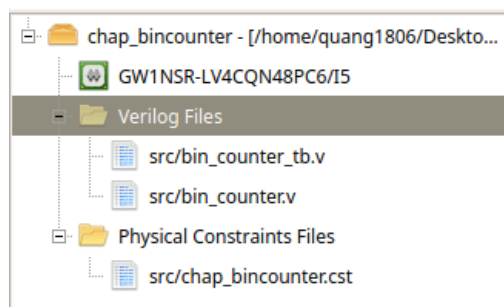


Figure 60: Schematic diagram of the successful creation of a simulation file

- The simulation excitation generates a clock CLK and reset signal with a period of 37,034ns, and the bin_counter.v that needs to be tested is instantiated.

```
`timescale 1ns/1ns
`define CLOCK_PERIOD 37.037
module bin_counter_tb();
```

```
    reg clk;
reg reset_n;
    wire led;
    bin_counter bin_counter_inst (
        .clk(clk),
        .reset_n(reset_n ),
        .led(led)
    );
    initial clk = 1;
    always #(`CLOCK_PERIOD/2) clk = ~clk;
    initial begin
        reset_n = 1'b0;
        #(`CLOCK_PERIOD *200 + 1);
        reset_n = 1'b1;
        #2000000000;
        $stop;
    end
endmodule
```

- After the simulation file is added, analyze and synthesize to see if there are any syntax errors, check that there are no errors, and then use Modelsim to simulate and verify the project.

4.2.1 Build a Modelsim simulation project

To set up a Modelsim simulation project, do the following:

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

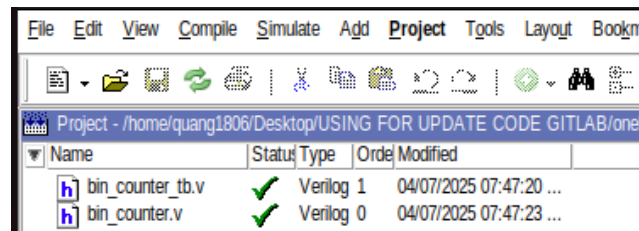


Figure 61: Schematic diagram of successful file compilation

4.2.2 Configure the simulation environment

After the simulation project is established and compiled, the next step is to configure the simulation environment as follows:

1. We right-click in the project status bar and select "Add to Project" -> "Simulation Configuration...".
2. Go to the Add Simulation Configuration page, select the "bin_counter_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.
3. Click on "Optimization Options...", and in "Optimization Options.." Select "Apply full visibility to all modules" in the settings bar and click "OK".
4. Click on the "libraries" setting bar and click "Add..." in the "Search libraries(-L)" column. Add our new high-cloud library file "gw1n", select the library file "gw1n" in "Search Libraries First(-lf)", and finally click "Save" to save the settings.

After the above steps have been completed, and after saving the configuration, we can see that the simulation configuration file "bin_counter_tb" has been generated in the "project" column, as shown in Figure below

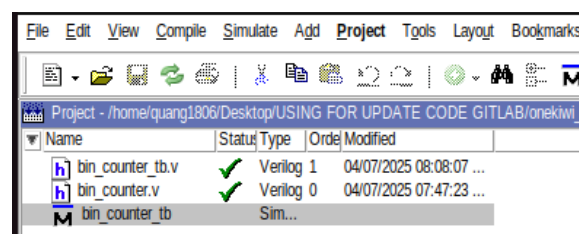


Figure 62: Schematic diagram of successful generation of simulation file

4.2.3 Simulation interface waveform observation

- We first need to double-click the newly generated simulation file to enter the "sim" interface, in the "sim" interface we can add the waveform we want to observe the module, select the module, right-click and select "Add Wave", enter the Wave window, click "Run at full speed" in the toolbar, click after the simulation stops, the waveform is shown in Figure below

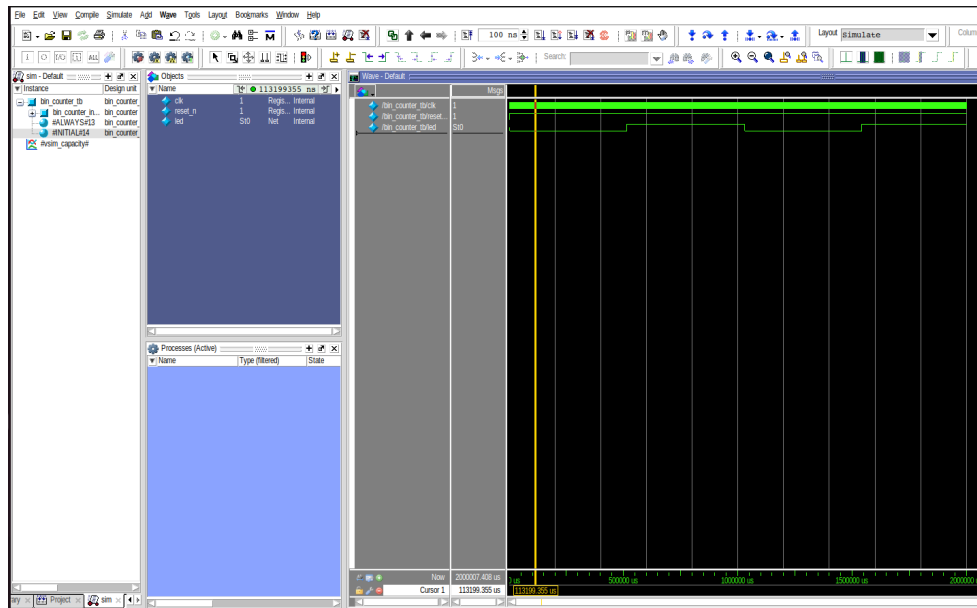


Figure 63: Waveform diagram

4.3 Board-level debugging and validation

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

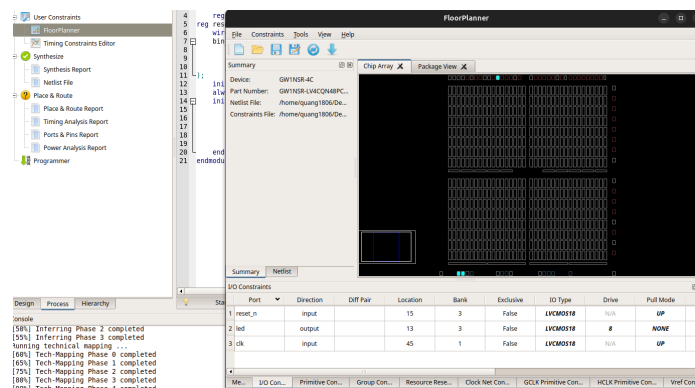


Figure 64: Pin assignment interface

- This completes the pin constraint addition. However, at this time, the content of the constraint is saved in memory, and has not yet been written to the file, click the save button in the toolbar, or directly Ctrl+S.
- After closing the FloorPlanner interface, click Design on the right to see the newly added .cst file, and double-click to view the file content, as shown in Figure below.

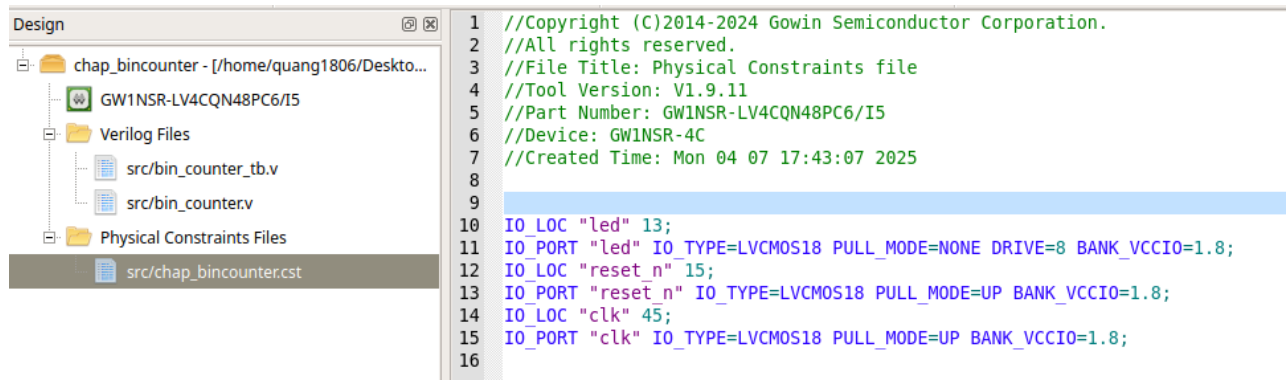


Figure 65: I/O constraint files

4.4 Reflection and summary

In this chapter, we learn the simple design of timing logic using counters as an example, and briefly introduce the methods of clock constraints. Based on this, the reader is asked to design a logic circuit to drive the LED lights to blink at different frequencies, and perform simulation and board-level verification

5 Passive Buzzer Drive Design and Verification

5.1 Passive Buzzer Circuit Design

In embedded systems, buzzers are commonly used as audio prompts. They can be divided into two types: active and passive buzzers.

- **Active buzzers** contain built-in oscillating circuits and emit sound when powered by DC voltage.
- **Passive buzzers** lack internal oscillators and must be driven by an external square wave, typically 2–5 kHz, to generate sound.

The passive buzzer used on the development board is a 3.3V DC electromagnetic buzzer. Its drive circuit includes:

- A capacitor (C89) to improve noise immunity
- A diode (D3) to protect the transistor from voltage spikes when the buzzer switches off

The buzzer is controlled through the **BEEP** pin of the FPGA. When this pin outputs a PWM signal within the 2–5 kHz frequency range, the buzzer will emit sound.

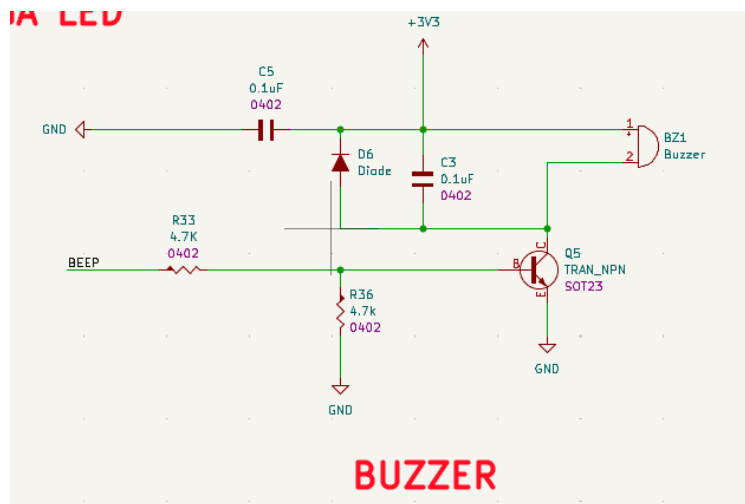


Figure 66: *Passive buzzer drive circuit diagram*

Passive buzzers cannot emit sound directly through a constant voltage—they require a square wave signal. This is achieved using a PWM (Pulse Width Modulation) signal.

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

The human ear perceives sound generated by this rapid switching of the buzzer's coil, and by adjusting the frequency, different tones can be created. This principle is the basis for playing musical notes with a passive buzzer.

To generate the square wave, the FPGA must output a periodic signal whose high and low levels alternate at the required rate (typically 2–5 kHz).

5.2 Principle of Passive Buzzer Driving

As discussed previously, a passive buzzer requires a PWM wave of a specific frequency on its control pin (BEEP) to produce sound. Therefore, the task of controlling a passive buzzer becomes one of designing a PWM wave generator.

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.

In the case of the passive buzzer, the FPGA provides PWM output on the BEEP pin, and the buzzer responds by vibrating and producing sound. This principle is fundamental to generating different tones and melodies with the buzzer.

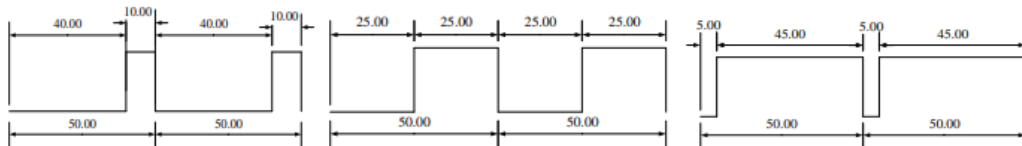


Figure 67: *PWM waveforms with different duty cycles*

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.

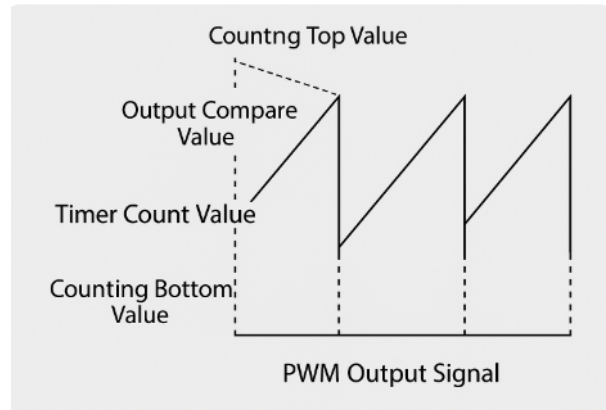


Figure 68: Principle of generating PWM using a timer

5.3 PWM Generator Module Design

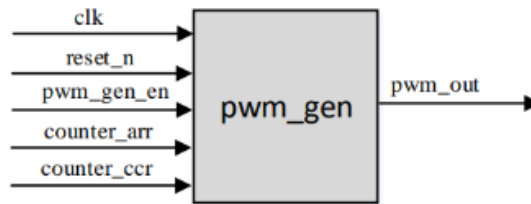


Figure 69: PWM module design

The signal	I/O	Description of the function
clk	I	The module global clock is 50MHz(27MHz for FPGA 4K)
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
```

5.4 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.

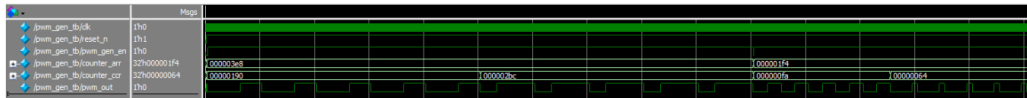


Figure 70: *pwm simulation*

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.

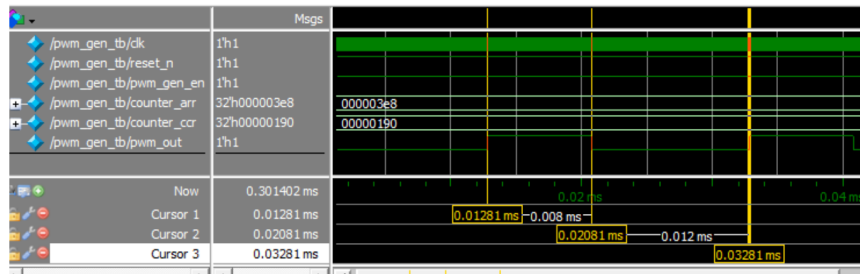


Figure 71: *50kHz duty cycle 0.4 simulation waveform chart*

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 $14/20 = 0.7$.

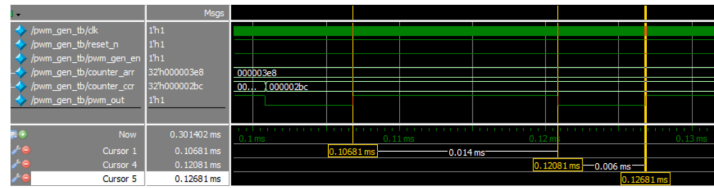


Figure 72: 50kHz duty cycle 0.7 simulation waveform chart

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 $5/10 = 0.5$.

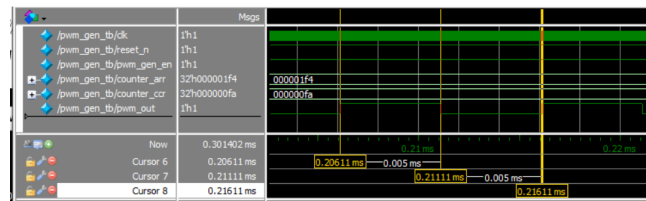


Figure 73: 100kHz duty cycle 0.5 simulation waveform chart

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 $2/10 = 0.2$.

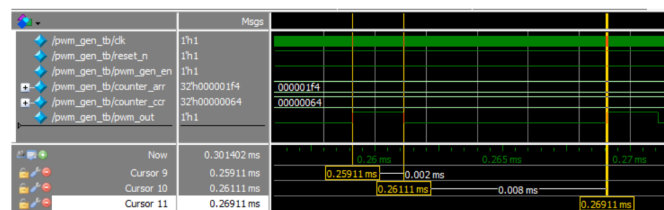


Figure 74: 100kHz duty cycle 0.5 simulation waveform chart

5.5 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_test(
    clk,
    reset_n,

    beep
);
    wire reset = ~reset_n;
    input clk;
    input reset_n;
    output beep;
    reg [31:0] counter_arr; // Auto-reload value register
    wire [31:0] counter_ccr; // Output compare value
    reg [24:0] t500ms_dly_cnt; // 500ms delay counter
    reg [2:0] pitch_num; // Tone number
    parameter T500ms_MAX_CNT = 25'd25000000;
    localparam
        D1 = 91837, // D scale tone 1 (294 Hz)
        D2 = 81818, // D scale tone 2 (330 Hz)
        D3 = 77143, // D scale tone 3 (350 Hz)
        D4 = 68702, // D scale tone 4 (393 Hz)
        D5 = 61224, // D scale tone 5 (441 Hz)
        D6 = 54545, // D scale tone 6 (495 Hz)
        D7 = 48525; // D scale tone 7 (556 Hz)
    // Output compare value is half of the auto-reload value
    assign counter_ccr = counter_arr >> 1;
    pwm_gen pwm_gen(
        .clk(clk),
        .reset_n(reset_n),
        .pwm_gen_en(1'b1),
        .counter_arr(counter_arr),
        .counter_ccr(counter_ccr),
        .pwm_out(beep)
    );
    // 500ms delay counter
    always @(posedge clk or posedge reset)
        if (reset)
            t500ms_dly_cnt <= 1'd0;
        else if (t500ms_dly_cnt == T500ms_MAX_CNT - 1'b1)
            t500ms_dly_cnt <= 1'd0;
        else
            t500ms_dly_cnt <= t500ms_dly_cnt + 1'b1;

    // Switch tone every 500ms
```

```
always @(posedge clk or posedge reset)
    if (reset)
        pitch_num <= 3'd0;
    else if (t500ms_dly_cnt == T500ms_MAX_CNT - 1'b1)
        pitch_num <= pitch_num + 1'd1;
    else
        pitch_num <= pitch_num;

// Assign auto-reload value based on tone number
always @(posedge clk or posedge reset)
    if (reset)
        counter_arr = 32'd1;
    else
        begin
            case (pitch_num)
                0 : counter_arr = 32'd1;
                1 : counter_arr = D1;
                2 : counter_arr = D6;
                3 : counter_arr = D6;
                4 : counter_arr = D6;
                5 : counter_arr = D6;
                6 : counter_arr = D7;
                7 : counter_arr = D7;
            endcase
        end
endmodule
```

5.6 Selection of pre-reload values and scale download verification

Through the simulation verification earlier, it is confirmed that the theoretical design of the PWM generation circuit is correct. Next, we will use this PWM generation module to drive the passive buzzer on the development board, allowing the passive buzzer to produce the tune "Do Re Mi Fa Sol La Si". First, we need to find the corresponding frequency values for the 7 tones of "Do Re Mi Fa Sol La Si". The frequencies of the tones are specified in the table below (categorized into bass, midrange, and treble).

Musical Note Frequency Tables for Passive Buzzer

Octave 1

Note	1	2	3	4	5	6	7
A	221	248	278	294	330	371	416
B	248	278	294	330	371	416	467
C	131	147	165	175	196	221	248
D	147	165	175	196	221	248	278
E	165	175	196	221	248	278	312
F	175	196	221	248	278	312	350
G	196	221	248	278	312	350	393

Octave 2

Note	1	2	3	4	5	6	7
A	441	495	556	589	661	742	832
B	495	556	589	661	742	832	935
C	262	294	330	350	393	441	495
D	294	330	350	393	441	495	556
E	330	350	393	441	495	556	624
F	350	393	441	495	556	624	700
G	393	441	495	556	624	700	786

Octave 3

Note	1	2	3	4	5	6	7
A	882	990	1112	1178	1322	1484	1665
B	990	1112	1178	1322	1484	1665	1870
C	525	589	661	700	786	882	990
D	589	661	700	786	882	990	1112
E	661	700	786	882	990	1112	1248
F	700	786	882	990	1112	1248	1400
G	786	882	990	1112	1248	1400	1572

The following uses D major as an example. Based on the frequency values of each note, the pre-reload values for the PWM sending module can be calculated. Table 20-4 below shows the calculated pitch frequencies and the corresponding pre-reload values for the PWM sending module.

Table 3: *Frequency to Preload Value Mapping For 50MHZ*

Frequency (Hz)	Preload Value	Frequency (Hz)	Preload Value	Frequency (Hz)	Preload Value
147	340136	294	170068	589	84889
165	303030	330	151515	661	75643
175	285714	350	142857	700	71429
196	255102	393	127272	786	63613
221	226244	441	113379	882	56689
248	201613	495	101010	990	50505
278	179856	556	89928	1112	44964

Table 4: *Frequency to Preload Value Mapping (27MHz Clock)*

Frequency (Hz)	Preload Value	Frequency (Hz)	Preload Value	Frequency (Hz)	Preload Value
147	183673	294	91837	589	45840
165	163636	330	81818	661	40847
175	154286	350	77143	700	38571
196	137755	393	68702	786	34351
221	122172	441	61224	882	30612
248	108871	495	54545	990	27273
278	97050	556	48525	1112	24281

6 Key filtering

6.1 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.

6.2 Key physical structure and circuit design

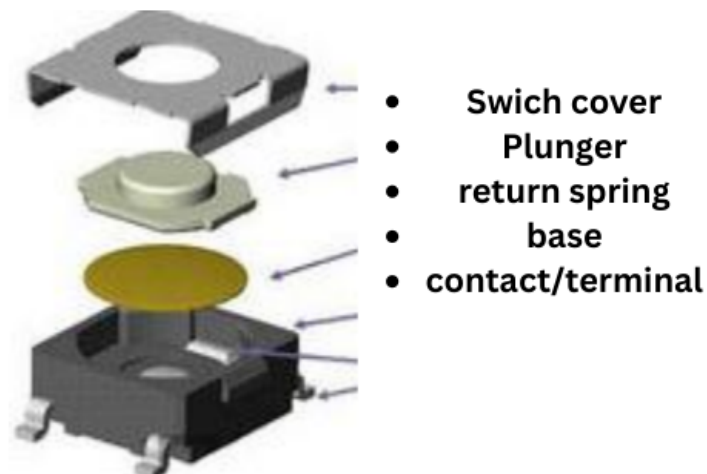


Figure 75: 1 Schematic diagram of the hardware structure of the button

- 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:

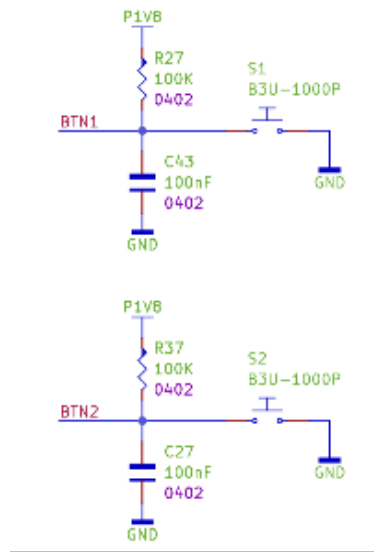


Figure 76: The change in the level of the key from pressing to release

- 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.

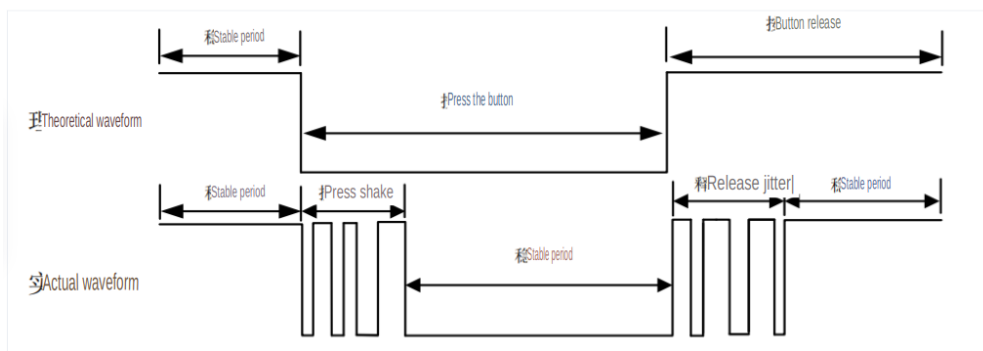


Figure 77: 3 The change in the level of the button from pressing to releasing it

- 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.

6.3 The state machine implements key debounce

6.3.1 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 78: *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 5: *Description of the interface function of the key debounce module*

6.3.2 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

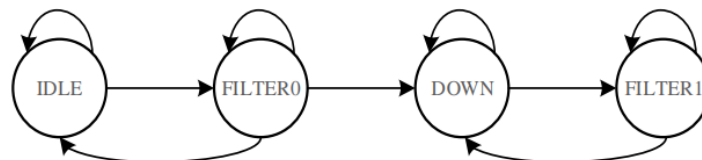


Figure 79: *State Transition Diagram*

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 6: State transitions for key debounce FSM

6.3.3 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
```

6.3.4 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;
```

6.3.5 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
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
```

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
```

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
```

6.4 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:

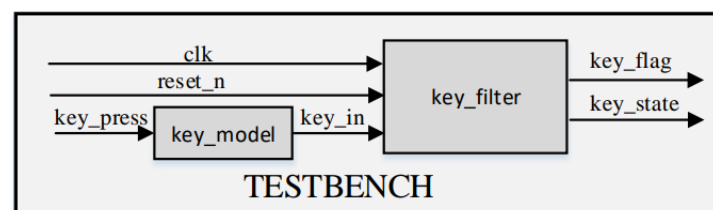


Figure 80: *Filtering testbench*

6.4.1 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.

6.4.2 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
```

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.

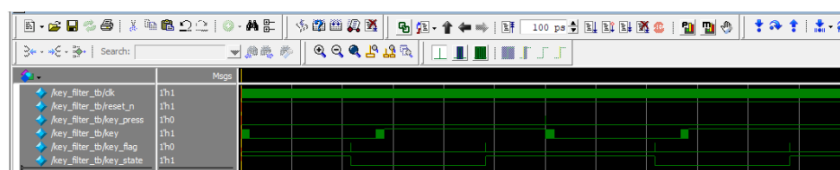


Figure 81: *Key debounce model simulation waveform diagram*

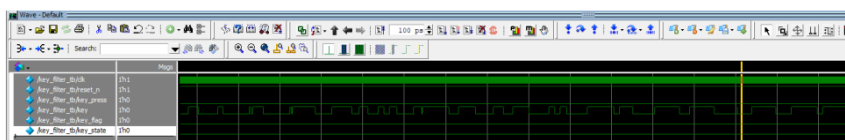


Figure 82: *Random jitter process*

7 UART_TX Design and verification of serial port transmitter module

7.1 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.

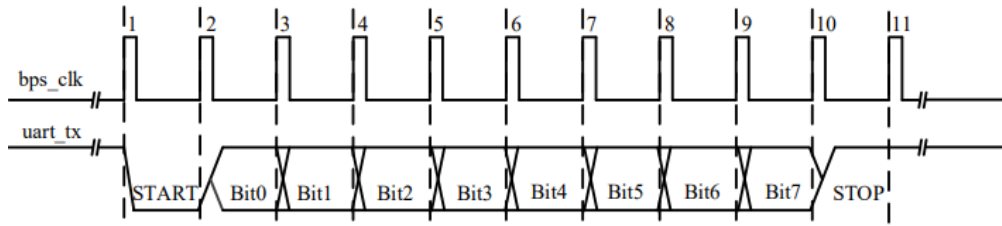


Figure 83: *process of uart transfer*

- 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.

7.2 Design and implementation of UART asynchronous serial communication transmission module

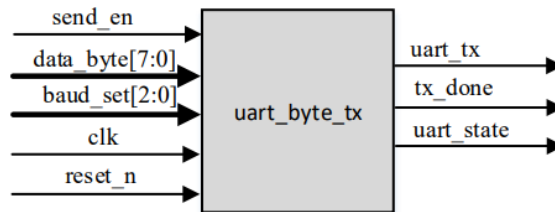


Figure 84: *Module of uart transfer*

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 7: *Table of module description*

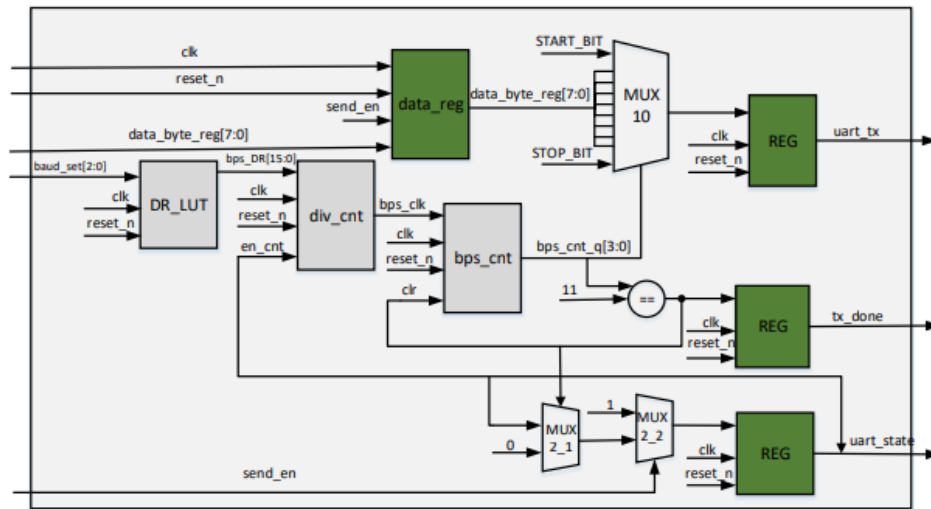


Figure 85: Module of uart transfer specifickly

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

7.3 Baurate clock module design

- Our system have clock frequency 27MHZ,formula:= $\frac{50 \times 10^6}{baudrate} - 1$ (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
    endcase
  end

```

```
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;
```

7.4 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
```

7.5 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
```

7.6 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
```

7.7 Simulation

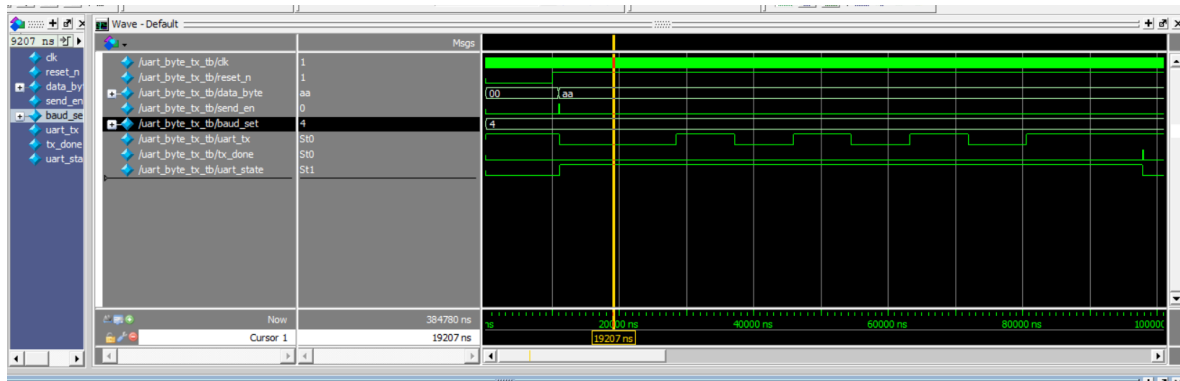


Figure 86: Simulation with modelsim

7.8 Testing with board

We using hercules software:

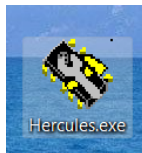


Figure 87: hercules software

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

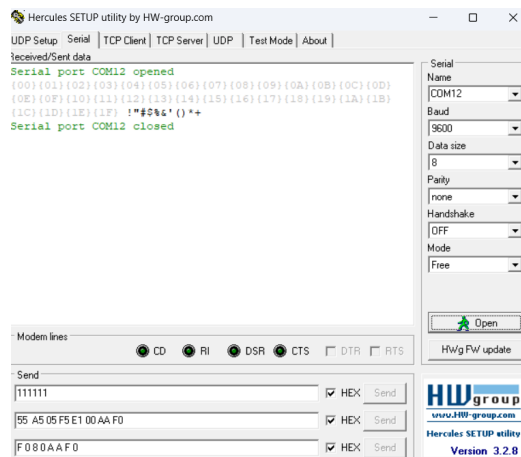


Figure 88: process of uart transfer

8 UART_RX Design and verification of serial port receiver module

8.1 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

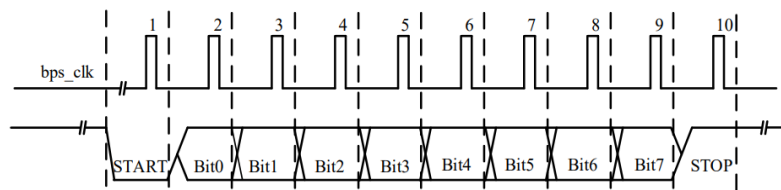


Figure 89: *Process of uart transfer*

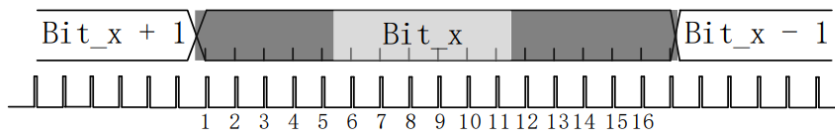


Figure 90: *Taking sample of 1 bit receive*

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.

8.2 Design and implementation of UART asynchronous serial communication receiving module

8.2.1 Serial port receiving module interface design



Figure 91: *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 8: Table of module receive description

8.2.2 Single-bit asynchronous signal synchronization design

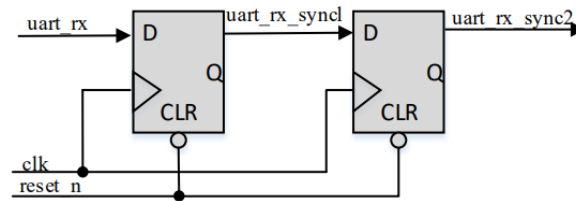


Figure 92: Schematic logic design

```

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.

8.3 Edge detection

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

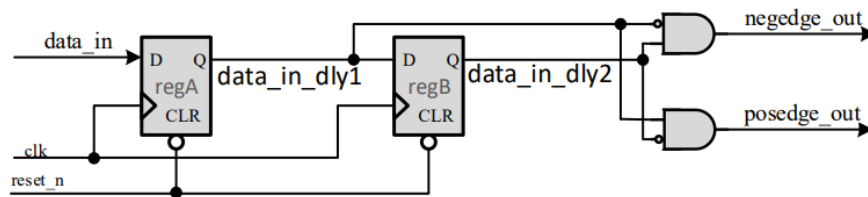


Figure 93: *principle of edge detection*

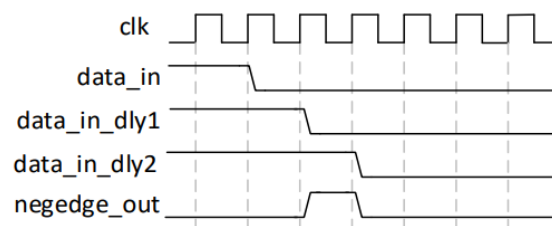


Figure 94: *Falling edge detection waveform*

```
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`

8.4 Design of the sample clock generation module

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

- 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.

8.5 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 .

8.6 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.)

8.7 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
```

8.8 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

```

8.9 Simulation

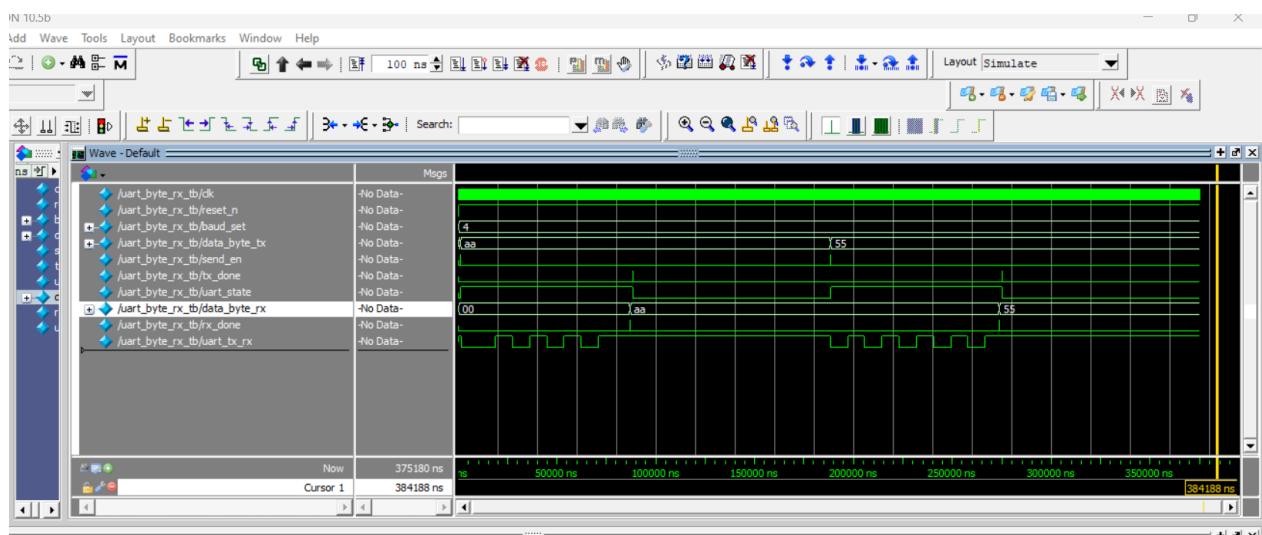


Figure 95: *UART receiver simulation with modelsim*

8.10 Testing with board

- Using the hercules software.

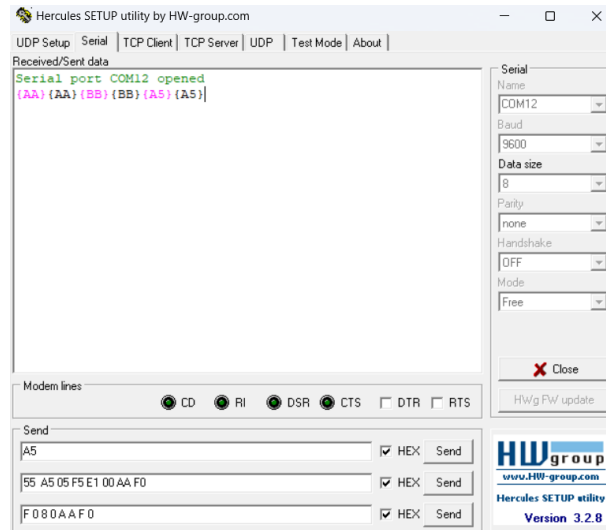


Figure 96: *UART receiver testing with board*

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

9.1 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

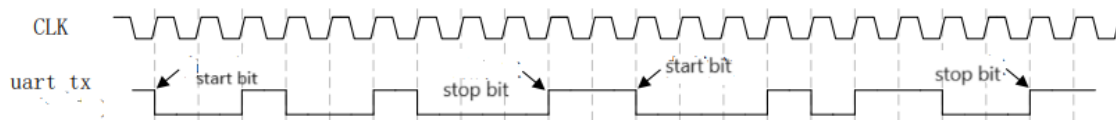


Figure 97: Serial port sends 0x1234

- 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

9.2 The principle of multi-byte serial port transmission

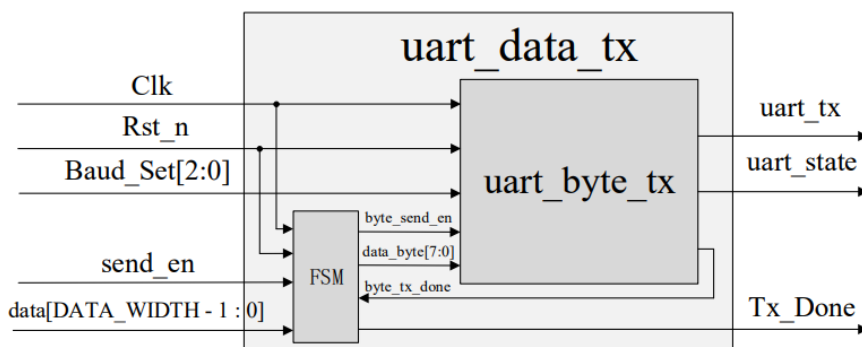


Figure 98: Multi-byte serial port transmitting module

- `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.

9.3 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.

9.4 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.

9.4.1 Multi-byte serial port receiving principle

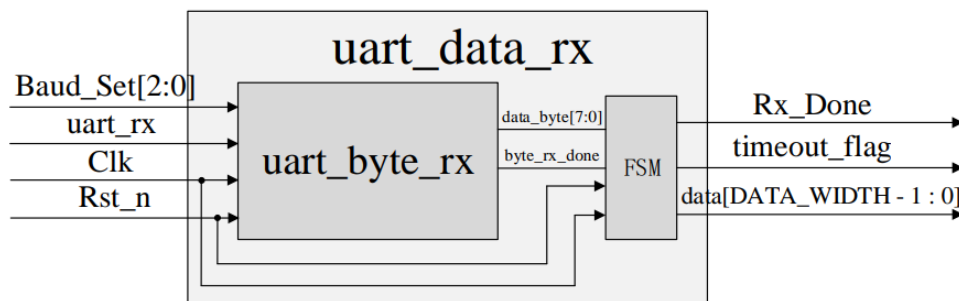


Figure 99: Block diagram of a multi-byte serial port receiving module

- 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.

9.5 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 [2:0] led;
    output uart_tx;
    wire [DATA_WIDTH-1:0] data;
    wire Rx_Done;
    wire [7:0] data_byte;
    uart_data_rx
    #(
        .DATA_WIDTH(DATA_WIDTH),
        .MSB_FIRST(MSB_FIRST)
    )
    uart_data_rx(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .uart_rx(uart_rx),
        .data(data),
```

```
.Rx_Done(Rx_Done),  
.timeout_flag(led[0]),  
.Baud_Set(3'd4)  
);  
uart_data_tx  
#(  
.DATA_WIDTH(DATA_WIDTH),  
.MSB_FIRST(MSB_FIRST)  
)uart_data_tx(  
.Clk(Clk),  
.Rst_n(Rst_n),  
.data(data),  
.send_en(Rx_Done),  
.Baud_Set(3'd4),  
.uart_tx(uart_tx),  
.Tx_Done(led[1]),  
.uart_state(led[2])  
);  
endmodule
```

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

10 I2C

10.1 I2C protocol parsing

- At the physical layer, the I2C interface only needs two bus lines, namely SCL (serial clock line) and SDA (serial data line), and the I2C bus is half-duplex, so there can only be one host at any one time. Each I2C device connected to the bus has a unique device address that communicates with each other. The transmission rate can reach 100 kbit/s in standard mode, 400 kb/s in fast mode, and 3.4 Mbit/s in high-speed mode. Data is transmitted bi-directionally between the master and slave devices on the bus in bytes (8 bits). Each device on the bus can act as either a master or a slave, and each device has a unique address (as can be seen in the I2C device data sheet) with which master and slave devices communicate with each other.
- The I2C agreement states:
 - The data bus (SDA) must remain stable when the clock (SCL) is high, so the data bus (SDA) can only be changed when the clock (SCL) is low.

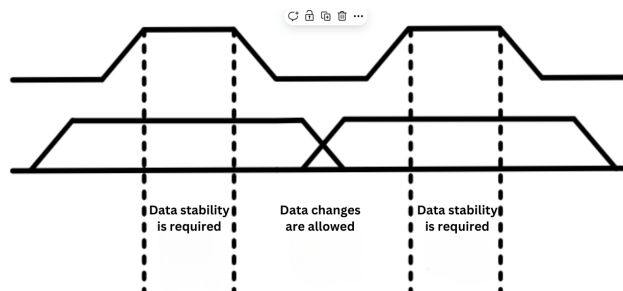


Figure 100: *I2C protocol data change timing requirements*

- When the clock (SCL) is high, the data bus (SDA) jumps from high to low to the bus start signal, and when the clock (SCL) is high, the data bus (SDA) jumps from low to high to the bus stop signal.

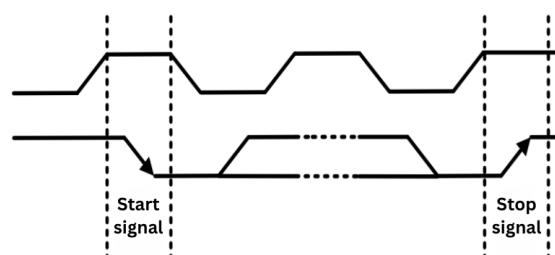


Figure 101: *Protocol start and end signal timing*

- Answer, when the IIC master (not necessarily the sender or receiver) sends 8 bits of data or commands out, the data bus (SDA) is released, i.e. set as input, and then waits for the slave to reply (0 low for answer, 1 for non-answer), at which point the clock is still provided by the master.

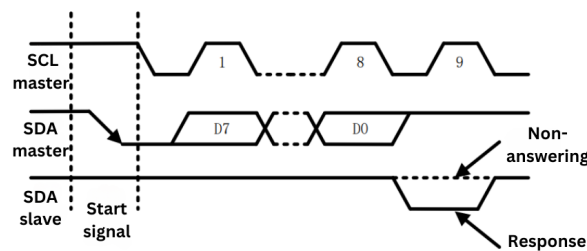


Figure 102: *Slave response signal timing*

- In the data frame format, when the I2C device communicates, the first thing is to send the "start signal", followed by the seven-bit device address, the eighth bit is the data transmission direction bit (0: represents write, 1: represents read), and then waits for the response of the slave. Of course, after the transfer is over, the "kill signal" is also generated by the host. When sending data, the high position is sent first.

10.2 I2C device ground

When the I2C protocol performs data transmission, the master first needs to send a control command to the bus, which includes the slave device address and read/write control, and then wait for a response from the slave. As shown in Figure below this is the data format for the I2C control command transmission.

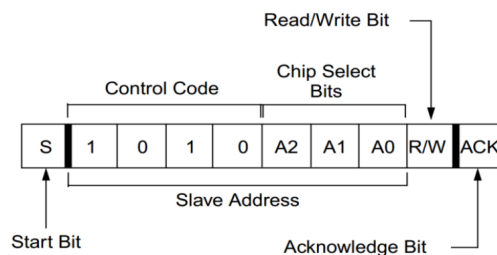


Figure 103: *The data format for transmitting control commands in the I2C protocol.*

10.3 I2C Memory Address

Every I2C-compatible device includes readable and writable registers or memory cells. For example, EEPROMs consist of sequentially addressed storage, and CMOS sensors like OV7670 (which uses SCCB, a compatible subset of I2C) include many readable/writable registers.

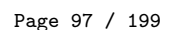
- **AT24C01** (1kbit) uses a 1byte address.
- **AT24C64** (64kbit) uses a 2byte address.



10.4.1 Single-byte Write Timing

Timing diagram for a write operation on the I2C bus. The diagram shows three signals: BUS ACTIVITY MASTER, SDA LINE, and BUS ACTIVITY. The sequence starts with a START condition (S), followed by a CONTROL BYTE (1010100), then a WORD ADDRESS (8 bits), then DATA (8 bits), and ends with a STOP condition (P). The BUS ACTIVITY MASTER signal shows the sequence of events: START, CONTROL BYTE, WORD ADDRESS, DATA, and STOP. The SDA LINE shows the corresponding data levels: high for START, low for CONTROL BYTE, high for WORD ADDRESS, low for DATA, and high for STOP. The BUS ACTIVITY signal shows the sequence of events: START, CONTROL BYTE, WORD ADDRESS, DATA, and STOP.





1. The master sets the **SDA** line as output.
2. The master initiates a **START** condition.
3. The master sends the device address byte, with the least significant bit (LSB) set to 0 to indicate a write operation.
4. The master sets **SDA** to high-impedance input mode to read the slave's ACK signal.
5. Upon receiving an ACK, the master sets **SDA** as output and sends the high byte of the memory/register address.

6. The master sets SDA to input mode to read the slave's ACK signal.
7. Upon receiving an ACK, the master sets SDA as output and sends the low byte of the address.
8. The master sets SDA to input mode to read the slave's ACK signal.
9. Upon receiving an ACK, the master sets SDA as output and transmits the data byte to be written.
10. The master sets SDA to input mode to read the slave's ACK signal.
11. Upon receiving ACK, the master generates a **STOP** condition to end the transmission.

10.4.2 Continuous Write Timing (Page Write)

Some EEPROMs support continuous (page) writes, where multiple data bytes can be written in succession.

Max page length for AT24Cxx series is typically 32 bytes.

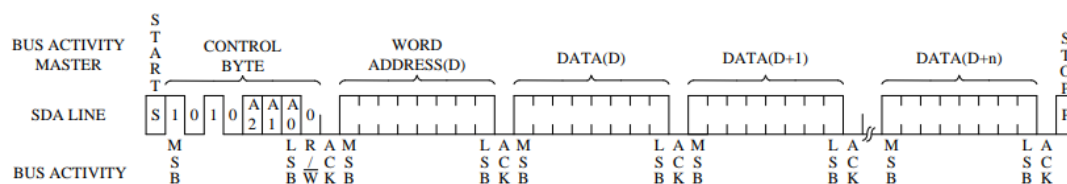


Figure 108: *Timing of continuous multi-byte write operations for the byte device*

Continuous Multi-Byte Write with Single-Byte Address (Step-by-Step)

1. The master sets the SDA line as output.
2. The master initiates a **START** condition.
3. The master sends the device address byte, with the least significant bit (LSB) set to 0 to indicate a write operation.
4. The master sets SDA to high-impedance input mode to read the slave's ACK signal.
5. Upon receiving an ACK, the master sets SDA as output and sends the 1-byte memory/register address.
6. The master sets SDA to input mode to read the slave's ACK signal.
7. Upon receiving an ACK, the master sets SDA as output and transmits the first data byte.
8. The master sets SDA to input mode to read the slave's ACK signal.

9. Upon receiving an ACK, the master sets **SDA** as output and transmits the next data byte.
10. The master sets **SDA** to input mode to read the slave's ACK signal.
 - If more data remains to be written, go back to step 9.
 - If all data has been written, proceed to step 11.
11. After the final ACK is received, the master generates a **STOP** condition to end the transmission.

10.5 I2C Read Timing

The read operation in I2C generally requires two phases. First, the master must write the target memory address to the slave. Then, the master initiates a second START condition and performs the read. This ensures that the slave knows from which address it should return data.

10.5.1 Single-Byte Read Timing

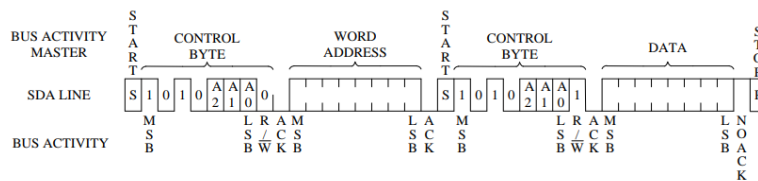


Figure 109: *Byte addressable device single-byte read operation timing*

The following steps describe how to read a single byte from a device with a 1-byte memory address:

1. The master sets **SDA** as output.
2. The master initiates a **START** condition.
3. The master sends the device address with R/W bit = 0 (write mode).
4. The master sets **SDA** to input mode and waits for the slave's ACK.
5. The master sends the memory/register address.
6. The master sets **SDA** to input mode and waits for ACK.
7. The master initiates a second **START** condition (called "repeated START").
8. The master sends the device address with R/W bit = 1 (read mode).

9. The master sets **SDA** to input mode and waits for **ACK**.
10. The slave sends the data byte. The master samples the bits.
11. The master does not acknowledge (**NACK**) the byte and sends a **STOP** condition to end the transaction.

10.5.2 Continuous Read Timing (Page Read)

For reading multiple consecutive bytes, the process is nearly identical to the single-byte read, but the master responds with an **ACK** after each byte (except the last), prompting the slave to continue transmission.

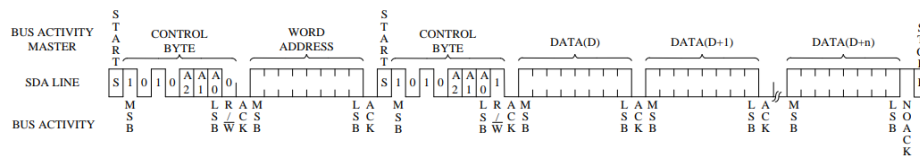


Figure 110: Timing sequence for multiple byte read operations of the byte addressable device.

Continuous Multi-Byte Read with Single-Byte Address (Step-by-Step)

1. The master sets the **SDA** line as output.
2. The master initiates a **START** condition.
3. The master sends the device address byte with the least significant bit (LSB) set to 0, indicating a write operation.
4. The master sets **SDA** to high-impedance input mode to read the slave's **ACK** signal.
5. Upon receiving an **ACK**, the master sets **SDA** as output and sends the 1-byte memory/register address.
6. The master sets **SDA** to input mode to read the slave's **ACK** signal.
7. Upon receiving an **ACK**, the master issues a **repeated START** condition.
8. The master sends the device address byte with the LSB set to 1, indicating a read operation.
9. The master sets **SDA** to input mode to read the slave's **ACK** signal.
10. Upon receiving **ACK**, the master sets **SDA** to input and reads the first byte of data from the **SDA** line.

11. The master sets **SDA** as output and sends an **ACK** to the slave.
12. The master sets **SDA** to input and reads the next byte of data from the **SDA** line.
 - If all n bytes have been read, proceed to step 13.
 - If more data remains, return to step 11.

(Note: For AT24Cxx EEPROMs, the maximum page read length is typically 32 bytes.)

13. The master sends a **NACK** (logic high) to indicate the last byte; since the **SDA** line is open-drain, this high state doesn't need to be actively driven.
14. The master generates a **STOP** condition to end the communication.

Note: This method applies to both 1-byte and 2-byte memory address devices. The only difference lies in how many address bytes the master sends before initiating the read phase.

10.6 i2c_bit_shift Module Design

In the previous section, we analyzed how to extract a general-purpose bit-level transfer unit based on the complete I2C timing sequence. This section focuses on designing the full transmission logic following that methodology.

10.6.1 Bit Transfer Unit Interface Design

According to the analysis of the low-level transmission unit, we can design a universal transmission logic module as shown in Figure ??.

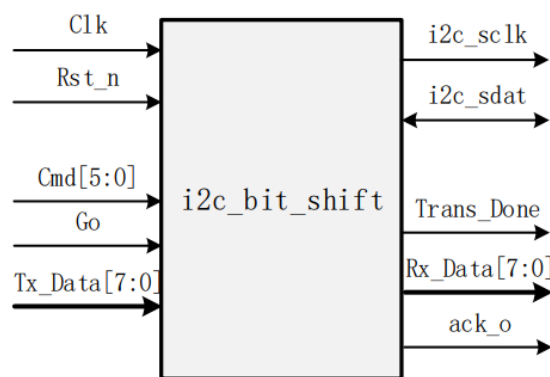


Figure 111: *i2c_bit_shift module*

Interface	I/O	Functional description
Clk	I	Module working clock, 50M clock
Rst_n	I	module reset signal
Cmd	I	control bus realizes the combination of various commands for various transmission operations (write, read, start, stop, answer, no answer)
Go	I	The start-up enable signal of the whole module, for the convenience of the interface, it is hoped that only a single clock cycle pulse needs to be generated on the port to start a byte complete transmission (including possible start bits, stop bits, and answer bits)
Rx_DATA	O	The 8-bit data received by the bus, which is output during the read operation
Tx_DATA	I	The 8-bit data to be sent by the bus is passed to the module through this port
Trans_Done	O	Send or receive an 8-bit data completion flag signal, and each transmission completion produces a single-cycle high-pulse signal
ack_o	O	slave answer flag, in this underlying logic, we do not give the correct reply signal on the bus for the time being, only the received reply signal state is stored and output, if the correct reply signal is not received, how to perform the next step, by the external other logic to decide.
i2c_sclk	O	I2c clock bus
i2c_sdat	I/O	I2c data bus

Table 9: *Description of the module interface i2c*

10.6.2 Design of Position Transmission Unit State Machine

- State Machine The core logic is implemented as a finite state machine (FSM) with states for each part of the I²C protocol:
 - IDLE: Waits for a Go signal to start the I²C operation.
 - GEN_STA: Generates the start condition.
 - WR_DATA: Sends the data byte bit-by-bit on the SDA line.
 - RD_DATA: Reads data from the slave.

- CHECK_ACK: Checks if the slave has acknowledged the transfer.
 - GEN_ACK: Generates an ACK or NACK signal after reading data.
 - GEN_STO: Generates the stop condition to end the I²C transaction.
- Data and Clock Line Control: The i2c_sdat line is controlled through a tri-state buffer using i2c_sdat_oe and i2c_sdat_o. i2c_sclk is toggled according to the I²C clock timing.
 - Command Processing: the Cmd input specifies the I²C operation (start, write, read, stop, etc.). The FSM transitions to the corresponding state based on the command.
 - Bit-Level Timing: each bit of data is transmitted or received in 4 clock cycles: Set data on SDA (or read SDA), toggle SCL high (positive edge), hold SCL high, toggle SCL low (negative edge).

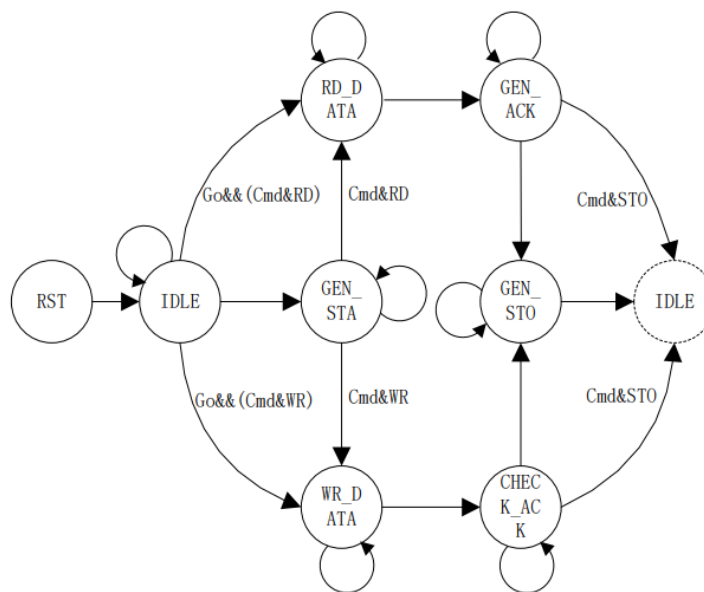


Figure 112: *module I2C FSM*

10.6.3 Bit Transfer Unit State Machine Design

The bit transfer unit uses a finite state machine (FSM) for control. Each specific action—such as generating a START condition, writing a byte, reading a byte, checking for ACK, generating ACK/NACK, and issuing a STOP—is implemented as a distinct state.

The FSM transitions between states based on the input command ‘Cmd’, which determines which type of I2C operation is to be performed.

The following is the state encoding used in the design:

Listing 3: *FSM State Definitions for i2c_bit_shift*

```
localparam
    IDLE      = 7'b00000001, // Idle state
    GEN_STA   = 7'b00000010, // Generate START condition
    WR_DATA   = 7'b00000100, // Write data state
    RD_DATA   = 7'b00001000, // Read data state
    CHECK_ACK = 7'b00010000, // Check ACK state
    GEN_ACK   = 7'b01000000, // Generate ACK/NACK state
    GEN_STO   = 7'b10000000; // Generate STOP condition
```

These states work together to handle the detailed protocol timing of the I2C bus. Each operation (START, WRITE, READ, etc.) will enter the FSM in the appropriate initial state, proceed through the required SCL/SDA toggling, and return to IDLE once complete.

Transitions are triggered by the 'Go' signal and follow a fixed sequence. When a byte is sent, it moves through: WR_DATA → CHECK_ACK → IDLE

When reading: RD_DATA → GEN_ACK → IDLE

This structure keeps the transfer logic modular and reusable for all high-level I2C control flows.

10.6.4 Key Logic Design of Bit Transfer Unit

In I2C, the data line (SDA) is an open-drain signal and requires an external pull-up resistor. To simulate this behavior in Verilog, we use bidirectional control with output enable.

The SDA line assignment logic is implemented as:

Listing 4: *SDA Line Drive Logic*

```
assign i2c_sdat = !i2c_sdat_o && i2c_sdat_oe ? 1'b0 : 1'bz;
```

This means: - When 'i2c_sdat_oe' (output enable) is '1' and 'i2c_sdat_o' is '0', drive the line low. - In all other cases, let the line float (high-impedance), allowing external pull-up.

Command Combination (Cmd) The 'Cmd' signal is 6 bits, and uses one-hot encoding to represent operations: - 'STA' (Start), 'WR' (Write), 'RD' (Read), 'ACK', 'NACK', 'STO' (Stop)

Multiple operations can be combined using bitwise OR:

Listing 5: *Example: Combining Cmd Operations*

```
Cmd = WR | STA | STO; // Write with start and stop
```

In the FSM, the 'Cmd' is decoded using bitwise AND to determine which states to enter:

```
if (Cmd & STA)
    state <= GEN_STA;
else if (Cmd & WR)
```

```
    state <= WR_DATA;
else if (Cmd & RD)
    state <= RD_DATA;
else
    state <= IDLE;
```

The write and read processes are further decoded within their respective states to decide whether to send ACK, issue STOP, or proceed with the next byte.

10.7 Design and Explanation of i2c_bit_shift Module

Module Overview

The `i2c_bit_shift` module implements low-level I2C protocol functionality, including generating START and STOP conditions, writing and reading data bits, and handling ACK/NACK. It is designed to be flexible by parameterizing the system and SCL clock frequencies.

Module Declaration and Interface

Listing 6: *Module Declaration*

```
module i2c_bit_shift(
    Clk,
    Rst_n,
    Cmd,
    Go,
    Rx_DATA,
    Tx_DATA,
    Trans_Done,
    ack_o,
    i2c_sclk,
    i2c_sdat
);
```

Port Definitions

Listing 7: *Port Definitions*

```
input Clk;
input Rst_n;
input [5:0] Cmd;
input Go;
output reg[7:0] Rx_DATA;
```

```
input [7:0]Tx_DATA;
output reg Trans_Done;
output reg ack_o;
output reg i2c_sclk;
inout i2c_sdat;
```

These are standard I/O signals required for driving the I2C protocol.

Open-Drain SDA Line Handling

Listing 8: *Open-Drain Simulation for SDA*

```
reg i2c_sdat_o;
reg i2c_sdat_oe;
assign i2c_sdat = !i2c_sdat_o && i2c_sdat_oe ? 1'b0 : 1'bz;
```

This simulates open-drain behavior: driving low or leaving the line floating.

Clock Divider for SCL Generation

Listing 9: *Clock Divider*

```
parameter SYS_CLOCK = 50_000_000;
parameter SCL_CLOCK = 400_000;
localparam SCL_CNT_M = SYS_CLOCK / SCL_CLOCK / 4 - 1;

reg [19:0]div_cnt;
reg en_div_cnt;
wire sclk_plus = div_cnt == SCL_CNT_M;

always@(posedge Clk or negedge Rst_n)
if(!Rst_n)
    div_cnt <= 20'd0;
else if(en_div_cnt) begin
    if(div_cnt < SCL_CNT_M)
        div_cnt <= div_cnt + 1'b1;
    else
        div_cnt <= 0;
end else
    div_cnt <= 0;
```

This block generates timing pulses at the required I2C frequency.

Command and State Encoding

Listing 10: *Command and State Encoding*

```
localparam
    WR    = 6'b000001,
    STA   = 6'b000010,
    RD    = 6'b000100,
    STO   = 6'b001000,
    ACK   = 6'b010000,
    NACK  = 6'b100000;

reg [7:0]state;
localparam
    IDLE      = 8'b00000001,
    GEN_STA   = 8'b00000010,
    WR_DATA   = 8'b00000100,
    RD_DATA   = 8'b00001000,
    CHECK_ACK = 8'b00010000,
    GEN_ACK   = 8'b00100000,
    GEN_STO   = 8'b01000000;
```

Each state represents a stage in the I2C transfer.

FSM and Operation

Listing 11: *Main FSM Implementation*

```
reg [4:0]cnt;

always@(posedge Clk or negedge Rst_n)
if(!Rst_n) begin
    Rx_DATA <= 0;
    i2c_sdat_oe <= 1'd0;
    en_div_cnt <= 1'b0;
    i2c_sdat_o <= 1'd1;
    Trans_Done <= 1'b0;
    ack_o <= 0;
    state <= IDLE;
    cnt <= 0;
end else begin
    case(state)
        IDLE: begin
            Trans_Done <= 1'b0;
```

```
i2c_sdat_oe <= 1'd1;
if(Go) begin
    en_div_cnt <= 1'b1;
    if(Cmd & STA) state <= GEN_STA;
    else if(Cmd & WR) state <= WR_DATA;
    else if(Cmd & RD) state <= RD_DATA;
    else state <= IDLE;
end else begin
    en_div_cnt <= 1'b0;
    state <= IDLE;
end
end
```

This is the FSM control that dispatches based on the input command.

FSM States of i2c_bit_shift Module

1. IDLE

Purpose: Waits for the Go signal to initiate a transaction.

Operations:

- Sets Trans_Done to 0, indicating no ongoing transaction.
- Enables SDA output (i2c_sdat_oe = 1).
- If Go is asserted:
 - Enables the clock divider (en_div_cnt = 1).
 - Transitions to:
 - * GEN_STA if Cmd includes STA.
 - * WR_DATA if Cmd includes WR.
 - * RD_DATA if Cmd includes RD.

2. GEN_STA (Generate Start Condition)

Purpose: Generates the I²C start condition to begin communication.

Operations:

- Over 4 steps (controlled by cnt):
 1. Set SDA high.
 2. Set SCL high.
 3. Pull SDA low while SCL is high (start condition).
 4. Pull SCL low.

- After completion, transitions to:
 - WR_DATA if Cmd includes WR.
 - RD_DATA if Cmd includes RD.

3. WR_DATA (Write Data)

Purpose: Transmits 8 bits of data (Tx_DATA) to the slave device.

Operations:

- For each bit:
 - Set SDA to the corresponding bit value.
 - Generate a clock pulse on SCL.
- After sending all bits, transitions to CHECK_ACK to verify acknowledgment from the slave.

4. RD_DATA (Read Data)

Purpose: Receives 8 bits of data from the slave device.

Operations:

- For each bit:
 - Release SDA (set as input).
 - Generate a clock pulse on SCL.
 - Sample SDA on SCL high to read the bit.
- After receiving all bits, transitions to GEN_ACK to send acknowledgment.

5. CHECK_ACK (Check Acknowledgment)

Purpose: Checks for acknowledgment (ACK) from the slave after data transmission.

Operations:

- Releases SDA.
- Generates a clock pulse on SCL.
- Samples SDA on SCL high to determine ACK (0) or NACK (1).
- If Cmd includes ST0, transitions to GEN_ST0; otherwise, returns to IDLE and sets Trans_Done to 1.

6. GEN_ACK (Generate Acknowledgment)

Purpose: Sends an ACK or NACK to the slave after data reception.

Operations:

- Sets SDA to 0 for ACK or 1 for NACK based on `Cmd`.
- Generates a clock pulse on SCL.
- If `Cmd` includes `STO`, transitions to `GEN_STO`; otherwise, returns to `IDLE` and sets `Trans_Done` to 1.

7. GEN_STO (Generate Stop Condition)

Purpose: Generates the I²C stop condition to end communication.

Operations:

- Pulls SDA low while SCL is low.
- Sets SCL high.
- Sets SDA high while SCL is high (stop condition).
- Returns to `IDLE` and sets `Trans_Done` to 1.

10.8 Design and Explanation of i2c_bit_shift Module

Module Overview

The `i2c_bit_shift` module implements low-level I²C protocol functionality, including generating START and STOP conditions, writing and reading data bits, and handling ACK/NACK. It is designed to be flexible by parameterizing the system and SCL clock frequencies.

FSM and State Explanations

State: `GEN_STA`

This state generates the I²C START condition.

```
GEN_STA: begin
  if(sclk_plus) begin
    if(cnt == 3) cnt <= 0;
    else cnt <= cnt + 1;
    case(cnt)
      0: begin i2c_sdat_o <= 1; i2c_sdat_oe <= 1; end
      1: begin i2c_sclk <= 1; end
      2: begin i2c_sdat_o <= 0; i2c_sclk <= 1; end
      3: begin i2c_sclk <= 0; end
      default: begin i2c_sdat_o <= 1; i2c_sclk <= 1; end
    endcase
    if(cnt == 3) begin
      if(Cmd & WR) state <= WR_DATA;
      else if(Cmd & RD) state <= RD_DATA;
    end
  end
```

```
end  
end
```

State: WR_DATA

This state transmits 8 bits of data from Tx_DATA to the I2C bus.

```
WR_DATA: begin  
  if(sclk_plus) begin  
    if(cnt == 31) cnt <= 0;  
    else cnt <= cnt + 1;  
    case(cnt)  
      0,4,8,12,16,20,24,28: begin  
        i2c_sdat_o <= Tx_DATA[7-cnt[4:2]];  
        i2c_sdat_oe <= 1;  
      end  
      1,5,9,13,17,21,25,29: i2c_sclk <= 1;  
      2,6,10,14,18,22,26,30: i2c_sclk <= 1;  
      3,7,11,15,19,23,27,31: i2c_sclk <= 0;  
      default: begin i2c_sdat_o <= 1; i2c_sclk <= 1; end  
    endcase  
    if(cnt == 31) state <= CHECK_ACK;  
  end  
end
```

State: RD_DATA

This state receives 8 bits of data into Rx_DATA from the I2C bus.

```
RD_DATA: begin  
  if(sclk_plus) begin  
    if(cnt == 31) cnt <= 0;  
    else cnt <= cnt + 1;  
    case(cnt)  
      0,4,8,12,16,20,24,28: begin  
        i2c_sdat_oe <= 0; i2c_sclk <= 0;  
      end  
      1,5,9,13,17,21,25,29: i2c_sclk <= 1;  
      2,6,10,14,18,22,26,30: begin  
        i2c_sclk <= 1;  
        Rx_DATA <= {Rx_DATA[6:0], i2c_sdat};  
      end  
      3,7,11,15,19,23,27,31: i2c_sclk <= 0;  
    endcase  
  end  
end
```

```
        default: begin i2c_sdat_o <= 1; i2c_sclk <= 1; end
    endcase
    if(cnt == 31) state <= GEN_ACK;
end
end
```

State: CHECK_ACK

Reads the ACK bit from the slave device.

```
CHECK_ACK: begin
    if(sclk_plus) begin
        if(cnt == 3) cnt <= 0;
        else cnt <= cnt + 1;
        case(cnt)
            0: begin i2c_sdat_oe <= 0; i2c_sclk <= 0; end
            1: i2c_sclk <= 1;
            2: begin ack_o <= i2c_sdat; i2c_sclk <= 1; end
            3: i2c_sclk <= 0;
            default: begin i2c_sdat_o <= 1; i2c_sclk <= 1; end
        endcase
        if(cnt == 3) begin
            if(Cmd & STO) state <= GEN_STO;
            else begin
                state <= IDLE;
                Trans_Done <= 1;
            end
        end
    end
end
end
```

State: GEN_ACK

Generates either ACK or NACK signal.

```
GEN_ACK: begin
    if(sclk_plus) begin
        if(cnt == 3) cnt <= 0;
        else cnt <= cnt + 1;
        case(cnt)
            0: begin
                i2c_sdat_oe <= 1;
                i2c_sclk <= 0;
            end
        endcase
    end
end
```

```
        if(Cmd & ACK) i2c_sdat_o <= 0;
        else if(Cmd & NACK) i2c_sdat_o <= 1;
    end
    1: i2c_sclk <= 1;
    2: i2c_sclk <= 1;
    3: i2c_sclk <= 0;
    default: begin i2c_sdat_o <= 1; i2c_sclk <= 1; end
endcase
if(cnt == 3) begin
    if(Cmd & STO) state <= GEN_STO;
    else begin
        state <= IDLE;
        Trans_Done <= 1;
    end
end
end
end
```

State: GEN_STO

Generates the STOP condition.

```
GEN_STO: begin
    if(sclk_plus) begin
        if(cnt == 3) cnt <= 0;
        else cnt <= cnt + 1;
        case(cnt)
            0: begin i2c_sdat_o <= 0; i2c_sdat_oe <= 1; end
            1: i2c_sclk <= 1;
            2: begin i2c_sdat_o <= 1; i2c_sclk <= 1; end
            3: i2c_sclk <= 1;
            default: begin i2c_sdat_o <= 1; i2c_sclk <= 1; end
        endcase
        if(cnt == 3) begin
            Trans_Done <= 1;
            state <= IDLE;
        end
    end
end
end
```

10.8.1 I2C control

For this module we create two task write_byte and read_byte for minimize coding

```
task read_byte;
    input [5:0] Ctrl_Cmd;
    begin
        Cmd <= Ctrl_Cmd;
        Go <= 1'b1;
    end
endtask

task write_byte;
    input [5:0] Ctrl_Cmd;
    input [7:0] Wr_Byte_Data;
    begin
        Cmd <= Ctrl_Cmd;
        Tx_DATA <= Wr_Byte_Data;
        Go <= 1'b1;
    end
endtask
```

Therefore, for writting process :

```
case(cnt)
0: write_byte(WR | STA, device_id<<1|0); // Device ID with Write Command
1: write_byte(WR, 8'hAA); //Normal write data
2: write_byte(WR|STO, 8'hFD); //write data with stop condition
```

For reading process, we will use two states to save data to register:

```
RD_REG: begin
    state <= WAIT_RD_DONE;
    case (cnt)
        0: write_byte(WR | STA, device_id<<1|1); // Device ID with Reading command
        1: read_byte(RD|ACK); //Normal read data
        2: read_byte(RD|STO|ACK); //read data with stop condition

WAIT_RD_DONE: begin
    Go <= 1'b0;
    if (Trans_Done) begin
        case (cnt)
            0: begin cnt <= 1; state <= RD_REG; end
            1: begin cnt <= 2; DATA[15:8]<=RX_DATA state <= RD_REG; end
```

```
2: begin state <= NEXT_STATE; DATA[7:0] <= RX_DATA; cnt <= 0; end
  default::;
endcase
end
end
```

10.9 Example

Base on the template you can modify to meet requirement of the device I2C such as time delay, data to write and read,...

```
i2c_write(i2c_addr=0x44, tx_bytes=[0xFD])
wait_seconds(0.01)
rx_bytes = i2c_read(i2c_addr=0x44, number_of_bytes=6)
t_ticks = rx_bytes[0] * 256 + rx_bytes[1]
checksum_t = rx_bytes[2]
rh_ticks = rx_bytes[3] * 256 + rx_bytes[4]
checksum_rh = rx_bytes[5]
t_degC = -45 + 175 * t_ticks/65535
rh_pRH = -6 + 125 * rh_ticks/65535
if (rh_pRH > 100):
    rh_pRH = 100
if (rh_pRH < 0):
    rh_pRH = 0
```

Figure 113: *SHT40 reading temp and humid*

11 Usage of the PLL IP Core

11.1 Principle of the PLL Circuit

In Verilog design, the execution of programs is usually driven by clocks. The more complex the design, the more clock domains are often required. However, development boards usually come with only a single crystal oscillator. For instance, the ACG525 development board includes a 50 MHz onboard active oscillator.

Although internal logic can divide or multiply this clock to generate various frequencies, the quality of these derived clocks is often poor and unsuitable for precise applications.

To solve this, FPGA vendors integrate dedicated clock management circuits within the chip. One such circuit is the **Phase-Locked Loop (PLL)**. These PLLs can produce high-quality, stable, and precise clocks through frequency multiplication and division.

In this chapter, we study the operating principle of PLLs and use the Gowin-provided PLL soft IP to demonstrate how to use it through a simple design.

Definition: A Phase-Locked Loop (PLL) is a feedback control circuit often used to control the frequency and phase of an internal oscillator based on an external reference signal. When the output signal's frequency matches the input reference, a fixed phase difference is maintained between them, thereby locking the phase.

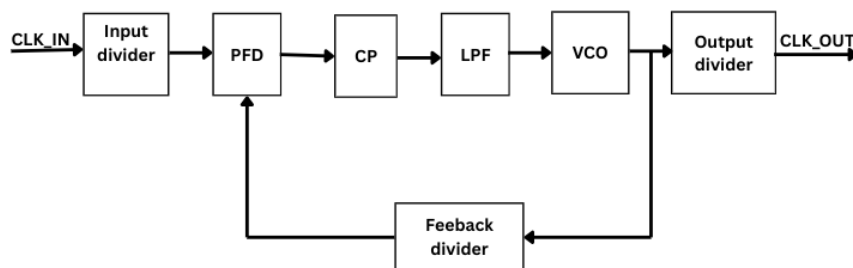


Figure 114: *PLL process*

A typical PLL consists of the following blocks:

Phase-Locked Loops (PLLs) in FPGAs and Their Impact on HDMI and Camera Interfaces

11.2 Introduction to Phase-Locked Loops (PLLs)

A Phase-Locked Loop (PLL) is a control system that generates an output signal whose phase is aligned with the phase of an input reference signal. In digital systems, PLLs are pivotal for clock generation, synchronization, and frequency multiplication. Within FPGAs, PLLs are integral for managing multiple clock domains, ensuring timing accuracy, and facilitating high-speed data transfers.

Table 10: *Functions of Blocks in a PLL Circuit*

Block	Abbreviation	Function
CLK_IN	–	Clock signal from a crystal oscillator (e.g., 27 MHz) is fed into the PLL
Input Divider	$\div D$	Divides the input frequency to match the phase comparator
PFD	PFD	Compares phase and frequency differences between the input signal and feedback signal
CP	CP	Generates a control current to adjust the voltage for the VCO
LPF	LPF	Filters the voltage signal from the charge pump to smooth and stabilize it
VCO	VCO	Generates a clock signal whose frequency varies with the input voltage
Output Divider	$\div V$	Divides the VCO frequency to produce the desired output frequency
CLK_OUT	–	The processed clock signal used to drive other circuits
Feedback Divider	–	Divides the output signal and sends it back to the PFD to complete the control loop

11.3 PLLs in FPGA Architectures

Modern FPGAs incorporate dedicated PLL blocks that offer functionalities such as:

- **Frequency Synthesis:** Generating multiple clock frequencies from a single reference clock.
- **Phase Alignment:** Ensuring precise phase relationships between different clock domains.
- **Jitter Reduction:** Minimizing clock signal variations to enhance signal integrity.
- **Clock Domain Crossing:** Facilitating safe data transfer between different clock domains.

These features are essential for applications requiring precise timing, such as video processing and high-speed communication interfaces.

11.4 Impact of PLLs on HDMI Interfaces

High-Definition Multimedia Interface (HDMI) is a widely used standard for transmitting uncompressed video and audio data. Implementing HDMI interfaces on FPGAs necessitates strict adherence to timing specifications, which is where PLLs play a crucial role:

- **Pixel Clock Generation:** HDMI requires specific pixel clock frequencies (e.g., 74.25 MHz for 720p). PLLs synthesize these frequencies from a base reference clock.

- **Data Serialization:** HDMI transmits data using Transition Minimized Differential Signaling (TMDS). PLLs provide the high-frequency clocks necessary for serializing parallel data into TMDS streams.
- **Phase Alignment:** Ensuring that the data and clock signals are phase-aligned is vital for HDMI signal integrity. PLLs adjust the phase to meet this requirement.

Proper configuration of PLLs ensures that the FPGA-generated HDMI signals meet the stringent timing and frequency standards required for reliable operation.

11.5 Impact of PLLs on Camera Interfaces

Interfacing cameras with FPGAs involves handling various video formats and synchronization signals. PLLs are instrumental in managing these aspects:

- **Clock Recovery:** Cameras often provide a pixel clock that needs to be synchronized with the FPGA's internal clock domains. PLLs facilitate this synchronization.
- **Frame Rate Matching:** To process video frames correctly, the FPGA must match the camera's frame rate. PLLs adjust the internal clocks to align with the incoming video stream.
- **Data Alignment:** Ensuring that data is captured at the correct time requires precise clocking. PLLs help in aligning the data capture clock with the incoming video signals.

By leveraging PLLs, FPGAs can effectively interface with various camera modules, handling different resolutions and frame rates with precision.

11.6 Considerations and Best Practices

When utilizing PLLs in FPGA designs, consider the following best practices:

- **Jitter Management:** Ensure that the PLL configuration minimizes jitter to maintain signal integrity, especially in high-speed interfaces like HDMI.
- **Lock Time:** Account for the PLL's lock time during system initialization to prevent data corruption.
- **Resource Utilization:** Be mindful of the number of available PLLs in the FPGA and plan the design accordingly.
- **Simulation and Testing:** Simulate the PLL behavior under various conditions to ensure reliable operation across all scenarios.

Adhering to these practices will enhance the reliability and performance of FPGA designs involving PLLs.

11.7 PLL IP module implementation

Clicking IP Core generator ,then search pll to find pll IP like figure below:

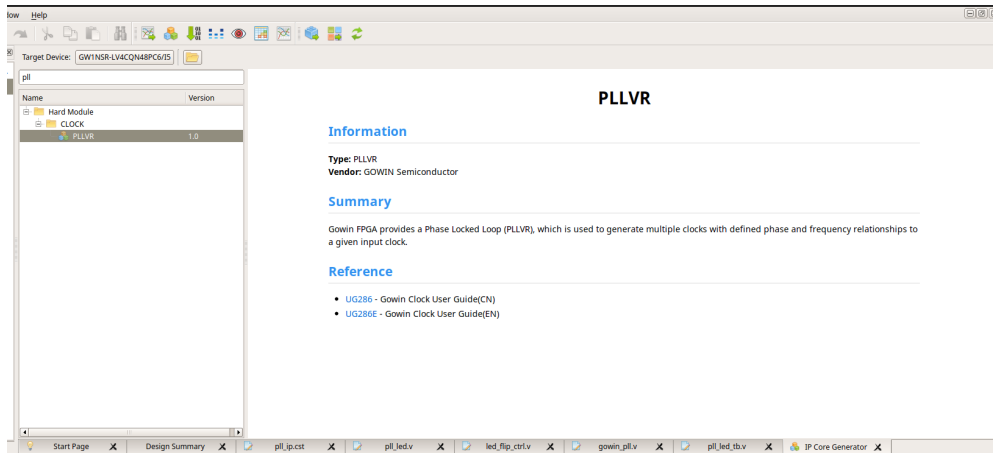


Figure 115: Finding PLL IP

Click the PLLVR you will see the table like figure below:

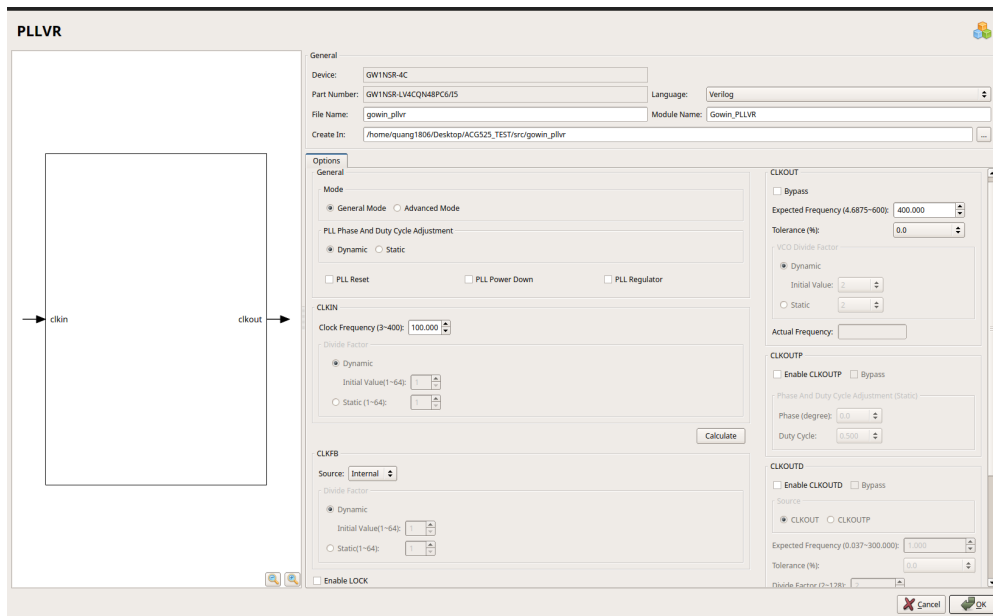


Figure 116: PLL IP setting table

1. The General Configuration Box is used to configure the relevant information of the generated IP design files. Readers can specify the file name, module name, and file save path in this dialog box.
2. Configure general mode and advanced mode. In general mode, the IP core will automatically

calculate the corresponding division factor based on the input clock and the expected output clock frequency to achieve the desired clock, as well as whether to adopt external feedback loops.

3. CLKIN:Configure the PLL input clock frequency and the division parameters.

- Clock Frequency: Configure the input clock frequency, here we select the 27M clock oscillator on the development board as the clock input source.
- Divide Factor: The division parameters can be configured in advanced mode, ranging from 1 to 64.

4. CLKFB:Configures the source and divide factor of rPLL.

- When configuring the source of the feedback clock, you can select Internal and External
- Divide Factor: Sets the Divide Factor as "Dynamic" or "Static" in advanced mode. In static mode, Divide Factor value can be set as a specific value, which ranges from 1 to 64. If the configuration is invalid, an error prompt will pop up when you click "Calculate" or "OK".

After that you can see the module and you can add it to your module:

```
Gowin_PLLVR your_instance_name(  
    .clkout(clkout), //output clkout  
    .lock(lock), //output lock  
    .clkkin(clkin), //input clkkin  
    .clkfb(clkfb) //input clkfb  
);
```

11.8 Conclusion

PLLs are indispensable components in FPGA-based systems, especially when dealing with high-speed interfaces like HDMI and camera modules. They provide the necessary clock generation and synchronization capabilities to ensure accurate data transmission and reception. Understanding and effectively implementing PLLs is crucial for the success of complex FPGA designs involving multimedia and communication interfaces.

12 VGA

12.1 Overview of VGA Development

12.1.1 Chapter Objective

This section introduces how to generate VGA display signals using FPGA, including:

- Principles of the VGA standard
- Synchronization and color signal structure
- Using counters to control display timing
- Laying the foundation for learning digital display interfaces like HDMI

12.1.2 Introduction to VGA

VGA (Video Graphics Array) is an **analog** display standard developed by IBM in 1987. Although HDMI and DisplayPort are more common today, many monitors still support VGA. It remains an ideal interface for learning basic video signal generation on FPGA.

Advantages:

- Simple interface
- Easy to debug using simulation or oscilloscope
- Easy to generate test patterns such as color bars

12.1.3 VGA Signal Pins

A typical VGA interface includes 5 key signals:

Signal	Description
HSYNC	Horizontal synchronization pulse
VSYNC	Vertical synchronization pulse
R	Red color channel (analog)
G	Green color channel (analog)
B	Blue color channel (analog)

12.1.4 Generating Analog Color Signals from FPGA

Since FPGAs output only digital signals, we need to convert them into analog voltages using a **resistor ladder DAC** (voltage divider network). This allows us to create analog signals for each color channel (R, G, B).

Example: Output 3 bits per color with different resistor values:

- MSB: $1k\Omega$
- Bit 1: $2k\Omega$
- LSB: $4k\Omega$

The combined voltage approximates an analog brightness level for that color.

12.1.5 Controlling VGA Timing with FPGA

Two counters are used:

- `h_cnt`: counts horizontal pixels per line
- `v_cnt`: counts vertical lines per frame

From these counters, we determine when to output color data (active display area) and when to generate synchronization pulses.

12.1.6 Why Learn VGA Before HDMI?

Although HDMI is a more modern and widely adopted standard, learning VGA first builds a strong foundation for understanding digital video systems in FPGA development.

Comparison Between VGA and HDMI:

Aspect	VGA	HDMI
Signal Type	Analog (RGB voltages)	Digital (TMDS encoded serial data)
Sync Signals	Separate HSYNC and VSYNC signals	HSYNC and VSYNC embedded in TMDS stream
Timing Structure	Uses visible area, front porch, sync pulse, back porch	Same timing structure
Pixel Clock	Used directly to control display timing	Pixel clock is base; TMDS clock is $10\times$ pixel clock
Data Transmission	Parallel analog lines (R, G, B)	High-speed serial transmission via 3 data + 1 clock channel
Complexity	Low: easier to implement using counters and resistors	High: requires PLL, TMDS encoder, DDR serializers
Ease of Debugging	Easy to debug using oscilloscope or LEDs	Harder to debug; requires logic analyzer or HDMI sink
Application for Learning	Ideal for beginners to understand timing and scan principles	Suitable after mastering VGA principles

Table 11: *Comparison of VGA and HDMI in FPGA design*

Conclusion: Learning VGA helps beginners:

- Understand raster-scan display timing
- Practice synchronizing video signals
- Work with counters and frame generation

These skills form the basis for mastering HDMI, where the same display concepts apply but require more advanced logic and encoding.

In our design, just apply the DVI_tx module to translate signal to TMDS

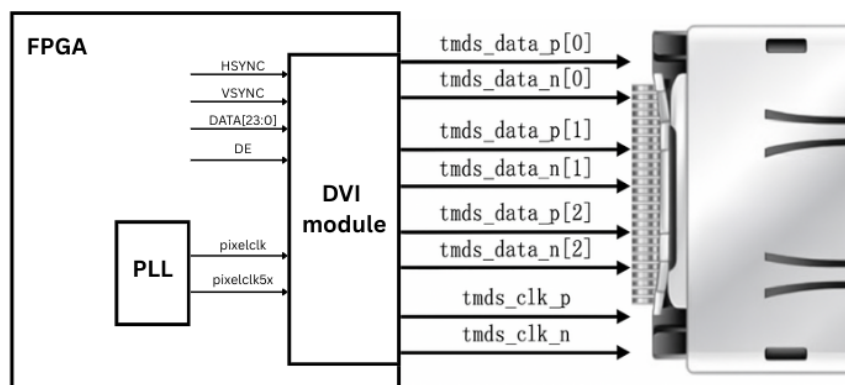


Figure 117: *hdmi module*

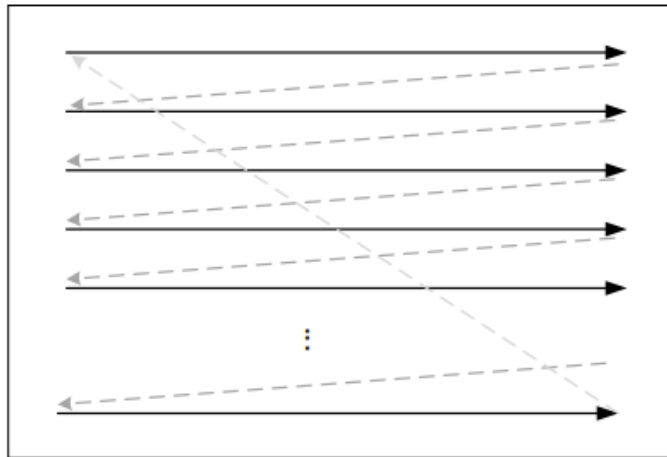
12.1.7 Conclusion

VGA is a simple analog interface, suitable for beginners in FPGA development. This section provides a foundational understanding of how to scan pixels, generate synchronization pulses, and control color output per pixel — essential concepts that will make learning HDMI much easier later.

12.2 VGA Display Imaging Principle

12.2.1 Raster Scan Principle

VGA monitors use a **raster scan** mechanism for imaging. This method scans pixels from left to right, line by line, from the top of the screen to the bottom. Once a frame is completed, the scan returns to the top-left corner to start the next frame. This process is repeated at a high frequency (typically 60Hz or more) to produce a stable image.

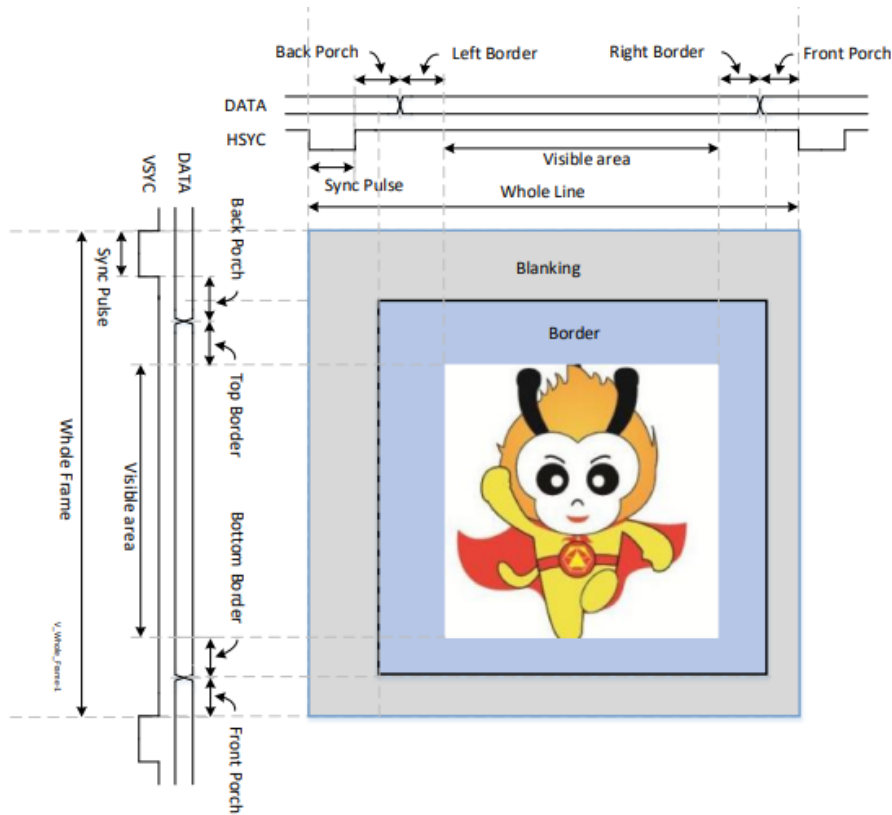
Figure 118: *Raster scan image*

12.2.2 Frame Structure and Timing Intervals

Region	Visible on Screen?	Role / Function
Active Area	Yes	Displays actual pixel data (RGB). This is the only part seen by the viewer.
Front Porch	No	A short pause after the active display before the sync pulse. Allows time for horizontal or vertical retrace to begin.
Sync Pulse (HSYNC / VSYNC)	No	Sends synchronization signal to inform the display to start a new line (HSYNC) or new frame (VSYNC).
Back Porch	No	Pause after the sync pulse to allow the display circuitry to settle before starting to show pixels again.

Table 12: *Roles of VGA Timing Regions*

Each frame on a VGA display consists of horizontal and vertical intervals. These intervals include the visible area where pixels are shown, and synchronization and porch intervals that control the timing.

**Figure 119:** *VGA Image display*

Horizontal Timing For a resolution of **640×480 @ 60Hz**, the horizontal timing is typically divided as follows:

Interval	Pixels (Clocks)
Visible Area	640
Front Porch	16
Sync Pulse (HSYNC)	96
Back Porch	48
Total	800

Table 13: *Horizontal timing structure (pixels per line)*

Vertical Timing Vertical timing (rows per frame) is also structured as follows:

Interval	Lines
Visible Area	480
Front Porch	10
Sync Pulse (VSYNC)	2
Back Porch	33
Total	525

Table 14: *Vertical timing structure (lines per frame)*

12.2.3 Synchronization Signals

Two key synchronization signals are generated:

- HSYNC – Indicates the end of a horizontal line.
- VSYNC – Indicates the end of a full frame.

These signals tell the display when to begin a new line or new frame.

12.2.4 Active Display Region

The **active display area** refers to the visible portion of the screen. Pixels are only displayed during this region. Outside of it (porches and sync pulses), the RGB signals are turned off or ignored.

12.2.5 Counter-Based Implementation in FPGA

In FPGA design, two counters are typically used:

- Horizontal counter (`h_cnt`) to track the current pixel position along the line.
- Vertical counter (`v_cnt`) to track the current line number.

By comparing counter values against timing thresholds, one can determine:

- When to assert HSYNC and VSYNC
- When to output RGB data (i.e., in the active area)

12.2.6 Timing Accuracy

To achieve 60Hz refresh rate and precise image generation, the system must operate with a **pixel clock of 25.175 MHz**. In most FPGA implementations, this is approximated to 25 MHz and generated using a PLL.

12.2.7 Conclusion

Understanding the VGA display's raster scan and timing structure is essential for generating accurate video signals. By mastering horizontal and vertical timing as well as synchronization principles, designers can create solid video output systems, which also form the basis for more complex standards such as HDMI.

12.3 VGA Timing Detail

12.3.1 VGA Timing Overview

In order to generate a standard VGA display (e.g., 640×480 @ 60Hz), accurate timing is essential. Each frame consists not only of the visible pixel area but also includes synchronization pulses and porches (blanking intervals). Understanding and calculating these parameters is key to proper image display on the screen.

12.3.2 Total Timing Calculation

The total time to complete one frame is determined by:

$$\text{Pixel Clock} = \text{Horizontal Total} \times \text{Vertical Total} \times \text{Frame Rate}$$

For a 640×480 resolution at 60Hz refresh rate, the timing parameters are as follows:

Horizontal Timing

- Active pixels: 640
- Front porch: 16
- Sync pulse (HSYNC): 96
- Back porch: 48
- **Total horizontal pixels: 800**

Vertical Timing

- Active lines: 480
- Front porch: 10
- Sync pulse (VSYNC): 2
- Back porch: 33
- **Total vertical lines: 525**

Resulting Pixel Clock

$$800 \times 525 \times 60 = 25,200,000 \text{ Hz} = 25.2 \text{ MHz}$$

This value is typically approximated to ****25 MHz**** in FPGA designs.

12.3.3 Timing Parameters Table

Parameter	Horizontal (pixels)	Vertical (lines)
Active Video	640	480
Front Porch	16	10
Sync Pulse	96	2
Back Porch	48	33
Total	800	525

Table 15: *VGA 640×480 Timing Parameters @ 60Hz*

Parameter	480×272	640×480	800×480	800×600	1024×600	1024×768	1280×720	1920×1080
H Right Borde	0	0	0	0	0	0	0	0
H Front Porch	2	8	40	40	24	24	110	88
H Sync Time	41	96	128	128	136	136	40	44
H Back Porch	2	40	88	88	160	160	220	148
H Left Border	0	8	0	0	0	0	0	0
H Data Time	480	640	800	800	1024	1024	1280	1920
H Total Time	525	800	1056	1056	1344	1344	1650	2200
V Bottom Borde	0	0	0	0	0	0	0	0
V Front Porch	2	2	2	1	1	1	5	4
V Sync Time	2	2	4	4	3	3	5	5
V Back Porch	2	25	25	23	38	38	20	36
V Top Border	0	8	0	0	0	0	0	0
V Data Time	272	480	480	600	600	768	720	1080
V Total Time	286	525	528	628	642	810	750	1125

Table 16: *User-defined VGA Timing Parameters for Common Resolutions*

12.3.4 Resolution vs. Total Frame Size

Note that:

- The resolution (e.g., 640×480) refers only to the visible pixel region.
- The counters in FPGA logic must cover the entire frame size (800×525), including blanking intervals.

12.3.5 Importance of Timing Accuracy

Accurate timing ensures:

- Correct synchronization with the display
- Stable image without tearing or distortion
- Proper placement of the visible image

12.3.6 640*480 resolution VGA controller timing analysis

Through the above analysis of the VGA timing, we have understood the basic characteristics of VGA timing. For a specific resolution timing standard, this refers to when the HS and VS pulse signals are generated.

At what time the valid image data is output is always certain, so we can use two counters to mark each time point in the entire cycle of the row and field according to the timing characteristics of that resolution. Then, at the corresponding time points, we can pull the HS and VS signals high or low, and output the RGB data to achieve the generation of VGA timing.

For line scanning, the main concern is when to generate the HS pulse signal and at what position within a line to output the image content. If we take the falling edge of the HS signal as the starting point of a line scan (this moment is considered as time 0), then we only need to know at what moment the HS pulse signal ends (VGA_HS_end), when the image data starts to output in a line (hdat_begin), when it ends outputting (hdat_end), and when the line scan stops (hpixel_end). Therefore, for the timing of a resolution of 640x480, we can deduce the following four parameters related to line image scanning:

Timing Node Parameter	VGA_HS_end	hdat_begin	hdat_end	hpixel_end
Timing Node Value (clk)	95	143	783	799
Composition of Timing Node	H Sync Time -1	VGA_HS_end + H Back Porch + H Left Border	hdat_begin + H Data Time	hdat_end + H Right Border + H Front Porch

Table 17: Horizontal Timing Node Parameters for VGA 640x480 @60Hz

Similarly, for field scanning, the main concern is at what moment the VS pulse signal is generated and at what position in a field the image content is output. If we take the falling edge of the VS signal as the starting moment of a line scan (which corresponds to 0 lines at this time), then

we only need to know at which line the VS pulse signal ends (VGA_VS_end), when the image data starts to be output in a line (vdat_begin), when it ends outputting (vdat_end), and until when the line scanning stops (vline_end). Therefore, based on the timing of 640x480 resolution, we can summarize the following four parameters related to field image scanning:

Timing Node Parameter	VGA_VS_end	vdat_begin	vdat_end	vline_end
Timing Node Value (clk)	1	34	514	524
Composition of Timing Node	V Sync Time -1	VGA_VS_end + V Back Porch + V Top Border	vdat_begin + V Data Time	vdat_end + V Bottom Border + V Front Porch

Table 18: Vertical Timing Node Parameters for VGA 640x480 @60Hz

12.4 Module design

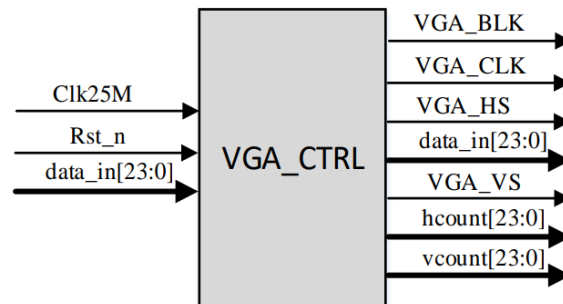


Figure 120: Caption

Port Name	Port Function
Clk25M	VGA pixel clock, 25MHz
Rst_n	System reset, active low
data_in[23:0]	Input port for display data
VGA_HS	VGA horizontal sync signal
VGA_VS	VGA vertical sync signal
VGA_RGB[23:0]	VGA 24-bit RGB color output
hcount[9:0]	Horizontal scan address in image area
vcount[9:0]	Vertical scan address in image area

Table 19: VGA Interface Signal Description

12.4.1 Line Scan Counter Design

The line scanning counter increments by 1 with each pixel clock. Once it reaches 799 (exactly 800 clock periods), which is after H Total Time clock cycles, the counter resets to zero and starts counting again. This part of the code.

It can be designed as follows:

```
always @(posedge Clk25M or negedge Rst_n) begin
    if (!Rst_n)
        hcount_r <= 10'd0;                // Reset
    else if (hcount_ov)
        hcount_r <= 10'd0;                // Restart after 799
    else
        hcount_r <= hcount_r + 10'd1;      // Increment every clock
end

assign hcount_ov = (hcount_r == hpixel_end); // Reset condition
```

12.4.2 Field scanning counter design

Since the field scan counter increments by 1 after each line scan is completed, the self-increment condition for the field scan counter is the overflow of the line scan counter. Therefore, the self-increment condition for the field scan counter is the completion of a line scan, which is "hcount_r == 10'd799". The code for the field scan counter is shown as follows:

```
always @(posedge Clk25M or negedge Rst_n) begin
    if (!Rst_n)
        vcount_r <= 10'd0;                // Reset
    else if (hcount_ov) begin              // Only count vertical on new line
        if (vcount_ov)
            vcount_r <= 10'd0;            // Restart after 524 lines
        else
            vcount_r <= vcount_r + 10'd1; // Increment line
    end
end
```

12.5 Synchronization signal design for operations

According to the VGA industrial standard timing, we know that each complete VGA frame includes a data segment and a blanking segment. During the blanking segment, the horizontal sync signal and the vertical sync signal each have a segment of horizontal sync pulse and vertical sync pulse. During the sync period, the corresponding horizontal sync signal or vertical sync signal

is at a low level. Therefore, we can determine the level state of the horizontal and vertical sync signals based on the values of the horizontal and vertical counters. For the horizontal sync signal, its horizontal sync pulse consists of the first 96 pixel clock cycles of a line scan, so the horizontal sync signal can be controlled in the following simple way:

```
assign VGA_HS=(hcount_r>10d95);
```

For field sync signals, its field sync head consists of the first 2 pixel clock cycles of a line scan, therefore the line sync signal can be controlled in the following simple way:

```
assign VGA_VS=(vcount_r>10d1);
```

12.5.1 Output data

The purpose of the VGA controller design is to drive the VGA display to show the required image content. Therefore, it is necessary to design the data output part, where the data source can be image signals generated from other parts, such as camera data or BMP image data. When driving the VGA, we only need to ensure that when scanning the correct pixel points, the image signals generated by other parts can correspond to the position of that pixel point, so it is not necessary to process the image data again. However, during the line and field blanking periods, it is essential to ensure that all data output to the RGB data lines of the VGA is 0. Therefore, a 2-to-1 multiplexer can be set, which allows the VGA controller to directly output the image data from other parts only during the non-blanking period, while during the blanking period, it forcibly outputs all 0.

We can first generate a valid flag signal for the image data, and then use that flag signal to control the content of the VGA output data, that is, to switch the channels of a two-to-one multiplexer, thus achieving the function of making the data all zero during the blanking period.

The code for generating the image data valid flag signal is shown below:

```
//Data and synchronization signal transmission
assign dat_act=((hcount_r>=10'd143)&&(hcount_r<10'd783))
&&((vcount_r>=10'd34)&&(vcount_r<10'd514));
```

dat_act is the valid flag signal for image data.

The VGA_BLANK signal and the dat_act signal serve the same function, so you can directly assign the dat_act signal to VGA_BLANK.

```
assign VGA_BLK = dat_act;
```

In addition, during the blanking period, you can also set the data on the RGB output port to 0 to force the blanking effect.

The code for the forced output 0 two-choice multiplexer is as follows:

```
assign VGA_RGB=(dat_act)?data_in:24'h000000;
```

Among them, VGA_RGB is the data outputted to the VGA interface, while data_in is the correct image data passed from other modules.

12.5.2 Output row and column scanning position

In order for other modules to correctly output image data based on the current scanning position, it is necessary to output the real-time scanning position of the VGA controller for use by other modules.

```
assign hcount=dat_act?(hcount_r- 10'd143):10'd0;  
assign vcount=dat_act?(vcount_r- 10'd34):10'd0;
```

12.5.3 Output data latch clock signal

For the VGA controller implemented on the FPGA, data is output on the rising edge of the clock signal. However, for the DAC chip, data needs to be sampled at the midpoint of the data. Based on the actual board-level debugging results, the clock of the VGA controller is inverted and then output as the data latch clock signal for the DAC, which ensures that the data is stable exactly when the DAC chip is latching the data, as shown below:

```
// Invert clock to generate VGA DAC latch clock  
assign VGA_CLK = ~Clk25M;
```

12.5.4 VGA Controller Design Example

The above is the driving circuit designed based on intuitive thinking. In the code, digital values are used directly for operations and comparisons. This coding method, while very intuitive, is not very conducive to modification. Therefore, to achieve an easily modifiable controller design that can be conveniently adapted to other resolutions after simple modifications, we optimized the code using parametric design. Some time-related numbers used in the code are defined directly using parameters like 'parameter'. This way, when modifications to the time parameters are needed in the future, only the parameter definitions need to be changed, and there is no need to go into

the code to modify each instance individually. We will not explain how to modify everything here; instead, we will provide the final modified code for your comparison and understanding.

```
//=====
// MODULE: VGA_CTRL
// DESCRIPTION: VGA driver for 640x480 @60Hz display timing
//=====
module VGA_CTRL(
    Clk25M,    // 25MHz system input clock
    Rst_n,     // Active-low reset
    data_in,   // 24-bit pixel input data (RGB)
    hcount,    // Horizontal pixel position (only valid during active video)
    vcount,    // Vertical line position (only valid during active video)
    VGA_RGB,   // VGA RGB output
    VGA_HS,    // VGA horizontal sync pulse
    VGA_VS,    // VGA vertical sync pulse
    VGA_BLK,   // VGA active display signal
    VGA_CLK    // VGA pixel clock (to DAC)
);

//===== Input Ports =====
input Clk25M;           // 25 MHz pixel clock
input Rst_n;           // Active-low reset
input [23:0] data_in;   // RGB pixel data: {R[7:0], G[7:0], B[7:0]}
//===== Output Ports =====
output [9:0] hcount;    // Pixel X position in visible area
output [9:0] vcount;    // Pixel Y position in visible area
output [23:0] VGA_RGB;  // VGA output RGB color
output VGA_HS;         // Horizontal sync
output VGA_VS;         // Vertical sync
output VGA_BLK;        // High when in visible region (blanking = low)
output VGA_CLK;        // Output pixel clock (for VGA DAC latch)
// Invert clock to generate VGA DAC latch clock
assign VGA_CLK = ~Clk25M;
//===== Internal Registers =====
reg [9:0] hcount_r;     // Horizontal pixel counter (0~799)
reg [9:0] vcount_r;     // Vertical line counter (0~524)
//===== Internal Wires =====
wire hcount_ov;        // Horizontal counter overflow
wire vcount_ov;        // Vertical counter overflow
wire dat_act;          // High when in active video region
//===== VGA 640x480 Timing Parameters =====
parameter VGA_HS_end = 10'd95, // Horizontal sync pulse ends at 95
           hdat_begin = 10'd143, // Visible area starts at 143
```

```
        hdat_end = 10'd783, // Visible area ends at 782 (640 pixels)
        hpixel_end = 10'd799, // Total horizontal pixels = 800
        VGA_VS_end = 10'd1, // Vertical sync pulse ends at line 1
        vdat_begin = 10'd34, // Visible vertical lines start at 34
        vdat_end = 10'd514, // Visible vertical lines end at 513 (480 lines)
        vline_end = 10'd524; // Total vertical lines = 525
//===== Display Position in Active Area =====
assign hcount = dat_act ? (hcount_r - hdat_begin) : 10'd0;
assign vcount = dat_act ? (vcount_r - vdat_begin) : 10'd0;
//===== Horizontal Counter =====
always @(posedge Clk25M or negedge Rst_n) begin
    if (!Rst_n)
        hcount_r <= 10'd0; // Reset
    else if (hcount_ov)
        hcount_r <= 10'd0; // Restart after 799
    else
        hcount_r <= hcount_r + 10'd1; // Increment every clock
end
assign hcount_ov = (hcount_r == hpixel_end); // Reset condition
//===== Vertical Counter =====
always @(posedge Clk25M or negedge Rst_n) begin
    if (!Rst_n)
        vcount_r <= 10'd0; // Reset
    else if (hcount_ov) begin // Only count vertical on new line
        if (vcount_ov)
            vcount_r <= 10'd0; // Restart after 524 lines
        else
            vcount_r <= vcount_r + 10'd1; // Increment line
    end
end
assign vcount_ov = (vcount_r == vline_end); // Reset condition
//===== Active Display Region =====
assign dat_act = ((hcount_r >= hdat_begin) && (hcount_r < hdat_end)) &&
                ((vcount_r >= vdat_begin) && (vcount_r < vdat_end));
assign VGA_BLK = dat_act; // High during visible display
//===== Sync Signal Generation =====
assign VGA_HS = (hcount_r > VGA_HS_end); // Active-low HSYNC
assign VGA_VS = (vcount_r > VGA_VS_end); // Active-low VSYNC
//===== RGB Output =====
assign VGA_RGB = dat_act ? data_in : 24'h000000; // Output pixel color if in active
area

endmodule
```

12.5.5 Simulation

Coding testbench for VGA:

```
`timescale 1ns/1ns
`define clk_period 40
module VGA_CTRL_tb;
    //----- Module Input Ports -----
    reg Clk25M;          // System input clock 25MHz
    reg Rst_n;
    reg [23:0]data_in;   // Data to be displayed
    //----- Module Output Ports -----
    wire [9:0]hcount;
    wire [9:0]vcount;
    wire [23:0]VGA_RGB;  // VGA data output
    wire VGA_HS;         // VGA horizontal sync signal
    wire VGA_VS;         // VGA vertical sync signal
    wire VGA_BLK;        // VGA vertical blanking signal
    wire VGA_CLK;        // VGA DAC output clock
    reg [11:0]V_cnt = 0; // Frame line scan counter
    VGA_CTRL VGA_CTRL(
        .Clk25M(Clk25M), // System input clock 25MHz
        .Rst_n(Rst_n),
        .data_in(data_in), // Data to be displayed
        .hcount(hcount), // VGA horizontal scan counter
        .vcount(vcount), // VGA vertical scan counter
        .VGA_RGB(VGA_RGB), // VGA data output
        .VGA_HS(VGA_HS), // VGA horizontal sync signal
        .VGA_VS(VGA_VS), // VGA vertical sync signal
        .VGA_BLK(VGA_BLK), // VGA vertical blanking signal
        .VGA_CLK(VGA_CLK) // VGA DAC output clock
    );
    initial Clk25M = 0;
    always #(`clk_period/2) Clk25M = ~Clk25M;
    initial begin
        Rst_n = 0;
        data_in = 24'd0;
        #(`clk_period *20 +1);
        Rst_n = 1;
        data_in = 24'hffffff;
    end
    initial begin
        wait(V_cnt == 3); // Wait until 2 frames are scanned before ending simulation
        $stop;
    end
endmodule
```

```

end
always @(posedge VGA_VS) // Count total number of scanned frames
    V_cnt <= V_cnt + 1'b1;
endmodule

```

- VGA_HS signal

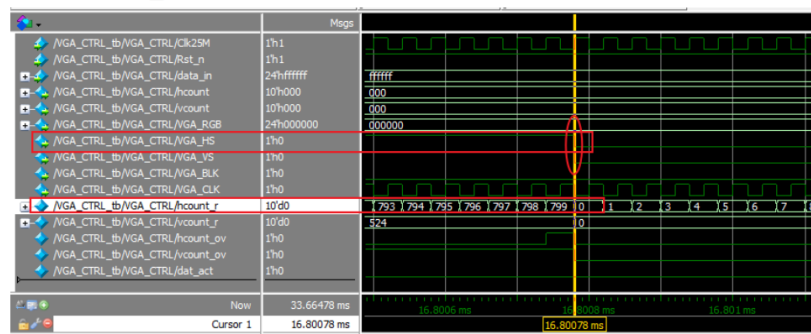


Figure 121: *Caption*

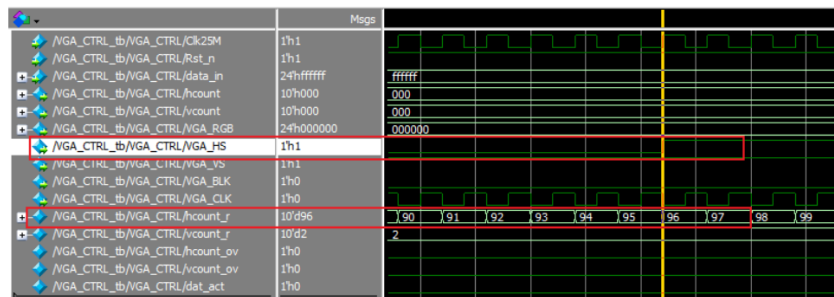


Figure 122: *Caption*

- As can be seen from the diagram, VGA_HS is at a low level in the horizontal scanning segment from 0 to 95, which is the line sync pulse; at other times, it is at a high level. The line scanning occurs sequentially, and the maximum count of the line scan counter is 799, which corresponds exactly to 800 pixel clock cycles, consistent with the design. Therefore, it can be concluded that the line scanning signal meets the timing design requirements.

- VGA_VS

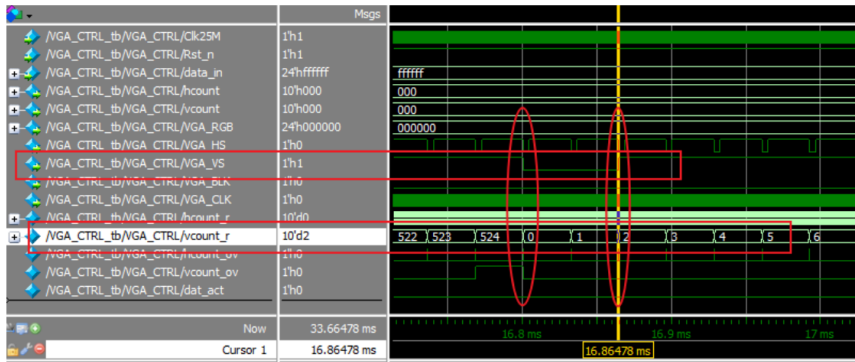


Figure 123: Caption

- As seen in the figure, the VGA_VS signal is at a low level during the field scanning time of 0-1, which indicates the field sync head, and is at a high level during the other times. For one field scan, the maximum count of the field scan counter is 524, which corresponds exactly to 525 line scanning cycles, consistent with the design and meeting the VGA industry standard. Therefore, it can be concluded that the field scanning signal satisfies the timing design requirements.
- Other signals will not be analyzed and compared in detail in this text. During board-level debugging, if the display effect is not correct, you can determine the error location based on the actual display effect, such as errors in the horizontal sync signal, vertical sync signal, etc.

13 Multi-resolution adaptive VGA controller design

In the last section, we explained in detail the origin and development of the VGA timing standard, and implemented a VGA controller for the standard VGA resolution (640*480) in an FPGA, verifying its correctness through simulation and board-level validation. However, many readers may also know that in today's era where 4K resolution is becoming nearly universal, a low-resolution display like 640*480 not only seems technically outdated but also provides poor display quality, which is hard to meet the increasingly demanding aesthetic standards of the human eye.

Given that the HDMI and TFT screen display system has a very wide application in digital systems, mastering the driving and application of TFT screens and HDMI is of great practical significance. Therefore, this section will learn how to use FPGA for RGB interface TFT screen driver design.

13.1 The design purpose of a multi-resolution adaptable VGA controller

To achieve 4K resolution output, it is nearly impossible to do so using traditional VGA drivers and transmission methods. However, for slightly lower resolutions such as 1080p (1920*1080 resolution) and 720p (1280*720 resolution), using VGA interfaces and transmission methods can still meet application needs. Currently, these two resolutions are also widely used in the multimedia and industrial control industries. Therefore, in this section, we will design a configurable VGA controller that supports resolutions from 480*272 up to 1920*1080.

The content of this section will be modified based on the content of the previous section. In the previous section "Common Timing Parameters for Various Resolutions," several numerical values of timing parameters corresponding to common resolutions were provided. Additionally, when designing a 640*480 resolution VGA controller, the function of each timing parameter in the controller logic was also analyzed. Therefore, during implementation, it is only necessary to modify the values of these timing parameter parameters according to different resolutions.

Here are the timing parameter values for various common resolutions listed again for everyone's further familiarity. The parameters related to rows are all based on the pixel update frequency, which is measured in pixel clock units.

	480	640	800	800	1024	1024	1280	1920
	272	480	480	600	600	768	720	1080
H Right Border	0	8	0	0	0	0	0	0
H Front Porch	2	8	40	40	24	24	110	88
H Sync Time	41	96	128	128	136	136	40	44
H Back Porch	2	40	88	88	160	160	220	148
H Left Border	0	8	0	0	0	0	0	0
H Data Time	480	640	800	800	1024	1024	1280	1920
H Total Time	525	800	1056	1056	1344	1344	1650	2200
V Bottom Border	0	8	8	0	0	0	0	0
V Front Porch	2	2	2	1	1	3	5	4
V Sync Time	10	2	2	4	4	6	5	5
V Back Porch	2	25	25	23	23	29	20	36
V Top Border	0	8	8	0	0	0	0	0
V Data Time	272	480	480	600	600	768	720	1080
V Total Time	286	525	525	628	628	806	750	1125

Figure 124: Common Resolution Timing Parameter Table

In the section "Timing Analysis of the VGA Controller at 640*480 Resolution", we realized the generation of various signals from the VGA controller and the accurate output of data by defining time node parameters such as VGA_HS_end, hdat_begin, hdat_end, hpixel_end, VGA_VS_end, vdat_begin, vdat_end, and vline_end. These time node parameters are obtained through simple addition operations from various border, porch, and timing parameters in the VGA resolution. Therefore, the first step in configurable parametric design is to represent the values of each time node parameter using the timing parameters in the VGA timing standard. Below are the relationships between several time node parameters and the VGA timing standard parameters.

```

hdat_begin = H_Sync_Time + H_Back_Porch + H_Left_Border - 1'b1;
hdat_end   = H_Total_Time - H_Right_Border - H_Front_Porch - 1'b1;
vdat_begin = V_Sync_Time + V_Back_Porch + V_Top_Border - 1'b1;
vdat_end   = V_Total_Time - V_Bottom_Border - V_Front_Porch - 1'b1;

```

Since $VGA_HS_end = H_Sync_Time - 1$, $VGA_VS_end = V_Sync_Time - 1$, $hpixel_end = H_Total_Time - 1$, and $vline_end = V_Total_Time - 1$, these timing node parameter values are not listed separately, but are instead directly represented by the timing parameter values.

With the above relationship expressions, we only need to set the values of the Border, Porch, and time parameters to the corresponding values for each resolution when using different resolutions. To implement this design in the simplest way, we can achieve it through predefined and condition-

compiled methods.

13.2 VGA timing parameters based on conditional compilation

Similarly, we only need to define the flags for each resolution as conditions in conditional compilation, and write the values of the various timing parameters for that resolution under each condition. Then, during usage, we only need to activate the flag for a certain resolution in the code, allowing all timing parameter values under that condition to be effective, thus achieving a "one-key" resolution modification. The following is the code to implement the definition of these parameter contents.

```
=====*/  
//`include "sys_cfg.v"  
// It is recommended to modify various parameter configurations directly in the sys_cfg  
// file,  
// rather than modifying them here.  
/*  
Select a few predefined parameters based on actual project requirements.  
MODE_RGB888 and MODE_RGB565: choose one of the two to decide whether the driver works  
in 16-bit or 24-bit color mode.  
For the series of display devices provided by Xiaomeige, the device parameters are as  
follows:  
4.3 inch screen: 16-bit RGB565 mode  
5 inch screen: 16-bit RGB565 mode  
GM7123 module: 24-bit RGB888 mode  
Resolution_xxxx macros define the resolution of the display devices.  
Common resolutions include:  
4.3-inch TFT screen: Resolution_480x272  
5-inch TFT screen: Resolution_800x480  
Common VGA resolutions:  
Resolution_640x480  
Resolution_800x600  
Resolution_1024x768  
Resolution_1280x720  
Resolution_1920x1080  
*/  
`define HW_VGA  
// Choose one of the following two display modes. Enable one and comment out the other.  
`define MODE_RGB888  
//`define MODE_RGB565  
// Choose one of the following 7 resolution options. Enable one and comment out the  
rest.
```

```
`ifdef HW_TFT43 // Using 4.3" 480x272 resolution display
    `define Resolution_480x272 1 // Clock = 9 MHz

`elsif HW_TFT50 // Using 5" 800x480 resolution display
    `define Resolution_800x480 1 // Clock = 33 MHz
`elsif HW_VGA // Using VGA monitor; default resolution is 640x480. Other resolutions
    are optional,
        // but current camera capture module does not adapt to them.
    //`define Resolution_640x480 1 // Clock = 25.175 MHz
    `define Resolution_800x480 1 // Clock = 33 MHz
    //`define Resolution_800x600 1 // Clock = 40 MHz
    //`define Resolution_1024x600 1 // Clock = 51 MHz
    //`define Resolution_1024x768 1 // Clock = 65 MHz
    //`define Resolution_1280x720 1 // Clock = 74.25 MHz
    //`define Resolution_1920x1080 1 // Clock = 148.5 MHz
`endif
// Define different color bit depths
`ifdef MODE_RGB888
    `define Red_Bits 8
    `define Green_Bits 8
    `define Blue_Bits 8
`elsif MODE_RGB565
    `define Red_Bits 5
    `define Green_Bits 6
    `define Blue_Bits 5
`endif
// Timing parameters for different resolutions
`ifdef Resolution_480x272
    `define H_Total_Time 12'd525
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd2
    `define H_Sync_Time 12'd41
    `define H_Back_Porch 12'd2
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd286
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd2
    `define V_Sync_Time 12'd10
    `define V_Back_Porch 12'd2
    `define V_Top_Border 12'd0

`elsif Resolution_640x480
```

```
`define H_Total_Time 12'd800
`define H_Right_Border 12'd8
`define H_Front_Porch 12'd8
`define H_Sync_Time 12'd96
`define H_Back_Porch 12'd40
`define H_Left_Border 12'd8

`define V_Total_Time 12'd525
`define V_Bottom_Border 12'd8
`define V_Front_Porch 12'd2
`define V_Sync_Time 12'd2
`define V_Back_Porch 12'd25
`define V_Top_Border 12'd8

`elsif Resolution_800x480
`define H_Total_Time 12'd1056
`define H_Right_Border 12'd0
`define H_Front_Porch 12'd40
`define H_Sync_Time 12'd128
`define H_Back_Porch 12'd88
`define H_Left_Border 12'd0

`define V_Total_Time 12'd525
`define V_Bottom_Border 12'd8
`define V_Front_Porch 12'd2
`define V_Sync_Time 12'd2
`define V_Back_Porch 12'd25
`define V_Top_Border 12'd8

`elsif Resolution_800x600
`define H_Total_Time 12'd1056
`define H_Right_Border 12'd0
`define H_Front_Porch 12'd40
`define H_Sync_Time 12'd128
`define H_Back_Porch 12'd88
`define H_Left_Border 12'd0

`define V_Total_Time 12'd628
`define V_Bottom_Border 12'd0
`define V_Front_Porch 12'd1
`define V_Sync_Time 12'd4
`define V_Back_Porch 12'd23
`define V_Top_Border 12'd0
```

```
`elsif Resolution_1024x600
    `define H_Total_Time 12'd1344
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd24
    `define H_Sync_Time 12'd136
    `define H_Back_Porch 12'd160
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd628
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd1
    `define V_Sync_Time 12'd4
    `define V_Back_Porch 12'd23
    `define V_Top_Border 12'd0

`elsif Resolution_1024x768
    `define H_Total_Time 12'd1344
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd24
    `define H_Sync_Time 12'd136
    `define H_Back_Porch 12'd160
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd806
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd3
    `define V_Sync_Time 12'd6
    `define V_Back_Porch 12'd29
    `define V_Top_Border 12'd0

`elsif Resolution_1280x720
    `define H_Total_Time 12'd1650
    `define H_Right_Border 12'd0
    `define H_Front_Porch 12'd110
    `define H_Sync_Time 12'd40
    `define H_Back_Porch 12'd220
    `define H_Left_Border 12'd0

    `define V_Total_Time 12'd750
    `define V_Bottom_Border 12'd0
    `define V_Front_Porch 12'd5
    `define V_Sync_Time 12'd5
```

```
`define V_Back_Porch 12'd20
`define V_Top_Border 12'd0

`elsif Resolution_1920x1080
`define H_Total_Time 12'd2200
`define H_Right_Border 12'd0
`define H_Front_Porch 12'd88
`define H_Sync_Time 12'd44
`define H_Back_Porch 12'd148
`define H_Left_Border 12'd0

`define V_Total_Time 12'd1125
`define V_Bottom_Border 12'd0
`define V_Front_Porch 12'd4
`define V_Sync_Time 12'd5
`define V_Back_Porch 12'd36
`define V_Top_Border 12'd0
`endif
```

This Verilog header file is used to configure hardware settings for display output in FPGA projects. It allows selection between two color modes: RGB888 (24-bit, with 8 bits for red, green, and blue) and RGB565 (16-bit, using 5 bits for red, 6 for green, and 5 for blue). Depending on the hardware in use—such as a 4.3-inch TFT, 5-inch TFT, or a VGA monitor—one of several predefined resolution settings can be activated. These include common display resolutions like 480×272 , 800×480 , 640×480 , 1024×768 , and up to 1920×1080 .

13.3 Design of VGA controller with multi-resolution support based on conditional compilation

In this way, when the VGA controller code is actually used, it is only necessary to define the relationships between `VGA_HS_end`, `hdat_begin`, `hdat_end`, `hpixel_end`, `VGA_VS_end`, `vdat_begin`, `vdat_end`, `vline_end`, and these timing parameters to support different resolutions. The complete controller code is shown below:

```
`include "disp_parameter_cfg.v"

module disp_driver(
    ClkDisp,
    Rst_n,
    Data,
    DataReq,
```

```
H_Addr,
V_Addr,
Disp_HS,
Disp_VS,
Disp_Red,
Disp_Green,
Disp_Blue,
Disp_DE,
Disp_PCLK
);
input ClkDisp;
input Rst_n;
input [`Red_Bits + `Green_Bits + `Blue_Bits - 1:0] Data;
output DataReq;
output [11:0] H_Addr;
output [11:0] V_Addr;
output reg Disp_HS;
output reg Disp_VS;
output reg [`Red_Bits - 1 :0]Disp_Red;
output reg [`Green_Bits - 1 :0]Disp_Green;
output reg [`Blue_Bits - 1 :0]Disp_Blue;
output reg Disp_DE;
output Disp_PCLK;
wire hcount_ov;
wire vcount_ov;
//----- Internal register definitions -----
reg [11:0] hcount_r; // Line scan counter
reg [11:0] vcount_r; // Frame scan counter
`ifdef HW_VGA
    assign Disp_PCLK = ~ClkDisp;
`else
    assign Disp_PCLK = ClkDisp;
`endif
assign DataReq = Disp_DE;

parameter hdat_begin = `H_Sync_Time + `H_Back_Porch + `H_Left_Border - 1'b1;
parameter hdat_end = `H_Total_Time - `H_Right_Border - `H_Front_Porch - 1'b1;
parameter vdat_begin = `V_Sync_Time + `V_Back_Porch + `V_Top_Border - 1'b1;
parameter vdat_end = `V_Total_Time - `V_Bottom_Border - `V_Front_Porch - 1'b1;
assign H_Addr = Disp_DE?(hcount_r - hdat_begin):12'd0;
assign V_Addr = Disp_DE?(vcount_r - vdat_begin):12'd0;
// Horizontal scanning
assign hcount_ov = (hcount_r >= `H_Total_Time - 1);
```

```
always@(posedge ClkDisp or negedge Rst_n)
if(!Rst_n)
    hcount_r <= 0;
else if(hcount_ov)
    hcount_r <= 0;
else
    hcount_r <= hcount_r + 1'b1;

// Vertical scanning
assign vcount_ov = (vcount_r >= `V_Total_Time - 1);

always@(posedge ClkDisp or negedge Rst_n)
if(!Rst_n)
    vcount_r <= 0;
else if(hcount_ov) begin
    if(vcount_ov)
        vcount_r <= 0;
    else
        vcount_r <= vcount_r + 1'd1;
end
else
    vcount_r <= vcount_r;

always@(posedge ClkDisp)
    Disp_DE <= ((hcount_r >= hdat_begin)&&(hcount_r < hdat_end))
                &&((vcount_r >= vdat_begin)&&(vcount_r < vdat_end));

always@(posedge ClkDisp) begin
    Disp_HS <= (hcount_r > `H_Sync_Time - 1);
    Disp_VS <= (vcount_r > `V_Sync_Time - 1);
    {Disp_Red,Disp_Green,Disp_Blue} <= (Disp_DE)?Data:1'd0;
end
endmodule
```

In order to enhance the readability of the code, the timing parameter values for various resolutions are separately written in a file named 'disp_parameter_cfg.v', and then at the beginning of the files where these parameters are used, simply include this file using 'include'.

Based on the above analysis, using a multi-resolution adaptive controller can effectively resolve the pixel issues of different displays under various resolution conditions. The principles learned in the VGA section can also be well adapted to the subsequent TFT and HDMI sections.

14 FPGA-based HDMI/DVI display

With the development of display technology, the data transmission methods and capabilities closely related to it are also continuously improving. The transmission method has evolved from the earliest VGA analog signal transmission to DVI, HDMI, DisplayPort interfaces, etc., with significant improvements in data quality and data capacity. The changes in transmission methods are mainly related to imaging principles and data rates.

14.1 FPGA-Based DVI Circuit Design

On the FPGA onekiwi 4K development board, a simple HDMI sending interface is designed. Using this interface, the FPGA can achieve image output of up to 1280*720@60Hz with only 8 I/O.

In order to achieve HDMI transmission while saving IO resources, some FPGA manufacturers and developers use FPGAs to implement the TMDS encoding and serial transmitter required for HDMI transmission. This design approach implements the core functions of the HDMI transmission chip internally within the FPGA, and ultimately uses the FPGA pins to output HDMI link signals that meet HDMI protocol specifications. Using this solution not only reduces the consumption of FPGA pin resources, but also simplifies circuit design and lowers hardware costs. In fact, even using low-end FPGA devices, such as Cyclone IV E or Spartan-6, it is possible to achieve image transmission of up to 720P. Moreover, using higher-performance FPGA chips or FPGA chips with high-speed transceivers can increase the possible image sizes transmitted. Therefore, mastering the method of using FPGA programming to implement an HDMI interface is not only practical but also cost-effective. The following diagram illustrates the concept of implementing an HDMI interface directly through FPGA programming.

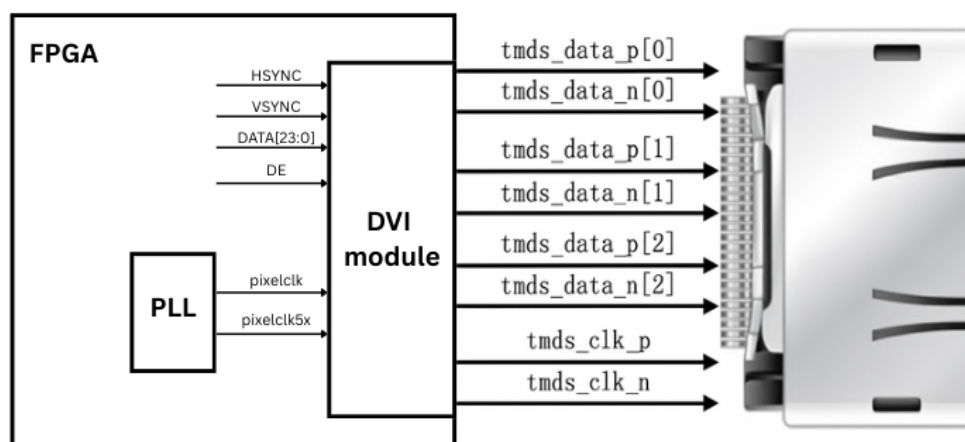


Figure 125: *TMDS encoder and serial transmitter*

The figure below is schematic of HDMI module in FPGA Onekiwi 4K:

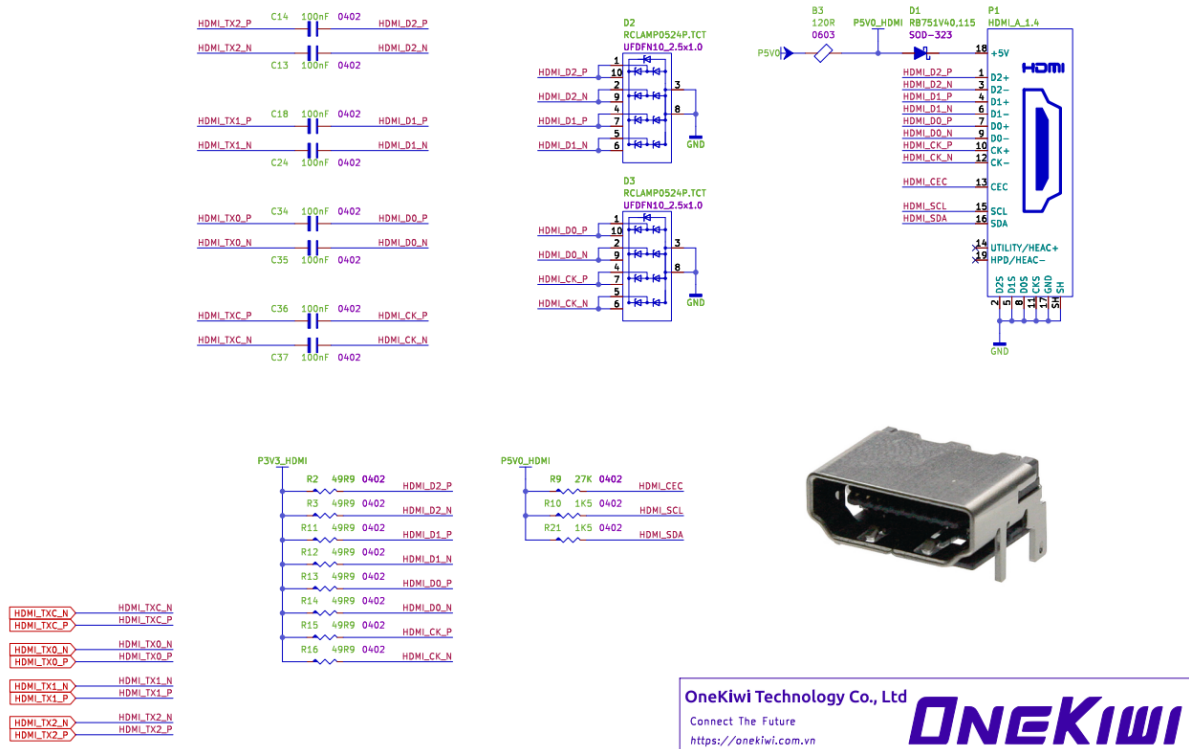


Figure 126: *Differential Line HDMI Transmission Circuit Design*

14.2 The differences and connections between HDMI and DVI

Meaning of DVI interface

DVI stands for Digital Visual Interface. It is a standard used to transmit digital video signals from a device (like a computer or FPGA board) to a monitor or display. DVI uses a method called TMDS (Transition-Minimized Differential Signaling) to send pixel data such as red, green, blue (RGB), and synchronization signals (HSync, VSync). DVI only carries video — it does not support audio. It was widely used before HDMI became popular. It's very suitable for FPGA projects because it is purely digital and relatively simple.

Meaning of HDMI Interface

High Definition Multimedia Interface (HDMI) is a fully digital interface for transmitting video and audio, capable of delivering uncompressed audio and video signals. The HDMI interface can provide data transfer bandwidth of up to 5Gbps, allowing for the transmission of uncompressed audio signals and high-resolution video signals. Additionally, it does not require analog-to-digital

or digital-to-analog conversion before signal transmission, ensuring the highest quality audio and video signal delivery.

Relationship between HDMI and DVI

- HDMI was developed based on DVI.
- The video signal format is the same: both use TMDS.
- That means: if your FPGA can output a valid DVI signal (video only), you can connect it to an HDMI monitor, and it will display just fine.

14.3 Introduction to DVI Data Link

Whether it's the HDMI or DVI specification, both use the TMDS encoding method for their data link layer. The output transmission process also includes the input interface layer, TMDS transmitter, TMDS receiver, and output interface layer.

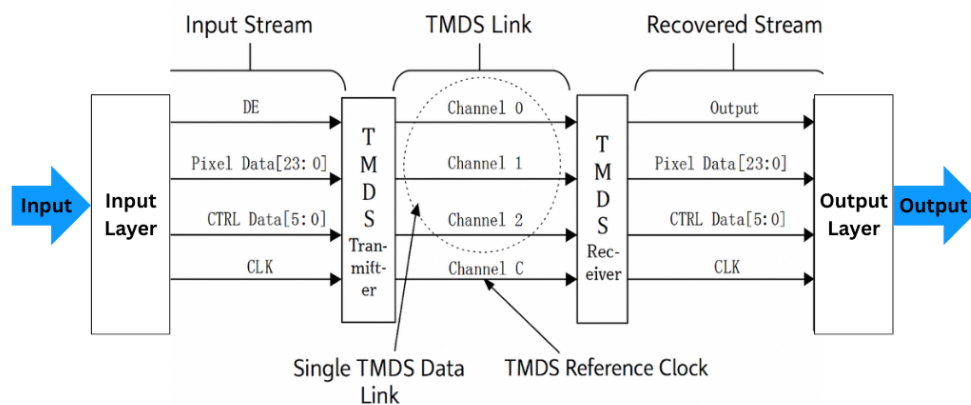


Figure 127: DVI data link

Input Interface Layer

The signal format of the input interface layer is a typical RGB line and field synchronization digital interface, which includes data.Enable signal DE (Data Enable), 24-bit pixel data (Pixel Data), 6-bit control data (including HSYNC, VSYNC, and blanking signal), and sync clock signal.

TMDS transmitter

The TMDS transmitter completes the encoding of data and control signals from the input interface layer according to the TMDS encoding method, and then outputs the encoded data through a high-speed serial interface, ultimately encoding the signals from the input interface

layer into four TMDS link layers. The encoding and serialization process of the signals from the input interface layer to the final TMDS link layer will be introduced in the next section.

TMDS receiver

In contrast to the TMDS transmitter, the function of the TMDS receiver is to receive high-speed serial data from the TMDS link, deserialize it, decode the TMDS signal, and obtain the same control signals and data as those at the input interface layer.

Output interface layer

The output interface layer transmits the data stream and control signals decoded from the TMDS receiver to the final data consumer, such as an LCD panel with an RGB interface.

14.4 Principles and Implementation of TMDS

In the introduction of the DVI data link layer above, it was mentioned that the TMDS transmitter encodes the data and control signals and then serializes them for transmission. So what is TMDS encoding? What kind of encoding method is TMDS encoding?

First of all, TMDS encoding includes two major components: the control segment for transmitting control signals and the data segment used to transmit image pixel data. A complete image transmission TMDS module consists of three identical encoding and sending modules. Each sending module contains 8 bits of pixel data, corresponding to the 8 bits of a single color in the 24-bit pixel data. There are also 2 control signals, which can correspond to horizontal synchronization (HSYNC) and vertical synchronization (VSYNC) signals, or may be set to 0. Additionally, there is a data enable signal (DE), which is used to distinguish between control data and pixel data. When the DE signal is high, it indicates that the current data is valid, and the encoder encodes the 8 bits of Data. When the DE signal is low, it encodes the 2 bits of control signals.

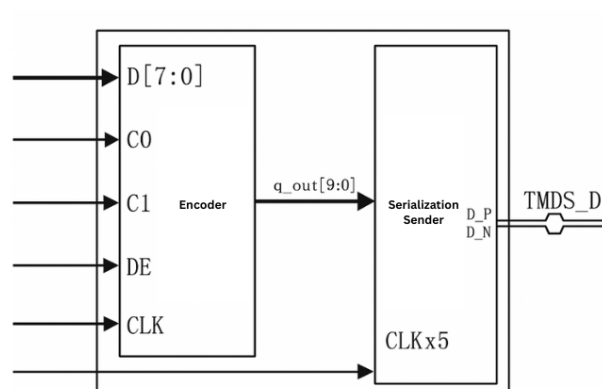


Figure 128: *TMDS Principle and Implementation Input-Output Schematic Diagram*

Here, the clock signal frequency used by the serial transmitter is 5 times that of the encoder. The reason it is 5 times is that the serial transmitter uses a double data rate transmission method when sending data, meaning that 2 bits of signal can be transmitted in one clock cycle. Therefore, only 5 times the clock of the encoder is needed to accomplish the timely transmission of 10 bits of data. The detailed sending principles of the serial transmitter are explained in the subsequent sections of this chapter.

The figure below is a diagram illustrating the transmission of video data using the TMDS method. From the perspective of the encoder, the TMDS transmitter divides the image data and control signals into 3 groups, with each group using one encoder plus a serial transmitter. Each encoder consists of 8 bits of image data and 2 bits of control signals. Additionally, it outputs a serial transmission clock, serving as the synchronization clock for the 3 data/control channels.

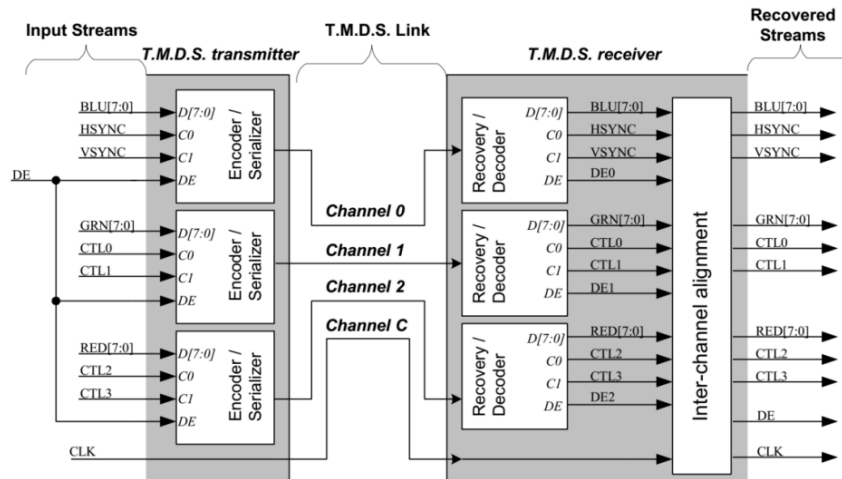


Figure 129: Overview of TMDS transmission channel encoding and decoding

In the first group, the blue (BLUE) component of the pixel data is combined with the line sync (HSYNC) and frame sync (VSYNC) signals from the control signal into one group, which is then output through an encoder and a serial modulator, called Channel 0.

In the second group, the green (GREEN) component in the pixel data along with two empty control signals CTL0 and CTL1 are grouped together. After passing through the encoder and serial modulator, it outputs, named Channel 1. In this group, CTL0 and CTL1 are connected to empty signals.

In the third group, the red (RED) component of the pixel data and two empty control signals CTL2 and CTL3 are grouped together. After passing through the encoder and serial modulator, it outputs, named Channel 2. In this group, CTL2 and CTL3 are connected to empty signals.

By this point, the division of data by TMDS is very clear. The focus of the design is on the encoder and the design of the serialized data transmitter.

14.5 Principle of TMDS Minimal Transmission Coding

What is minimized transmission, and why use minimized transmission? Like everyone else, the author had this question pop into their mind the first time they heard this concept.

The so-called minimized transmission essentially aims to obtain a new data set with the least number of state transitions by transforming the input data. What does it mean to have the least number of transitions? Let me give you an example. For the 8-bit data 0x55, its binary value is 01010101b. In this data, if two adjacent bits have different values, it is considered one transition. Starting from the least significant bit, bit[0] = 1, bit[1] = 0, since 0 and 1 are different states, that counts as one transition. Next, we look at bit[1] = 0, bit[2] = 1; again, 1 and 0 are different, which counts as another transition. For 0x55, since every pair of adjacent bits has different values, the number of transitions for this data is 7. On the other hand, for the 8-bit value 0x80, the binary value is 10000000b, which has only 1 transition. Thus, different data sets have different transition counts. Therefore, minimizing transmission is about finding ways to re-encode the data so that all encoded data have as few transitions as possible.

Why use minimal transmission? This involves the issue of signal transmission in specific cables. For the TMDS specification, the final data is output serially, bit by bit, for each 10-bit data. During the transmission process, if we assume the previous data is 0 and the next data is 1, then after transmitting the previous bit of data, the signal line needs to immediately set the level to represent a 1. This process will inevitably result in changes in the high and low levels of the signal line, and the changes in the signal will generate a magnetic field. This causes electromagnetic interference on the signal line. Therefore, the fewer signal transitions there are, the less electromagnetic interference will be generated.

14.6 Principle of Minimizing Transmission Implementation

TMDS encoding works by minimizing transitions to reduce signal noise and by maintaining approximate DC balance. For **pixel data**, TMDS characters are designed to have **five or fewer transitions**, while **control characters** (transmitted during blanking periods) contain **seven or more transitions**. These high-transition control characters allow the decoder to detect character boundaries in the serial stream, even though the characters themselves may not be individually unique.

The **TMDS data channel** continuously transmits **10-bit characters**. During idle periods,

four special patterns are sent, corresponding to the four possible combinations of the two control signals. When valid pixel data is being transmitted, the encoder applies a **two-stage encoding process**:

1. The first stage converts 8-bit input data into a **9-bit codeword** with the fewest possible transitions.
2. The second stage adds a **10th bit** to indicate whether the codeword has been inverted for DC balance purposes.

To maintain DC balance, the encoder may **invert the 9-bit codeword** depending on the running disparity between transmitted 1s and 0s. The **10th bit** signals whether this inversion occurred. The receiver uses this bit to determine if the codeword needs to be inverted back to its original form, allowing for easy decoding.

The decision to invert is based on the cumulative imbalance of 1s and 0s transmitted so far, and on the current codeword's bit composition. If too many 1s have been transmitted and the current input contains more 1s than 0s, the encoder will invert the codeword to maintain balance. This dynamic encoding approach ensures signal integrity and is more intuitive when studied alongside the encoder's flowchart.

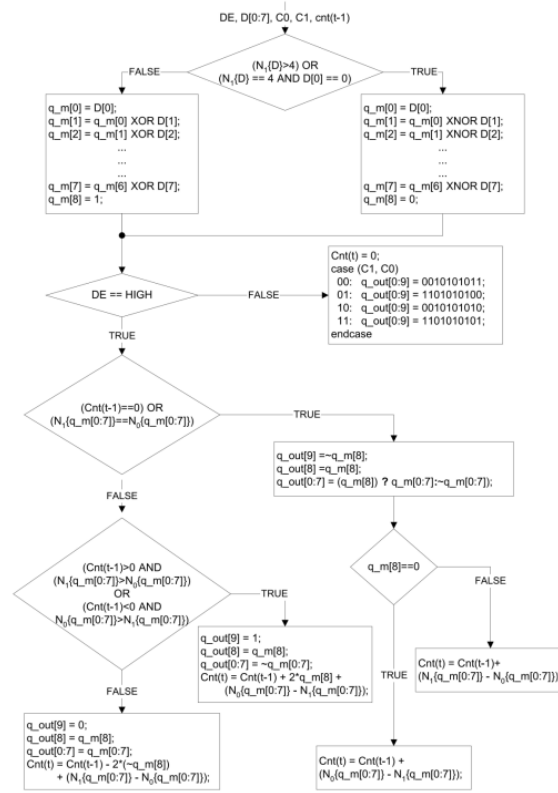


Figure 130: Minimize transmission encoding

Signal Name	Function Description
D, C0, C1, DE	Encoder input signals. D is 8-bit pixel data. C0 and C1 are control signals. DE is the data enable signal.
cnt	A register used to track the polarity of the data stream. A value greater than 0 indicates more 1s were transmitted; a value less than 0 indicates more 0s were transmitted. $\text{cnt}\{t-1\}$ is the polarity of the previous data; $\text{cnt}\{t\}$ is the polarity of the current input data.
q_m	The result of minimized transition encoding, 9 bits, derived from the 8-bit input after applying the transition minimization algorithm.
q_out	The final 10-bit TMDS-encoded result. The 9-bit minimized transition data is appended with one bit for DC balancing.
$N1\{x\}$	A function that counts the number of 1s in the input parameter x .
$N0\{x\}$	A function that counts the number of 0s in the input parameter x .

Table 20: *TMDS Encoder Signal Definitions***Data encoding algorithm process**

1. First, count how many 1s are in the input data.
2. Determine the encoding direction based on the number of 1s in the input data and the value of the least significant bit.
3. According to the encoding direction, the input data is encoded. If the direction is 1, the XOR encoding method is used; if the direction is 0, the XNOR encoding method is used. The encoding direction value is treated as the 9th bit of the encoded data, written as $q_m[8]$. That is, in the 8-bit to 9-bit conversion: $q_m[7:0]$ represents the encoded data, and $q_m[8]$ indicates the direction. When $q_m[8] = 0$, the XNOR operation is used; when $q_m[8] = 1$, the XOR operation is used. This procedure constitutes the first stage of TMDS encoding — **transition minimization** — which converts 8-bit input data into a 9-bit output.

Direct Current Balanced Coding

In differential signaling, the encoding process primarily relies on the difference between the number of 1s and 0s in the entire data stream as determined by the previous encoding process, to guide the differential encoding in the current process. This ensures that the total number of transmitted 1s and 0s during the transmission of the entire data stream does not vary significantly. This helps to ensure the DC balance of the entire transmission link.

The so-called DC balance refers to the situation where the number of 0s and 1s in the trans-

mitted signal is the same, which means that there will be no direct DC current transfer between the sender and receiver. In communication systems, DC balance can effectively avoid issues caused by unstable voltages at both ends of the transmission.

Control data encoding

During TMDS encoding, the entire transmission content is divided into pixel data and control data. When the DE signal is high, the data being encoded and transmitted is pixel data; when DE is low, the data being encoded and transmitted is control data. Whether pixel data or control data is being encoded during any given input clock cycle depends on the state of the data enable signal DE. When DE is high, 8-bit pixel data is encoded into a 10-bit TMDS sequence that minimizes transitions and maintains DC balance. In control cycles, when DE is low, the DVI transmitter encodes 2 bits of control data (C0, C1) into a 10-bit sequence. The diagram below shows the relationship between the DVI cycle and the DE signal.

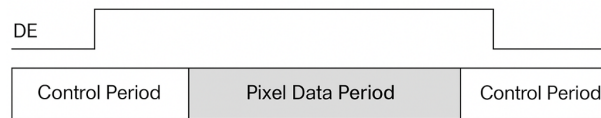


Figure 131: *Switching control signals for pixel data cycle and control cycle*

In the above process, the encoding method of pixel data cycles has been discussed, while in the control cycle, the encoding is much simpler. It only requires outputting different fixed data based on the states of input CTL0 and CTL1, which can be implemented using a lookup table without the need for encoding algorithms. Table shows the signal content corresponding to the control signals for each of the three channels in TMDS.

TMDS Channel	C0	C1
0	HSYNC	VSYNC
1	CTL0 (0)	CTL1 (0)
2	CTL2 (0)	CTL3 (0)

Table 21: *Contents of Control Signals for Each TMDS Channel*

Table shows the encoded output values corresponding to the different states of the 2-bit control signals for each channel.

14.7 TMDS encoding implementation

In fact, as long as the corresponding logic circuit is designed in the order shown in the above flowchart, this encoder can be realized. The following code is the encoder we developed based on this flowchart, which has been tested and can function properly in image data transmission.

C1	C0	Encoding Value
0	0	10'b1101010100
0	1	10'b0010101011
1	0	10'b0101010100
1	1	10'b1010101011

Table 22: *Encoding Values Corresponding to 2-Bit Control Signal States (Table 26-3-2)*

```

module encode(
    clk,
    rst_n,
    din,
    c0,
    c1,
    de,
    dout
);
    input          clk;          // Pixel clock input
    input          rst_n;        // Asynchronous active-high reset
    input [7:0]    din;          // 8-bit input data, to be registered
    input          c0;           // Control signal c0
    input          c1;           // Control signal c1
    input          de;           // Data enable
    output reg [9:0] dout;       // 10-bit TMDS encoded output
    parameter CTL0 = 10'b1101010100;
    parameter CTL1 = 10'b0010101011;
    parameter CTL2 = 10'b0101010100;
    parameter CTL3 = 10'b1010101011;
    reg [3:0] n1d;               // Counts the number of 1s in the input 8-bit data
    reg [7:0] din_q;             // Synchronously registered 8-bit input data (needed for
                                // counting)
    // Count the number of 1s and 0s in each 8-bit input data (pipelined output)
    always @(posedge clk) begin
        din_q <= `D din;
        n1d <= `D din[0] + din[1] + din[2] + din[3] + din[4] + din[5] + din[6] + din[7];
    end
    // Step 1: 8-bit -> 9-bit transition-minimized encoding
    // Reference: DVI Spec 1.0, Page 29, Figure 3-5
    wire decision1;
    assign decision1 = (n1d > 4'h4) || ((n1d == 4'h4) && (din_q[0] == 1'b0));
    // q_m[0] = din_q[0];
    // q_m[i+1] = q_m[i] ^ din_q[i+1]; => q_m[8] = 1

```

```
// q_m[i+1] = q_m[i] ^^ din_q[i+1]; => q_m[8] = 0
wire [8:0] q_m;
assign q_m[0] = din_q[0];
assign q_m[1] = (decision1) ? ~(q_m[0] ^ din_q[1]) : (q_m[0] ^ din_q[1]);
assign q_m[2] = (decision1) ? ~(q_m[1] ^ din_q[2]) : (q_m[1] ^ din_q[2]);
assign q_m[3] = (decision1) ? ~(q_m[2] ^ din_q[3]) : (q_m[2] ^ din_q[3]);
assign q_m[4] = (decision1) ? ~(q_m[3] ^ din_q[4]) : (q_m[3] ^ din_q[4]);
assign q_m[5] = (decision1) ? ~(q_m[4] ^ din_q[5]) : (q_m[4] ^ din_q[5]);
assign q_m[6] = (decision1) ? ~(q_m[5] ^ din_q[6]) : (q_m[5] ^ din_q[6]);
assign q_m[7] = (decision1) ? ~(q_m[6] ^ din_q[7]) : (q_m[6] ^ din_q[7]);
assign q_m[8] = (decision1) ? 1'b0 : 1'b1;
// Step 2: 9-bit -> 10-bit encoding with DC balancing
// Reference: DVI Spec 1.0, Page 29, Figure 3-5
reg [3:0] n1q_m, n0q_m; // Count 1s and 0s in q_m
always @(posedge clk) begin
    n1q_m <= `D q_m[0] + q_m[1] + q_m[2] + q_m[3] + q_m[4] + q_m[5] + q_m[6] + q_m[7];
    n0q_m <= `D 4'h8 - (q_m[0] + q_m[1] + q_m[2] + q_m[3] + q_m[4] + q_m[5] + q_m[6] +
        q_m[7]);
end

reg [4:0] cnt; // Running disparity counter: tracks excess 1s or 0s; MSB (cnt[4]) is
    sign
wire decision2, decision3;
assign decision2 = (cnt == 5'h0) || (n1q_m == n0q_m);
assign decision3 = (~cnt[4] && (n1q_m > n0q_m)) || (cnt[4] && (n0q_m > n1q_m));

// Pipeline alignment (2-stage register sync)
reg [1:0] de_reg;
reg [1:0] c0_reg;
reg [1:0] c1_reg;
reg [8:0] q_m_reg;
always @(posedge clk) begin
    de_reg <= `D {de_reg[0], de};
    c0_reg <= `D {c0_reg[0], c0};
    c1_reg <= `D {c1_reg[0], c1};
    q_m_reg <= `D q_m;
end

// 10-bit TMDS data output logic
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        dout <= 10'h0;
        cnt <= 5'd0;
    end
end
```

```
end else begin
  if (de_reg[1]) begin // Data active period: encode pixel data
    if (decision2) begin
      dout[9] <= `D ~q_m_reg[8];
      dout[8] <= `D q_m_reg[8];
      dout[7:0] <= `D (q_m_reg[8]) ? q_m_reg[7:0] : ~q_m_reg[7:0];
      cnt <= `D (~q_m_reg[8]) ? (cnt + n0q_m - n1q_m) : (cnt + n1q_m - n0q_m);
    end else if (decision3) begin
      dout[9] <= `D 1'b1;
      dout[8] <= `D q_m_reg[8];
      dout[7:0] <= `D ~q_m_reg;
      cnt <= `D cnt + {q_m_reg[8], 1'b0} + (n0q_m - n1q_m);
    end else begin
      dout[9] <= `D 1'b0;
      dout[8] <= `D q_m_reg[8];
      dout[7:0] <= `D q_m_reg[7:0];
      cnt <= `D cnt - {~q_m_reg[8], 1'b0} + (n1q_m - n0q_m);
    end
  end else begin // Control period: send control symbols
    cnt <= `D 5'd0;
    case ({c1_reg[1], c0_reg[1]})
      2'b00: dout <= `D CTL0;
      2'b01: dout <= `D CTL1;
      2'b10: dout <= `D CTL2;
      default: dout <= `D CTL3;
    endcase
  end
end
end
endmodule
```

14.8 Serial Transmission Module

14.8.1 Principle of Serial Transmission

After completing the minimized transmission coding, the next step is to send the encoded content in a serial manner.

For the transmitter, using a 5 times encoding clock rate for the encoder outputs the encoded 10-bit data in a bit-by-bit Dual Data Rate (DDR) format. Therefore, the entire TMDS encoding transmission module requires a total of 2 clock paths: one for the encoder, with a clock rate equal to the input interface layer's clock rate, and another for the serial transmitter, with a clock rate

equal to 5 times that of the input interface layer.

As for the common question of why the frequency of the serial clock is 5 times that of the encoder instead of 10 times, those who have experience with UART serial communication know that to send a 10-bit data (8 data bits + 1 start bit + 1 stop bit) through the serial port, it needs to be sent in 10 steps, sending 1 bit each time. So why is it only 5 steps here? This is because, in this case, each bit is sent on both the rising and falling edges of the clock. That is, during one clock cycle, 2 bits of data can be sent, so 10 bits of data can be sent in just 5 clock cycles. In this way, each 10-bit data can be completed in one clock cycle of the encoder by the serial transmitter, as illustrated in the diagram below .

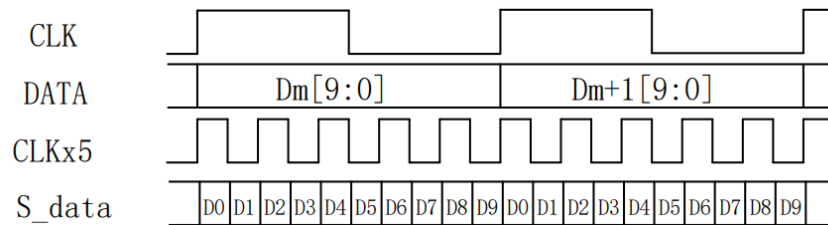


Figure 132: Serial transmission timing diagram

Data is sent on both the rising and falling edges of the clock, which is the DDR (Double Data Rate) interface. The DDR interface can double the amount of data transmitted on a single signal line without changing the transmission clock frequency. As for the DDR SDRAM and SDRAM that everyone is interested in, the biggest structural difference is that DDR SDRAM uses the DDR interface to transmit data, resulting in a data transfer bandwidth for DDR SDRAM that is twice that of SDRAM at the same clock frequency.

14.8.2 FPGA Implementation of DDR Interface

Almost all currently popular FPGAs support double data rate interfaces, and Gowin's FPGAs are no exception. When using them, we can utilize double data rate IO by calling primitives in the Gowin software. Double data rate IO includes IDDR (Input Double Data Rate IO) and ODDR (Output Double Data Rate IO). By calling the double rate data IO primitives, we can achieve double data rate transmission.

This experiment requires the use of ODDR (Dual Data Rate Output) to achieve double data rate output. The ODDR mode is used to transmit double data rate signals from the FPGA device. Q0 is the double-rate data output, and Q1 is used for the OEN signal of the IOBUF/TBUF connected to Q0. The logic block diagram of ODDR and the timing diagram are shown in figures below.

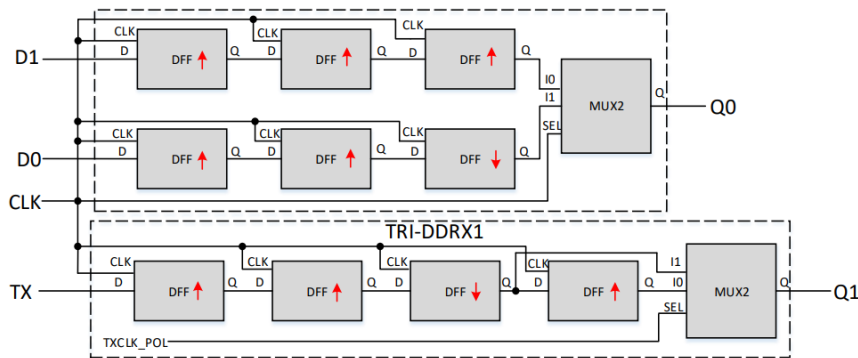


Figure 133: ODDR Logic Block Diagram

Of course, we can use ODDR by calling the IP. In the IP Core Generator interface, search for DDR, select DDR under IO, and enter the IP configuration interface

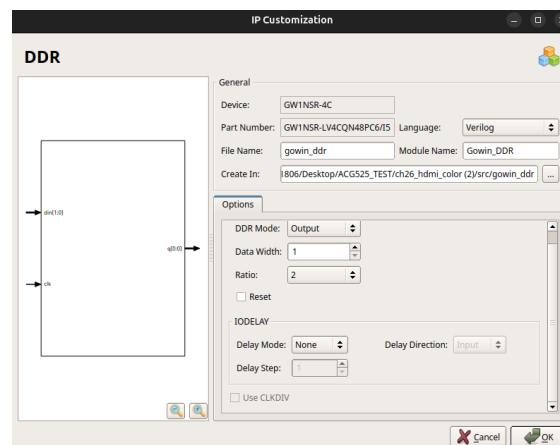


Figure 134: IO_DDR Configuration Interface

Brief Explanation of Configuration Items in the Diagram

1. **DDR Mode:** Configures the DDR mode, including “Input”, “Output”, “Tristate”, and “Bi-directional”.
2. **Data Width:** Sets the data width for DDR, supporting values from 1 to 64.
3. **Ratio:** Sets the data conversion ratio of the DDR module. Supported ratios include 2, 4, 7, 8, 10, 14, 16, and 32.
4. **Reset:** When the Ratio is set to 2, this option can be enabled or disabled. If enabled, it will instantiate an IDDR or ODDR module.
5. **Input IODELAY:**

- **Enable Input IODELAY:** Specifies whether the input delay module is used for DDR input.
- **Delay Mode:** Configures the delay mode. “Dynamic” means using IODELAY with dynamic delay step adjustment. “Static” means using IODELAY with fixed delay steps. “Adaptive” means using IODELAY with adaptive delay control.
- **Delay Step:** Used for static delay configuration, the step range is from 0 to 255.

6. Output IODELAY:

- **Enable Output IODELAY:** Specifies whether the output delay module is used for DDR output.
- **Delay Mode:** Configures the delay mode. “Dynamic” uses IODELAY with dynamic adjustment; “Static” uses fixed steps; “Adaptive” applies self-adjusting delay.
- **Delay Step:** Sets static delay steps in the range 0–255.

7. **Use CLKDIV:** When enabled, instantiates the CLKDIV module to divide and allocate the fclk signal. This should be disabled when the Ratio is set to 2.

14.8.3 TMDs data bits correspond to the DDR interface

In TMDs transmission, data is sent from the FPGA, so only ODDR is needed. The following figure shows a schematic of a 1-bit ODDR.

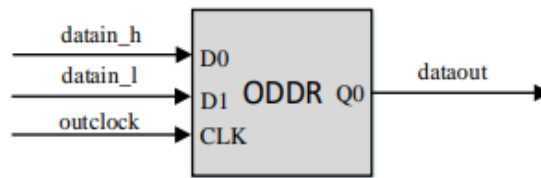


Figure 135: Usage method of the ODDR position

DVI has a total of 4 channels, of which three are data channels. Since each channel needs to output 10 bits of data within 5 clock cycles, and the output is sequential starting from the least significant bit, the data sent to the datain_h and datain_l ports needs to be switched once for each clock cycle. The specific data connected to datain_h and datain_l for each clock cycle is shown in the table below:

	CLK0	CLK1	CLK2	CLK3	CLK4
datain_h	q_out[0]	q_out[2]	q_out[4]	q_out[6]	q_out[8]
datain_l	q_out[1]	q_out[3]	q_out[5]	q_out[7]	q_out[9]

Table 23: Mapping Between Each Clock Cycle and datain_h / datain_l

So when implementing this, you only need to use a counter to count the 5 clock cycles in a loop (the counter resets after counting up to 4, also known as a mod 5 counter), and use the output value of the counter as the selection input for the multiplexer to choose which two bits of data from `q_out` will be connected to `datain_h` and `datain_l`. Based on this, the logic circuit diagram shown in Figure can be drawn:

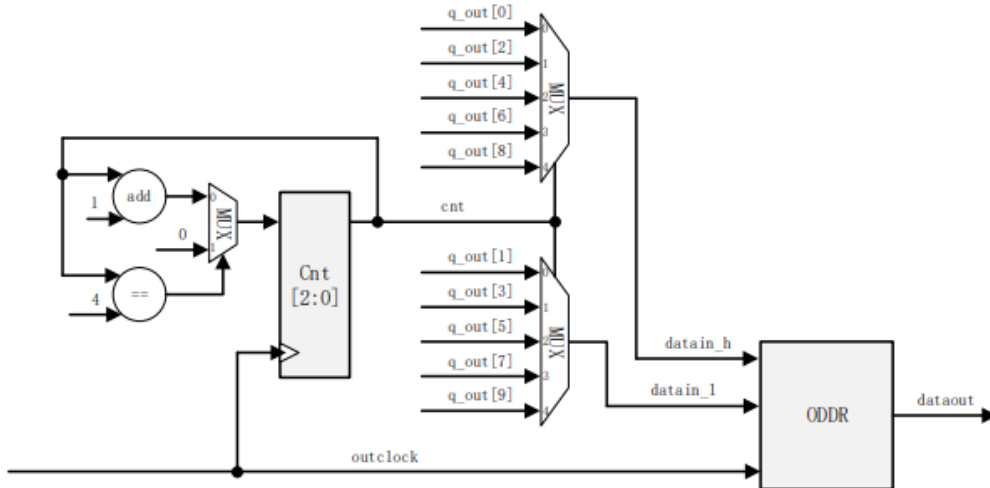


Figure 136: The multiplexer switches the input bits.

In the above diagram, a multiplexer approach is used to select and switch each bit in `q_out`. In fact, a shift register method can also be used to achieve shifting.

14.8.4 Implementation of serial transmission module coding

From the above analysis, it can be understood that the data sent to the `datain_l` port of DDR is the 1st, 3rd, 5th, 7th, and 9th bits of `q_out`, while the data sent to the `datain_h` port of DDR is the 0th, 2nd, 4th, 6th, and 8th bits of `q_out`. If a shift register method is used, it is necessary to first extract the required high and low bits from each channel to form two new 5-bit data sets, and then shift them separately. The reference design we provided uses this shift register approach.

First, split the 10bit data from the three channels that just entered into odd and even paths, and place them into `TMDS_x_l` and `TMDS_x_h` respectively.

```
wire [4:0] TMDS_0_l={datain_0[9],datain_0[7],datain_0[5],datain_0[3],datain_0[1]};
wire [4:0] TMDS_0_h={datain_0[8],datain_0[6],datain_0[4],datain_0[2],datain_0[0]};
wire [4:0] TMDS_1_l={datain_1[9],datain_1[7],datain_1[5],datain_1[3],datain_1[1]};
wire [4:0] TMDS_1_h={datain_1[8],datain_1[6],datain_1[4],datain_1[2],datain_1[0]};
wire [4:0] TMDS_2_l={datain_2[9],datain_2[7],datain_2[5],datain_2[3],datain_2[1]};
wire [4:0] TMDS_2_h={datain_2[8],datain_2[6],datain_2[4],datain_2[2],datain_2[0]};
wire [4:0] TMDS_3_l={datain_3[9],datain_3[7],datain_3[5],datain_3[3],datain_3[1]};
```

```
wire [4:0]TMDS_3_h={datain_3[8],datain_3[6],datain_3[4],datain_3[2],datain_3[0]};
```

Secondly, the data is input to the shift register using a clock rate of 5 times, while a mod 5 counter TMDS_mod5 is used, which updates the data every time it counts to 5.

```
// Modulo-5 counter
always @(posedge clkx5) begin
    if (TMDS_mod5 >= 3'd4)
        TMDS_mod5 <= 3'd0;
    else
        TMDS_mod5 <= TMDS_mod5 + 3'd1;
end

// At 5x clock speed, shift and transmit data
always @(posedge clkx5) begin
    if (TMDS_mod5 == 3'd4) begin
        TMDS_shift_0h <= TMDS_0_h;
        TMDS_shift_0l <= TMDS_0_l;
        TMDS_shift_1h <= TMDS_1_h;
        TMDS_shift_1l <= TMDS_1_l;
        TMDS_shift_2h <= TMDS_2_h;
        TMDS_shift_2l <= TMDS_2_l;
        TMDS_shift_3h <= TMDS_3_h;
        TMDS_shift_3l <= TMDS_3_l;
    end else begin
        TMDS_shift_0h <= TMDS_shift_0h[4:1];
        TMDS_shift_0l <= TMDS_shift_0l[4:1];
        TMDS_shift_1h <= TMDS_shift_1h[4:1];
        TMDS_shift_1l <= TMDS_shift_1l[4:1];
        TMDS_shift_2h <= TMDS_shift_2h[4:1];
        TMDS_shift_2l <= TMDS_shift_2l[4:1];
        TMDS_shift_3h <= TMDS_shift_3h[4:1];
        TMDS_shift_3l <= TMDS_shift_3l[4:1];
    end
end
end
```

Thirdly, After the offset is completed, it is sufficient to send the values of the high and low displacement registers for each channel to the datain_h and datain_l ports of the ODDR. **Choosing output mode and data Width 4 in the IP DDR setting.**

```
// Gowin DDR output instance for positive signals
Gowin_DDR Gowin_DDR_1(
    .din({TMDS_shift_3l[0], TMDS_shift_2l[0], TMDS_shift_1l[0], TMDS_shift_0l[0],
```

```
        TMDS_shift_3h[0], TMDS_shift_2h[0], TMDS_shift_1h[0], TMDS_shift_0h[0]}), //
        input [7:0] din
    .clk(clkx5), // input clock
    .q({dataout_3_p, dataout_2_p, dataout_1_p, dataout_0_p}) // output [3:0] q
);
// Gowin DDR output instance for negative signals (inverted)
Gowin_DDR Gowin_DDR_2(
    .din(~{TMDS_shift_3l[0], TMDS_shift_2l[0], TMDS_shift_1l[0], TMDS_shift_0l[0],
        TMDS_shift_3h[0], TMDS_shift_2h[0], TMDS_shift_1h[0], TMDS_shift_0h[0]}), //
        input [7:0] din
    .clk(clkx5), // input clock
    .q({dataout_3_n, dataout_2_n, dataout_1_n, dataout_0_n}) // output [3:0] q
);
```

The full module is :

```
module serdes_4b_10to1(
    input          clkx5,          // 5x clock input
    input [9:0]    datain_0,       // input data for serialization
    input [9:0]    datain_1,       // input data for serialization
    input [9:0]    datain_2,       // input data for serialization
    input [9:0]    datain_3,       // input data for serialization
    output         dataout_0_p,     // DDR output (positive)
    output         dataout_0_n,     // DDR output (negative)
    output         dataout_1_p,     // DDR output (positive)
    output         dataout_1_n,     // DDR output (negative)
    output         dataout_2_p,     // DDR output (positive)
    output         dataout_2_n,     // DDR output (negative)
    output         dataout_3_p,     // DDR output (positive)
    output         dataout_3_n     // DDR output (negative)
);
    reg [2:0] TMDS_mod5 = 0; // Modulo-5 counter
    // Shift registers for high and low bits of each data channel
    reg [4:0] TMDS_shift_0h = 0, TMDS_shift_0l = 0;
    reg [4:0] TMDS_shift_1h = 0, TMDS_shift_1l = 0;
    reg [4:0] TMDS_shift_2h = 0, TMDS_shift_2l = 0;
    reg [4:0] TMDS_shift_3h = 0, TMDS_shift_3l = 0;
    // Bit mapping for odd/even (high/low) positions of each datain_X
    wire [4:0] TMDS_0_l = {datain_0[9], datain_0[7], datain_0[5], datain_0[3],
        datain_0[1]};
    wire [4:0] TMDS_0_h = {datain_0[8], datain_0[6], datain_0[4], datain_0[2],
        datain_0[0]};
```

```
wire [4:0] TMDS_1_l = {datain_1[9], datain_1[7], datain_1[5], datain_1[3],
    datain_1[1]};
wire [4:0] TMDS_1_h = {datain_1[8], datain_1[6], datain_1[4], datain_1[2],
    datain_1[0]};

wire [4:0] TMDS_2_l = {datain_2[9], datain_2[7], datain_2[5], datain_2[3],
    datain_3[1]};
wire [4:0] TMDS_2_h = {datain_2[8], datain_2[6], datain_2[4], datain_2[2],
    datain_3[0]};

wire [4:0] TMDS_3_l = {datain_3[9], datain_3[7], datain_3[5], datain_3[3],
    datain_3[1]};
wire [4:0] TMDS_3_h = {datain_3[8], datain_3[6], datain_3[4], datain_3[2],
    datain_3[0]};
// Modulo-5 counter
always @(posedge clkx5) begin
    if (TMDS_mod5 >= 3'd4)
        TMDS_mod5 <= 3'd0;
    else
        TMDS_mod5 <= TMDS_mod5 + 3'd1;
end

// At 5x clock speed, shift and transmit data
always @(posedge clkx5) begin
    if (TMDS_mod5 == 3'd4) begin
        TMDS_shift_0h <= TMDS_0_h;
        TMDS_shift_0l <= TMDS_0_l;
        TMDS_shift_1h <= TMDS_1_h;
        TMDS_shift_1l <= TMDS_1_l;
        TMDS_shift_2h <= TMDS_2_h;
        TMDS_shift_2l <= TMDS_2_l;
        TMDS_shift_3h <= TMDS_3_h;
        TMDS_shift_3l <= TMDS_3_l;
    end else begin
        TMDS_shift_0h <= TMDS_shift_0h[4:1];
        TMDS_shift_0l <= TMDS_shift_0l[4:1];
        TMDS_shift_1h <= TMDS_shift_1h[4:1];
        TMDS_shift_1l <= TMDS_shift_1l[4:1];
        TMDS_shift_2h <= TMDS_shift_2h[4:1];
        TMDS_shift_2l <= TMDS_shift_2l[4:1];
        TMDS_shift_3h <= TMDS_shift_3h[4:1];
        TMDS_shift_3l <= TMDS_shift_3l[4:1];
    end
end
```

```
end

// Gowin DDR output instance for positive signals
Gowin_DDR Gowin_DDR_1(
    .din({TMDS_shift_3l[0], TMDS_shift_2l[0], TMDS_shift_1l[0], TMDS_shift_0l[0],
        TMDS_shift_3h[0], TMDS_shift_2h[0], TMDS_shift_1h[0], TMDS_shift_0h[0]}), //
        input [7:0] din
    .clk(clkx5), // input clock
    .q({dataout_3_p, dataout_2_p, dataout_1_p, dataout_0_p}) // output [3:0] q
);
// Gowin DDR output instance for negative signals (inverted)
Gowin_DDR Gowin_DDR_2(
    .din(~{TMDS_shift_3l[0], TMDS_shift_2l[0], TMDS_shift_1l[0], TMDS_shift_0l[0],
        TMDS_shift_3h[0], TMDS_shift_2h[0], TMDS_shift_1h[0], TMDS_shift_0h[0]}), //
        input [7:0] din
    .clk(clkx5), // input clock
    .q({dataout_3_n, dataout_2_n, dataout_1_n, dataout_0_n}) // output [3:0] q
);
endmodule
```

14.9 DVI transmitter implementation

After completing the design of the lower-level encoding and serial transmitter, for the DVI interface, it is only necessary to instantiate the encoder and transmitter and link them to the image data stream's data and control signals according to the DVI specifications. As shown in the figure below:

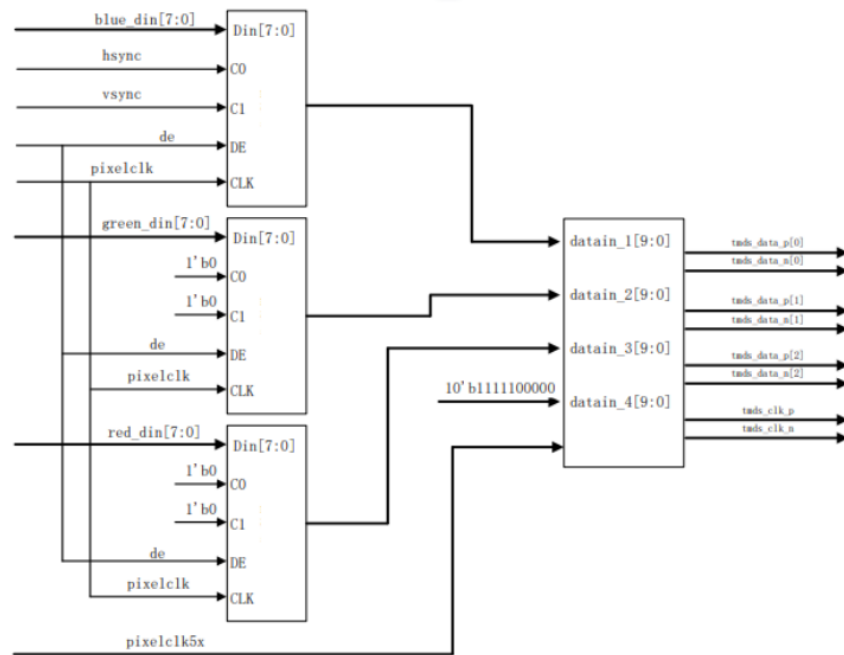


Figure 137: *DVI Sending Schematic*

TMDs requires that the pixel clock be sent along with the data at the time of transmission. Note that the clock frequency here should be consistent with the frequency of pixelclk, not pixelclk5x. This means that tmds_clk is not the bit synchronization clock for the data, but rather a synchronization clock for the complete encoded data. Each falling edge of tmds_clk marks the beginning of the transmission of a new 10-bit data.

To generate tmds_clk, there are two methods. One method is to directly output the pixelclk and its inverted signal as tmds_clk, while the other method is to use ODDR to generate it in data form. Using ODDR to generate in data form can ensure that there is a certain degree of phase relationship between the output tmds_clk and tmds_data, making it easier for the data to be synchronized at the receiving end, so we use ODDR to generate it in data form here. The method to generate this waveform is quite simple: let the first 2.5 clock cycles of the ODDR pin output low level, and the following 2.5 clock cycles output high level. At this point, you only need to assign a constant value of 10'b1111100000 to the datain_3 port of the serial sender. The complete top-level DVI encoder is shown below.

```
module dvi_encoder(
    input    pixelclk,      // system clock
    input    pixelclk5x,    // system clock x5
    input    rst_n,         // reset
    input[7:0] blue_din,    // Blue data in
    input[7:0] green_din,   // Green data in
```

```
input[7:0] red_din,      // Red data in
input      hsync,       // hsync data
input      vsync,       // vsync data
input      de,          // data enable
output     tmds_clk_p,
output     tmds_clk_n,
output[2:0] tmds_data_p, //rgb
output[2:0] tmds_data_n //rgb
);
wire [9:0] red;
wire [9:0] green;
wire [9:0] blue;
encode encb(
    .clk(pixelclk),
    .rst_n(rst_n),
    .din(blue_din),
    .c0(hsync),
    .c1(vsync),
    .de(de),
    .dout(blue)
);
encode encg(
    .clk(pixelclk),
    .rst_n(rst_n),
    .din(green_din),
    .c0(1'b0),
    .c1(1'b0),
    .de(de),
    .dout(green)
);
encode encr(
    .clk(pixelclk),
    .rst_n(rst_n),
    .din(red_din),
    .c0(1'b0),
    .c1(1'b0),
    .de(de),
    .dout(red)
);
serdes_4b_10to1 serdes_4b_10to1_inst(
    .clkx5(pixelclk5x), // 5x clock input
    .datain_0(blue), // input data for serialisation
    .datain_1(green), // input data for serialisation
```

```
.datain_2(red), // input data for serialisation
.datain_3(10'b1111100000), // input data for serialisation
.dataout_0_p(tmds_data_p[0]), // out DDR data
.dataout_0_n(tmds_data_n[0]), // out DDR data
.dataout_1_p(tmds_data_p[1]), // out DDR data
.dataout_1_n(tmds_data_n[1]), // out DDR data
.dataout_2_p(tmds_data_p[2]), // out DDR data
.dataout_2_n(tmds_data_n[2]), // out DDR data
.dataout_3_p(tmds_clk_p), // out DDR data
.dataout_3_n(tmds_clk_n) // out DDR data
);
endmodule
```

14.10 Display Color Bar Experiment Based on DVI Interface

After the design of the controller is completed, during use, it is only necessary to connect the relevant signals of the VGA or TFT controller logic to the corresponding ports of the dvi_encoder, based on the basic VGA or TFT display system. As shown in Figure below.

In this experiment, the VGA controller is configured to output timing signals at a resolution of 800x480. To enable the 'Resolution_800x480' option, it can be set in the 'disp_parameter_cfg.v' file. Additionally, since TMDS encoding requires 2 clock signals, one is synchronized with the VGA controller as the basic logic working clock, and the other is 5 times the VGA controller's clock frequency, used as the transmission clock for TMDS serial encoding. Therefore, compared to directly driving a VGA display which requires a PLL to generate the corresponding frequency clock signal (33MHz for 480p), when using the DVI interface, a PLL is also needed to generate an additional clock signal at 5 times the frequency, which is 165MHz for 480p resolution.

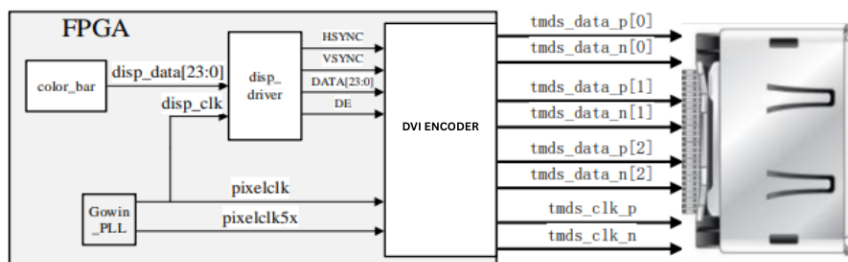


Figure 138: *Diagram of hdmi module*

Setting physical constraint

```
IO_LOC "tmds_data_p[0]" 30,29;  
IO_PORT "tmds_data_p[0]" IO_TYPE=LVC MOS33D PULL_MODE=NONE DRIVE=8 BANK_VCCIO=3.3;  
IO_LOC "tmds_data_p[1]" 32,31;  
IO_PORT "tmds_data_p[1]" IO_TYPE=LVC MOS33D PULL_MODE=NONE DRIVE=8 BANK_VCCIO=3.3;  
IO_LOC "tmds_data_p[2]" 35,34;  
IO_PORT "tmds_data_p[2]" IO_TYPE=LVC MOS33D PULL_MODE=NONE DRIVE=8 BANK_VCCIO=3.3;  
IO_LOC "tmds_clk_p" 28,27;  
IO_PORT "tmds_clk_p" IO_TYPE=LVC MOS33D PULL_MODE=NONE DRIVE=8 BANK_VCCIO=3.3;  
IO_LOC "rst_n" 14;  
IO_PORT "rst_n" IO_TYPE=LVC MOS18 PULL_MODE=UP BANK_VCCIO=1.8;  
IO_LOC "clk" 45;  
IO_PORT "clk" IO_TYPE=LVC MOS33 PULL_MODE=UP BANK_VCCIO=3.3;
```

You can setting the display in `hdmi_color.v` .The result display sucessfully is figure below:

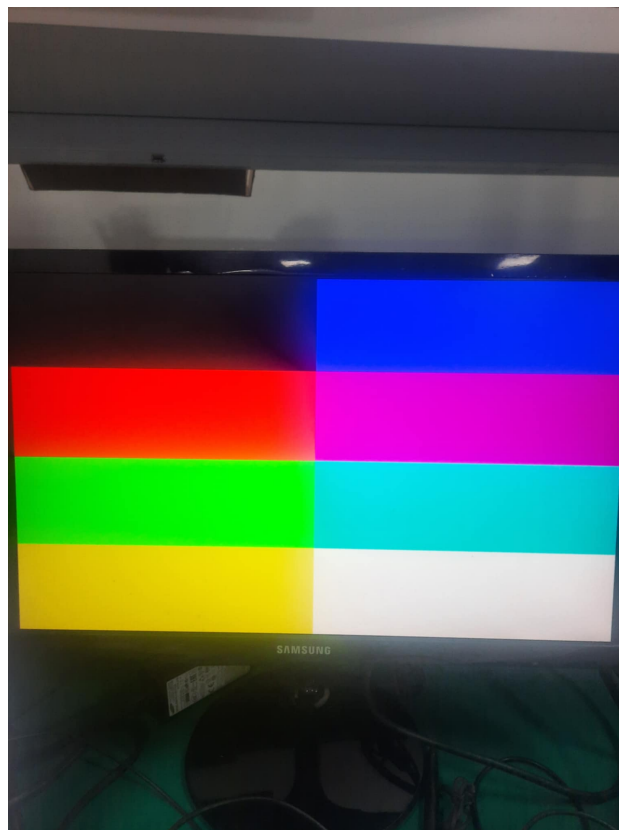


Figure 139: *HDMI result*

15 Camera OV2640 and HDMI Implementation

This chapter will guide you to complete the HDMI display design for the OV2640 image acquisition using these two ports.

The figure below is overall module design:

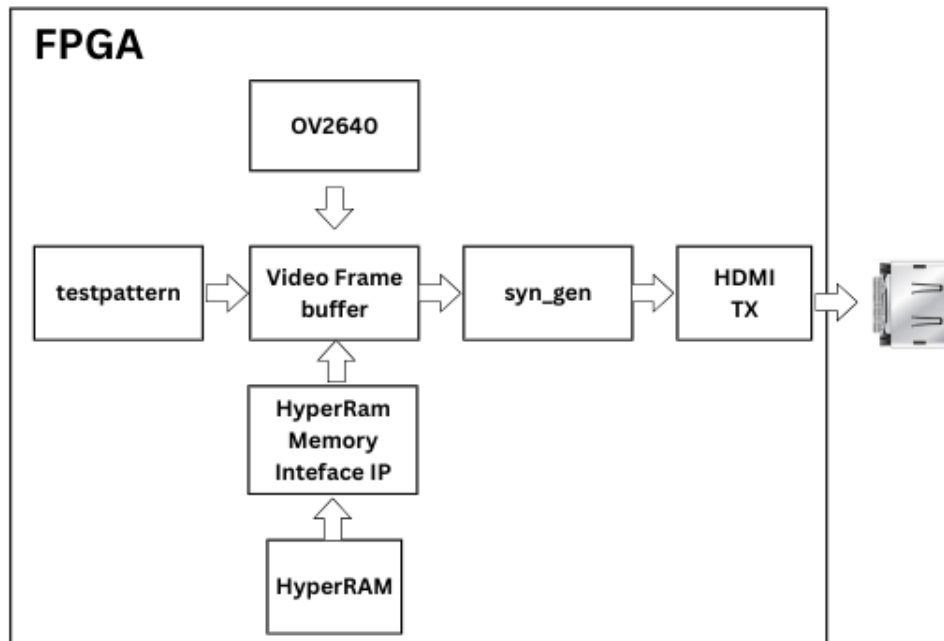


Figure 140: *Camera_hdmi block diagram*

- **OV2640** – CMOS camera module responsible for capturing live image frames and outputting pixel data along with synchronization signals (VSYNC, HREF, PIXCLK). It provides the raw video input to the system.
- **testpattern** – Generates a predefined video pattern (e.g., color bars) for testing and debugging purposes. This module can replace the camera as a video source when needed.
- **Video Frame Buffer** – Serves as a bridge between the input data stream and the HDMI output. It buffers incoming pixel data from either the camera or the testpattern and handles reading data out in sync with video timing.
- **HyperRAM Memory Interface IP** – Manages communication between the FPGA and external HyperRAM. It handles memory addressing, read/write commands, and synchronization for storing full video frames.
- **HyperRAM** – External memory component used to store captured frames temporarily. This enables real-time video buffering even for high-resolution formats like 720p.

- **syn_gen** – Generates synchronization signals for HDMI display, including HSYNC , VSYNC , and DE (data enable). It ensures that video data is read out at the correct timing intervals for proper display.
- **HDMI TX** – HDMI transmitter module that converts RGB video data into TMDS-encoded signals. It outputs high-speed differential signals to drive the HDMI display.
- **Display** – The external HDMI monitor or television that receives and displays the final video output from the system.

15.1 DVI Transmit IP implementation

You can read reference document at https://www.gowinsemi.com/en/support/ip_detail/48/

The DVI TX IP is used to transmit the video timing signals and data into Transition Minimized Differential Signaling(TMDS), and transmit it over the Digital Visual Interface(DVI). In DVI, three channels are designated for red, green, and blue color components(RGB888) in each video pixel data.

Choosing IP customization then find DVI_Tx and click, then the table setting will appear like figure below:

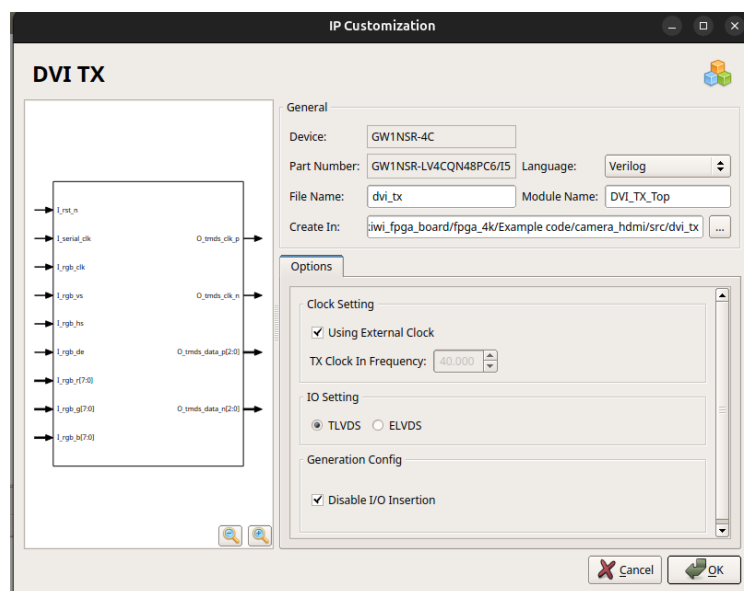


Figure 141: *DVI_TX configuring table*

In this project we will choose using external clock ,TLVDS IO setting

Table 24: DVI TX Module Signal Descriptions

No.	Signal Name	I/O	Description	Remarks
1	I_rst_n	I	Reset signal, active low.	The I/O of all the signals takes DVI TX IP as reference.
2	I_serial_clk	I	This signal is valid when external clock is used. $I_serial_clk = I_rgb_clk \times 5$	
3	I_rgb_clk	I	Video input pixel clock	
4	I_rgb_vs	I	Video input field sync vs signal	
5	I_rgb_hs	I	Video input line sync hs signal	
6	I_rgb_de	I	Video input data enable signal	
7	I_rgb_r	I	Video input data R component	
8	I_rgb_g	I	Video input data G component	
9	I_rgb_b	I	Video input data B component	
10	O_tmids_clk_p	O	Output TMDS differential signal clock positive	
11	O_tmids_clk_n	O	Output TMDS differential signal clock negative	
12	O_tmids_data_p	O	Output TMDS differential signal data positive: - Channel 0 corresponds to blue component - Channel 1 corresponds to green component - Channel 2 corresponds to red component	
13	O_tmids_data_n	O	Output TMDS differential signal data negative: - Channel 0 corresponds to blue component - Channel 1 corresponds to green component - Channel 2 corresponds to red component	

The DVI_TX ip is designed like the previous chapter, so you can read again the previous chapter to understand the IP design of DVI transmitter.

15.2 HyperRAM Memory Interface IP Structure

You can read reference at :https://www.gowinsemi.com/en/support/ip_detail/179/
Gowin HyperRAM Memory Interface IP is a common used HyperRAM interface IP, in compliance with HyperRAM standard protocol. The IP includes the HyperRAM MCL (Memory Controller Logic) and the corresponding PHY (Physical Interface) design. Gowin HyperRAM Memory Interface IP provides you a common command interface to connect with the HyperRAM chip for data

access and storage.

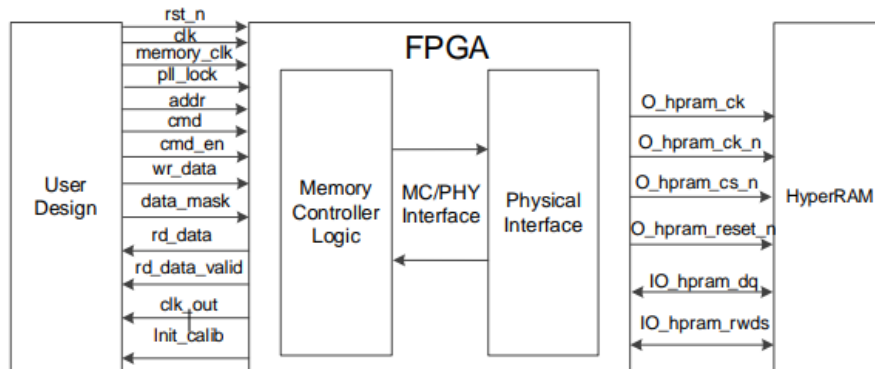


Figure 142: Gowin HyperRAM Memory Interface IP Structure

Memory controller logic

The write, read and other commands that you sent are sorted and reorganized in Memory Controller Logic to combine a data form that complies with the HyperRAM protocol. During write operations, the Memory Controller Logic reorganizes and buffers the data to meet the initial latency requirements between commands and data. During read operations, it samples and reorganizes the retrieved data to restore it to the correct format. The HyperRAM Memory Controller consists of the following main modules, including the CMD unit, the WR_Data unit, the RD_Data unit, etc. The main structure is shown in Figure

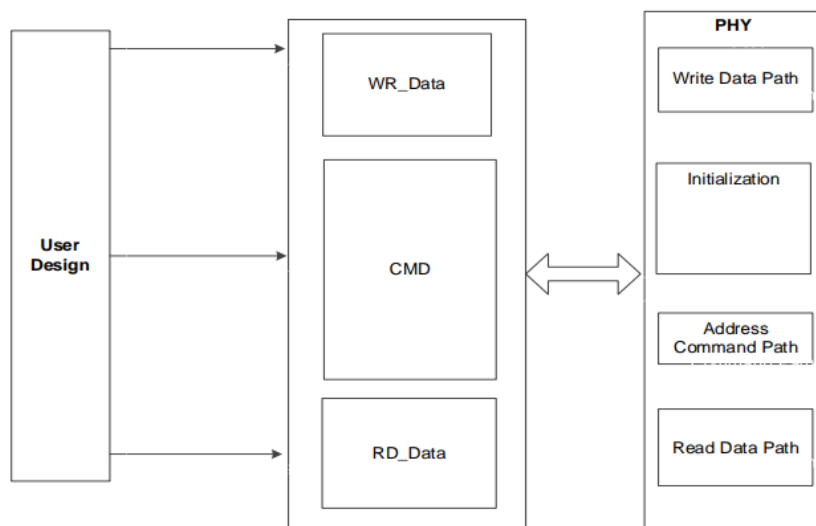


Figure 143: HyperRAM Memory Controller Logic Basic Structure

When choosing IP HyperRam memory you will see the table below, the version of Gowin EDA

can make different UI of setting but you just set the exact value like this.

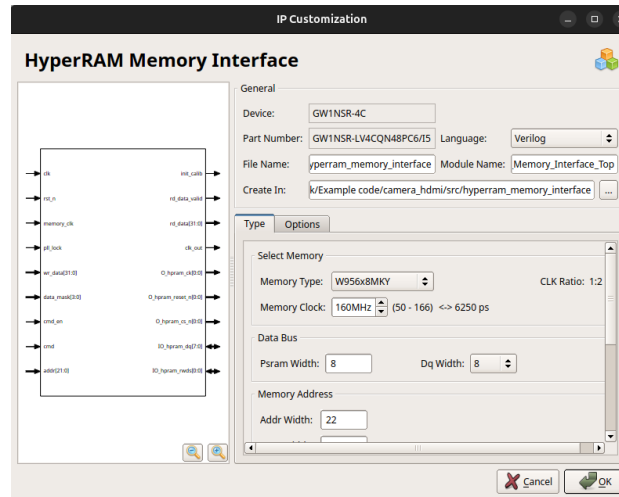


Figure 144: Ram ip setting 1

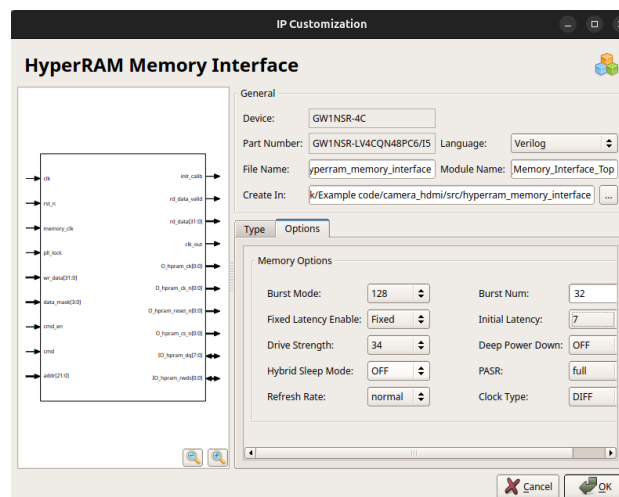


Figure 145: Ram ip setting 2

Remember to choose memory type W95, memory clock 160MHZ. Then go to options choose Burst mode 128, initial latency 7, clock type diff, ..

15.3 Video Frame Buffer IP

Video Frame Buffer is used to receive parallel video input data, then cache it to memory, and output parallel video data at the same time, thus realizing the frame cache.

Gowin Video Frame Buffer IP is located between VESA standard video input/output interface and the memory interface controller IP, so that you can realize the video frame buffer based on

memory.

You can read reference at https://www.gowinsemi.com/en/support/ip_detail/89/

Table 25: *I/O signal Video Frame Buffer IP*

No.	Signal Name	I/O	Description
1	I_rst_n	I	Reset signal, active-low.
2	I_dma_clk	I	Memory Write/Read clock signal.
3	I_wr_halt	I	Write pointer halt signal. Valid in three-frame buffer mode. If not used, assign 0.
4	I_rd_halt	I	Read pointer halt signal. Valid in three-frame buffer mode. If not used, assign 0.
5	I_vin0_clk	I	Input video clock signal.
6	I_vin0_vs_n	I	Input vertical sync, negative polarity.
7	I_vin0_de	I	Input data valid signal.
8	I_vin0_data	I	Input video data signal.
9	O_vin0_fifo_full	O	Output FIFO full signal.
10	I_vout0_clk	I	Output video clock signal.
11	I_vout0_vs_n	I	Output vertical sync, negative polarity.
12	I_vout0_de	I	Output data read enable signal.
13	O_vout0_den	O	Output data valid signal, 2 clock cycles delayed from I_vout0_de.
14	O_vout0_data	O	Output video data signal.
15	O_vout0_fifo_empty	O	Output FIFO empty signal.
16	O_cmd	O	Command channel.
17	O_cmd_en	O	Command and address enable signal.
18	O_addr	O	Address input.
19	O_wr_data	O	Write data channel.
20	O_data_mask	O	Write data mask signal.
21	I_rd_data_valid	I	Read data valid signal.
22	I_rd_data	I	Read data channel.
23	I_init_calib	I	Initialization completed signal.

Then when choosing Ip video_frame_buffer the table will appear just set the memory to HyperRAM memory and click OK

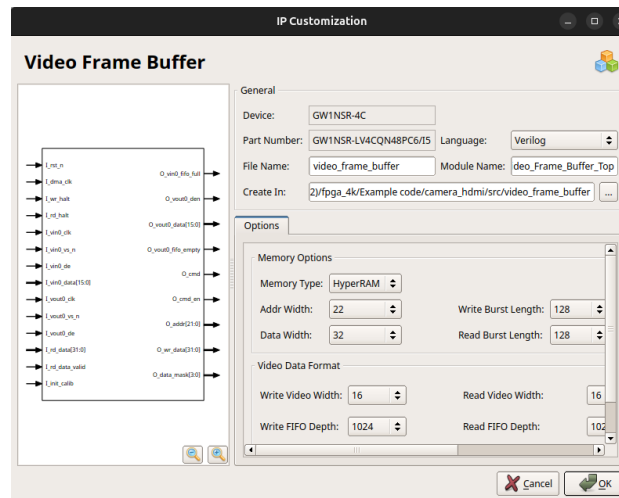


Figure 146: Video frame buffer ip setting

15.4 Phase locked loop(PLL) and clock divide setting

In this project, we need different clock using for TMDS,pixel clock HDMI,Ov2640 camera clock,hyperRam clock.

hyperRam Clock

For hyperRam clock just use ip PLL to create Clock_in is our board 27MHZ , for clock out is 159MHZ,enable lock like figure below

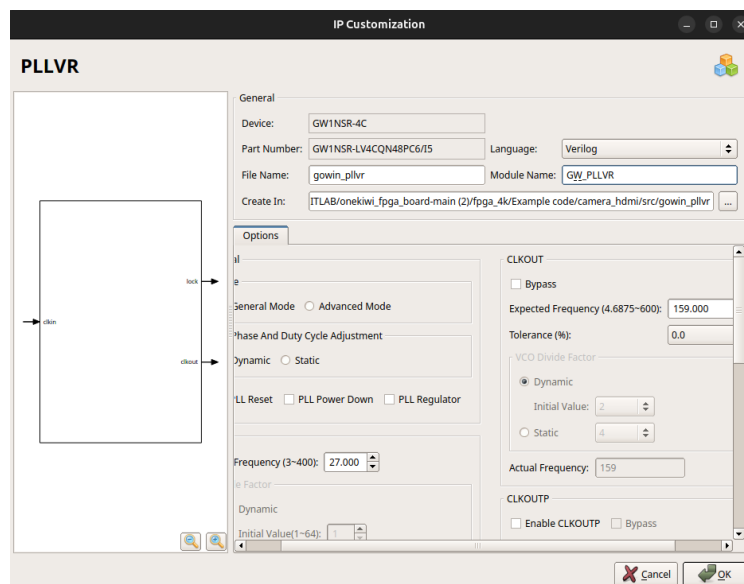


Figure 147: PLL clock for hyperRam memory

TMDS clock and OV2640 clock

For these clock , setting clock in 27MHZ(our FPGA clock),clockout is 371.25MHZ,enable clock-outD 12.375MHZ ,enable lock like figures below :

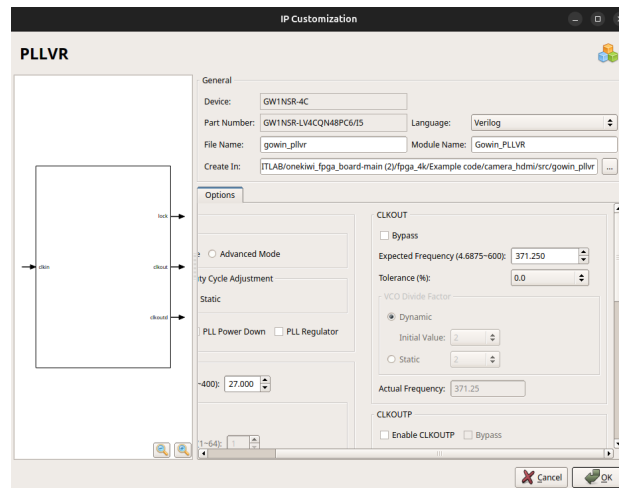


Figure 148: PLL clock for TMDS

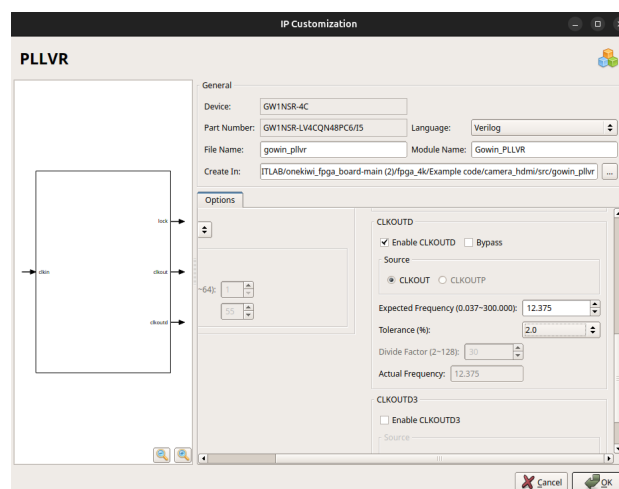


Figure 149: PLL clock for OV2640

15.4.1 Clock_Div for pixel clock HDMI

The pixel clock for HDMI is $371.25/5=74.25$ MHZ , so just use IP clockdiv like figure below:

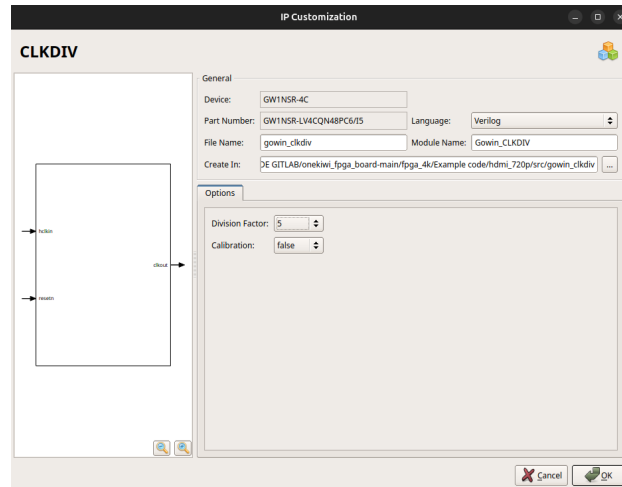


Figure 150: Clock divider for pixel clock

When you finish setting clock, just call it in the top module like:

```
GW_PLLVR GW_PLLVR_inst
(
    .clkout(memory_clk ), //output clkout      159MHZ
    .lock  (mem_pll_lock ), //output lock
    .clkin (I_clk       ) //input clk
);
TMDS_PLLVR TMDS_PLLVR_inst
(.clkin  (I_clk      ) //input clk
,.clkout  (serial_clk) //output clk 371.25Mhz
,.clkoutd (clk_12M   ) //output clkoutd 12.375mhz
,.lock    (pll_lock  ) //output lock
);
CLKDIV u_clkdiv
(.RESETN(hdmi_rst_n)
,.HCLKIN(serial_clk) //clk x5
,.CLKOUT(pix_clk)    //clk x1
,.CALIB (1'b1)
);
```

15.5 Camera OV2640 Module

The OV2640 is a compact and cost-effective camera module widely used in embedded vision systems. It integrates a 2-megapixel CMOS image sensor and an image signal processor (ISP), capable of outputting JPEG, YUV, or RGB formatted images. One of its most attractive features is its SCCB interface (a variant of I²C) used for internal register configuration, making it ideal

```
module OV2640\_Registers (  
input clk,  
input resend,  
input advance,  
output \[15:0] command,  
output finished  
);
```

Table 26: *I/O Description of I2C_Interface.v Module*

Signal	Direction	Purpose
clk	Input	12Hz base clock
siod	Inout	Data line (SDA)
sioc	Output	Clock line (SCL)
taken	Output	Pulse = 1 when one write is initiated (used to advance LUT)
send	Input	Triggers a write operation
rega	Input	Register address to write to
value	Input	Value to write into the register

```
reg [7:0] divider = 8'b00000001;  
reg [31:0] busy_sr = {32{1'b0}};  
reg [31:0] data_sr = {32{1'b1}};  
reg sioc_temp;  
reg taken_temp;  
reg siod_temp;
```

```
localparam id = SID;
```

- divider: Slows down 50MHz to 200kHz (for SCCB timing)
- busy_sr: 1's shift register → tracks how many bits still active
- data_sr: Holds the full 32-bit sequence to shift out
- siod_temp/sioc_temp: internal signals to control the I/O linesid: Device ID (e.g., 0x42 for OV7670)

```
assign siod = siod_temp;  
assign sioc = sioc_temp;  
assign taken = taken_temp;
```

- Simple output assignment from temporary registers.

```
always @ (busy_sr or data_sr[31]) begin
    if(busy_sr[11:10] == 2'b10 || busy_sr[20:19] == 2'b10 || busy_sr[29:28] == 2'b10)
        siod_temp <= 1'bZ; // tri-state during idle
    else
        siod_temp <= data_sr[31]; // drive from MSB
end
```

- SCCB requires that SDA (siod) be tri-stated between phases. This logic enables that.

```
always @ (posedge clk) begin
    taken_temp <= 1'b0;
    if(busy_sr[31] == 0) begin
        sioc_temp <= 1;
        if(send == 1) begin
            if(divider == 8'b00000000) begin
                data_sr <= {3'b100, id, 1'b0, rega, 1'b0, value, 1'b0, 2'b01};
                busy_sr <= {3'b111, 9'b11111111, 9'b11111111, 9'b11111111, 2'b11};
                taken_temp <= 1'b1;
            end else begin
                divider <= divider + 1;
            end
        end
    end
end
```

- If nothing is busy, and send==1, then build a 32-bit SCCB transaction:start, id, rega, value, stop
- taken_temp = 1 signals that a new write started.

```
else begin
    case ({busy_sr[31:29], busy_sr[2:0]})
```

- This checks special bit positions (e.g., start and stop conditions) to precisely control transitions on SDA/SCL (e.g., SDA falling/rising while SCL high).

FSM Description for I2C_Interface

The FSM in the `I2C_Interface` module controls the timing and sequence of a 3-phase SCCB write transmission to an OV2640 camera. The FSM is implicitly implemented using the `busy_sr`

and `data_sr` shift registers.

States Overview

State Name	Trigger Condition	Description
IDLE	<code>busy_sr[31] == 0</code>	The system is idle. Waits for <code>send == 1</code> and <code>divider == 0</code> to load a new 32-bit data word into <code>data_sr</code> and begin the write cycle. <code>taken</code> is asserted to indicate a new transmission.
START	<code>data_sr[31:29] == 3'b100</code>	Initiates the SCCB start condition. <code>SIOD</code> transitions from high to low while <code>SIOC</code> is high. This signals the beginning of a transmission.
WRITE_ID	Bit range [28:21]	Transmits the 8-bit device ID (0x42 for OV2640). <code>SIOC</code> toggles at 200kHz. <code>SIOD</code> shifts bits out MSB first.
WRITE_REG	Bit range [19:12]	Transmits the 8-bit register address to configure in the OV2640.
WRITE_VALUE	Bit range [10:3]	Transmits the 8-bit data value to write into the specified register.
STOP	<code>data_sr[1:0] == 2'b01</code>	Signals the end of transmission. <code>SIOD</code> transitions from low to high while <code>SIOC</code> is high.
COMPLETE	<code>busy_sr == 0</code>	The transaction is complete. The FSM returns to IDLE and waits for the next <code>send</code> pulse.

Table 27: *FSM States of the SCCB Write Protocol*

Clocking and Timing

- The FSM operates on a 12MHz input clock.
- A divider register is used to generate 200kHz timing for SCCB.
- Each bit of data is transmitted over a 4-phase cycle using `divider[7:6]`:
 - 2'b00: `SIOC = 0`
 - 2'b01: `SIOC = 1` (rising edge)
 - 2'b10: `SIOC = 1` (hold)
 - 2'b11: `SIOC = 0` (falling edge)

Transmission Format

`data_sr = {3'b100, ID, 1'b0, rega, 1'b0, value, 1'b0, 2'b01}`

- Total 32 bits, sent one bit per SCCB clock cycle.

- Bits include start condition, device address, register address, data, and stop condition.

```
case ({busy_sr[31:29],busy_sr[2:0]})    // Checking for the start and stop condition
    // For START condition
    6'b111_111: begin                    // bit 31th of data_sr is
        transmitted, SIOC must be high
        case (divider[7:6])              // --> SIOD goes from tri-state to
            high
            2'b00: sioc_temp <= 1;
            2'b01: sioc_temp <= 1;
            2'b10: sioc_temp <= 1;
            default: sioc_temp <= 1;
        endcase
    end

    6'b111_110: begin                    // bit 30th of data_sr is transmitted
        case (divider[7:6])              // --> SIOD goes from high to low,
            SIOC must be high
            2'b00: sioc_temp <= 1;        // --> complete START condition
            2'b01: sioc_temp <= 1;
            2'b10: sioc_temp <= 1;
            default: sioc_temp <= 1;
        endcase
    end

    6'b111_100: begin                    // bit 29th of data_sr is transmitted
        (don't care bit)
        case (divider[7:6])              // --> SIOD goes from tri-state to
            high, then high to low
            2'b00: sioc_temp <= 0;        // after SIOC goes from high to low
            2'b01: sioc_temp <= 0;        // --> Ready for first transmission
            from Master to Slave
            2'b10: sioc_temp <= 0;
            default: sioc_temp <= 0;
        endcase
    end

    6'b100_000: begin                    // bit 1st of data_sr is transmitted
        case (divider[7:6])              // SIOD is low
            2'b00: sioc_temp <= 1;        // SIOC must be high
            2'b01: sioc_temp <= 1;
            2'b10: sioc_temp <= 1;
```

```
        default: sioc_temp <= 1;
    endcase
end

6'b000_000: begin                                // bit 0th of data_sr is transmitted
    case (divider[7:6])                          // SIOD is high
        2'b00: sioc_temp <= 1;                  // --> SIOD goes from low to high
            while SIOC is high
        2'b01: sioc_temp <= 1;                  // --> complete STOP condition for
            SCCB protocol
        2'b10: sioc_temp <= 1;
        default: sioc_temp <= 1;
    endcase
end

// Between START and STOP condition
// SIOC is high on 2 cycles of 400Khz and low on 2 cycle of 400Khz
// --> SIOC is 200Khz
default: begin
    case (divider[7:6])
        2'b00: sioc_temp <= 0;
        2'b01: sioc_temp <= 1;
        2'b10: sioc_temp <= 1;
        default: sioc_temp <= 0;
    endcase
end
endcase

// Create a frequency for SCCB with is 200KHz
if(divider == 8'b11111111) begin
    busy_sr <= {busy_sr[30:0], 1'b0};           // Update number of bit that get
        transmitted
    data_sr <= {data_sr[30:0], 1'b1};           // Update new bit 31th by bit 30th
    divider <= {8{1'b0}};                      // Reset counter for clock divider
end
else begin
    divider <= divider + 1;
end

end
```

15.5.2 Module OV2640_Register

This module outputs a sequence of 16-bit configuration commands to initialize the OV2640 camera.

```
module OV2640_Registers (  
    input clk,  
    input resend,  
    input advance,  
    output [15:0] command,  
    output finished  
);
```

Table 28: *I/O Description of OV2640_Registers Module*

Signal	Direction	Description
clk	Input	Clock signal for sequencing register access. Typically 50MHz.
resend	Input	Resets the address pointer to restart the configuration sequence from the beginning.
advance	Input	Advances to the next register-value pair in the LUT table.
command	Output	16-bit output containing the current register-value pair {register[7:0], value[7:0]}.
finished	Output	Flag set to 1 when all configuration commands have been output (i.e., current command is 0xFFFF).

Internal signal

```
reg [15:0] sreg;  
reg finished_temp;  
reg [8:0] address = {9{1'b0}};
```

- sreg: Holds the current configuration command (register and value).
 - finished_temp: Flag set when configuration ends.
 - address: LUT (lookup table) index counter. 9 bits → can address up to 512 configuration lines.
-

```
always @ (sreg) begin  
    if(sreg == 16'hFFFF)  
        finished_temp <= 1;  
    else
```

```
        finished_temp <= 0;
end
```

- When sreg is set to 0xFFFF, the module signals it has reached the end of the configuration sequence.

```
always @ (posedge clk) begin
    if(resend == 1)
        address <= {8{1'b0}};
    else if(advance == 1)
        address <= address+1;
```

- On resend, it resets address to 0 to start from the top.
- On advance, it goes to the next register-value pair.

```
case (address)
    000 : sreg <= 16'hFF_01;
    001 : sreg <= 16'h12_80;
    ...
    230 : sreg <= 16'hE0_00;
    default : sreg <= 16'hFF_FF;
endcase
```

- This is a large list of predefined register-value pairs.
- Each 16-bit value sreg contains:

15:8 : Register address

7:0 : Value to be

15.5.3 Module OV2640_Controller

This top-level controller orchestrates the register configuration of the OV2640 camera

```
module OV2640_Controller (
    input clk,           // 50MHz system clock
    input resend,        // Manual reconfiguration trigger
    output config_finished, // Flag when camera setup is done
    output sioc,         // SCCB clock (SCL)
    inout siod,          // SCCB data (SDA)
```

```
output reset,          // OV2640 reset pin
output pwn             // OV2640 power down pin
);
```

Table 29: *I/O Description of OV2640_Controller Module*

Signal	Direction	Description
clk	Input	50 MHz clock signal used to drive the configuration logic.
resend	Input	Triggers reconfiguration from the beginning of the register table.
config_finished	Output	High when all configuration commands have been sent to the camera.
sioc	Output	SCCB clock line (SCL) used for communication with the OV2640.
siod	Inout	SCCB data line (SDA) used to send commands to the OV2640.
reset	Output	Active-high reset for the OV2640 module. Typically held high.
pwn	Output	Power-down control for the OV2640. Must be low for normal operation.

Internal Signals

- **command** – A 16-bit wire carrying the concatenated register address and data value to be written to the OV2640. This signal comes from the `OV2640_Registers` module.
- **finished** – A wire that goes high when all register-value pairs have been sent. It is generated by the LUT module.
- **taken** – A wire pulsed high by the `I2C_Interface` after completing one write transaction, prompting the next value from the LUT.
- **send** – A register used to control whether the `I2C_Interface` should continue sending configuration data.

Output Assignments

- **config_finished** is assigned directly from **finished**, signaling the user logic when configuration is complete.
- **reset** is assigned to logic high (1), meaning the OV2640 is not being held in reset.
- **pwn** is assigned to logic low (0), keeping the OV2640 powered on.

Configuration Control Logic

```
always @ (finished) begin
    send = ~finished;
end
```

This block sets the **send** signal to 1 whenever the configuration is not finished, and to 0 when it is complete.

Register Lookup Table Instance

```
OV2640_Registers LUT (
    .clk(clk),
    .advance(taken),
    .command(command),
    .finished(finished),
    .resend(resend)
);
```

This instantiates the **OV2640_Registers** module, which serves as a LUT (Lookup Table) of configuration values. When **taken** is asserted, the table advances to the next entry. If **resend** is high, the address pointer resets to zero.

SCCB Communication Interface

```
I2C_Interface #(.SID(8'h60)) I2C (
    .clk(clk),
    .taken(taken),
    .siod(siod),
    .sioc(sioc),
    .send(send),
    .rega(command[15:8]),
    .value(command[7:0])
);
```

This instance handles the physical transmission of SCCB data to the OV2640:

- **SID** = 0x60, the I²C address of the OV2640 in write mode.
- **siod** and **sioc** connect to the camera's SCCB interface.
- The **send** signal enables or disables transmission.

- The `taken` signal goes high for one cycle after a register is written, triggering the next command from the LUT.

15.5.4 Syn_gen Module

This module synchronization signal generator typically used in video display systems

Table 30: *I/O Description of `syn_gen` Module*

Signal	Direction	Description
I_pxl_clk	Input	Pixel clock signal, used as the main timing reference for counters and output sync signals.
I_rst_n	Input	Active-low reset signal to initialize counters and registers.
I_h_total	Input	Total horizontal period (in pixels), including sync, back porch, resolution, and front porch.
I_h_sync	Input	Horizontal sync pulse width (in pixels).
I_h_bporch	Input	Horizontal back porch duration (in pixels).
I_h_res	Input	Active horizontal resolution (visible display width in pixels).
I_v_total	Input	Total vertical period (in lines), including sync, back porch, resolution, and front porch.
I_v_sync	Input	Vertical sync pulse width (in lines).
I_v_bporch	Input	Vertical back porch duration (in lines).
I_v_res	Input	Active vertical resolution (visible display height in lines).
I_rd_hres	Input	Read horizontal resolution window (typically matches or clips horizontal resolution).
I_rd_vres	Input	Read vertical resolution window (typically matches or clips vertical resolution).
I_hs_pol	Input	Horizontal sync polarity (0 = active low, 1 = active high).
I_vs_pol	Input	Vertical sync polarity (0 = active low, 1 = active high).
O_rden	Output	Read enable signal, high during the valid read region.
O_de	Output	Data enable signal, high during the visible display region.
O_hs	Output	Horizontal sync pulse output. Polarity determined by <code>I_hs_pol</code> .
O_vs	Output	Vertical sync pulse output. Polarity determined by <code>I_vs_pol</code> .

Pixel and Line Counters

```
reg [15:0] V_cnt;  
reg [15:0] H_cnt;
```

These two counters track the horizontal and vertical positions of the current pixel within a frame. They are reset at the end of each frame and line respectively.

```
always@(posedge I_px1_clk or negedge I_rst_n)
begin
    if(!I_rst_n)
        V_cnt <= 16'd0;
    else if((V_cnt >= (I_v_total-1)) && (H_cnt >= (I_h_total-1)))
        V_cnt <= 16'd0;
    else if(H_cnt >= (I_h_total-1))
        V_cnt <= V_cnt + 1;
end
```

```
always @(posedge I_px1_clk or negedge I_rst_n)
begin
    if(!I_rst_n)
        H_cnt <= 16'd0;
    else if(H_cnt >= (I_h_total-1))
        H_cnt <= 16'd0;
    else
        H_cnt <= H_cnt + 1;
end
```

These blocks increment horizontal and vertical counters. Horizontal counter resets every line; vertical counter resets every frame.

Signal Windows (DE, HS, VS)

```
assign Pout_de_w = ((H_cnt >= (I_h_sync + I_h_bporch)) &&
                    (H_cnt <= (I_h_sync + I_h_bporch + I_h_res - 1))) &&
                    ((V_cnt >= (I_v_sync + I_v_bporch)) &&
                    (V_cnt <= (I_v_sync + I_v_bporch + I_v_res - 1)));
```

Pout_de_w is high only during the active display window.

```
assign Pout_hs_w = ~(H_cnt < I_h_sync);
assign Pout_vs_w = ~(V_cnt < I_v_sync);
```

HS and VS pulses are active-low by default (negated above). The output polarity is corrected later.

Read Enable Area

```
assign Rden_w = ((H_cnt >= (I_h_sync + I_h_bporch)) &&
                 (H_cnt <= (I_h_sync + I_h_bporch + I_rd_hres - 1))) &&
                 ((V_cnt >= (I_v_sync + I_v_bporch)) &&
```

```
(V_cnt <= (I_v_sync + I_v_bporch + I_rd_vres - 1)));
```

This defines a window from which the controller should read pixel data — typically matches the visible display, but may be smaller.

Output Signal Registration

```
always@(posedge I_px1_clk or negedge I_rst_n)
begin
    if(!I_rst_n)
    begin
        Pout_de_dn <= 1'b0;
        Pout_hs_dn <= 1'b1;
        Pout_vs_dn <= 1'b1;
        Rden_dn    <= 1'b0;
    end
    else
    begin
        Pout_de_dn <= Pout_de_w;
        Pout_hs_dn <= Pout_hs_w;
        Pout_vs_dn <= Pout_vs_w;
        Rden_dn    <= Rden_w;
    end
end
```

Registers the intermediate signals by one clock cycle to avoid glitches and improve timing closure.

Final Outputs (With Polarity) and Assignment

```
always@(posedge I_px1_clk or negedge I_rst_n)
begin
    if(!I_rst_n)
    begin
        O_de    <= 1'b0;
        O_hs    <= 1'b1;
        O_vs    <= 1'b1;
        O_rden  <= 1'b0;
    end
    else
    begin
        O_de    <= Pout_de_dn;
        O_hs    <= I_hs_pol ? ~Pout_hs_dn : Pout_hs_dn;
    end
end
```

```

    O_vs  <= I_vs_pol ? ~Pout_vs_dn : Pout_vs_dn;
    O_rden <= Rden_dn;
end
end

```

Final registered outputs. The horizontal and vertical sync signals are inverted if the corresponding polarity input is set to 1.

15.5.5 Top module

The `video_top` module is the top-level integration for an FPGA video processing system. It supports both a test pattern and OV2640 camera input, stores video in HyperRAM via a frame buffer, and outputs video through HDMI using TMDS encoding.

Table 31: *I/O Description of `video_top` Module*

Signal	Direction	Description
I_clk	Input	Main system clock input (27 MHz).
I_rst_n	Input	Global active-low reset.
O_led[1:0]	Output	Status LEDs: O_led[0] = running heartbeat, O_led[1] = HyperRAM init done.
SDA	Inout	SCCB data line for OV2640 configuration.
SCL	Inout	SCCB clock line for OV2640 configuration.
VSYNC	Input	Vertical sync from camera (used in pixel sync).
HREF	Input	Data enable from camera (horizontal reference).
PIXDATA[9:0]	Input	10-bit raw pixel data from OV2640.
PIXCLK	Input	Pixel clock from OV2640.
XCLK	Output	External clock to drive OV2640 (12 MHz).
O_hpram_ck	Output	HyperRAM clock output.
O_hpram_ck_n	Output	Inverted HyperRAM clock output.
O_hpram_cs_n	Output	HyperRAM chip select (active low).
O_hpram_reset_n	Output	HyperRAM reset (active low).
IO_hpram_dq[7:0]	Inout	HyperRAM data bus (DQ).
IO_hpram_rwds	Inout	HyperRAM read/write data strobe.
O_tmds_clk_p	Output	HDMI TMDS differential clock (positive).
O_tmds_clk_n	Output	HDMI TMDS differential clock (negative).
O_tmds_data_p[2:0]	Output	HDMI TMDS data lanes (positive): {R, G, B}.
O_tmds_data_n[2:0]	Output	HDMI TMDS data lanes (negative): {R, G, B}.
key	Input	User button input to toggle between test pattern and camera input.

1. Clock and Reset Management

```
input I_clk;    // 27MHz system clock
input I_rst_n;  // Active-low reset
```

A global system clock (27 MHz) is distributed to generate other necessary clocks: - ‘clk_12M’ for camera XCLK - ‘memory_clk’ for HyperRAM - ‘serial_clk’ for TMDS serialization - ‘pix_clk’ for video pixel domain

Reset is synchronized using the `Reset_Sync` module.

2. LED Heartbeat Indicator

```
always @(posedge I_clk or negedge sys_resetn)
    if (!sys_resetn)
        run_cnt <= 0;
    else if (run_cnt >= 32'd27_000_000)
        run_cnt <= 0;
    else
        run_cnt <= run_cnt + 1;

assign O_led[0] = (run_cnt < 13_500_000);
assign O_led[1] = ~init_calib;
```

- `O_led[0]` blinks at 1Hz as a heartbeat. - `O_led[1]` indicates HyperRAM initialization status.

3. Test Pattern Generator

```
testpattern testpattern_inst (
    .I_mode ({1'b0, cnt_vs[7:6]}),
    ...
    .O_data_r(tp0_data_r), .O_data_g(tp0_data_g), .O_data_b(tp0_data_b)
);
```

This module generates selectable patterns (bars, colors, etc.) for testing. It outputs RGB data and timing signals (VS, HS, DE).

4. OV2640 Camera Interface and Configuration

```
OV2640_Controller u_OV2640_Controller (
    .clk(clk_12M), .sioc(SCL), .siod(SDA), ...
);
```

- Configures the OV2640 via SCCB protocol. - Uses `clk_12M` as the camera driving clock (XCLK).

Pixel data is captured on PIXCLK using:

```
always @(posedge PIXCLK)
    pixdata_d1 <= PIXDATA;
```

Raw pixel data is formatted into 16-bit values (e.g., RAW10).

5. Key-Driven Source Switching

```
key_flag key_flag_inst (.key(key), .key_flag(key_flag));
```

The user can toggle between test pattern and live camera feed by pressing a push button.

6. Frame Buffer System (HyperRAM)

```
Video_Frame_Buffer_Top Video_Frame_Buffer_Top_inst (
    .I_vin0_clk(...), .O_vout0_data(off0_syn_data), ...
);
```

- Captures RGB565 input stream into HyperRAM. - Allows asynchronous reading via another clock domain ('`pix_clk`'). - Uses AXI-like control signals (`cmd`, `addr`, `wr_data`, etc.)

7. HyperRAM Interface

```
HyperRAM_Memory_Interface_Top HyperRAM_Memory_Interface_Top_inst (...);
```

Connects to physical HyperRAM device. Manages reads/writes and calibration. PLL generates a memory clock ('`memory_clk`') from 27 MHz base.

8. Video Timing Generator

```
syn_gen syn_gen_inst (
    .I_px1_clk(pix_clk), .I_h_res(1280), .I_v_res(720), ...
);
```

Generates standard DE/HS/VS signals for a selected video resolution (e.g., 720p).

9. HDMI Output via TMDS

```
DVI_TX_Top DVI_TX_Top_inst (  
    .I_rgb_clk(pix_clk), .I_rgb_r(rgb_data[23:16]), ...  
);
```

TMDS transmitter module outputs: - 0_tmds_clk_p/n - 0_tmds_data_p/n[2:0]
TMDS_PLLVR and CLKDIV generate the required 5× pixel serial clock.

10. Reset Synchronizer

```
module Reset_Sync (...);
```

Ensures clean synchronization of asynchronous reset signal to internal clock domains.

Pll calculate website:https://juj.github.io/gowin_fpga_code_generators/pll_calculator.html

15.6 Physical constraint setting

```
IO_LOC "0_tmds_clk_p" 28,27;  
IO_PORT "0_tmds_clk_p" PULL_MODE=NONE DRIVE=8;  
IO_LOC "0_tmds_data_p[0]" 30,29;  
IO_PORT "0_tmds_data_p[0]" PULL_MODE=NONE DRIVE=8;  
IO_LOC "0_tmds_data_p[1]" 32,31;  
IO_PORT "0_tmds_data_p[1]" PULL_MODE=NONE DRIVE=8;  
IO_LOC "0_tmds_data_p[2]" 35,34;  
IO_PORT "0_tmds_data_p[2]" PULL_MODE=NONE DRIVE=8;  
IO_LOC "XCLK" 33;  
IO_PORT "XCLK" IO_TYPE=LVC MOS25 PULL_MODE=NONE DRIVE=8;  
IO_LOC "0_led[0]" 10;  
IO_PORT "0_led[0]" IO_TYPE=LVC MOS33 PULL_MODE=NONE DRIVE=8;  
IO_LOC "SCL" 44;  
IO_PORT "SCL" IO_TYPE=LVC MOS33 PULL_MODE=NONE DRIVE=8;  
IO_LOC "SDA" 46;  
IO_PORT "SDA" IO_TYPE=LVC MOS33 PULL_MODE=NONE DRIVE=8;  
IO_LOC "PIXCLK" 41;  
IO_PORT "PIXCLK" IO_TYPE=LVC MOS33 PULL_MODE=UP;  
IO_LOC "PIXDATA[9]" 40;  
IO_PORT "PIXDATA[9]" PULL_MODE=UP;  
IO_LOC "PIXDATA[8]" 39;  
IO_PORT "PIXDATA[8]" PULL_MODE=UP;
```

```
IO_LOC "PIXDATA[7]" 23;
IO_PORT "PIXDATA[7]" PULL_MODE=UP;
IO_LOC "PIXDATA[6]" 16;
IO_PORT "PIXDATA[6]" PULL_MODE=UP;
IO_LOC "PIXDATA[5]" 18;
IO_PORT "PIXDATA[5]" PULL_MODE=UP;
IO_LOC "PIXDATA[4]" 20;
IO_PORT "PIXDATA[4]" PULL_MODE=UP;
IO_LOC "HREF" 42;
IO_PORT "HREF" IO_TYPE=LVCMS33 PULL_MODE=UP;
IO_LOC "VSYNC" 43;
IO_PORT "VSYNC" IO_TYPE=LVCMS33 PULL_MODE=UP;
IO_LOC "I_rst_n" 14;
IO_PORT "I_rst_n" PULL_MODE=UP;
IO_LOC "key" 15;
IO_PORT "key" PULL_MODE=UP;
IO_LOC "I_clk" 45;
IO_PORT "I_clk" IO_TYPE=LVCMS33 PULL_MODE=UP;
IO_LOC "PIXDATA[3]" 19;
IO_PORT "PIXDATA[3]" PULL_MODE=UP;
IO_LOC "PIXDATA[2]" 17;
IO_PORT "PIXDATA[2]" PULL_MODE=UP;
IO_LOC "PIXDATA[1]" 21;
IO_PORT "PIXDATA[1]" PULL_MODE=UP;
IO_LOC "PIXDATA[0]" 22;
IO_PORT "PIXDATA[0]" PULL_MODE=UP;
```

15.6.1 Result



Figure 153: *Camera result*

16 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.