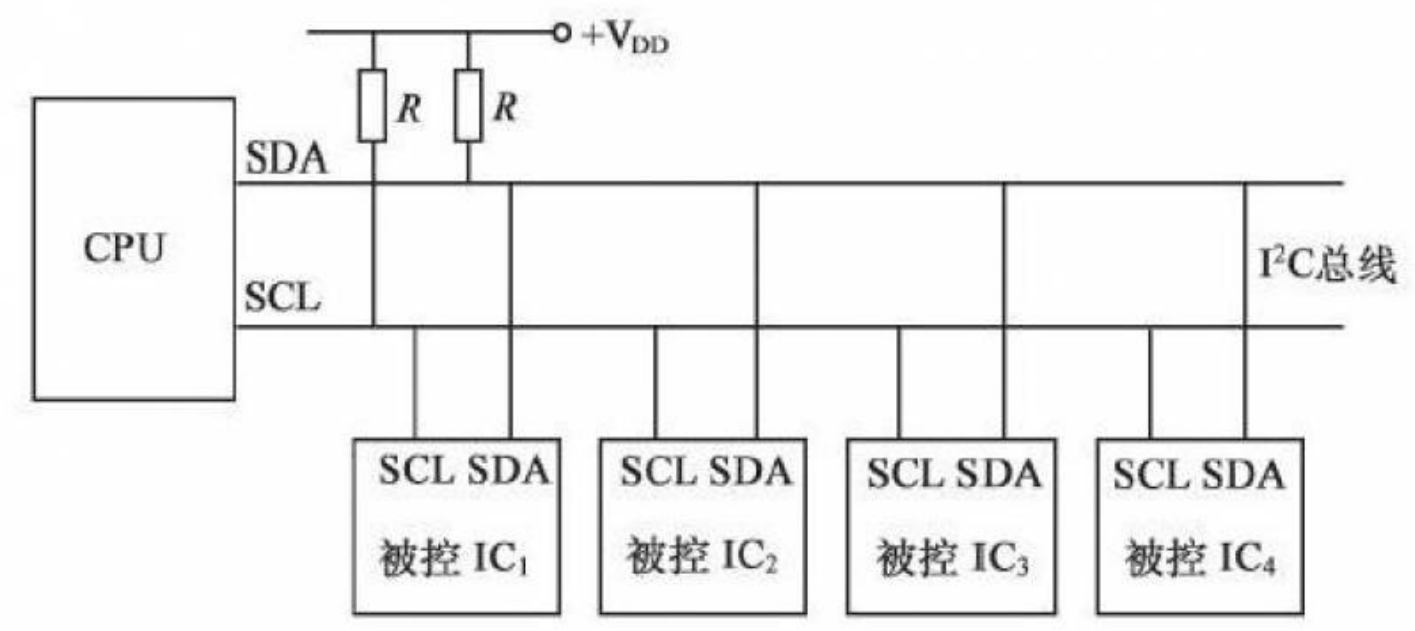
DAC (I2C) function driver of PCF8591 based on STEP FPGA

This section will use the DAC conversion (I2C) function of the PCF8591 on the FPGA driver backplane with everyone.

====Hardware description====

PCF8591 is a chip that integrates 4 ADCs and 1 DAC, and uses I2C bus communication.
The I2C bus is a simple, two-way two-wire synchronous serial bus developed by Philips. It only needs two wires to transfer information between devices connected to the bus. The master device is used to start the bus to transmit data and generate a clock to open the device for transmission. At this time, any addressed device is regarded as a slave device. If the host wants to send data to the slave device, the host first addresses the slave device, then actively sends the data to the slave device, and finally the host terminates the data transfer; if the host wants to receive data from the slave device, the master device first addresses the slave device, Then the host receives the data sent from the device, and finally the host terminates the receiving process. Not too much to explain here, hardware connection as follows: The design of the hardware connection as this design FPGA I2C master, as PCF8591 I2C slave, the slave address by the fixed address and programmable address components, peripherals our The backplane has grounded the programmable addresses A0, A1, and A2, so the 7-bit address is 7'h48, plus the lowest bit of read and write control, so the addressing address when writing data to PCF8591 is 8'h90, and reading data to PCF8591 When the addressing address is 8'h91. The following PCF8591 integrates many functions. When different functions are needed, the PCF8591 should be configured accordingly. The configuration data is stored in a register named CONTROL BYTE. The following figure shows the functions of some bits in the register. For details, please refer to the datasheet of PCF8591 In this design, we only use the DAC function, and the configuration data is 8'h40. The design of the communication process we need to specifically: start - write addressing - read response - write configuration data - read response - [Write DAC data - read response] cycle - end everyone should be on how to drive PCF8591 DAC introduced by the above The sampling has an overall concept, and some details are the timing details of I2C communication, as shown below



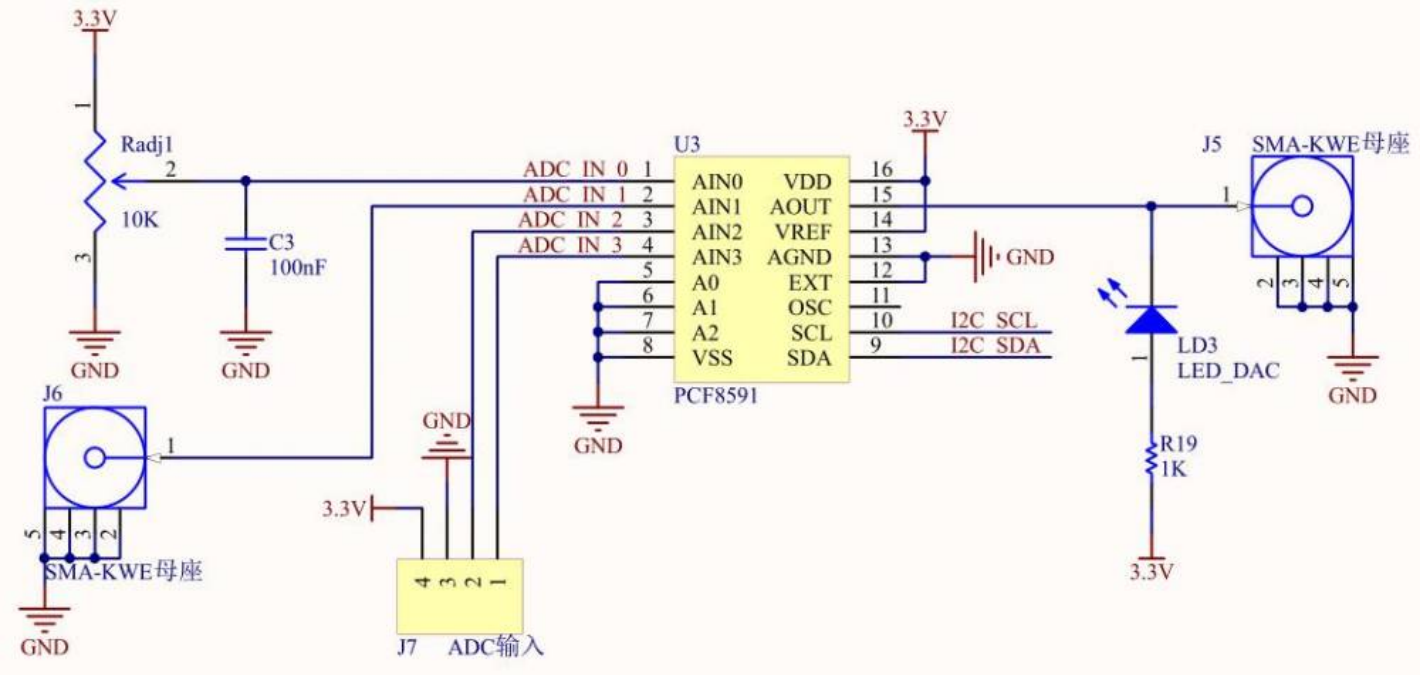


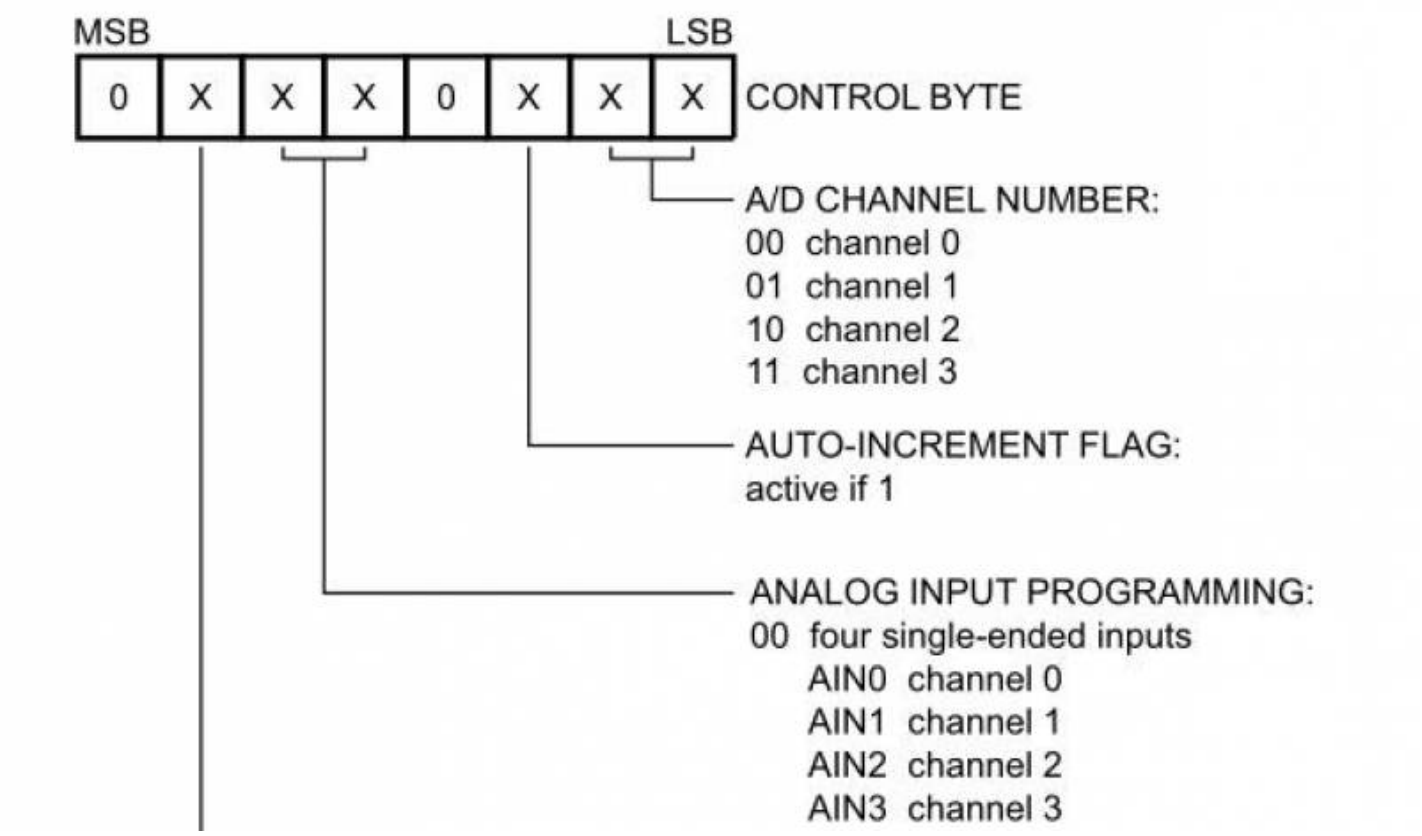
Table 5. I²C slave address byte

Bit	Slave address							
	7 MSB	6	5	4	3	2	1	0 LSB
slave address	1	0	0	1	A2	A1	A0	R/$\overline{W}$

The least significant bit of the slave address byte is bit R/$\overline{W}$ (see Table 6).

Table 6. R/$\overline{W}$-bit description

R/$\overline{W}$	Description
0	write data
1	read data



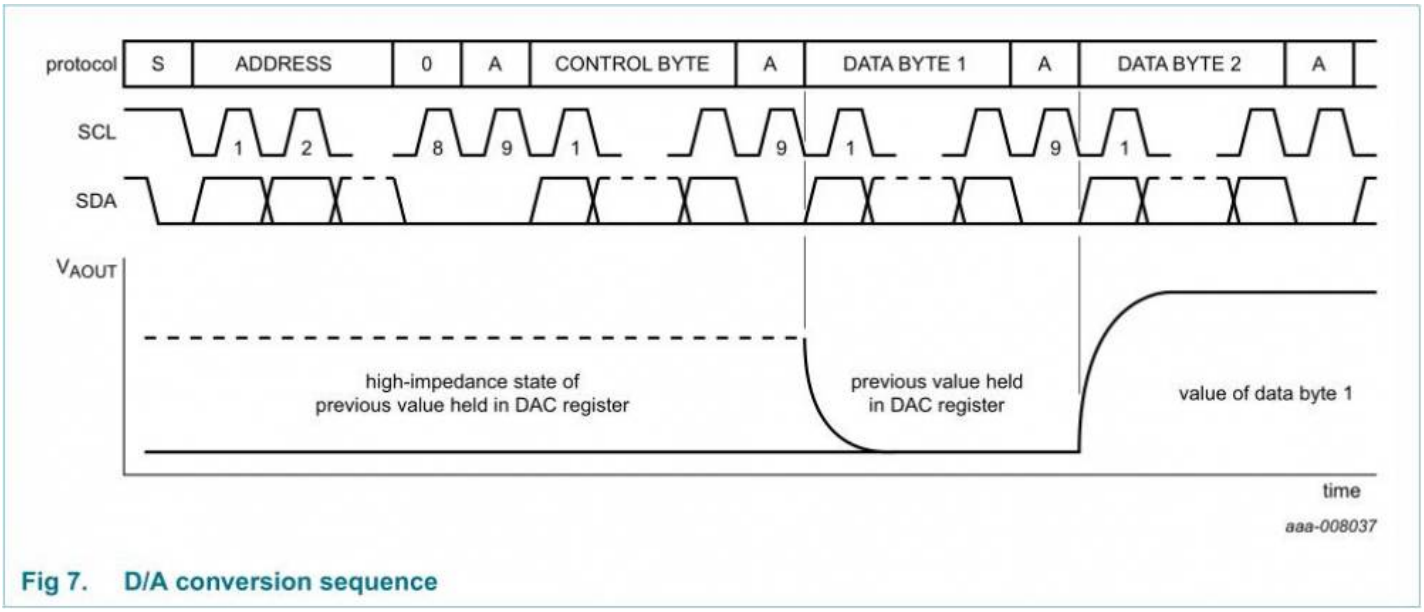


Fig 7. D/A conversion sequence

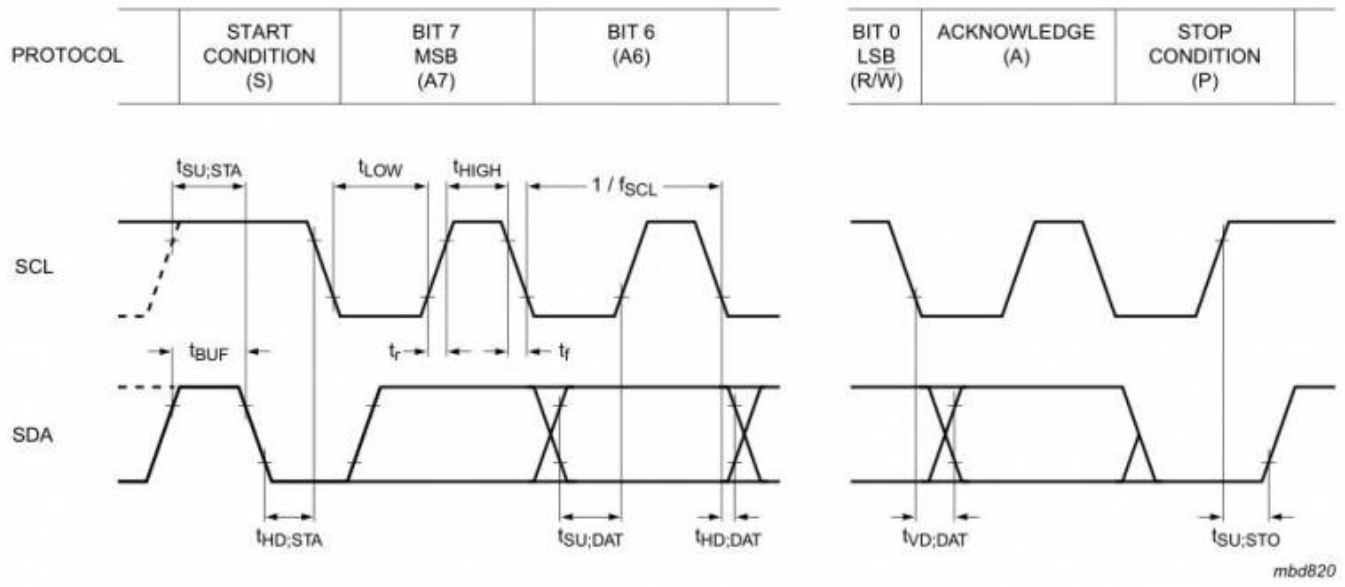


Fig 21. I²C bus timing diagram; rise and fall times refer to V_{IL} and V_{IH}

Symbol	Parameter		Min	Typ	Max	Unit
I²C bus timing (see Figure 21)		[1]				
f_{SCL}	SCL clock frequency		-	-	100	kHz
t_{SP}	pulse width of spikes that must be suppressed by the input filter		-	-	100	ns
t_{BUF}	bus free time between a STOP and START condition		4.7	-	-	µs
$t_{SU;STA}$	set-up time for a repeated START condition		4.7	-	-	µs
$t_{HD;STA}$	hold time (repeated) START condition		4.0	-	-	µs
t_{LOW}	LOW period of the SCL clock		4.7	-	-	µs
t_{HIGH}	HIGH period of the SCL clock		4.0	-	-	µs
t_r	rise time of both SDA and SCL signals		-	-	1.0	µs
t_f	fall time of both SDA and SCL signals		-	-	0.3	µs
$t_{SU;DAT}$	data set-up time		250	-	-	µs
$t_{HD;DAT}$	data hold time		0	-	-	µs
$t_{VD;DAT}$	data valid time		-	-	3.4	µs
$t_{SU;STO}$	set-up time for STOP condition		4.0	-	-	µs

====Verilog code====

```
// ---------------------------------------------- --------------------
// >>>>>>>>>>>>>>>>>>>>>>>> COPYRIGHT NOTICE < <<<<<<<<<<<<<<<<<<<<<<
// ------------------------ ----------------------------------------------
// Module: DAC_I2C
/ /
// Author: Step
//
// Description: DAC_I2C
//
// Web: www.stepfpga.com
//
// --------------------- ----------------------------------------------
// Code Revision History:
// ---------------------------------------- ----------------------
// Version: |Mod. Date: |Changes Made:
// V1.1 |2016/10/30 |Initial ver
// ------------------------------------ ----------------------------
module DAC_I2C
 (
        input                           clk_in ,                //System clock
        input                           rst_n_in ,      //System reset, active low

        output  reg                     dac_done ,      //DAC sampling completion flag
        input           [ 7 : 0 ]       dac_data ,      //DAC sampling data

        output                          scl_out ,       //I2C bus SCL
        inout                           sda_out          //I2C bus SDA
) ;

        parameter       CNT_NUM  =      15 ;

        localparam      IDLE     =      3'd0 ;
        localparam      MAIN     =      3'd1 ;
        localparam      START    =      3'd2 ;
        localparam      WRITE    =      3'd3 ;
        localparam      STOP     =      3'd4 ;

        //According to the PCF8591 datasheet, the I2C frequency is up to 100KHz.
        //We are going to use 4 beats to complete the transmission of 1bit data, so we need a
 400KHz clock to trigger the
        design.//Use counter frequency division to generate 400KHz clock signal clk_400khz
        reg                                     clk_400khz ;
        reg             [ 9 : 0 ]               cnt_400khz ;
        always @ ( posedge clk_in or  negedge rst_n_in )  begin
                if ( ! rst_n_in )  begin
                        cnt_400khz <=  1 0'd0 ;
                        clk_400khz <=  1'b0 ;
                end  else  if ( cnt_400khz>= CNT_NUM - 1 )  begin
                        cnt_400khz <=  1 0'd0 ;
                        clk_400khz <=  ~ clk_400khz ;
                end  else  begin
                        cnt_400khz <= cnt_400khz +  1'b1 ;
```

```
                end
        end

        reg             [ 7 : 0 ]               adc_data_r ;
        reg                                     scl_out_r ;
        reg                                     sda_out_r ;
        reg             [ 2 : 0 ]               cnt ;
        reg             [ 2 : 0 ]               cnt_main ;
        reg             [ 7 : 0 ]               data_wr ;
        reg             [ 2 : 0 ]               cnt_start ;
        reg             [ 2 : 0 ]               cnt_write ;
        reg             [ 2 : 0 ]               cnt_stop ;
        reg             [ 2 : 0 ]               state ;

        always @ ( posedge clk_400khz or  negedge rst_n_in )  begin
                if ( ! rst_n_in )  begin        //If the button is reset, initialize the relev
ant data
                        scl_out_r <=  1'd1 ;
                        sda_out_r <=  1'd1 ;
                        cnt <=  1'b0 ;
                        cnt_main <=  1'b0 ;
                        cnt_start <=  1'b0 ;
                        cnt_write <=  3'd0 ;
                        cnt_stop <=  1'd0 ;
                        dac_done <= 1'b1 ;
                        state <= IDLE ;
                end  else  begin
                        case ( state )
                                IDLE : begin    //Software self-reset, mainly used for process
ing after program runaway
                                                scl_out_r <=  1'd1 ;
                                                sda_out_r <=  1'd1 ;
                                                cnt < =  1'b0 ;
                                                cnt_main <=  1'b0 ;
                                                cnt_start <=  1'b0 ;
                                                cnt_write <=  3'd0 ;
                                                cnt_stop <=  1'd0 ;
                                                dac_done<=  1'b1 ;
                                                state <= MAIN ;
                                        end
                                MAIN : begin
                                                if ( cnt_main >=  3'd3 ) cnt_main <=  3'd3 ;
//Execute control of the sub-state in MAIN cnt_main
                                                else cnt_main <= cnt_main +  1' b1 ;
                                                case ( cnt_main )
                                                        3'd0 :  begin state <= START ;  end
//
                                                         START in I2C communication sequence
3'd1 :  begin data_wr <=  8'h90; state <= WRITE ;  end  //A0, A1, A2 are all connected to GND,
and the write address is 8'h90
                                                        3'd2 :  begin data_wr <=  8'h40 ; stat
e <= WRITE ;  end       //control byte is 8' h40, open the DAC function
                                                        3'd3 :  begin data_wr <= dac_data ; st
```

```
ate <= WRITE ; dac_done <=  1'b0 ;  end         //Data that needs to be converted by DAC
                                                        3'd4 :  begin state <= STOP ;  end
//I2C End of communication sequence STOP
                                                        default : state <= IDLE ;       //If t
he program is out of control, enter the IDLE self-reset state
                                                endcase
                                        end
                                START : begin   //Start START in I2C communication sequence
                                                if ( cnt_start >=  3'd5 ) cnt_start <=  1'b0 ;
/ / Perform control of the sub-states in START cnt_start
                                                else cnt_start <= cnt_start +  1'b1 ;
                                                case ( cnt_start )
                                                        3'd0 :  begin sda_out_r <=  1'b1 ; scl
_out_r <=  1'b1 ;  end  //Pull SCL and SDA high and keep it above
                                                        4.7us 3'd1 :    begin sda_out_r <=
1'b1 ; scl_out_r <=  1'b1 ;  end        //clk_400khz every cycle 2.5us, requires two cycles
                                                        3'd2 :  begin sda_out_r <=  1'b0 ;  en
d       // SDA pulls down to SCL pulls down and stays above
                                                        4.0us 3'd3 :    begin sda_out_r <=
1'b0 ;  end     //clk_400khz every cycle 2.5us, requires two cycles
                                                        3 'd4 :         begin scl_out_r <=
1'b0 ;  end     //SCL is pulled down and stays above
                                                        4.7us 3'd5 :    beginscl_out_r <=  1'b
0 ; state <= MAIN ;  end        //clk_400khz 2.5us per cycle, it takes two cycles to return to
MAIN
                                                        default : state <= IDLE ;       //If t
he program is out of control, enter IDLE self-reset state
                                                endcase
                                        end
                                WRITE : begin   //Write operation WRITE and corresponding judg
ment operation ACK in I2C communication sequence
                                                if ( cnt <=  3'd6 )  begin      //A total of 8
bit data needs to be sent, here the number of cycles is controlled
                                                        if ( cnt_write >=  3'd3 )  begin cnt_w
rite <=  1'b0 ; cnt <= cnt+  1'b1 ;  end
                                                        else  begin cnt_write <= cnt_write +
1'b1 ; cnt <= cnt ;  end
                                                end  else  begin
                                                        if ( cnt_write >=  3'd7 )  begin cnt_w
rite <=  1'b0 ; cnt <=  1'b0 ;  end     // Both variables are restored to their initial values
                                                        else  begin cnt_write <= cnt_write +
1'b1 ; cnt <= cnt ;  end
                                                end
                                                case ( cnt_write )
                                                        //Transfer data according to I2C timin
g
                                                        3'd0 :  begin scl_out_r <=  1'b0 ; sda
_out_r <= data_wr [ 7 - cnt ] ;  end    //SCL is pulled low and controls the SDA output corres
ponding to bit
                                                        3 'd1 :         begin scl_out_r <=
1'b1 ;  end     //SCL is pulled high and stay above
                                                        4.0us 3'd2 :    begin scl_out_r <=
1'b1 ;  end     //clk_400khz every cycle 2.5us, requires two cycles
```

```
                                                    3' d3 :         begin scl_out_r <= 1'b
0 ;  end        //SCL is pulled low, ready to send the next 1bit of data
                                                    //Get the response signal from the sla
ve device and judge
                                                    3'd4 :  begin sda_out_r <=  1'bz ; dac
_done <=  1'b1 ;  end   //release SDA line, ready to receive the response signal from the slav
e device
                                                    3'd5 :  begin scl_out_r <=  1'b1 ;  en
d       //SCL is pulled high and stay above
                                                    4.0us 3'd6 :    begin  if ( sda_out )
 state <= IDLE ;  else state < = state ;  end   //Get the response signal from the slave devic
e and judge
                                                    3'd7 :  begin scl_out_r <=  1'b0 ; sta
te <= MAIN ;  end       //SCL pulls low and returns to MAIN state
                                                    default : state <= IDLE ;       //If t
he program is out of control, enter IDLE self-reset state
                                                endcase
                                        end
                                STOP : begin    //I2C communication The end of the sequence ST
OP
                                                if ( cnt_stop >=  3'd5 ) cnt_stop <=  1'b0 ;
//Execute control of the sub-states in STOP cnt_stop
                                                else cnt_stop <= cnt_stop +  1'b1 ;
                                                case ( cnt_stop )
                                                    3'd0 :  begin sda_out_r <=  1'b0 ;  en
d       //SDA pulls down and prepares to STOP
                                                    3'd1 :  begin sda_out_r <=  1'b0 ;  en
d       //SDA pulls low and prepares to STOP
                                                    3'd2 :  begin scl_out_r <=  1'b1 ;  en
d       //SCL advance SDA to raise 4.0us
                                                    3'd3 :  begin scl_out_r <=  1'b1 ;  en
d       //SCL advance SDA to raise 4.0us
                                                    3'd4 :  begin sda_out_r <=  1'b1 ;  en
d       / /SDA pull up
                                                    3'd5 :  beginsda_out_r <=  1'b1 ; stat
e <= MAIN ;  end        //Complete STOP operation and return to MAIN state
                                                    default : state <= IDLE ;       //If t
he program is out of control, enter IDLE self-reset state
                                                endcase
                                        end
                                default :;
                        endcase
                end
        end

        assign  scl_out = scl_out_r ;   //
        assign SCL port assign  sda_out = sda_out_r ;   // assign SDA port

endmodule
```

====Summary====

This section mainly explains the principle and software design of using I2C to drive the DAC function of PCF8591. You need to create your own project while mastering it, and generate FPGA configuration file loading test through the entire design process.
If you are not familiar with the use of Diamond software, please refer to here: Use of Diamond .

====Related Information====

Use STEP-MXO2 second generation of the DAC driver PCF8591: subsequent download connection will be updated
using the STEP-MAX10 of the DAC driver PCF8591: subsequent download link will be updated