

# RUNBER Development Board

## Instructions for Experiments

---

---

Version

<b>Date of revision</b>	<b>Version of revision</b>	<b>Reason for revision</b>
5/20/2020	v1.0	Document Creation
10/20/2020	v1.1	Addition of 4 instructions for experiment

## Contents

1	Control of LEDs.....	7
1.1	Purpose of the experiment.....	7
1.2	Requirements of the experiment .....	7
1.3	Principle of the experiment .....	7
1.4	Design of the source code of the experiment .....	8
1.4.1	<i>Design of the file header</i> .....	8
1.4.2	<i>Design of module</i> .....	9
1.4.3	<i>A complete Module (without comment header)</i> .....	10
1.4.4	<i>Distribution of hardware pins</i> .....	11
1.5	Steps of the experiment.....	11
1.5.1	<i>Open Gowin to create project</i> .....	11
1.5.2	<i>Add design file</i> .....	13
1.5.3	<i>Synthesize</i> .....	13
1.5.4	<i>Project constraint</i> .....	14
1.6	Result of the experiment .....	17
2	Flowing Water LED .....	18
2.1	Purpose of the experiment.....	18
2.2	Requirements of the experiment .....	18
2.3	Principle of the experiment .....	18
2.4	Design of the source code of the experiment .....	18
2.5	Steps of the experiment.....	19
2.6	Result of the experiment .....	19
3	Key-vibration elimination .....	20
3.1	Purpose of the experiment.....	20
3.2	Requirements of the experiment .....	20
3.3	Principle of the experiment .....	20
3.4	Design of the source code of the experiment .....	21
4	Key-controlled Flowing Water Lights.....	22
4.1	Purpose of the experiment.....	22
4.2	Requirements of the experiment .....	22
4.3	Principle of the experiment .....	22
4.3.1	<i>Function of the key control module</i> .....	22
4.3.2	<i>Function of the LED control module</i> .....	23
4.4	Design of the source code of the experiment .....	23
4.4.1	<i>Source code of the top files</i> .....	23
4.4.2	<i>Key control module</i> .....	24
4.4.3	<i>LED control module</i> .....	25
4.5	Result of the experiment .....	26
5	Static State of Numeric Displays .....	27
5.1	Purpose of the experiment.....	27
5.2	Requirements of the experiment .....	27
5.3	Principle of the experiment .....	27

5.3.1	<i>Operating principle of numeric displays</i> .....	27
5.3.2	<i>Scheme design</i> .....	28
5.4	Source code of the experiment.....	29
5.5	Result of the experiment.....	29
6	Dynamic Display of Numeric Display.....	30
6.1	Purpose of the experiment.....	30
6.2	Requirements of the experiment.....	30
6.3	Principle of the experiment.....	30
6.4	Source code of the experiment.....	31
6.4.1	<i>Modules at top level</i> .....	31
6.4.2	<i>Key control module</i> .....	34
6.4.3	<i>Numeric display control module</i> .....	34
6.5	Result of the experiment.....	35
7	UART Serial Port Communication.....	36
7.1	Purpose of the experiment.....	36
7.2	Requirements of the experiment.....	36
7.3	Principle of the experiment.....	36
7.3.1	<i>Principle of the serial port</i> .....	36
7.3.2	<i>Steps of serial port transmission</i> .....	38
7.3.3	<i>Characters sent via serial port</i> .....	38
7.4	Design of the source code of the experiment.....	38
7.4.1	<i>Design of serial port transmission module</i> .....	39
7.4.2	<i>Design of serial port receiving module</i> .....	49
7.4.3	<i>Design of serial port transmission control module</i> .....	53
7.4.4	<i>Design of top level module of serial port experiment</i> .....	55
7.5	Result of the experiment.....	57
8	Sequence Detector.....	58
8.1	Purpose of the experiment.....	58
8.2	Requirements of the experiment.....	58
8.3	Principle of the experiment.....	58
8.4	Design of the source code of the experiment.....	58
8.4.1	<i>Scheme design</i> .....	58
8.4.2	<i>Design of top level modules (including the numeric display module)</i> .....	59
8.4.3	<i>LED key control module</i> .....	61
8.4.4	<i>Design of sequence detection module</i> .....	63
8.5	Result of the experiment.....	65
9	Coded Lock.....	66
9.1	Purpose of the experiment.....	66
9.2	Requirements of the experiment.....	66
9.3	Principle of the experiment.....	66
9.4	Source code of the experiment.....	66
9.4.1	<i>Design of top level modules</i> .....	67
9.4.2	<i>Design of key control</i> .....	69
9.4.3	<i>Design of comparison module</i> .....	71

9.4.4	<i>Design of display module</i> .....	72
9.5	Result of the experiment .....	76
10	Digital Clock .....	77
10.1	Purpose of the experiment .....	77
10.2	Requirements of the experiment .....	77
10.3	Principle of the experiment .....	77
10.4	Source code of the experiment .....	78
10.4.1	<i>Top level design</i> .....	78
10.4.2	<i>Design of clock timing and control module</i> .....	81
10.4.3	<i>Design of the numeric display module</i> .....	83
10.5	Result of the experiment .....	84
11	Frequency meter .....	86
11.1	Purpose of the experiment .....	86
11.2	Requirements of the experiment .....	86
11.3	Principle of the experiment .....	86
11.3.1	<i>Key input control module</i> .....	86
11.3.2	<i>Frequency generation and measurement module</i> .....	87
11.3.3	<i>Control of display on the numeric display</i> .....	87
11.4	Source code of the experiment .....	87
11.4.1	<i>Modules at top level</i> .....	87
11.4.2	<i>Key input control module</i> .....	88
11.4.3	<i>Frequency generation and measurement module</i> .....	88
11.5	Result of the experiment .....	90
12	Reaction time tester .....	91
12.1	Purpose of the experiment .....	91
12.2	Requirements of the experiment .....	91
12.3	Principle of the experiment .....	91
12.4	Source code of the experiment .....	91
12.4.1	<i>Top level design</i> .....	91
12.4.2	<i>Key-vibration elimination</i> .....	93
12.4.3	<i>LED display control module</i> .....	93
12.4.4	<i>Timing comparison module</i> .....	95
12.4.5	<i>Display module of the numeric display</i> .....	97
12.5	Result of the experiment .....	97
13	Servo Experiment .....	98
13.1	Purpose of the experiment .....	98
13.2	Requirements of the experiment .....	98
13.3	Principle of the experiment .....	98
13.4	Source code of the experiment .....	99
13.4.1	<i>Modules at top level</i> .....	99
13.4.2	<i>Key-vibration elimination module</i> .....	99
13.4.3	<i>Module for angle adjustment via keys</i> .....	99
13.4.4	<i>PWM output module</i> .....	100
13.5	Result of the experiment .....	102

14	Ultrasonic ranging.....	103
14.1	Purpose of the experiment.....	103
14.2	Requirements of the experiment .....	103
14.3	Principle of the experiment .....	103
	14.3.1 Input/output control.....	103
	14.3.2 Calculation of the distance.....	104
	14.3.3 Control of display on the numeric display .....	105
14.4	Source code of the experiment.....	105
	14.4.1 Modules at top level .....	105
	14.4.2 Input/output control.....	106
	14.4.3 Calculation of velocity .....	108
14.5	Result of the experiment .....	110
15	Measurement of temperature and humidity .....	111
15.1	Purpose of the experiment.....	111
15.2	Requirements of the experiment .....	111
15.3	Principle of the experiment .....	111
	15.3.1 Input/output control.....	111
	15.3.2 Data conversion .....	111
	15.3.3 Control of display on the numeric display .....	112
15.4	Source code of the experiment.....	112
	15.4.1 Modules at top level .....	112
	15.4.2 Input/output control.....	115
	15.4.3 Data reading .....	116
	15.4.4 Data conversion .....	119
15.5	Result of the experiment .....	123

## 1 Control of LEDs

### 1.1 Purpose of the experiment

To realize control of multiple LEDs

### 1.2 Requirements of the experiment

Control of 8 LEDs for periodical twinkles over a cycle of 1s (on for 0.5s and off for 0.5s)

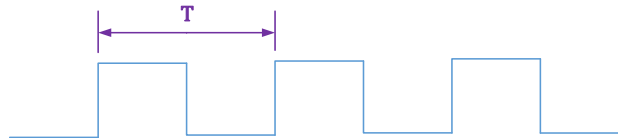
### 1.3 Principle of the experiment

We are all familiar with the units of time (hour, minute and second) as well as the relationship between them:



1 hour = 60 minutes = 3600seconds, which means the second hand of a clock will go 3600 steps forward when the hour hand goes 1 step forward.

There is also a fixed pace of clock signal in the digital circuit. The period between the start and the end of the pace is called a cycle (T).



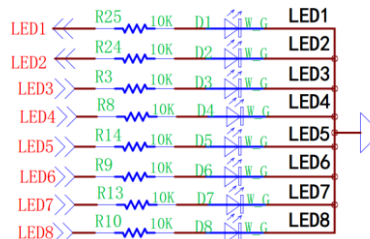
The frequency of the clock is of concern in the digital system, and the relationship between frequency and cycle is shown below:

$$f = \frac{1}{T}$$

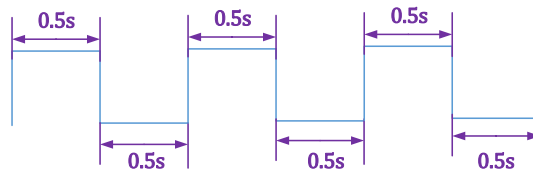
The Runber board features a 12MHz crystal oscillator that provides clock for GW1N.

### Experiment analysis:

Whether the LED is on or off is controlled by the level of the IO output (it is on when the output at high level, and off at low level), as shown in the schematic diagram below:



To keep the LED on for 0.5s and off for 0.5s periodically, we need to control the IO output at high level for 0.5s and at low level for 0.5s alternately, as shown by the waveform below.



The cycle of the external input 12MHz clock is 83.333ns (The timing principle of the counter in design of verilog is basically the same; when the cycle of the input clock and the target length of timing are set, we can find the count value of the counter needed to get the timing width):

$$0.5s = 6000000 * 83.33ns = 6000000 * T_{12MHz};$$

There are only 2 values of IO output: 1 or 0. We can use a counter, and the value of the IO output will change when the counting reaches 6000000 clock cycles. That is, the output value will skip every 0.5s, and the LED will be on and off alternately every 0.5s.

## 1.4 Design of the source code of the experiment

### 1.4.1 Design of the file header

A file header will be added before the module is created, which includes the company, author, time, design name, project name, module name, target device, EDA tool (version), module description, version description (revision description) and other information, as well as the definition of the unit of simulation time:

```

1  `timescale 1ns / 1ps
2  ///////////////////////////////////////////////////////////////////
3  //      Company      : MYMINIEYE Technology CO., Ltd
4  //      Internet site  : www.myminieye.com
5  //      WeChat public account : MYMINIEYE
6  //      Engineer      : Nil
7  //
8  //      Create Date   : 2020-06-04 14:34
9  //      Design Name   : led_light
10 //      Module Name   : led_light
11 //      Project Name  : led_light
12 //      Target Devices : Gowin@GW1N-UV4LQ144C6/15
13 //      Tool Versions  : Gowin1.9.2
14 //      Description   :
15 //
16 //      Dependencies  :
17 //
18 //      Revision      : Revision 0.01 - File Created
19 //      Additional Comments :
20 //
21 ///////////////////////////////////////////////////////////////////
22 `define UD #1
    
```

`timescale 1ns / 1ps means the simulation precision is 1ns and the display precision is 1ps;

`define UD #1 defines UD as #1, which is valid for simulation only, meaning a delay of a simulation precision, or 1ns as shown in the statement above.



## 1.4.2 Design of module

### 1.4.2.1 Creation of module, and determination of input/output signal

```
module led_light(  
    input      clk,      // input clock, the frequency is 12MHz  
    input      rstn,     // input reset signal, active at low  
    output [7:0] led     // output LED control signal , lighting at high  
);  
endmodule
```

The code above is a standard model for creation of a module, which needs determination of input/output signal and definition of bit width, before the specific logic design of the module. The space mark “,” is needed between pins, and no space mark is needed after the last pin.

Input/output signal is to be defined when the module is created. The input in this experiment is the clock and reset, and the output is the control of the state of the LED (on or off). There are 8 LEDs on the Runber board, so the output is the signal of a width of 8 bits.

### 1.4.2.2 Design of a counter

The counting for a single state is 6000000, and that for a cycle (on and off) is 12000000 = 24'hB71B00. So the width of the counter can be 24 bits. A larger bit width is needed for compatibility with higher frequency (the bit width of the counter in the code below is designed as 25 bits for compatibility with up to 50MHz, 25'd25000000 = 25'h17D7840). For analysis, please refer to the operating principle of synchronous counter in digital circuit.

```
1    reg [24:0] led_light_cnt;  
2  
3    // time counter  
4    always @(posedge clk) // Triggering condition; posedge is rising edge, and negedge  
    is falling edge  
5    begin  
6        if(!rstn)  
7            led_light_cnt <= `UD 25'd0;  
8        else if(led_light_cnt == 25'd599_9999)  
9            led_light_cnt <= `UD 25'd0;  
10       else  
11           led_light_cnt <= `UD led_light_cnt + 25'd1;  
12    end
```

When the counting of the counter reaches 25'd5999999, the cycle of the counting contains 0-25'd5999999 clock cycles, so the total time length is  $25'd6000000 * T_{clk}$ ; the frequency of the hardware input clock is 12MHz, so the technical cycle of the counter is 0.5s.

### 1.4.2.3 Control of the display of LED

The state of LED is changed at specific time scale to control the regular display of LED (on or off).

The timing cycle of led\_light\_cnt is 0.5s, so the display of LED can be changed every 0.5s if the state of LED is changed at a fixed point on led\_light\_cnt. As LED has only 2 states (on or off), the

change of the state from On to Off (or from Off to On) can be realized by negation of the register in assignment.

```

1  reg [24:0] led_status;
2
3  always @(posedge clk)
4  begin
5      if(!rstn)
6          led_status <= `UD 8'd0;
7      else if(led_light_cnt == 25'd599_9999)
8          led_status <= `UD ~led_status;
9  end
10
11 assign led = led_status;

```

### 1.4.3 A complete Module (without comment header)

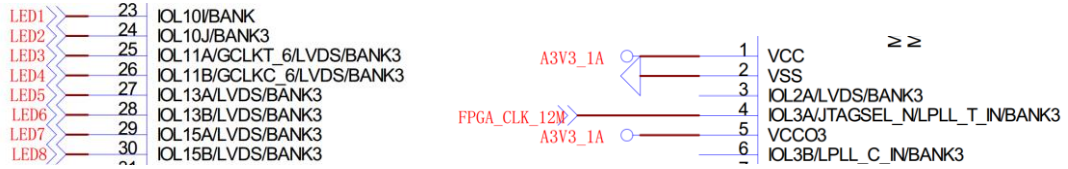
```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module led_light(
4      input      clk,
5      input      rstn,
6      output [7:0] led
7  );
8
9  //=====
10     //reg and wire
11     reg [24:0] led_light_cnt;
12     reg [ 7:0] led_status;
13     // time counter
14     always @(posedge clk)
15     begin
16         if(!rstn)
17             led_light_cnt <= `UD 25'd0;
18         else if(led_light_cnt == 25'd599_9999)
19             led_light_cnt <= `UD 25'd0;
20         else
21             led_light_cnt <= `UD led_light_cnt + 25'd1;
22     end
23
24     // led status change
25     always @(posedge clk)
26     begin
27         if(!rstn)
28             led_status <= `UD 8'd0;
29         else if(led_light_cnt == 25'd599_9999)
30             led_status <= `UD ~led_status;
31     end
32     assign led = led_status;
33
34 endmodule
35

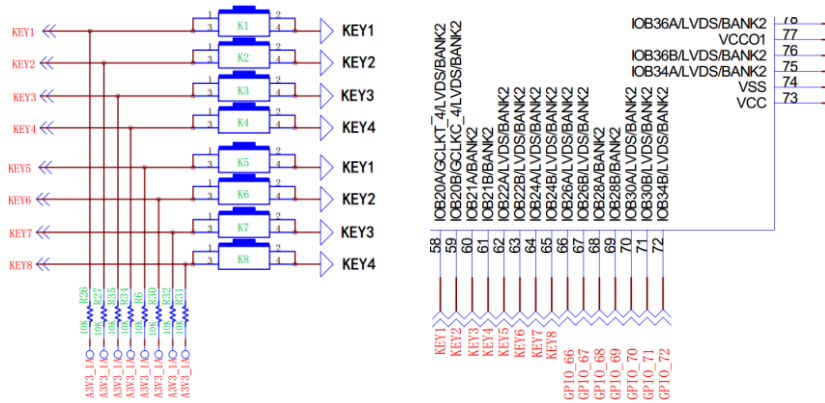
```

### 1.4.4 Distribution of hardware pins

The schematic diagram of the connection between the LED and CLK of Runber and the IO of FPGA is shown below (for physical constraint in the project, please refer to the distribution of the FPGA pins in the schematic diagram):



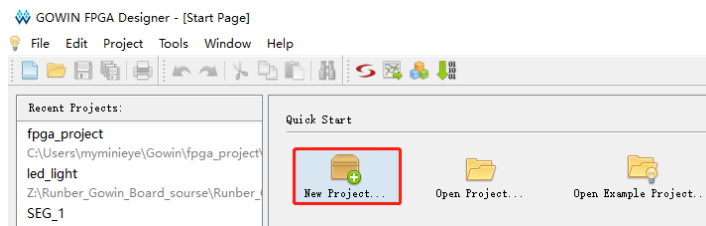
The design of reset is valid for low level. The low level is obtained when the key on the Runber board is pressed down, and high level is got when the key is released. The input of the key can be used as the reset signal, and KEY1 can be used as resetting key.



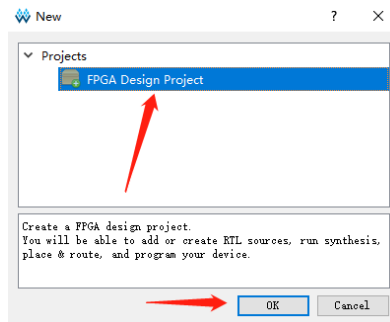
## 1.5 Steps of the experiment

### 1.5.1 Open Gowin to create project

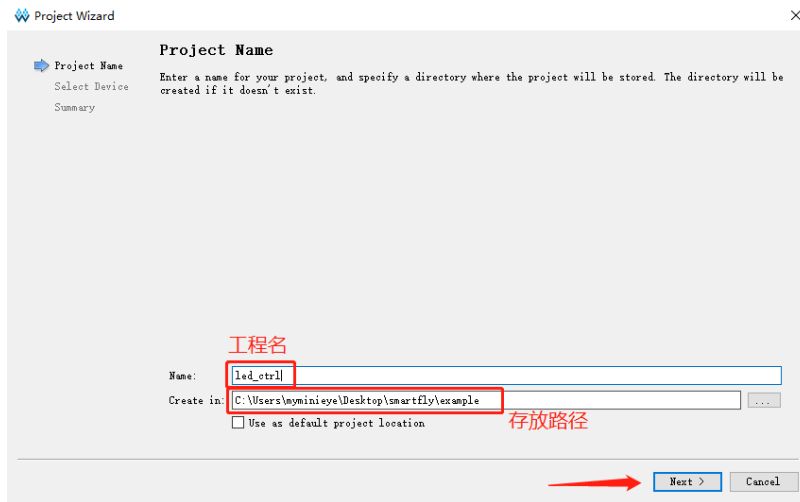
Step1: Open gowin software, and click New Project....



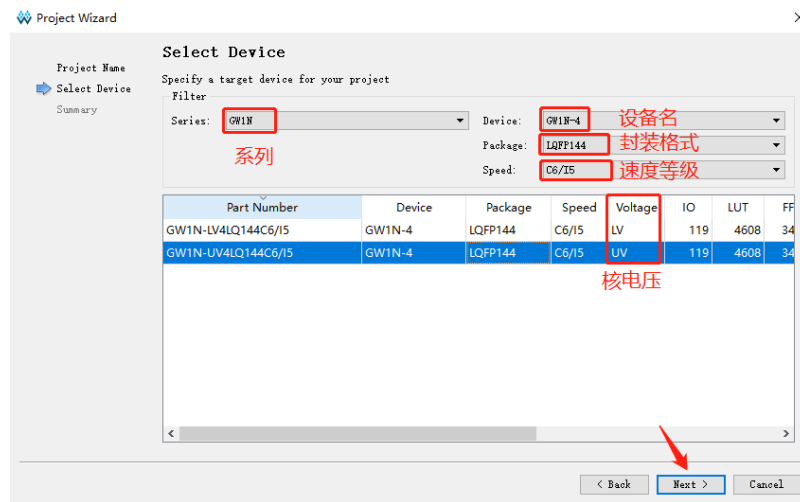
Step2: Select FPGA Design Project, and click OK



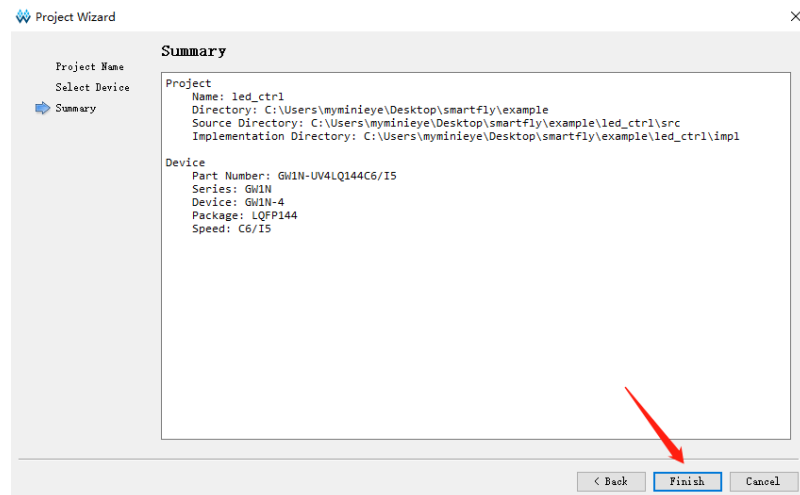
Step3: Create a project named led\_ctrl to the corresponding file directory, and then click Next



Step4: Select package format and speed level, and then click Next

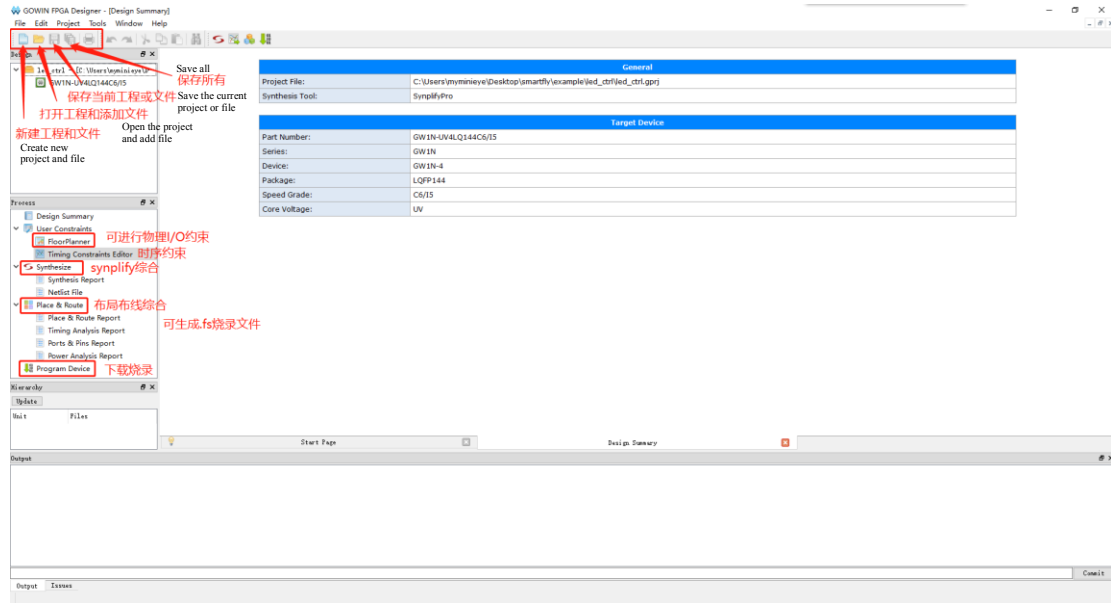


Step5: Click Finish to finish the project creation



### 1.5.2 Add design file

The interface of the Gowin cloud source software is shown below:

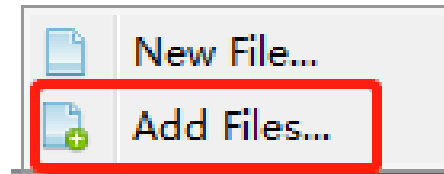


Add verilog file, copy the module in the previous design to the file, or add the previously edited verilog file to the project:

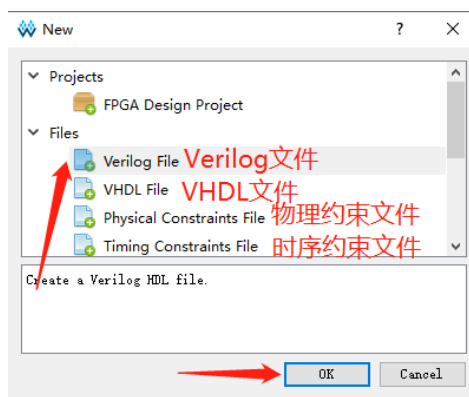
Add file to project:

Right click in the Design window, and select ADD FILE;

Create file to project:



Click New File above, or click File/New..., or right click a space in the Design window and select New File...; click Verilog File when the New File window pops up, and then click OK



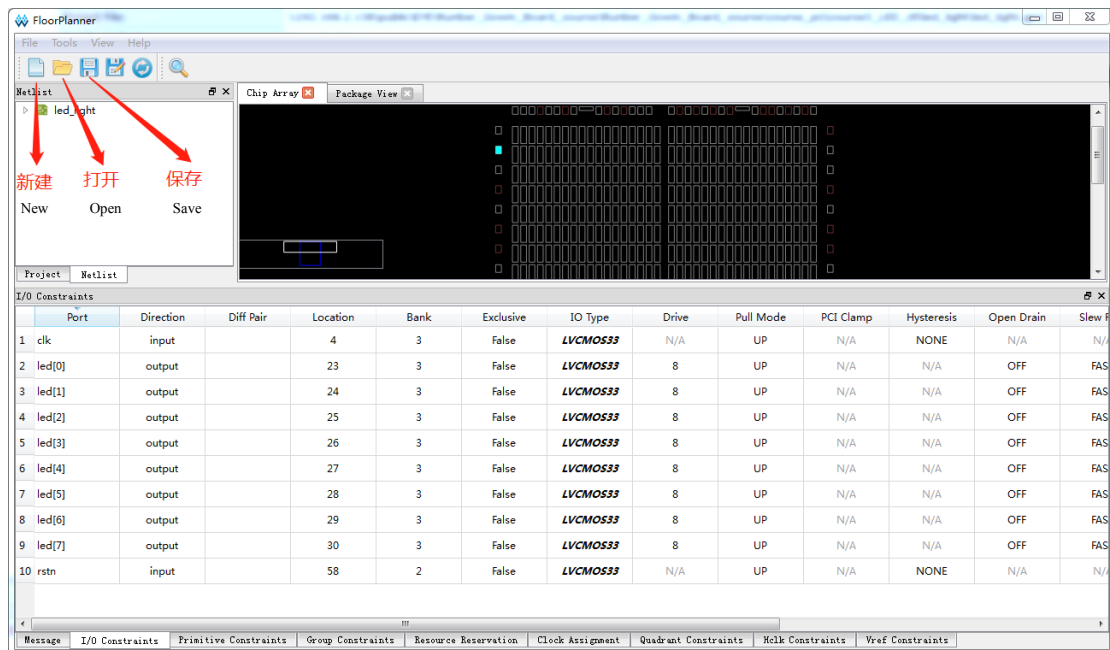
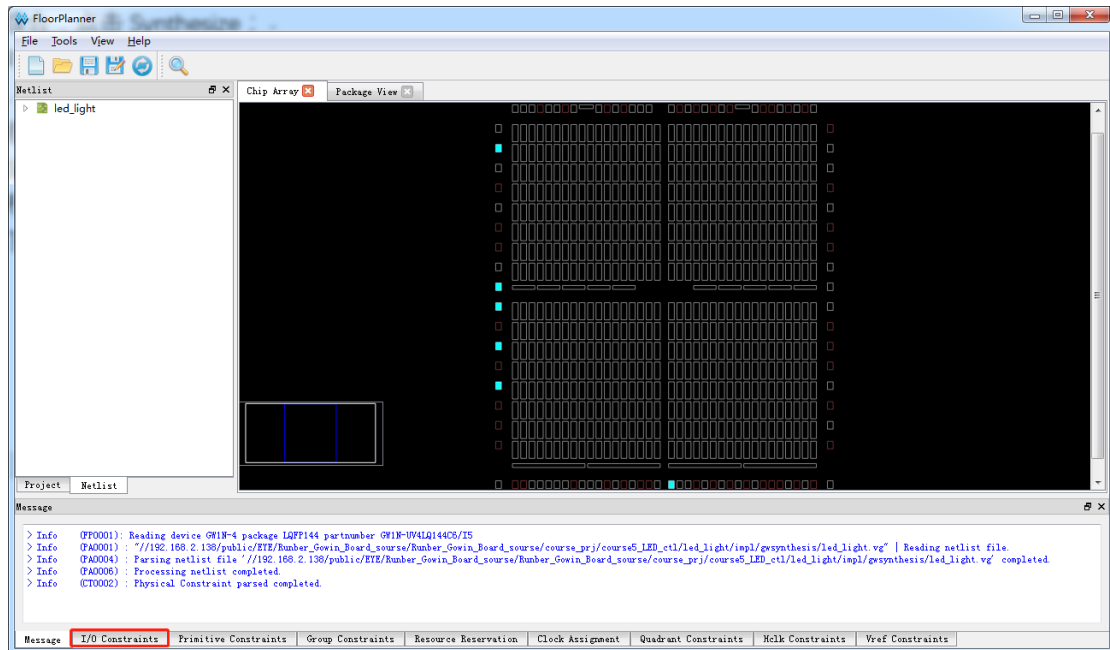
### 1.5.3 Synthesize

Click Synthesize.

## 1.5.4 Project constraint

### 1.5.4.1 Physical constraint on the project

Double click FloorPlanner, and then click I/O Constraints at the bottom of the popup window to edit IO distribution.



When IO distribution is modified in accordance with the schematic diagram, click Save to save a cst file (for the grammar of the constraint please refer to the official Gowin document: Guide to Constraint on Gowin Design). The contents of the file is as follows (A new cst file can also be created directly. The contents in the editor and the grammar of the constraint can be compared with those in the file configured and generated by the tool):

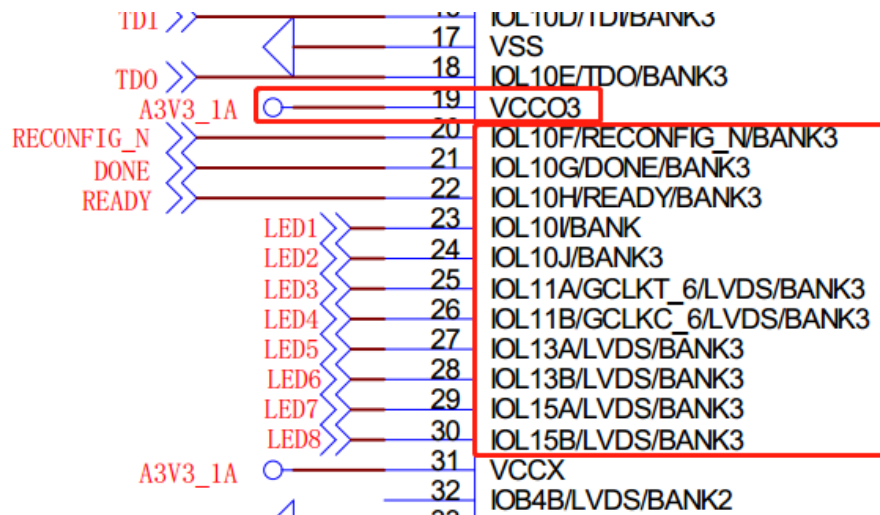
```
1 //Copyright (C)2014-2019 Gowin Semiconductor Corporation.
2 //All rights reserved.
3 //File Title: Physical Constraints file
4 //GOWIN Version: V1.9.1.01Beta
5 //Part Number: GW1N-LV4LQ144C6/I5
6 //Created Time: Wed Sep 04 20:36:36 2019
7 IO_LOC "clk" 4;
8 IO_LOC "led[0]" 23;
9 IO_LOC "led[1]" 24;
10 IO_LOC "led[2]" 25;
11 IO_LOC "led[3]" 26;
12 IO_LOC "led[4]" 27;
13 IO_LOC "led[5]" 28;
14 IO_LOC "led[6]" 29;
15 IO_LOC "led[7]" 30;
16 IO_LOC "rstn" 58;
17 IO_PORT "clk" IO_TYPE=LVCMOS33;
18 IO_PORT "rstn" IO_TYPE=LVCMOS33;
19 IO_PORT "led[0]" IO_TYPE=LVCMOS33;
20 IO_PORT "led[1]" IO_TYPE=LVCMOS33;
21 IO_PORT "led[2]" IO_TYPE=LVCMOS33;
22 IO_PORT "led[3]" IO_TYPE=LVCMOS33;
23 IO_PORT "led[4]" IO_TYPE=LVCMOS33;
24 IO_PORT "led[5]" IO_TYPE=LVCMOS33;
25 IO_PORT "led[6]" IO_TYPE=LVCMOS33;
26 IO_PORT "led[7]" IO_TYPE=LVCMOS33;
```

IO\_LOC "signal" pin: means connecting signal to pins.

Naming of pins is different for different package, and it should be noted that pin represents the PAD (pin) No. of the chip.

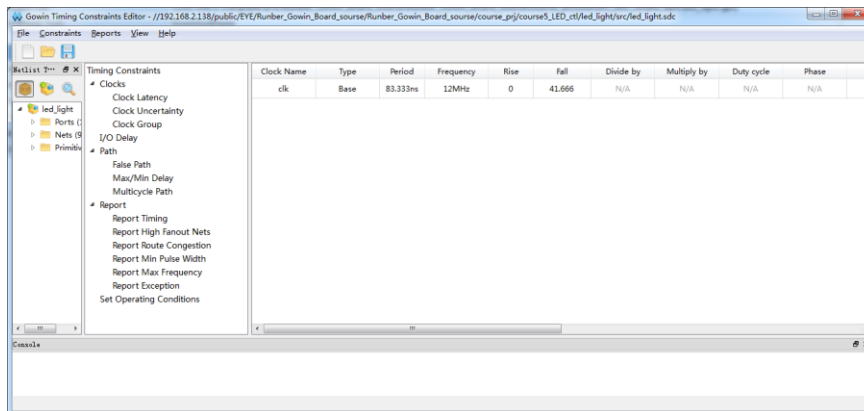
IO\_PORT "signal" IO\_TYPE=Level: means that the standard for the level of the signal is set as Level.

IO\_TYPE is the standard for the level, the voltage of which should be determined in combination with the bank power supply. (As shown in the figure below, the LED on the Runber board is assigned to Bank3; VCCO3 is the bank power IO for Bank3, and the power connected to it is 3.3V.) The type of IO can be LVDS, LVCMOS, LVTTTL or others as needed, so we define the IO\_TYPE of LED as LVCOMS33.

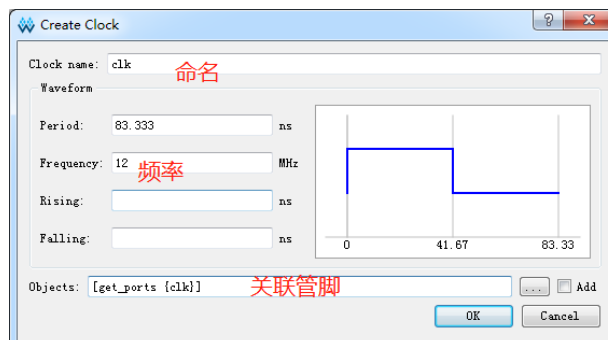


### 1.5.4.2 Edit timing constraints file

Double click Timing Constraints Editor in the box of Process in Gowin.



While this project is relatively simple, we need to tell the EDA tool the clock frequency we enter. Right click in the blank window in the upper left part, and then select creat clock; name the clock in the popup window, edit the clock frequency, and connect the input pin.



A clock constraint will be created after clicking OK. Click Save in the main window of the clock constraint to save a cst file with the contents as follows (for the grammar of the constraint please refer to the official Gowin document: Guide to Constraint on Gowin Design):



```
1 //Copyright (C)2014-2020 GOWIN Semiconductor Corporation.  
2 //All rights reserved.  
3 //File Title: Timing Constraints file  
4 //GOWIN Version: 1.9.2.02 Beta  
5 //Created Time: 2020-01-07 11:18:21  
6 create_clock -name clk -period 83.333 -waveform {0 41.666} [get_ports  
  {clk}]
```

- 1) Placement and routing: Double click Place & Route;
- 2) Download burn tool to the board: Double click Program Device. Detailed description of the corresponding interface can be found in the materials of the first week, which is not described here. (The repeated part in the subsequent documents will be presented summarily.)

### 1.6 Result of the experiment

The 8 LEDs get on and off alternately at the same time, with a time interval of 0.5s.

## 2 Flowing Water LED

### 2.1 Purpose of the experiment

To master the principle of the flowing water light and realize it

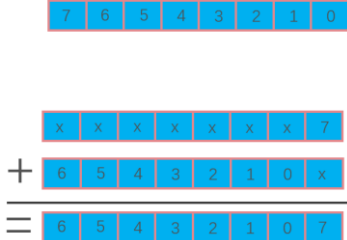
### 2.2 Requirements of the experiment

8 LEDs form a flowing water light with a time interval of 0.5s

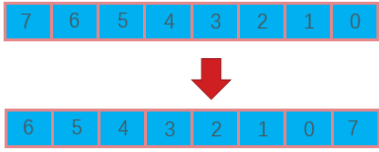
### 2.3 Principle of the experiment

Compared with the twinkle of the LEDs in the previous experiment, we just need to change the states of the LEDs to get the 8 LEDs on in turn.

An intermediate variable is needed to conduct the flowing water light experiment in C language. The code is shown in the left side below, and the transfer of the data bits is shown in the right figure below:

<pre> 1  int data; 2  int temp1,temp2; 3 4  data = 0x01; 5  temp1 = data &gt;&gt; 7; 6  temp2 = data &lt;&lt;1; 7  data = temp1   temp2;                 </pre>	
---	---

The development in FPGA is hardware-based, with the language being hardware description language (HDL), and the processing unit of verilog is 1bit. Data with a bit width of 8 bits can be considered as 8 separate signal lines, and the ordering of the 8 lines and the assignment of values to them can be carried out in random combination. The code is shown below:

<pre> 1  reg [7:0] data; 2 3  always @(posedge clk) 4  data &lt;= {data[6:0],data[7]}; 5  // or: 6  wire [7:0] data1; 7  wire [7:0] out; 8  assign out = {data1[6:0],data1[7]};                 </pre>	
--	--

### 2.4 Design of the source code of the experiment

The contents of the Module are shown below:



```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module water_led (
4      input      clk,
5      input      rstn,
6      output [7:0] led
7  );
8
9  //=====
10 //reg and wire
11     reg [24:0] led_light_cnt;
12     reg [ 7:0] led_status;
13
14     // time counter
15     always @(posedge clk)
16     begin
17         if(!rstn)
18             led_light_cnt <= `UD 25'd0;
19         else if(led_light_cnt == 25'd599_9999)
20             led_light_cnt <= `UD 25'd0;
21         else
22             led_light_cnt <= `UD led_light_cnt + 25'd1;
23     end
24
25     // led status change
26     always @(posedge clk)
27     begin
28         if(!rstn)
29             led_status <= `UD 8'b0000_0001;
30         else if(led_light_cnt == 25'd599_9999)
31             led_status <= `UD {led_status[6:0], led_status[7]};
32     end
33
34     assign led = led_status;
35
36 Endmodule
```

## 2.5 Steps of the experiment

Creation of the project and process of the compilation are the same as the twinkle of the LEDs in the previous experiment. When adding the file, we just need to add the verilog file of water\_led in this experiment. The distribution of the pins is the same as the twinkle of the LEDs in the previous experiment.

## 2.6 Result of the experiment

The 8 LEDs are lightened in turn, with the previous LED getting off as the next gets on, and so on. It seems that the LED keeping on is flowing along the line of the 8 LEDs, thus the name of “Flowing Light”.

### 3 Key-vibration elimination

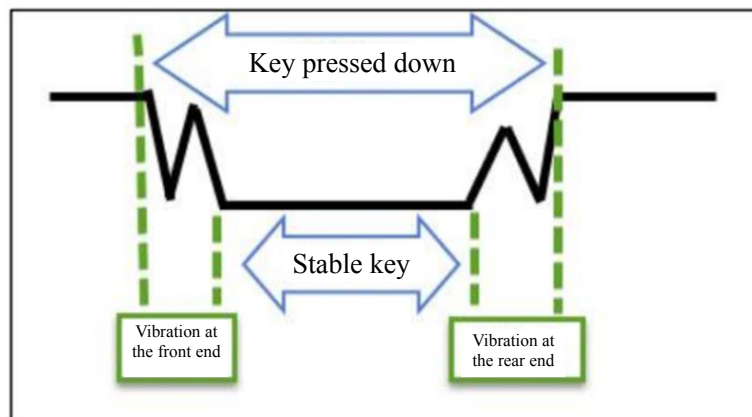
#### 3.1 Purpose of the experiment

There is mechanical vibration when a mechanical dome key is pressed down or released, leading to an unstable state of the key and rapid changes. When the key is used to input a signal, the state of vibration will lead to uncontrollable change to the operation of the project. So we need to eliminate the vibration from the signal, and that is the reason for the name of this experiment.

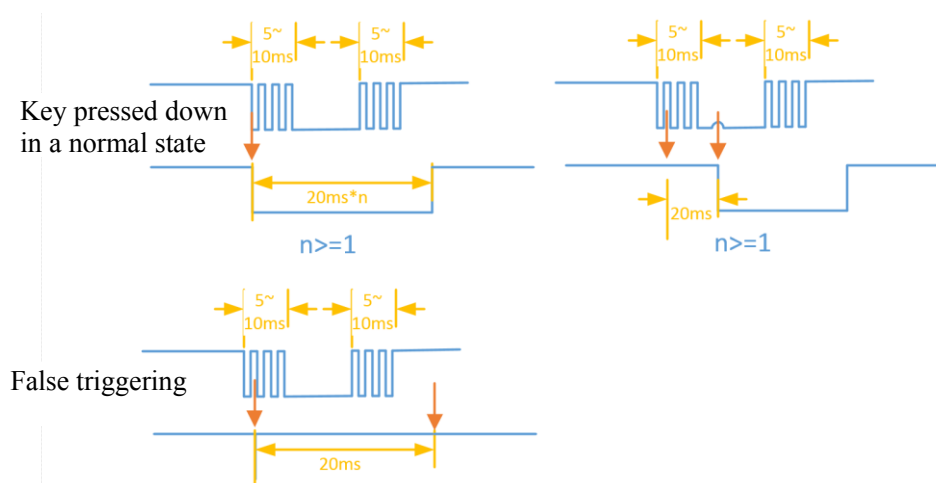
#### 3.2 Requirements of the experiment

To realize key-vibration elimination

#### 3.3 Principle of the experiment



Vibration at the front and rear end each lasts for about 5-10ms, with a total time of within 20ms. In the design for the maximum time of 20ms, a counter that returns to zero upon reaching N is used as a timescale. The input signal of the key is collected with an interval of 20ms to prevent the rapid change in the signal resulting from the vibration of the key.



Design a counter of 18 bits with the maximum counting being  $N = 18'h3FFFF = 18'd262143$

At the maximum counting, the time  $t = N * T = N / f = 262143 / 12M = 21.84525ms > 20ms$ .

Note: There is detailed description of the timing function of the counter in the control of LEDs. Attention should be paid to the frequency of the input clock and the length of the target timing, so as to decide the scope of the counting.

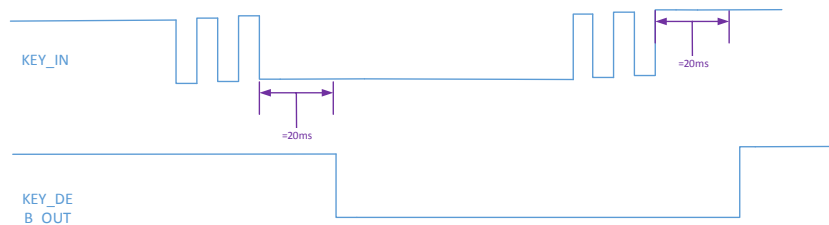
### 3.4 Design of the source code of the experiment

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module btn_deb#(
4      parameter          BTN_WIDTH = 4'd8 // key width
5      常量
6  )
7  (
8      input              clk,          //12MHz
9      Input [BTN_WIDTH-1:0] btn_in,    // key input
10     output reg [BTN_WIDTH-1:0] btn_deb // key output
11 );
12 //=====
13     reg [17:0] time_cnt= 18'd0;
14     always @ (posedge clk)
15     begin
16         if(time_cnt == 18'h3A980) // reset at 20ms counter over
17             time_cnt <= `UD 20'd0;
18         else
19             time_cnt <= `UD time_cnt + 20'd1;
20     end
21
22     always @(posedge clk)
23     begin
24         if(time_cnt == 20'd0)
25             btn_deb <= `UD btn_in; // latch the key input every 20ms
26     end
27
28     endmodule

```

There is a certain probability of false triggering in this method, which can be improved as indicated below (a realization with expansion):



A new grammar (**parameter**) can be added to the design of this module. The parameter defines a constant in verilog. Delivery of the module can be carried out by placing the definition of the parameter into the port of the module. For the method of delivery, please see the module instantiation in the subsequent part.

## 4 Key-controlled Flowing Water Lights

### 4.1 Purpose of the experiment

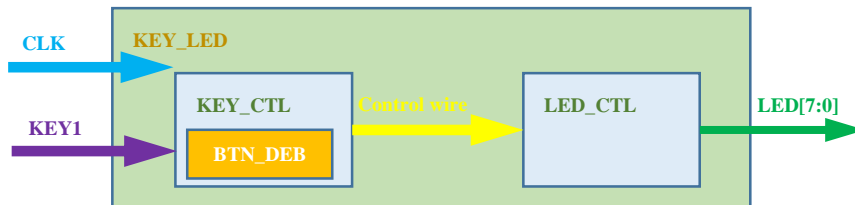
1. To design 4 patterns of flowing water lights, which can be controlled by a choice of the key.
2. To realize a circulation between the 4 patterns with a key as the control input, which can change the light to the next pattern at a press.

### 4.2 Requirements of the experiment

1. Experiment platform: Hummingbird series development board.
2. K1 provides the key input, and the LED output is D1-D8.

### 4.3 Principle of the experiment

The framework for realization is shown below:



1. Switch of the state of flowing LED by the key is realized on the top level.
2. Design of an input control module and an output control module is needed.

This experiment is about integrating multiple modules into a project, which involves sub-module design and module instantiation. The sub-module design is mainly about determination of the input and output based on their functions and the subsequent detailed design.

The method of module instantiation is shown below:

```

1  module_name # (
2      .PARAM      ( PARAM_SET )      // PARAM is the constant port of the instantiated module;
    PARAM_SET is the contents of assignment to the constant
3  ) uint_name(      // module_name is the instantiated module name; uint_name is the instantiated unit name
4      .port      ( signal      )      // port is the pin in the instantiated module; signal is the
    signal of the current module
5  );

```

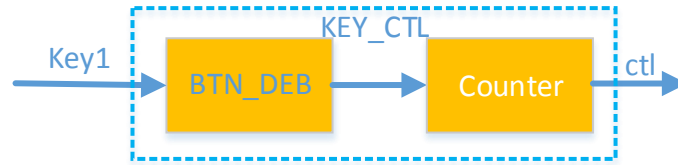
#### 4.3.1 Function of the key control module

Receiving input signal from the key. Counting the times of pressing on the key: As there are 4 patterns of the flowing light, the scope of counting is 0-3; the result of the counting is sent to the LED control module.

According to the requirement, the input signals are the clock and the key, and the output signal is the light control signal.

Internal functional processing:

- (1) Vibration in the key signal needs to be eliminated internally;
- (2) Press on the key triggers the change of the counter (output of the counting), thus altering the pattern of the flowing light.



### 4.3.2 Function of the LED control module

Switch of the 4 patterns of the flowing light is controlled by the counting sent from the key; initialization of the next pattern begins when the state of each LED is completed. According to the requirement, the following information can be obtained:

Input signals: clock, signal to control the pattern of the flowing light; output signal: LED control signal with a width of 8 bits.

Notes for functional processing: switch point of the state of the flowing light, and the method of initialization for switch between different states.

## 4.4 Design of the source code of the experiment

### 4.4.1 Source code of the top files

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module key_led_top(
4      input      clk,
5      input      key,
6      output [7:0] led
7  );
8
9      wire [1:0] ctrl;
10
11     key_ctl key_ctl(
12         .clk      ( clk ),//input      clk,
13         .key      ( key ),//input      key,
14         .ctrl     ( ctrl )//output [1:0] ctrl
15     );
16
17     led u_led(
18         .clk      ( clk ),//input      clk,
19         .ctrl     ( ctrl ),//input [1:0] ctrl,
20
21         .led      ( led )//output[7:0] led
22     );
23
24 endmodule
25
  
```

## 4.4.2 Key control module

```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module key_ctl(
4      input          clk,
5      input          key,
6
7      output [1:0] ctrl
8  );
9
10     wire btn_deb;
11     // Key-vibration elimination
12     btn_deb#(
13         .BTN_WIDTH ( 4'd1 ) //parameter          BTN_WIDTH = 4'd8
14     ) U_btn_deb
15     (
16         .clk      ( clk ) //input          clk,
17         .btn_in   ( key ) //input [BTN_WIDTH-1:0] btn_in,
18
19         .btn_deb  ( btn_deb ) //output reg [BTN_WIDTH-1:0] btn_deb
20     );
21
22     reg btn_deb_1d;
23     always @(posedge clk)
24     begin
25         btn_deb_1d <= `UD btn_deb; //get the btn_deb delay one clock cycle
26     End
27
28     // The way to obtain the falling edge: the previous clock cycle is at high level, and the current clock
29     // cycle is at low level;
30     // So the signal after the key-vibration elimination is delayed for a cycle (the state of the previous
31     // clock cycle is kept)
32     //
33     // sig          _____|_____
34     //
35     //sig_reg      _____|_____
36     //
37     //falling      _____|_____
38
39     reg [1:0] key_push_cnt=2'd0;
40     always @(posedge clk)
41     begin
42         if(~btn_deb & btn_deb_1d) //get he falling edge of btn_deb
43         begin
44             key_push_cnt <= `UD key_push_cnt + 2'd1;
45         end
46     end
47
48     assign ctrl = key_push_cnt;
49 endmodule
```

The description of the key-vibration elimination module will not be repeated here. Please see chapter 3 for the principle of the key-vibration elimination module.





#### 4.4.3 LED control module

```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module led(
4      input          clk,
5      input  [1:0]  ctrl,
6      output [7:0]  led
7  );
8
9      reg [24:0] led_light_cnt = 25'd0;
10     reg [ 7:0] led_status = 8'b1000_0000;
11
12     // time counter
13     always @(posedge clk)
14     begin
15         if(led_light_cnt == 25'd19_999_999)
16             led_light_cnt <= `UD 25'd0;
17         else
18             led_light_cnt <= `UD led_light_cnt + 25'd1;
19     end
20
21     reg [1:0] ctrl_1d; // Save the ctrl value of the previous led state cycle
22     always @(posedge clk)
23     begin
24         if(led_light_cnt == 25'd19_999_999)
25             ctrl_1d <= ctrl; // The design here can ensure that the state of the next pattern begins since time 0
26     end
27
28     // led status change
29     always @(posedge clk)
30     begin
31         if(led_light_cnt == 25'd19_999_999) // 0.5s cycle
32         begin
33             case(ctrl)
34                 2'd0 : // Flowing water LED from high level to low level
35                 begin
36                     if(ctrl_1d != ctrl)
37                         led_status <= `UD 8'b1000_0000;
38                     else
39                         led_status <= `UD {led_status[0],led_status[7:1]};
40                 end
31                 2'd1 : // Flowing water LED from low level to high level
42                 begin
43                     if(ctrl_1d != ctrl)
44                         led_status <= `UD 8'b0000_0001;
45                     else
46                         led_status <= `UD {led_status[6:0],led_status[7]};
47                 end
48             end
49         end
50     end
```



```
48         2'd2 : // Increase of number of lights on from low level to high level
49         begin
50             if(ctrl_1d != ctrl || led_status == 8'b1111_1111)
51                 led_status <= `UD 8'b0000_0000;
52             else
53                 led_status <= `UD {led_status[6:0],1'b1};
54             end
55         2'd3 : // Increase of number of lights off from high level to low level
56         begin
57             if(ctrl_1d != ctrl || led_status == 8'b0000_0000)
58                 led_status <= `UD 8'b1111_1111;
59             else
60                 led_status <= `UD {1'b0,led_status[7:1]};
61             end
62         endcase
63     end
64 end
65
66     assign led = led_status;
67
68 endmodule
69
70 endmodule
71
```

#### 4.5 Result of the experiment

After the firmware is downloaded, the light is on in a manner of flowing from LED8 - LED1. Each time KEY1 is pressed on, the pattern of the light is switched between the 4 states periodically.

## 5 Static State of Numeric Displays

### 5.1 Purpose of the experiment

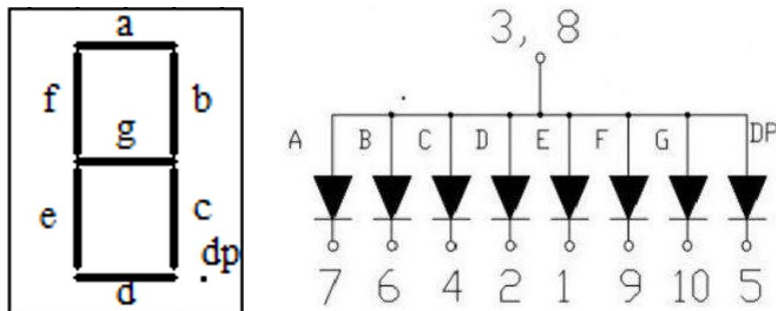
### 5.2 Requirements of the experiment

4 numeric displays display 0-9 at the same time, with the number changing every 1s.

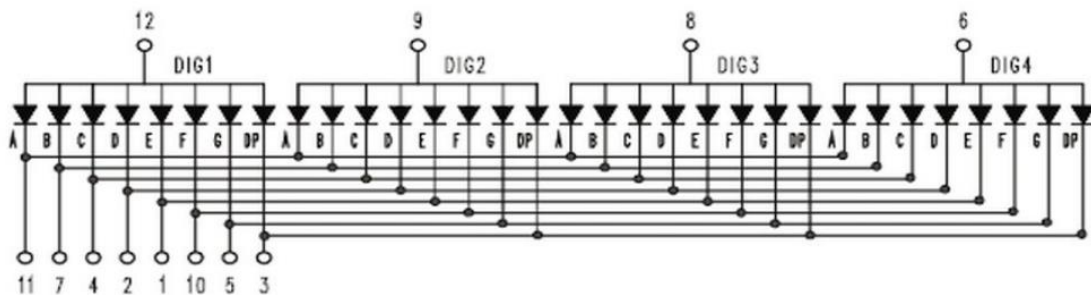
### 5.3 Principle of the experiment

#### 5.3.1 Operating principle of numeric displays

A numeric display is a semiconductor luminescent device, the basic unit of which is the light emitting diode (LED). A numeric display that displays 4 numbers is a 4-bit numeric display. Numeric displays can be divided into 7-segment numeric displays and 8-segment ones, and an 8-segment numeric display has one more light emitting diode (display of one more decimal point) than the 7-segment numeric one. Numeric displays can be also divided into common anode or common cathode numeric displays based on the connection of the LED units. A common anode numeric display is one in which the anodes of all the LEDs are connected together to form a common anode (COM). When a common anode numeric display is used, the common anode (COM) should be connected to +5V. When the cathode of a certain segment of the LED is at low level, the corresponding segment is on. When the cathode of a certain segment of the LED is at high level, the corresponding segment is off. A common cathode numeric display is one in which the cathodes of all the LEDs are connected together to form a common cathode (COM). When a common cathode numeric display is used, the common anode (COM) should be connected to GND. When the anode of a certain segment of the LED is at high level, the corresponding segment is on. When the anode of a certain segment of the LED is at low level, the corresponding segment is off.



The connection of the internal pins of a 4-digit common cathode numeric display is shown in the figure below:



Segment selection: The segment selection consists of 8 led displays: a, b, c, d, e, f, g, and dp.

The segment selection signal controls whether a segment of the numeric display is on.

Digit selection: The digit selection consists of 4 set of 8 segment selection LEDs: seg1, seg2, seg3, and seg4.

The gating signal controls which digit of the numeric display is on.

For example: to get only A at the first digit on, we need to place the pin 11 at high level and pins at other segments (1-5, 7, 10, and 11) at low level; pin 12 at low level and pins at other digits (6, 8, and 9) at high level.

Principle for getting the numeric display on:

Enter the corresponding level to light on each of the segment a-b-c-d-e-f-g-dp. If the numeric display is common cathode, the high level 1 can get it on; on the contrary, if the numeric display is common anode, the low level 0 can get it on. The numeric display of the Runber board is common anode, so 0-9 are displayed with the configuration below:

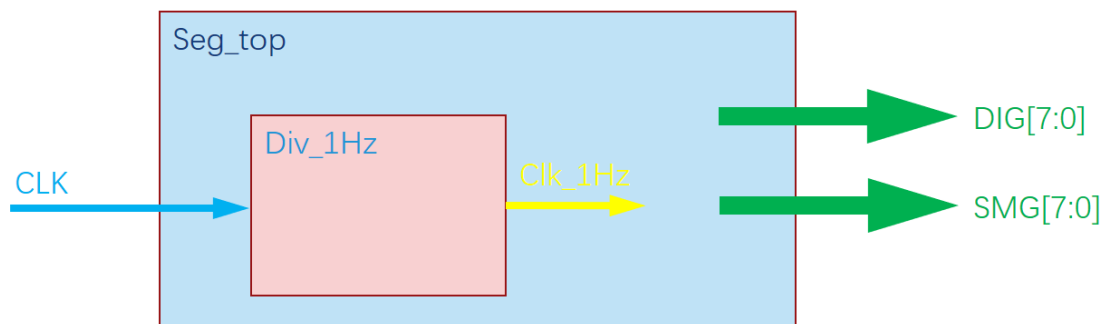
```

1      assign dig = 4'b0000;
2          // 0 1 2 3 4 5 6 7
3          // G F E D C B A P
4      case(counter)
5          4'd0:smg = 8'b0111_1110;/"0"  8'b1000_0001
6          4'd1:smg = 8'b0011_0000;/"1"  8'b1100_1111
7          4'd2:smg = 8'b0110_1101;/"2"  8'b1001_0010
8          4'd3:smg = 8'b0111_1001;/"3"  8'b1000_0110
9          4'd4:smg = 8'b0011_0011;/"4"  8'b1100_1100
10         4'd5:smg = 8'b0101_1011;/"5"  8'b1010_0100
11         4'd6:smg = 8'b0101_1111;/"6"  8'b1010_0000
12         4'd7:smg = 8'b0111_0000;/"7"  8'b1000_1111
13         4'd8:smg = 8'b0111_1111;/"8"  8'b1000_0000
14         4'd9:smg = 8'b0111_1011;/"9"  8'b1000_0100
15         default:smg = 8'b0111_1110;
16     endcase

```

### 5.3.2 Scheme design

1. Switch of display by the numeric display is realized at top level;
2. We need to design a timer of 1s.



For generation of a 1Hz clock please refer to the experiment of twinkle of LEDs. The output of the LED control signals are 8 signals of 1Hz.

#### 5.4 Source code of the experiment

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module top_seq
4  (
5      input          clk, //12MHz 83.33ns
6      output [3:0]   dig,
7      output reg [7:0] smg
8  );
9
10     wire clk_1hz;
11     div_clk u_div_clk
12     (
13         .clk(clk),
14         .clk_1hz(clk_1hz)
15     );
16
17     reg [3:0] counter=0;
18     always @(posedge clk_1hz)
19     begin
20         if(counter==4'd9)
21             counter <= `UD 4'd0;
22         else
23             counter <= `UD counter + 1'b1;
24     end
25
26     assign dig = 4'b0000;
27     always @(*)
28     begin
29         case(counter)
30             4'd0: smg = 8'b0111_1110; // "0" 8'b1000_0001
31             4'd1: smg = 8'b0011_0000; // "1" 8'b1100_1111
32             4'd2: smg = 8'b0110_1101; // "2" 8'b1001_0010
33             4'd3: smg = 8'b0111_1001; // "3" 8'b1000_0110
34             4'd4: smg = 8'b0011_0011; // "4" 8'b1100_1100
35             4'd5: smg = 8'b0101_1011; // "5" 8'b1010_0100
36             4'd6: smg = 8'b0101_1111; // "6" 8'b1010_0000
37             4'd7: smg = 8'b0111_0000; // "7" 8'b1000_1111
38             4'd8: smg = 8'b0111_1111; // "8" 8'b1000_0000
39             4'd9: smg = 8'b0111_1011; // "9" 8'b1000_0100
40             default: smg = 8'b0111_1110;
41         endcase
42     end
43 endmodule

```

#### 5.5 Result of the experiment

After the firmware is burnt, the numbers displayed at the 4 digits of the numeric display are the same, which change periodically every second from 0-9 in turn.

---

## 6 Dynamic Display of Numeric Display

### 6.1 Purpose of the experiment

Dynamic control of the 4-digit numeric display to display different numbers.

### 6.2 Requirements of the experiment

Different numbers are displayed from the 4 numeric displays. The key K0 controls the 1st numeric display, a press on which increases the number by 1 (from 0 to 9); the key K1 controls the 2nd numeric display, a press on which increases the number by 1 (from 0 to 9); the key K2 controls the 3rd numeric display, and the key K3 the 4th numeric display.

### 6.3 Principle of the experiment

From the previous chapters we know that a numeric display consists of several LEDs, the combination of which present different results. In the previous experiment, the 4 numeric displays get on at the same time and display the same number. Now we'll explore how they can display different numbers.

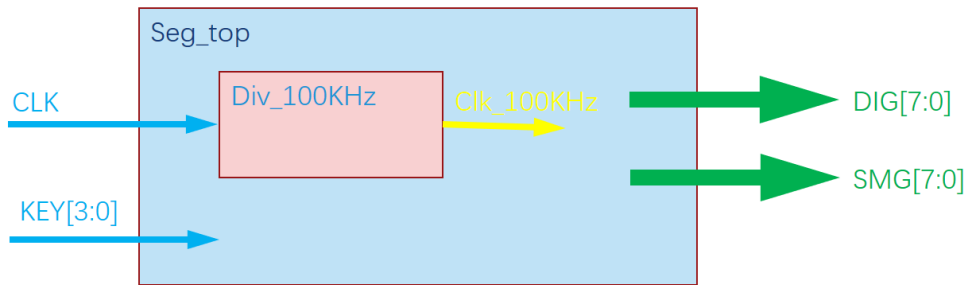
In terms of hardware connection, different numbers cannot be displayed at the same time. But we can get a visual effect of display of different numbers at the same time by refreshing the display. The basis for the aforesaid method is as follows:

The response of the human eye to time frequency is like a filter, which is most sensitive to the signal of 15-20Hz (with strong flicker) in general strong indoor light and not responsive to the signal higher than 75Hz (no flicker). The frequency at which flicker begins to disappear is called critical flicker frequency (CFF). In a dark environment with low pass and low CFF, the human eye is most sensitive to signal of 5Hz and not responsive to signal higher than 25Hz. A cinema is a dark environment, where the refresh rate of 24Hz of the projector does not bring flicker. This characteristic can explain persistence of vision, that is, when the image disappears/changes before us, it stays for a while in our mind rather than disappearing right away.

In design of the display of a numeric display in a mode of flicker, a higher frequency is better for the view of the human eye. But the lighting of the LED bead in the numeric display places a requirement on the high level (it should be noted that the response time of the numeric display on the Runber board is at a level of 0.1us), so the highest frequency is not the best. A proper refresh rate will work, and we make it 100KHz in the experiment.

Scheme design:

1. Key-vibration elimination;
2. Key counting;
3. Time sharing display of numeric display



## 6.4 Source code of the experiment

### 6.4.1 Modules at top level

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module top_seq2
4  (
5      input clk,//50MHZ 20ns
6      input [3:0]button,
7      output reg [3:0]dig,
8      output reg [7:0]smg
9  );
10 /*=====
11      Key-vibration elimination
12 =====*/
13 wire [3:0]key;
14 btn_deb
15 #(
16     .BT_WIDTH(4'd4)
17 )
18 u_btn_deb
19 (
20     .clk(clk),
21     .btn_in(button),
22     .btn_out(key)
23 );
24 /*=====
25      Counting of 4 keys
26 =====*/
27 wire [3:0] key0_cnt;
28 key_cnt key0
29 (
30     .clk(clk),
31     .key(key[0]),
32     .key_times(key0_cnt)
33 );

```

```

34 wire [3:0] key1_cnt;
35 key_cnt key1
36 (
37     .clk(clk),
38     .key(key[1]),
39     .key_times(key1_cnt)
40 );
41
42 wire [3:0] key2_cnt;
43 key_cnt key2
44 (
45     .clk(clk),
46     .key(key[2]),
47     .key_times(key2_cnt)
48 );
49
50 wire [3:0] key3_cnt;
51 key_cnt key3
52 (
53     .clk(clk),
54     .key(key[3]),
55     .key_times(key3_cnt)
56 );
57 /*=====
58                               Clock frequency division
59                               =====*/
60 wire clk_100khz;
61 div_clk div_clk
62 (
63     .clk      (clk),
64     .clk_100khz (clk_100khz)
65 );
66 /*=====
67                               Display by the numeric display
68                               =====*/
69 reg  [1:0]sel=0;
70 wire [3:0]dig0;
71 wire [7:0]smg0;
72
73 always @(posedge clk_100khz)
74 begin
75     sel <= `UD sel+1'b1;
76 end
77
78 seq_control seq_control_0
79 (
80     .sel(2'd3),
81     .key(key0_cnt),
82     .dig(dig0),
83     .smg(smg0)
84 );
85

```



```
86 wire [3:0]dig2;
87 wire [7:0]smg2;
88
89 seq_control seq_control_2
90 (
91     .sel(2'd1),
92     .key(key2_cnt),
93     .dig(dig2),
94     .smg(smg2)
95 );
96
97 wire [3:0]dig3;
98 wire [7:0]smg3;
99
100 seq_control seq_control_3
101 (
102     .sel(2'd0),
103     .key(key3_cnt),
104     .dig(dig3),
105     .smg(smg3)
106 );
107
108 // Select display seq;
109 always @(posedge clk_100khz)
110 begin
111     if(sel==2'b00)
112         dig <= `UD dig0;
113     else if(sel==2'b01)
114         dig <= `UD dig1;
115     else if(sel==2'b10)
116         dig <= `UD dig2;
117     else if(sel==2'b11)
118         dig <= `UD dig3;
119 end
120
121 always @(posedge clk_100khz)
122 begin
123     if(sel==2'b00)
124         smg <= `UD smg0;
125     else if(sel==2'b01)
126         smg <= `UD smg1;
127     else if(sel==2'b10)
128         smg <= `UD smg2;
129     else if(sel==2'b11)
130         smg <= `UD smg3;
131 end
132
133 endmodule
134
```



#### 6.4.2 Key control module

```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module key_cnt
4  (
5      input          clk,
6      input          rstn_key,
7      input          key,
8      output reg [3:0] key_times
9  );
10
11     reg key_reg;
12     always @(posedge clk)
13     begin
14         key_reg <= `UD key; //get key one clock delay value
15     end
16
17     always @(posedge clk )
18     begin
19         if(key_reg&&~key) //falling edge
20         begin
21             if(key_times==4'd9)
22                 key_times <=`UD 4'd0;
23             else
24                 key_times <=`UD key_times + 1'b1;
25         end
26         else
27             key_times <=`UD key_times;
28     end
29
30 endmodule
31
```

#### 6.4.3 Numeric display control module

```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module seq_control
4  (
5      input [1:0]sel,
6      input [3:0]key,
7      output reg [3:0]dig,
8      output reg [7:0]smg
9  );
10
```



```
11  /*=====
12  Digit selection mapping
13  =====*/
14  always @(*)
15  begin
16  case(sel)
17  2'd0:dig = 4'b1110;
18  2'd1:dig = 4'b1101;
19  2'd2:dig = 4'b1011;
20  2'd3:dig = 4'b0111;
21  default:dig = 4'b1111;
22  endcase
23  end
24  //=====
25  // 0 1 2 3 4 5 6 7
26  // G F E D C B A P
27  // Common anode numeric display, where 0 is valid
28  (on)//=====
29  always @(*)
30  begin
31  case(key)
32  4'd0:smg = 8'b0111_1110;/"0" 8'b1000_0001
33  4'd1:smg = 8'b0011_0000;/"1" 8'b1100_1111
34  4'd2:smg = 8'b0110_1101;/"2" 8'b1001_0010
35  4'd3:smg = 8'b0111_1001;/"3" 8'b1000_0110
36  4'd4:smg = 8'b0011_0011;/"4" 8'b1100_1100
37  4'd5:smg = 8'b0101_1011;/"5" 8'b1010_0100
38  4'd6:smg = 8'b0101_1111;/"6" 8'b1010_0000
39  4'd7:smg = 8'b0111_0000;/"7" 8'b1000_1111
40  4'd8:smg = 8'b0111_1111;/"8" 8'b1000_0000
41  4'd9:smg = 8'b0111_1011;/"9" 8'b1000_0100
42  default:smg = 8'b1111_1111;
43  endcase
44  end
45  endmodule
46
47
48
```

## 6.5 Result of the experiment

KEY0-3 control the display of the numeric display from left to right. The key K0 controls the 1st numeric display, a press on which increases the number by 1 (from 0 to 9); the key K1 controls the 2nd numeric display, a press on which increases the number by 1 (from 0 to 9); the key K2 controls the 3rd numeric display, and the key K3 the 4th numeric display.

## 7 UART Serial Port Communication

### 7.1 Purpose of the experiment

Data interaction with PC via serial port using Runber board, and a simple validation.

### 7.2 Requirements of the experiment

The Baut rate in the serial port communication is set as 115200bps, the data format contains 1 start bit, 8 data bits, 0 parity bit and 1 stop bit. The board sends a “===HELLO WORD===” displayed in a decimal format to the serial assistant every second. Numbers in hexadecimal format are sent to the board via the serial assistant, which are displayed with the numeric display, and the LED gets on in hexadecimal format.

### 7.3 Principle of the experiment

#### 7.3.1 Principle of the serial port

From the figure on the right we can see that a standard serial port has 9 lines, with the meaning as follows:

Data line:

TXD (pin 3): Transmit Data

RXD (pin 2): Receive Data

Handshake:

RTS (pin 7): Request to Send

CTS (pin 8): Clear to Send

DSR (pin 6): Data Send Ready

DCD (pin 1): Data Carrier Detect

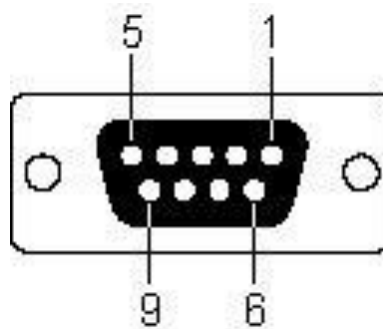
DTR (pin 4): Data Terminal Ready

Ground wire:

GND (pin 5): Ground Wire

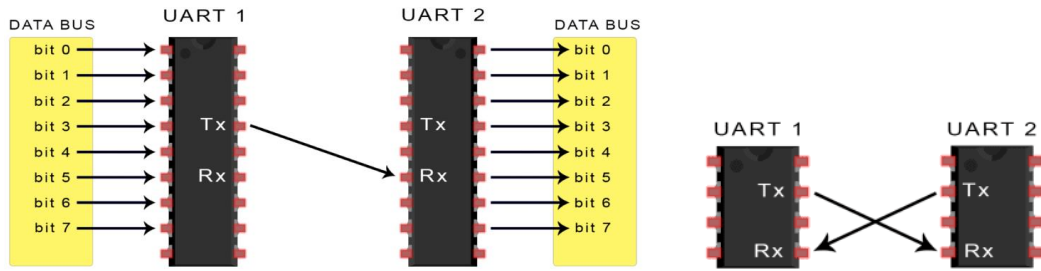
Other

RI (pin 9): Indicate Ring



When using RS232 serial port, we often use only 3 of the 9 transmission lines: TXD, RXD, and GND. For data transfer, however, both sides need to use the same Baut rate and the same mode of transfer (transfer architecture, handshake protocols, etc.) While this method is adequate for most applications, it is limited in the case of overload on part of the receiver.

Connection of RS232 serial port



The serial port transport protocols are as follows:

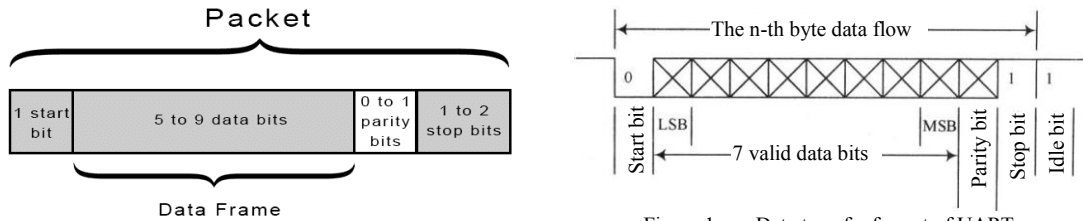


Figure 1 Data transfer format of UART

Start bit: a logic signal “0” is first sent to indicate the start of transmission characters.

Data bits: may be 5-8 bits of logical “0” or “1”, such as the ASCII code (7 bits) and extended BCD code (8 bits).

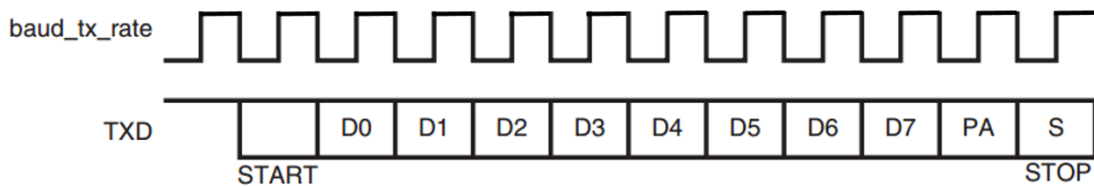
Parity bit: when this bit is added to the data, the number of “1” becomes an even number (even parity) or odd number (odd parity).

Stop bit: the end mark of a character data. It can be a high level of 1 bit, 1.5 bits or 2 bits.

Idle bit: a bit in the logic state of “1”, indicating no data is transferred along the current line.

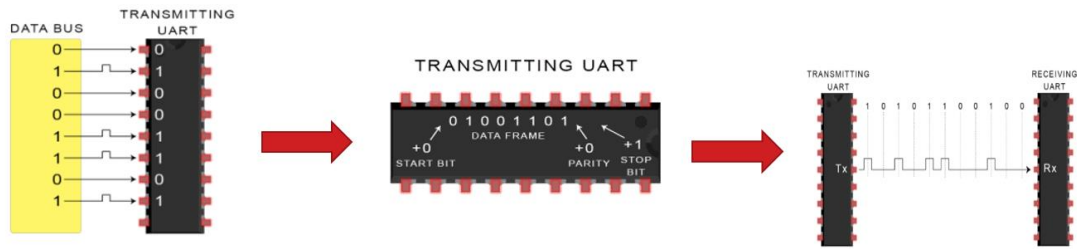
Baut rate: the Baut rate in UART can be considered bit rate, the bits transferred per second. There are generally a choice of Baut rate of 9600, 19200, 115200, etc. It means how many bits are transferred per second.

The introduction of Baut rate leads to the following process of transmission of the serial port:

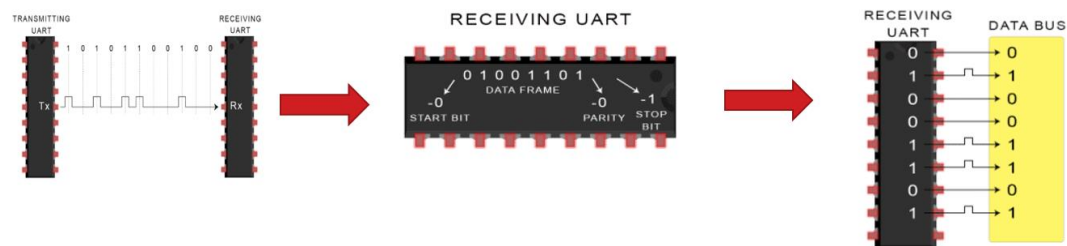


### 7.3.2 Steps of serial port transmission

#### 7.3.2.1 Process of serial port sending



#### 7.3.2.2 Process of serial port receiving



### 7.3.3 Characters sent via serial port

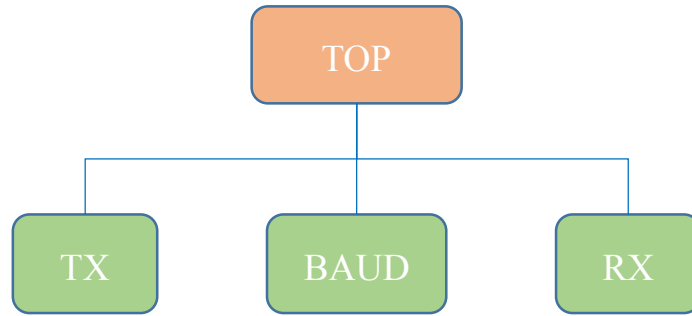
From the above serial Port protocol we see that 5-8 bits of data can be transferred each time via serial port. Characters are generally expressed with ASCII code (7 bits) in computers, so they can be sent in form of ASCII code.

We can find the ASCII codes of the characters used in “==HELLO WORD==” by looking up the ASCII table:

Character	ASCII code (hex)	Character	ASCII code (hex)	Character	ASCII code (hex)	Character	ASCII code (hex)
=	3D	H	48	E	45	L	4C
O	4F	W	57	R	52	D	44
Carriage Return	0D	Line Feed	0A	Space	20		

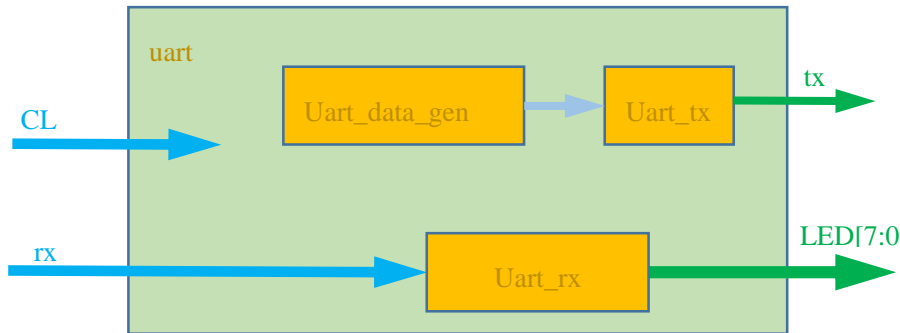
### 7.4 Design of the source code of the experiment

The experiment can be divided as follows based on analysis of its purpose:



In term of the principle, the calculation of Baut rate can be seen as a counter, the sending and receiving of which is reusable. In our design, to maintain the integrity of TX or RX we integrate the Baut cycle counter in the respective module.

The above analysis only helps us build the bridge for communication between RUNBER and PC (UART), but does not involve the data transmission. So we need to add the modules to send and receive data.



You should integrate the numeric display after class. If you do not understand that part very well, please refer to chapters 5 and 6.

#### 7.4.1 Design of serial port transmission module

Objective: upon receiving a signal of SendCommand, send data[7:0] -> in turn : {start,data[0:7],stop} (data of 10 bits in total: no parity bit, 1 stop bit).

There are 2 methods to serialize the parallel data.

Method 1: control of shift register output via bit counting and baud counting.



```
1 // transmit bit
2 always@(posedge clk)
3 begin
4     if(!rstn)
5         txd <= `UD 1'b1;
6     else
7         begin
8             if(trans_en)
9                 Begin
10 // The statement below can be used to integrate the start mark and stop mark with the
11 // transmission data
12 // The statement below can be used for bit control only
13                 case(trans_bit)
14                     4'h0 :txd <= `UD 1'b0;
15                     4'h1 :txd <= `UD tx_data_reg[0];
16                     4'h2 :txd <= `UD tx_data_reg[1];
17                     4'h3 :txd <= `UD tx_data_reg[2];
18                     4'h4 :txd <= `UD tx_data_reg[3];
19                     4'h5 :txd <= `UD tx_data_reg[4];
20                     4'h6 :txd <= `UD tx_data_reg[5];
21                     4'h7 :txd <= `UD tx_data_reg[6];
22                     4'h8 :txd <= `UD tx_data_reg[7];
23                     4'h9 :txd <= `UD 1'b1;
24                     default :txd <= `UD 1'b1;
25                 endcase
26             end
27         else
28             txd <= `UD 1'b1;
29         end
30     end
31
```

Method 2: control of state change via bit counting and baud counting, and output from state machine.



```

1 // logical ouput Output from state machine
2 always @ (posedge clk)
3 begin
4     if(tx_en)
5         begin
6             case(tx_state)
7                 IDLE      : uart_tx  <= `UD 1'h1; // Output of high level in idle state
8                 SEND_START : uart_tx  <= `UD 1'h0; // Sending of low level of a Baud cycle in start state
9                 SEND_DATA  :          // Send a bit every Baud cycle in send state;
10                begin
11                    case(tx_bit_cnt)
12                        3'h0 : uart_tx  <= `UD trans_data[0];
13                        3'h1 : uart_tx  <= `UD trans_data[1];
14                        3'h2 : uart_tx  <= `UD trans_data[2];
15                        3'h3 : uart_tx  <= `UD trans_data[3];
16                        3'h4 : uart_tx  <= `UD trans_data[4];
17                        3'h5 : uart_tx  <= `UD trans_data[5];
18                        3'h6 : uart_tx  <= `UD trans_data[6];
19                        3'h7 : uart_tx  <= `UD trans_data[7];
20                        default: uart_tx  <= `UD 1'h1;
21                    endcase
22                end
23                SEND_STOP  : uart_tx  <= `UD 1'h1; // Sending of stop state, and output of high level of a Baud
cycle
24                default   : uart_tx  <= `UD 1'h1; // Other states are the same as the idle state by default, where
output of high level maintained
25            endcase
26        end
27    else
28        uart_tx <= `UD 1'h1;
29    end
30

```

The module of method 1 is as follows:

```

1 `timescale 1ns / 1ps
2 `define UD #1
3 module uart_tx #(
4     parameter BAUND_RATE_CNT = 12'd1250
5     //115200 : 12MHz, 12000000/115200 = 10'd104
6     //9600  :      12000000/9600 = 11'd1250
7 )
8 (
9     input      clk,
10    input      rstn,
11    input      trig, // active posedge
12
13    input [7:0] tx_data,
14
15    output reg  txd,
16    output      tx_busy
17 );
18

```

```

19 //=====
20 // baud rate set
21 reg [11:0] baud_cnt;
22 always @(posedge clk)
23 begin
24     if(!rstn)
25         baud_cnt <= `UD 12'd0;
26     else
27         begin
28             if(baud_cnt == BAUND_RATE_CNT - 1'b1)
29                 baud_cnt <= `UD 12'd0;
30             else
31                 baud_cnt <= `UD baud_cnt + 12'd1;
32         end
33     end
34
35     wire baud_over = (baud_cnt == BAUND_RATE_CNT - 1'b1) ? 1'b1 : 1'b0;
36
37 //=====
38 //transmit start
39 reg trig_1d;
40 reg [7:0] tx_data_reg;
41 always @(posedge clk)
42 begin
43     trig_1d <= `UD trig;
44     end
45
46     reg start_en;
47     wire start;
48     always @(posedge clk)
49     begin
50         if(!rstn)
51             start_en <= `UD 1'b0;
52         else if(~trig_1d & trig & ~start_en)
53             start_en <= `UD 1'b1;
54         else if(baud_over)
55             start_en <= `UD 1'b0;
56     end
57     assign start = ~trig_1d & trig; //start_en & baud_over;
58
59     // Latch the data when sending is triggered
60     always @(posedge clk)
61     begin
62         if(!rstn)
63             tx_data_reg <= `UD 8'h3f;
64         else if(~trig_1d & trig)
65             tx_data_reg <= `UD tx_data;
66     end
67

```



```
68 //=====  
69 // trasmit data  Latch the start stop data in a latch.  
70     reg [9:0] trans_data;  
71  
72     always @(posedge clk)  
73     begin  
74         if(!rstn)  
75             trans_data <= `UD 10'h3f;  
76         else if(~trig_1d & trig)  
77             trans_data <= `UD {1'b1,tx_data,1'b0};  
78     end  
79  
80 //=====  
81 // transmit control  
82     reg         trans_en;  
83     reg [3:0] trans_bit;  
84     always @(posedge clk)  
85     begin  
86         if(!rstn)  
87             trans_en <= `UD 1'b0;  
88         else if(~trig_1d & trig)  
89             trans_en <= `UD 1'b1;  
90         else if(trans_bit == 4'd9 && baund_over)  
91             trans_en <= `UD 1'b0;  
92         else  
93             trans_en <= `UD trans_en;  
94     end  
95  
96     assign tx_busy = ~trans_en;  
97  
98     always @(posedge clk)  
99     begin  
100        if(!rstn)  
101            trans_bit <= `UD 4'd0;  
102        else  
103            begin  
104                if(trans_en && baund_over)  
105                    begin  
106                        if(trans_bit == 4'd9)  
107                            trans_bit <= `UD 4'd0;  
108                        else  
109                            trans_bit <= `UD trans_bit + 4'd1;  
110                    end  
111                else if(!trans_en)  
112                    trans_bit <= `UD 4'd0;  
113            end  
114        end  
115
```

```
116 //=====
117 // transmit bit
118     always@(posedge clk)
119     begin
120         if(!rstn)
121             txd <= `UD 1'b1;
122         else
123             begin
124                 if(trans_en)
125                     begin
126 // The statement below can be used to integrate the start mark and stop mark with the transmission
127 // data
128 // The statement below can be used for bit control only
129                         case(trans_bit)
130                             4'h0 :txd <= `UD 1'b0;
131                             4'h1 :txd <= `UD tx_data_reg[0];
132                             4'h2 :txd <= `UD tx_data_reg[1];
133                             4'h3 :txd <= `UD tx_data_reg[2];
134                             4'h4 :txd <= `UD tx_data_reg[3];
135                             4'h5 :txd <= `UD tx_data_reg[4];
136                             4'h6 :txd <= `UD tx_data_reg[5];
137                             4'h7 :txd <= `UD tx_data_reg[6];
138                             4'h8 :txd <= `UD tx_data_reg[7];
139                             4'h9 :txd <= `UD 1'b1;
140                             default :txd <= `UD 1'b1;
141                         endcase
142                     end
143                 else
144                     txd <= `UD 1'b1;
145             end
146     end
147
```

The module design of method 2 is as follows:

```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module uart_tx #(
4      parameter          BPS_NUM = 16'd434
5      // When Baud rate is set as 4800, the bit width is the number of clock cycles: 50MHz set 10417 40MHz set 8333
6      // When Baud rate is set as 9600, the bit width is the number of clock cycles: 50MHz set 5208 40MHz set 4167
7      // When Baud rate is set as 115200, the bit width is the number of clock cycles: 50MHz set 434 40MHz set 347
8      // 12M set 104
9  )
10 (
11     input          clk,          // clock                      Clock signal
12     input [7:0]    tx_data,      // uart tx data signal byte;          Byte data to be transmitted
13     input          tx_pluse,     // uart tx enable signal,rising is active; Sending module sends trigger signal
14     output reg     uart_tx,     // uart tx transmit data line          Serial port transmitting signal line of
15     // sending module
16     output         tx_busy      // uart tx module work states,high is busy; State indication of busy sending
17     // module
18 );
```



```
18 //=====
19 //wire and reg in the module
20 //=====
21 reg          tx_pluse_reg =0;
22 reg [7:0]    trans_data=0;
23
24 reg [2:0]    tx_bit_cnt=0; //the bits number has transmitted.
25
26 reg [2:0]    tx_state=0; //current state of tx state machine.
27 reg [2:0]    tx_state_n=0; //next state of tx state machine.
28
29 reg [3:0]    pluse_delay_cnt=0;
30 reg          tx_en = 0;
31
32 // uart tx state machine's state
33 localparam IDLE      = 4'h0; //tx state machine's state. Idle state
34 localparam SEND_START = 4'h1; //tx state machine's state. Send start state
35 localparam SEND_DATA  = 4'h2; //tx state machine's state. Send data state
36 localparam SEND_STOP  = 4'h3; //tx state machine's state. Send stop state
37 localparam SEND_END   = 4'h4; //tx state machine's state. Send stop state
38
39 // uart bps set the clk's frequency is 50MHZ
40 reg [15:0]  clk_div_cnt=0; //count for division the clock.
41
42 //=====
43 //logic
44 //=====
45 assign tx_busy = (tx_state != IDLE);
46 //some control single.
47
48 always @(posedge clk)
49 begin
50     tx_pluse_reg <= `UD tx_pluse;
51 end
52
53 // uart Latch data to be transmitted
54
55 always @(posedge clk)
56 begin
57     if(~tx_pluse_reg & tx_pluse)
58         trans_data <= `UD tx_data;
59 end
60
61 // uart Marking signal of module sending enabled
62 always @(posedge clk)
63 begin
64     if(~tx_pluse_reg & tx_pluse)
65         tx_en <= `UD 1'b1;
66     else if(tx_state == SEND_END)
67         tx_en <= `UD 1'b0;
68 end
69
```



```
70 //division the clock to satisfy baud rate. Baut cycle counter
71 always @ (posedge clk)
72 begin
73     if(clk_div_cnt == BPS_NUM || (~tx_pluse_reg & tx_pluse))
74         clk_div_cnt <= `UD 16'h0;
75     else
76         clk_div_cnt <= `UD clk_div_cnt + 16'h1;
77 end
78
79 //count the number has transmitted. Counting of bits transmitted in send data state, sum of Baut cycles
80 always @ (posedge clk)
81 begin
82     if(!tx_en)
83         tx_bit_cnt <= `UD 3'h0;
84     else if((tx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
85         tx_bit_cnt <= `UD 3'h0;
86     else if((tx_state == SEND_DATA) && (clk_div_cnt == BPS_NUM))
87         tx_bit_cnt <= `UD tx_bit_cnt + 3'h1;
88     else
89         tx_bit_cnt <= `UD tx_bit_cnt;
90 end
91
92
93 //transmitter state machine
94
95 // state change State change
96 always @(posedge clk)
97 begin
98     tx_state <= tx_state_n;
99 end
100
101 // state change condition Condition and rule of state change
102 always @ (*)
103 begin
104     case(tx_state)
105     IDLE :
106     begin
107         if(~tx_pluse_reg & tx_pluse) // Trigger send changes to send start state after 16 clock delays
108             tx_state_n = SEND_START;
109         else
110             tx_state_n = tx_state;
111     end
112     SEND_START :
113     begin
114         if(clk_div_cnt == BPS_NUM) // Enter send data state after sending of low level of a Baut
115             tx_state_n = SEND_DATA;
116         else
117             tx_state_n = tx_state;
118     end
```

```

119     SEND_DATA    :
120     begin
121         if(tx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM)
122             // Change to send stop state after 8 Baud cycles (8 bits of data sent)
123             tx_state_n = SEND_STOP;
124         else
125             tx_state_n = tx_state;
126         end
127     SEND_STOP    :
128     begin
129         if(clk_div_cnt == BPS_NUM)
130             // Stop bit is set as a Baud cycle. Change to send stop state after sending of high level of a Baud cycle
131             tx_state_n = SEND_END;
132         else
133             tx_state_n = tx_state;
134         end
135     SEND_END     : tx_state_n = IDLE;
136     default      : tx_state_n = IDLE;
137 endcase
138 end
139
140 // logical output - Output from state machine
141 always @ (posedge clk)
142 begin
143     if(tx_en)
144     begin
145         case(tx_state)
146             IDLE      : uart_tx <= `UD 1'h1; // Output of high level in idle state
147             SEND_START : uart_tx <= `UD 1'h0; // Sending of low level of a Baud cycle in start state
148             SEND_DATA  : // Send a bit every Baud cycle in
149 send state;
150         begin
151             case(tx_bit_cnt)
152                 3'h0 : uart_tx <= `UD trans_data[0];
153                 3'h1 : uart_tx <= `UD trans_data[1];
154                 3'h2 : uart_tx <= `UD trans_data[2];
155                 3'h3 : uart_tx <= `UD trans_data[3];
156                 3'h4 : uart_tx <= `UD trans_data[4];
157                 3'h5 : uart_tx <= `UD trans_data[5];
158                 3'h6 : uart_tx <= `UD trans_data[6];
159                 3'h7 : uart_tx <= `UD trans_data[7];
160             default: uart_tx <= `UD 1'h1;
161             endcase
162         end
163         SEND_STOP : uart_tx <= `UD 1'h1; // Output of high level of a Baud cycle in send stop state
164         default   : uart_tx <= `UD 1'h1;
165         // Other states are the same as the idle state by default, where output of high level
166 maintained
167     endcase
168     end
169     else
170     uart_tx <= `UD 1'h1;
171     end
172 endmodule

```



#### 7.4.2 Design of serial port receiving module

Serial port receiving module is the reverse process of the transmitting module, with similar approach to design. However, attention should be paid to the following:

1. Receive start signal: maintenance of low level of several clock cycles after rx falling edge comes, indicating the state of receive start.
2. Received data extraction location: in the above description of data transmission, data is changed when the Baut cycle begins. Extraction of received data needs to be conducted at a time when rx is stable (the middle of a Baut cycle).
3. Latching of final output data: received data should be latched when the last bit is stored in the register, and then a data enable signal should be sent, indicating the validity of the output data.

The module design is as follows:

```

1  `timescale 1ns / 1ps
2  `define UD #1
3
4  module uart_rx # (
5      parameter          BPS_NUM      =    16'd433
6      // When Baut rate is set as 4800, the bit width is the number of clock cycles: 50MHz set 10417
7      // 40MHz set 8333
8      // When Baut rate is set as 9600, the bit width is the number of clock cycles: 50MHz set 5208
9      // 40MHz set 4167
10     // When Baut rate is set as 115200, the bit width is the number of clock cycles: 50MHz set 434
11     // 40MHz set 347
12 )
13 (
14     //input ports
15     input          clk,
16     input          rstn,
17     input          uart_rx,
18
19     //output ports
20     output reg [7:0] rx_data,
21     output reg      rx_en,
22     output          rx_finish
23 );
24
25 // uart rx state machine's state
26 localparam IDLE          = 4'h0; // Idle state, waiting for coming of the start signal.
27 localparam RECEIV_START = 4'h1; // Receive Uart start signal, low level for a Baut
28 cycle.
29 localparam RECEIV_DATA  = 4'h2; // Receive Uart data transmission signal,
30 localparam RECEIV_STOP  = 4'h3; // Data line at high level in stop state,
31 localparam RECEIV_END   = 4'h4; // End of transfer state.
32
33 //=====
34 =
35 //wire and reg in the module
36 //=====
37 =
38 reg    [2:0]    rx_state=0; //current state of tx state machine. Current state
39 reg    [2:0]    rx_state_n=0;//next state of tx state machine. Next state
40 reg    [7:0]    rx_data_reg; // Buffer register for received data
41 reg          uart_rx_1d; //save uart_rx one cycle. Save uart_rx for a clock cycle
42 reg          uart_rx_2d; //save uart_rx one cycle. Save uart_rx for the first 2 clock cycles
43 wire          start; //active when start a byte receive. Start signal marker detected
44 reg    [15:0]   clk_div_cnt; //count for division the clock. Baut cycle counter

```



```
40
//=====
41 //logic
42 //=====
43 //some control single.
44 always @ (posedge clk)
45 begin
46     uart_rx_1d <= `UD uart_rx;
47     uart_rx_2d <= `UD  uart_rx_1d;
48 end
49
50 assign start      = (!uart_rx) && (uart_rx_1d || uart_rx_2d);
51 assign rx_finish = (rx_state == RECEIV_END);
52
53 //division the clock to satisfy baud rate. Baut cycle counter
54 always @ (posedge clk)
55 begin
56     if(rx_state == IDLE || clk_div_cnt == BPS_NUM)
57         clk_div_cnt  <= `UD 16'h0;
58     else
59         clk_div_cnt  <= `UD clk_div_cnt + 16'h1;
60 end
61
62 // receive bit data numbers
63 // Counting of bits received in receive data state, increasing by 1 every Baut cycle
64 reg  [2:0]    rx_bit_cnt=0; //the bits number has transmitted.
65 always @ (posedge clk)
66 begin
67     if(rx_state == IDLE)
68         rx_bit_cnt <= `UD 3'h0;
69     else if((rx_bit_cnt == 3'h7) && (clk_div_cnt == BPS_NUM))
70         rx_bit_cnt <= `UD 3'h0;
71     else if((rx_state == RECEIV_DATA) && (clk_div_cnt == BPS_NUM))
72         rx_bit_cnt <= `UD rx_bit_cnt + 3'h1;
73     else
74         rx_bit_cnt <= `UD rx_bit_cnt;
75 end
76 //=====
77 //receive state machine
78 //=====
79 // State change of the state machine
80 always @(posedge clk)
81 begin
82     rx_state <= rx_state_n;
83 end
84
85 // Condition and rule of state change of the state machine
86 always @ (*)
87 begin
88     case(rx_state)
89         IDLE      :
90             begin
91                 if(start) // Start signal detected, and change to start state next
92                     rx_state_n = RECEIV_START;
93             else
94                 rx_state_n = rx_state;
95             end
96     end
```

```

96     RECEIV_START    :
97     begin
98         if(clk_div_cnt == BPS_NUM)           // Receiving of start marking signal
completed
99             rx_state_n = RECEIV_DATA;
100        else
101            rx_state_n = rx_state;
102        end
103     RECEIV_DATA    :
104     begin
105         if(rx_bit_cnt == 3'h7 && clk_div_cnt == BPS_NUM)
106             // Transmission of 8-bit data completed
107             rx_state_n = RECEIV_STOP;
108         else
109             rx_state_n = rx_state;
110         end
111     RECEIV_STOP    :
112     begin
113         if(clk_div_cnt == BPS_NUM) // Receiving of stop marking signal completed
114             rx_state_n = RECEIV_END;
115         else
116             rx_state_n = rx_state;
117         end
118     RECEIV_END    :
119     begin
120         if(!uart_rx_1d)
121             // The data line is pulled down again, meaning start marking signal is sent again for new
data transmission and change to start state is needed
122             rx_state_n = RECEIV_START;
123         else
124             // Return to idle state when nothing else happens, waiting for start signal
125             rx_state_n = IDLE;
126         end
127     default        : rx_state_n = IDLE;
128 endcase
129 end
130
131 // Output from state machine
132 always @ (posedge clk)
133 begin
134     case(rx_state)
135         IDLE        ,
136         RECEIV_START :
137             // Setting of buffer register of received data and the data enable in idle and start
states;
138         begin
139             rx_en <= `UD 1'b0;
140             rx_data_reg <= `UD 8'h0;
141         end
142         RECEIV_DATA :
143         begin
144             if(clk_div_cnt == BPS_NUM[15:1])
145                 // Extraction of data transmitted via data line at the middle of a Baud
cycle;
146                 rx_data_reg <= `UD {uart_rx , rx_data_reg[7:1]}; //以 Fill uart_rx data
into the highest bit of the buffer register via ring shift right (the low bit is in front in Uart transmission, and the last bit
is the highest)
147         end

```

```

148     RECEIV_STOP  :
149     begin
150         rx_en    <= `UD 1'b1;      // Output of enable signal, indicating the validity
of the latest data output
151         rx_data <= `UD rx_data_reg; // Assign the value of buffer register to output
register
152     end
153     RECEIV_END   :
154     begin
155         rx_data_reg <= `UD 8'h0;
156     end
157     default:    rx_en <= `UD 1'b0;
158 endcase
159 end
160
161 endmodule
162

```

### 7.4.3 Design of serial port transmission control module

Objective: generation of trigger signal with an interval of 1s and output of the first byte, and output of the next byte at busy falling edge.

The module is as follows:

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module uart_data_gen(
4      input          clk,
5      input          rstn,
6      input [7:0]    read_data,
7      input          tx_busy,
8      input [7:0]    write_max_num,
9      output reg [7:0] write_data,
10     output reg      write_en
11 );
12     // set every second send a string,"====HELLO WORLD===="
13     // Setting of sending a string about every second
14     reg [23:0] time_cnt=0;
15     reg [ 7:0] data_num;
16     always @(posedge clk)
17     begin
18         time_cnt <= `UD time_cnt + 24'd1;
19     end
20
21     // Set workspace of serial port transmission
22     reg      work_en=0;
23     reg      work_en_1d=0;
24     always @(posedge clk)
25     begin
26         if(time_cnt == 25'd2048)
27             work_en <= `UD 1'b1;
28         else if(data_num == write_max_num-1'b1)
29             work_en <= `UD 1'b0;
30     end

```



```
32     always @(posedge clk)
33     begin
34         work_en_1d <= `UD work_en;
35     end
36
37     // get the tx_busy's falling edge    Get falling edge of tx_busy
38     reg          tx_busy_reg=0;
39     wire         tx_busy_f;
40     always @ (posedge clk) tx_busy_reg <= `UD tx_busy;
41
42     assign tx_busy_f = (!tx_busy) && (tx_busy_reg);
43
44     // Trigger signal of serial port transmission data
45     reg write_pluse;
46     always @ (posedge clk)
47     begin
48         if(!rstn)
49             write_pluse <= `UD 1'b0;
50         else if(work_en)
51             begin
52                 if(~work_en_1d || tx_busy_f)
53                     write_pluse <= `UD 1'b1;
54                 else
55                     write_pluse <= `UD 1'b0;
56             end
57         else
58             write_pluse <= `UD 1'b0;
59     end
60
61     always @ (posedge clk)
62     begin
63         if(!rstn)
64             data_num    <= `UD 8'h0;
65         else if(~work_en & tx_busy_f)
66             data_num    <= 7'h0;
67         else if(write_pluse)
68             data_num    <= data_num + 8'h1;
69     end
70
71     always @(posedge clk)
72     begin
73         write_en <= `UD write_pluse;
74     end
75
76     // Corresponding ASCII of characters
77     // H:0x48    E:0x45    L:0x4C    O:0x4F    W:0x57    R:0x52    D:0x44
78     always @ (posedge clk)
79     begin
80         case(data_num)
81             8'h0    ,
82             8'h1    ,
83             8'h2    ,
84             8'h3    : write_data <= `UD 8'h3D; // ASCII code is =
85             8'h4    : write_data <= `UD 8'h48; // ASCII code is H
86             8'h5    : write_data <= `UD 8'h45; // ASCII code is E
```

```
87      8'h6 : write_data <= `UD 8'h4C;// ASCII code is L
88      8'h7 : write_data <= `UD 8'h4C;// ASCII code is L
89      8'h8 : write_data <= `UD 8'h4F;// ASCII code is O
90      8'h9 : write_data <= `UD 8'h20;// ASCII code is
91      8'ha : write_data <= `UD 8'h57;// ASCII code is W
92      8'hb : write_data <= `UD 8'h4F;// ASCII code is O
93      8'hc : write_data <= `UD 8'h52;// ASCII code is R
94      8'hd : write_data <= `UD 8'h4C;// ASCII code is L
95      8'he : write_data <= `UD 8'h44;// ASCII code is D
96      8'hf ,
97      8'h10 ,
98      8'h11 : write_data <= `UD 8'h3D;// ASCII code is =
99      8'h12 : write_data <= `UD 8'h0d;
100     8'h13 : write_data <= `UD 8'h0a;
101     default : write_data <= `UD read_data;
102     endcase
103 end
104
105 endmodule
106
```

#### 7.4.4 Design of top level module of serial port experiment

Objective: The board sends a “====HELLO WORD====” displayed in a decimal format to the serial assistant every second. Numbers in hexadecimal format are sent to the board via the serial assistant, and the LED gets on in hexadecimal format.

Uart\_data\_gen module generates a trigger signal with an interval of 1s and outputs the first byte to be sent at the same time, waiting for busy falling edge output from uart\_tx. When learning uart\_tx enters idle state and the next byte can be sent, it sends trigger pulse for serial port transmission and outputs the next byte.

After receiving the data, Uart\_rx module outputs an rx\_en signal (Receive Data Enable signal) and a set of Receive Data signals. The received data signal is latched, which can light on the LED directly.

The module is realized as follows:

```
1  `timescale 1ns / 1ps
2  `define UD #1
3
4  module uart_top(
5      //input ports
6      input clk,
7      input rstn,
8      input uart_rx,
9
10     //output ports
11     output [7:0] led,
12     output uart_tx
13 );
14
```

```

15 parameter BPS_NUM = 16'd104;
16 // When Baut rate is set as 4800, the bit width is the number of clock cycles: 50MHz set 10417 40MHz set 8333
17 // When Baut rate is set as 9600, the bit width is the number of clock cycles: 50MHz set 5208 40MHz set 4167
18 // When Baut rate is set as 115200, the bit width is the number of clock cycles:50MHz set 434 40MHz set 347.12M set 104
19 //=====
20 //wire and reg in the module
21 //=====
22 wire tx_busy; //transmitter is free.
23 wire rx_finish; //receiver is free.
24 wire [7:0] rx_data; //the data receive from uart_rx.
25 wire [7:0] tx_data;
26 wire tx_en; //enable transmit.
27 wire rx_en;
28 //=====
29 //instance
30 //=====
31 reg [7:0] receive_data;
32 always @(posedge clk) receive_data <= led;
33 uart_data_gen uart_data_gen(
34 .clk ( clk ),//input clk,
35 .rstn ( rstn ),//input rstn,
36 .read_data ( receive_data ),//input [7:0] read_data,
37 .tx_busy ( tx_busy ),//input tx_busy,
38 .write_max_num ( 8'h14 ),//input [7:0] write_max_num,
39 .write_data ( tx_data ),//output reg [7:0] write_data
40 .write_en ( tx_en )//output reg write_en
41 );
42
43 //uart transmit data module.
44 uart_tx #(
45 .BPS_NUM ( BPS_NUM )//parameter BPS_NUM = 16'd434
46 )
47 u_uart_tx(
48 .clk ( clk ),// input clk,
49 .tx_data ( tx_data ),// input [7:0] tx_data,
50 .tx_pluse ( tx_en ),// input tx_pluse,
51 .uart_tx ( uart_tx ),// output reg uart_tx,
52 .tx_busy ( tx_busy )// output tx_busy
53 );
54
55 //Uart receive data module.
56 uart_rx #(
57 .BPS_NUM ( BPS_NUM )//parameter BPS_NUM = 16'd434
58 )
59 u_uart_rx (
60 .clk ( clk ),// input clk,
61 .rstn ( rstn ),// input rstn,
62 .uart_rx ( uart_rx ),// input uart_rx,
63 .rx_data ( rx_data ),// output reg [7:0] rx_data,
64 .rx_en ( rx_en ),// output reg rx_en,
65 .rx_finish ( rx_finish )// output rx_finish
66 );
67 assign led = rx_data;
68
69 endmodule
70

```



## 7.5 Result of the experiment

When serial port configuration is set via SSCOM and the corresponding serial port is connected, the following results can be found:

1. “==HELLO WORD==” and a CRLF is printed on the serial port every second.
2. When 55 is sent in Hex format on the serial port tool, we can see D1, D3, D5, and D7 on the RUNBER board get on and D2, D4, D6 and D8 are off. When AA is sent, we can see D2, D4, D6 and D8 on the RUNBER board get on and D1, D3, D5 and D7 are off.
3. Numeric display serial port receives data (Please realize this result after class, and carry out the extended exercise on module instantiation and module connection).

---

## 8 Sequence Detector

### 8.1 Purpose of the experiment

To detect whether a particular sequence exists in a continuous signal (for example, whether “11011000” contains “101”).

### 8.2 Requirements of the experiment

1. DIP switches SW0-SW7 are the input of continuous signal.
2. KEY1-KEY3 are the particular signal input sequence. When the KEY is pressed down the corresponding LED gets on, indicating the corresponding bit is 1; the LED gets off when the KEY is pressed down again, indicating the corresponding bit is 0.
3. K8 is the key for start and end of the sequence detection. The detection starts when KEY8 is pressed down for the first time, and LED8 gets on at the same time to indicate the current state. When KEY8 is pressed down again, the detection stops and LED8 gets off. After the end of the detection, the number of times the particular sequence appears in the string is displayed on the numeric display.

### 8.3 Principle of the experiment

The states of SW0-SW7 are the detection sequences;

LED1-LED3 are the particular sequences;

The result displayed on the numeric display is the number of times LED[3:1] appears in SW[7:0].

### 8.4 Design of the source code of the experiment

#### 8.4.1 Scheme design

3 functional modules are needed based on analysis of the purpose of the experiment.

##### 1. Key LED module

The keys adjust the particular sequences: KEY[2:0] controls the values of the particular sequences; KEY8 controls whether the detection is carried out. The particular sequences are displayed and saved as output via LED, which at the same time sends the particular sequence and detection enable signal to the detection module.

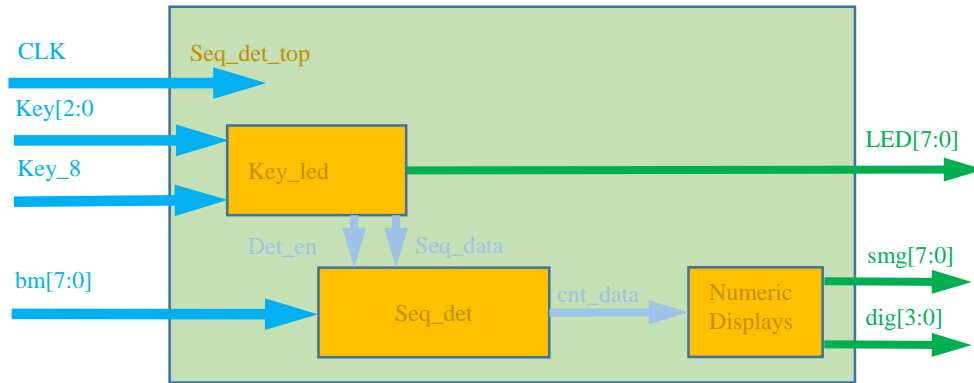
##### 2. Sequence comparison module:

The DIP switch provides the sequence to be detected, receives the particular sequence and detection enable signal sent from the key control module and compares it with the sequence to be detected. The result of the comparison is output to the numeric display module for display.

##### 3. Numeric display control module

The objective of the numeric display module is to display the result of the statistics. Display with a dynamic numeric display is adopted.

Connection lines between the modules are shown in the block diagram below:



### 8.4.2 Design of top level modules (including the numeric display module)

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module top_seq_det
4  (
5      input      clk,          // input clock
6      input      key_8,        // KEY 8 input control detected enable
7      input [2:0] key_in,      // KEY[2:0] input control detected sequence
8      input [7:0] bm,         // DIP Switch[7:0] input used to detecting
9      output     key8_led,     // display the detecte status
10     output [6:0] key_in_led, // display the detected sequence
11     output reg [3:0] dig,     // output Digital tube Bit selection
12     output reg [7:0] smg     // output Digital tube segment selection
13 );
14
15 /*=====
16                      Key-vibration elimination and state marking
17 =====*/
18 wire [2:0]seq_data;
19 key_control key_control
20 (
21     .clk      ( clk      ),
22     .key_8    ( key_8    ),
23     .key_in   ( key_in   ),
24     .key8_led ( key8_led ),
25     .key_in_led ( key_in_led[2:0] ),
26     .seq_data ( seq_data )
27 );
28
29 assign key_in_led[6:3] = 4'd0;
30
31 /*=====
32                      Sequence detection
33 =====*/
34 wire [3:0]data;
35

```

```

36 seq_det seq_det
37 (
38     .clk          ( clk          ),
39     .key8_led     ( key8_led),// Detection state marking
40     .key_in_led   ( seq_data ),// Sequence to be detected
41     .bm           ( bm           ),// Input sequence
42     .data         ( data         )
43 );
44 /*=====
45                      Clock frequency division
46 =====*/
47 wire clk_100khz;
48 div_clk div_clk
49 (
50     .clk          ( clk          ),
51     .clk_100khz  ( clk_100khz  )
52 );
53 /*=====
54                      Display on the numeric display
55 =====*/
56 reg [1:0]sel=0;
57 wire [3:0]dig0;
58 wire [7:0]smg0;
59
60 always @(posedge clk_100khz)
61 begin
62     sel <= `UD sel+1'b1;
63 end
64 // Digital Tube 0 output
65 seq_control seq_control_0
66 (
67     .sel(sel),
68     .key(data),
69     .dig(dig0),
70     .smg(smg0)
71 );
72 // Digital Tube 1 output
73 wire [3:0]dig1;
74 wire [7:0]smg1;
75 seq_control seq_control_1
76 (
77     .sel(sel),
78     .key(4'd0),
79     .dig(dig1),
80     .smg(smg1)
81 );
82 // Digital Tube 2 output
83 wire [3:0]dig2;
84 wire [7:0]smg2;
85 seq_control seq_control_2
86 (
87     .sel(sel),
88     .key(4'd0),
89     .dig(dig2),
90     .smg(smg2)
91 );

```



```
93 wire [3:0]dig3;
94 wire [7:0]smg3; // Digital Tube 3 output
95 seq_control seq_control_3
96 (
97     .sel(sel),
98     .key(4'd0),
99     .dig(dig3),
100    .smg(smg3)
101 );
102 // display by Digital Tube
103 always @(posedge clk_100khz)
104 begin
105     if(sel==2'b00)
106         dig <= `UD dig0;
107     else if(sel==2'b01)
108         dig <= `UD dig1;
109     else if(sel==2'b10)
110         dig <= `UD dig2;
111     else if(sel==2'b11)
112         dig <= `UD dig3;
113 end
114
115 always @(posedge clk_100khz)
116 begin
117     if(sel==2'b00)
118         smg <= `UD smg0;
119     else if(sel==2'b01)
120         smg <= `UD smg1;
121     else if(sel==2'b10)
122         smg <= `UD smg2;
123     else if(sel==2'b11)
124         smg <= `UD smg3;
125 end
126
127 endmodule
128
```

### 8.4.3 LED key control module

```
1 `timescale 1ns / 1ps
2 `define UD #1
3 module key_control
4 (
5     input          clk,          // input clock
6     input          key_8,        // KEY 8 input
7     input [2:0]    key_in,      // KEY[2:0] input
8     output reg     key8_led,    // LED8 control signal
9     output reg [2:0] key_in_led, // LED[2:0] control signal
10    output reg [2:0] seq_data    // Sequence to be detected
11 );
12
```



```
13  /*=====
14  Key-vibration elimination
15  =====*/
16  wire [2:0]key_out;
17  btn_deb #(
18    .BT_WIDTH(4'd3)
19  ) u_btn_deb_key1 (
20    .clk(clk),
21    .btn_in(key_in),
22    .btn_out(key_out)
23  );
24
25  btn_deb #(
26    .BT_WIDTH(4'd1)
27  ) u_btn_deb_key8 (
28    .clk(clk),
29    .btn_in(key_8),
30    .btn_out(key_8_out)
31  );
32
33  /*=====
34  =====*/
35  reg [2:0]key_out_reg;
36  reg key_8_out_reg;
37
38  always @(posedge clk)
39  begin
40    key_out_reg <= `UD key_out;
41    key_8_out_reg<= `UD key_8_out;
42  end
43
44  reg key_8_flag=0;
45  always @(posedge clk)
46  begin
47    if(!key_8_out && key_8_out_reg)
48      key_8_flag <= `UD ~key_8_flag;
49    else
50      key_8_flag <= `UD key_8_flag;
51  end
52
53  always @(posedge clk)
54  begin
55    key8_led <= `UD key_8_flag;
56  end
57
58  reg [2:0]key_flag=3'b000;
59  always @(posedge clk)
60  begin
61    if(key_8_flag==1'b0)
62      key_flag[0] <= `UD 1'b0;
63    else if(!key_out[0] && key_out_reg[0])
64      key_flag[0] <= `UD ~key_flag[0];
65    else
66      key_flag[0] <= `UD key_flag[0];
67  end
68
```



```
69 always @(posedge clk)
70 begin
71     if(key_8_flag==1'b0)
72         key_flag[1] <= `UD 1'b0;
73     else if(!key_out[1] && key_out_reg[1])
74         key_flag[1] <= `UD ~key_flag[1];
75     else
76         key_flag[1] <= `UD key_flag[1];
77 end
78
79 always @(posedge clk)
80 begin
81     if(key_8_flag==1'b0)
82         key_flag[2] <= `UD 1'b0;
83     else if(!key_out[2] && key_out_reg[2])
84         key_flag[2] <= `UD ~key_flag[2];
85     else
86         key_flag[2] <= `UD key_flag[2];
87 end
88 always @(posedge clk)
89 begin
90     key_in_led <= `UD key_flag;
91 end
92
93 //seq_data
94 always @(posedge clk)
95 begin
96     if(key_8_flag)
97         seq_data <= `UD key_in_led;
98 end
99
100 endmodule
101
```

#### 8.4.4 Design of sequence detection module

```
1 `timescale 1ns / 1ps
2 `define UD #1
3 module seq_det
4 (
5     input          clk,
6     input          key8_led, // Detection state marking
7     input [2:0]    key_in_led, // Sequence to be detected
8     input [7:0]    bm, // Input sequence
9     output reg [3:0] data
10 );
11
12 // 8bit data detecte 3 bit sequence. we need compare six numbers
13 reg [5:0]flag;
14
```

```
15 always @(posedge clk)
16 begin
17     if(!key8_led&&bm[7:5]==key_in_led)
18         flag[0] <= `UD 1'b1;
19     else
20         flag[0] <= `UD 1'b0;
21 end
22
23 always @(posedge clk)
24 begin
25     if(!key8_led&&bm[6:4]==key_in_led)
26         flag[1] <= `UD 1'b1;
27     else
28         flag[1] <= `UD 1'b0;
29 end
30
31 always @(posedge clk)
32 begin
33     if(!key8_led&&bm[5:3]==key_in_led)
34         flag[2] <= `UD 1'b1;
35     else
36         flag[2] <= `UD 1'b0;
37 end
38
39 always @(posedge clk)
40 begin
41     if(!key8_led&&bm[4:2]==key_in_led)
42         flag[3] <= `UD 1'b1;
43     else
44         flag[3] <= `UD 1'b0;
45 end
46
47 always @(posedge clk)
48 begin
49     if(!key8_led&&bm[3:1]==key_in_led)
50         flag[4] <= `UD 1'b1;
51     else
52         flag[4] <= `UD 1'b0;
53 end
54
55 always @(posedge clk)
56 begin
57     if(!key8_led&&bm[2:0]==key_in_led)
58         flag[5] <= `UD 1'b1;
59     else
60         flag[5] <= `UD 1'b0;
61 end
62
63 always @(posedge clk)
64 begin
65     data <= `UD flag[5] + flag[4] + flag[3] + flag[2] + flag[1] + flag[0];
66 end
67
68 endmodule
69
```



---

## 8.5 Result of the experiment

Steps of the experiment:

1. Adjust the input sequence and change the input value of the DIP switch (SW[7:0]);
2. Adjust the fixed sequence and change the state of LED (LED[2:0]) via the soft touch key;
3. Press down the soft touch KEY8 to start detection, and view the statistics displayed on the numeric display;
4. Press down the soft touch KEY8 to exit the detection, and repeat the above 3 steps.

Result of the experiment

When SW[7:0]=8'b10101010 and LED[2:0]=3'b101, the number 3 is displayed on the numeric display after Key8 is pressed down;

When SW[7:0]=8'b10101010 and LED[2:0]=3'b100, the number 0 is displayed on the numeric display after Key8 is pressed down.

---

## 9 Coded Lock

### 9.1 Purpose of the experiment

To make a simple coded lock with the keys, DIP switches and numeric displays on the Runber board.

### 9.2 Requirements of the experiment

Set the password with the DIP switches, and enter the unlocking password with the keys. The unlocking is successful when the unlocking password is the same as the one set, and 8888 is displayed on the numeric displays; otherwise 7777 is displayed.

BM1-BM8 set 4 digits of the password, with every 2 keys setting a digit: BM[0:1] for the 1st digit (corresponding to BM1 and BM2), BM[2:3] for the 2nd digit, BM[4:5] for the 3rd, and BM[6:7] for the 4th. So the password is a 4-digit number consisting of 0, 1, 2, and 3.

KEY0-KEY3 are the password input, and the number increases by 1 when the key is pressed down once. The number is displayed on the numeric display, which changes in a cycle of 0, 1, 2 and 3.

K7 is the key for confirmation. When it is pressed down, the input password is compared with the set password. If the 2 passwords are the same, 8888 is displayed; otherwise 7777 is displayed.

K6 is for clearing to zero. 0000 is displayed on the numeric display when it is pressed down, and a new password can be entered.

### 9.3 Principle of the experiment

The principle is similar to that of sequence detection in the previous chapter. This experiment is an extension on the basis of the previous one.

There is a change in the width of the sequence comparison. As a data is placed in 2 bits, the key-controlled input password data can be set as 2 bits. Comparison and reset are realized with 2 keys in this experiment: one for confirmation of the comparison, and the other for clearing to zero.

### 9.4 Source code of the experiment

3 sub-modules are needed based on the requirements.

#### a) Key control module

1. Key-vibration elimination for input signal of the 6 keys;
2. Falling edge input is taken for KEY7 and KEY6;
3. KEY[3:0] change the respective input password with falling edge, with the number increasing by 1 each time (in a cycle of 0-3, and 2 bits can be used)

#### b) Display module of the numeric display

There are 2 states of display:

State of password input:

1. Default state of power on:

2. Falling edge of KEY6 triggers reset state;
3. The 4-digit input password is displayed real time.

Password validation state:

1. Falling edge of KEY7 triggers the state;
2. Result of password validation is displayed. 8888 is displayed if the password validation is correct; otherwise 7777 is displayed.

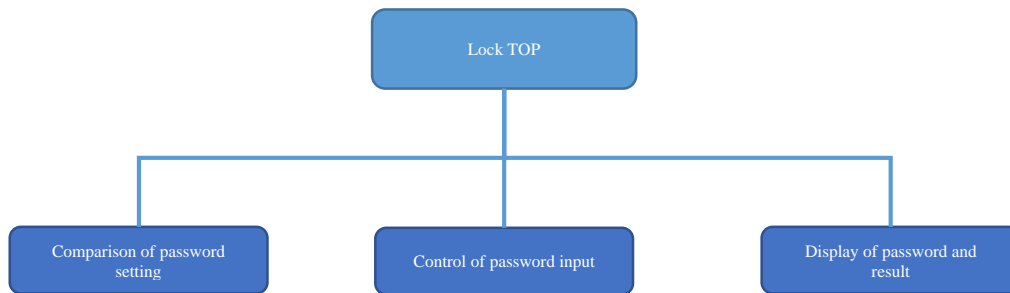
c) Password validation module:

Falling edge of KEY7 triggers enabling. And it triggers the input password saved and compares it with the password set by the DIP switch;

The result of the comparison is output, and sent to the display module of the numeric display

### 9.4.1 Design of top level modules

The relationship between the top level modules and the above 3 modules is shown in the figure below:



The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
key	4	Input	Input signal from the soft touch key, input from K1-K4 on the Runber board
enter	1	Input	Comparison signal for password confirmation, input from K7 on the Runber board
init	1	Input	Re-input signal for password confirmation, input from K6 on the Runber board
sw	8	Input	Input signal for password setting, input from SW1-8 on the Runber board
smg	8	Output	Display of password comparison result, output of numeric display segment selection signal
dig	4	Output	Display of password comparison result, output of numeric display digit selection signal

The module design is as follows:

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module lock_top(
4      input      clk,
5      input  [3:0] key,
6      input      enter,
7      input      init,
8      input  [7:0] sw,
9
10     output [7:0] smg,
11     output [3:0] dig
12 );
13
14     wire      enter_trig;
15     wire      init_trig;
16     wire [7:0] ctrl;
17     wire      com_result;
18
19     key_ctl key_ctl(
20         .clk      ( clk      ),//input      clk,
21         .key      ( key      ),//input      [3:0] key,
22         .enter    ( enter    ),//input      enter,
23         .init     ( init     ),//input      init,
24
25         .enter_trig ( enter_trig ),//output      enter_trig,
26         .init_trig  ( init_trig  ),//output      init_trig,
27         .ctrl      ( ctrl      )//output      [7:0] ctrl
28     );
29
30     compare compare(
31         .clk      ( clk      ),//input      clk,
32         .sw       ( sw       ),//input [7:0] sw,
33         .ctrl     ( ctrl     ),//input [7:0] ctrl,
34         .enter_trig ( enter_trig ),//input      enter_trig,
35
36         .com_result ( com_result )//output      com_result
37     );
38
39     seq_display(
40         .clk      ( clk      ),//input      clk,
41         .enter_trig ( enter_trig ),//input      enter_trig,
42         .init_trig  ( init_trig  ),//input      init_trig,
43         .com_result ( com_result ),//input      com_result,
44         .ctrl      ( ctrl      ),//input      [7:0] ctrl,
45
46         .smg      ( smg      ),//output reg [7:0] smg,
47         .dig      ( dig      )//output reg [3:0] dig
48     );
49
50     endmodule
51

```

#### 9.4.2 Design of key control

The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
key	4	Input	Input signal from the soft touch key, input from K1-K4 on the Runber board
enter	1	Input	Comparison signal for password confirmation, input from K7 on the Runber board
init	1	Input	Password re-input signal, input from K6 on the Runber board
enter_trig	1	Output	Comparison confirmation, which triggers pulse signal output
init_trig	1	Output	Entering password re-input, which triggers pulse signal output
ctrl	8	Output	Output of the 4-digit password input, with the same bit definition as password setting

The module design is as follows:

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module key_ctl(
4      input          clk,
5      input          [3:0] key,
6      input          enter,
7      input          init,
8
9      output         enter_trig,
10     output         init_trig,
11     output         [7:0] ctrl
12 );
13
14     wire [5:0] btn_deb;
15     // Key-vibration elimination
16     btn_deb#(
17         .BTN_WIDTH ( 4'd6                ) //parameter BTN_WIDTH = 4'd8
18     ) U_btn_deb
19     (
20         .clk        ( clk                ), //input          clk,
21         .btn_in     ( {enter,init,key}    ), //input [BTN_WIDTH-1:0] btn_in,
22
23         .btn_deb    ( btn_deb            ) //output reg [BTN_WIDTH-1:0] btn_deb
24     );
25
26     reg [1:0] key1_push_cnt=2'd0;
27     reg [1:0] key2_push_cnt=2'd0;
28     reg [1:0] key3_push_cnt=2'd0;
29     reg [1:0] key4_push_cnt=2'd0;
30
31     reg btn1_deb_1d,btn2_deb_1d,btn3_deb_1d,btn4_deb_1d;
32     reg enter_deb_1d,init_deb_1d;
33
34     // Password confirmation and re-input, which triggers pulse signal acquisition
35     assign enter_trig = ~btn_deb[5] & enter_deb_1d;
36     assign init_trig = ~btn_deb[4] & init_deb_1d;
37
38     always @(posedge clk)
39     begin
40         btn1_deb_1d <= `UD btn_deb[0];
41         btn2_deb_1d <= `UD btn_deb[1];
42         btn3_deb_1d <= `UD btn_deb[2];
43         btn4_deb_1d <= `UD btn_deb[3];
44         init_deb_1d  <= `UD btn_deb[4];
45         enter_deb_1d <= `UD btn_deb[5];
46     end
47

```

```

48     always @(posedge clk)
49     begin
50         if(~btn_deb[4] & init_deb_1d)
51             key1_push_cnt <= `UD 2'd0;
52         else if(~btn_deb[0] & btn1_deb_1d)
53             begin
54                 key1_push_cnt <= `UD key1_push_cnt + 2'd1;
55             end
56     end
57
58     always @(posedge clk)
59     begin
60         if(~btn_deb[4] & init_deb_1d)
61             key2_push_cnt <= `UD 2'd0;
62         else if(~btn_deb[1] & btn2_deb_1d)
63             begin
64                 key2_push_cnt <= `UD key2_push_cnt + 2'd1;
65             end
66     end
67
68     always @(posedge clk)
69     begin
70         if(~btn_deb[4] & init_deb_1d)
71             key3_push_cnt <= `UD 2'd0;
72         else if(~btn_deb[2] & btn3_deb_1d)
73             begin
74                 key3_push_cnt <= `UD key3_push_cnt + 2'd1;
75             end
76     end
77
78     always @(posedge clk)
79     begin
80         if(~btn_deb[4] & init_deb_1d)
81             key4_push_cnt <= `UD 2'd0;
82         else if(~btn_deb[3] & btn4_deb_1d)
83             begin
84                 key4_push_cnt <= `UD key4_push_cnt + 2'd1;
85             end
86     end
87     // Password output: {the 4th digit, the 3rd digit, the 2nd digit, the 1st digit}, with each digit
88     // occupying 2 bits
89     assign ctrl = {key4_push_cnt,key3_push_cnt,key2_push_cnt,key1_push_cnt};
90 endmodule

```

### 9.4.3 Design of comparison module

The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board

sw	8	Input	Setting of 4-digit password
enter_trig	1	Input	Comparison confirmation, which triggers pulse signal input
ctrl	8	Input	Input of 4-digit password
com_result	1	Output	Output of result of comparison

### Module design

```

1  `timescale 1ns / 1ps
2  `define UD #1
3  module compare(
4      input      clk,
5      input [7:0] sw,
6      input [7:0] ctrl,
7      input      enter_trig,
8
9      output      com_result
10 );
11
12 //=====
13 // Latch current password input;
14 reg [7:0] ctrl_1d;
15 always @(posedge clk)
16 begin
17     if(enter_trig)
18         ctrl_1d <= `UD ctrl;
19 end
20
21 assign com_result = (ctrl_1d == sw);
22 endmodule
23

```

### 9.4.4 Design of display module

The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
enter_trig	1	Input	Comparison confirmation, which triggers pulse signal input
init_trig	1	Input	Password re-input, which triggers pulse signal input
com_result	1	Input	Input of result of comparison from the comparison module



ctrl	8	Input	Input of 4-digit password
smg	8	Output	Output of the numeric display segment selection
dig	4	Output	Output of the numeric display digit selection

In design of this module, attention should be paid to the 2 display modes of the numeric display: password input mode and comparison result display mode; the switch between the 2 modes is triggered by enter\_trig and init\_trig.

We do not repeat the description of the display control module of the numeric display here.

```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module seq_display(
4      input          clk,
5      input          enter_trig,
6      input          init_trig,
7      input          com_result,
8      input          [7:0]  ctrl,
9
10     output reg [7:0]  smg,
11     output reg [3:0]  dig
12 );
13
14
15 //=====
16 // Display difference between states
17 reg      seq_status=1'b0;
18 always @(posedge clk)
19 begin
20     if(enter_trig)
21         seq_status <= `UD 1'b1;
22     else if(init_trig)
23         seq_status <= `UD 1'b0;
24 end
25
26 //=====
27 // Control of display on the numeric display
28 reg [3:0] key0_cnt=4'd0,key1_cnt=4'd0,key2_cnt=4'd0,key3_cnt=4'd0;
29 always @(posedge clk)
30 begin
31     if(seq_status)
32     begin
33         if(com_result)
34         begin
35             key0_cnt <= `UD 4'd8;
36             key1_cnt <= `UD 4'd8;
37             key2_cnt <= `UD 4'd8;
38             key3_cnt <= `UD 4'd8;
39         end
40     else
41     begin
42         key0_cnt <= `UD 4'd7;
43         key1_cnt <= `UD 4'd7;
44         key2_cnt <= `UD 4'd7;
45         key3_cnt <= `UD 4'd7;
46     end
47     end
48     begin
49         key0_cnt <= `UD {2'd0,ctrl[1:0]};
50         key1_cnt <= `UD {2'd0,ctrl[3:2]};
51         key2_cnt <= `UD {2'd0,ctrl[5:4]};
52         key3_cnt <= `UD {2'd0,ctrl[7:6]};
53     end
54 end
55
```

```
56 //=====
57 //                               Clock frequency division
58 wire clk_100khz;
59 div_clk div_clk (
60     .clk      (clk),
61     .clk_100khz (clk_100khz)
62 );
63 //=====
64 //                               Display on the numeric display
65 reg [1:0]sel=0;
66 always @(posedge clk_100khz)
67 begin
68     sel <= `UD sel+1'b1;
69 end
70
71 wire [3:0]dig0;
72 wire [7:0]smg0;
73 seq_control seq_control_0
74 (
75     .sel(2'd3),
76     .key(key0_cnt),
77     .dig(dig0),
78     .smg(smg0)
79 );
80
81 wire [3:0]dig1;
82 wire [7:0]smg1;
83 seq_control seq_control_1
84 (
85     .sel(2'd2),
86     .key(key1_cnt),
87     .dig(dig1),
88     .smg(smg1)
89 );
90
91 wire [3:0]dig2;
92 wire [7:0]smg2;
93 seq_control seq_control_2
94 (
95     .sel(2'd1),
96     .key(key2_cnt),
97     .dig(dig2),
98     .smg(smg2)
99 );
100
101 wire [3:0]dig3;
102 wire [7:0]smg3;
103 seq_control seq_control_3
104 (
105     .sel(2'd0),
106     .key(key3_cnt),
107     .dig(dig3),
108     .smg(smg3)
109 );
110
```

```

111     always @(posedge clk_100khz)
112     begin
113         if(sel==2'b00)
114             dig <= `UD dig0;
115         else if(sel==2'b01)
116             dig <= `UD dig1;
117         else if(sel==2'b10)
118             dig <= `UD dig2;
119         else if(sel==2'b11)
120             dig <= `UD dig3;
121     end
122
123     always @(posedge clk_100khz)
124     begin
125         if(sel==2'b00)
126             smg <= `UD smg0;
127         else if(sel==2'b01)
128             smg <= `UD smg1;
129         else if(sel==2'b10)
130             smg <= `UD smg2;
131         else if(sel==2'b11)
132             smg <= `UD smg3;
133     end
134
135 endmodule
136

```

## 9.5 Result of the experiment

Steps of the experiment:

1. Adjust the input sequence and change the input value of the DIP switches (SW[7:0]);
2. Adjust fixed sequence, and adjust password input via soft touch key. The numeric display will display the password input real time.
3. Press down the soft touch KEY7 to trigger password comparison, and the numeric display will display the result of the comparison.
4. Press down the soft touch KEY6 to enter the state of password re-input, and repeat the above 3 steps.

Result of the experiment

When SW[7:0]=8'b10101010, if four 2 are displayed in the password input state, the numeric display will display 8888 after Key7 is pressed down; if the numbers displayed in the password input state are not four 2, the numeric display will display 7777 after Key7 is pressed down. We can enter the password input state to adjust the password by pressing down Key6.

## 10 Digital Clock

### 10.1 Purpose of the experiment

To design a digital clock with the functions of timing and clock calibration.

### 10.2 Requirements of the experiment

The hour and minute readings are displayed on the numeric display, and the second is marked with the twinkle of LED.

3 keys are used for clock calibration.

K1 is used for switch between normal timing and calibration of the hour and minute readings;

K2 is used for “+” of the clock;

And K3 is used for “-” of the clock;

When a scale is being calibrated, the corresponding numeric display will twinkle.

### 10.3 Principle of the experiment

Based on the analysis of the experiment requirements, attention should be paid to 2 functions in realization of the digital clock:

1. The time display function: the twinkle of LED indicates the reading of seconds, the 2 digits on the right side of the numeric display indicate the reading of minutes, and the 2 digits on the left side indicate the reading of hours.

The realization of this function is based on 2 sub-functions: (1) control of timing of 1s, which can be realized in the same way as the timing module the previous experiment. It should be noted that the cycle of timing here is 1s. (2) carrying control in the process of timing. There are 4 points where carrying occurs:

Second $\rightleftharpoons$ Minute	ON and OFF of the LED per time is counted as 1, with a cycle of 1s. When the LED get ON and OFF 60 times, the reading of minutes is added by 1.
Minute <sub>ones</sub> $\rightleftharpoons$ Minute <sub>tens</sub>	When the reading of ones reaches 9 and a number is carried from the second to minute, the reading of tens should be added by 1 and the reading of ones be set to 0.
Minute $\rightleftharpoons$ Hour	1 hour=60 minutes. When the reading of minutes reaches 59 and a number is carried from the second to minute, the reading of ones of hours should be added by 1 and the reading of minutes be set to 0.
Hour <sub>ones</sub> $\rightleftharpoons$ Hour <sub>tens</sub>	Reading of hours is displayed in 24-hour system. When the reading of tens of hours is less than 2, reading of ones of hours reaches 9 and a number is carried from the minute to hour, the reading of tens of hours

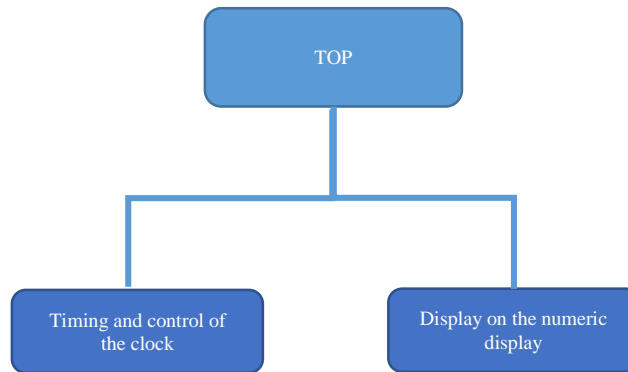
	should be added by 1.
Return to zero at the full range of the 24-hour system	When the reading of tens of hours is 2, reading of ones of hours reaches 3 and a number is carried from the minute to hour, the reading of hours, minutes and second should all return to zero.

2. Clock calibration function: readings of minutes and hours are controlled via the corresponding keys, and the corresponding digit should twinkle during the adjustment.

Attention should be paid to 2 aspects in this process: (1) When a digit is being adjusted, the numeric display should twinkle at this digit; (2) carrying of numbers during the adjustment.

Based on the analysis above, this project is divided into 2 parts:

1. Timing and control of the clock
2. Control of display on the numeric display



## 10.4 Source code of the experiment

### 10.4.1 Top level design

The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
key	3	Input	Clock calibration signal input (soft touch key)
led	1	Output	Display of seconds with twinkles (a twinkle of LED is 1s)
smg	8	Output	Output of the numeric display segment selection
dig	4	Output	Output of the numeric display digit selection

```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module top_watch
4  (
5      input      clk,
6      input [2:0] key,
7      output     led,
8      output [3:0] dig,
9      output [7:0] smg
10 );
11
12     parameter CLK_FRE = 26'd12_000_000;
13
```

```

14      /*=====
15          Creation of reset signal
16      =====*/
17      reg [4:0] rstn_cnt=0;
18      always @(posedge clk)
19      begin
20          if(rstn_cnt==5'h1f)
21              rstn_cnt <= `UD rstn_cnt;
22          else
23              rstn_cnt <= `UD rstn_cnt + 1'b1;
24      end
25
26      wire rstn;
27      assign rstn = rstn_cnt[4];
28      /*=====
29          Creation and control of digital clock
30      =====*/
31      wire [3:0] hour_h, hour_l, minutes_h, minutes_l;
32      wire [2:0] dig_ctl;
33      watch_data_gen #(
34          .CLK_FRE          ( CLK_FRE )//parameter CLK_FRE=26'd12_000_000
35      ) u_watch_data_gen
36      (
37          .clk              ( clk          ), //input clk,
38          .rstn             ( rstn         ), //input rstn,
39          .key              ( key          ), //input [2:0]key,
40          .hour_h_o         ( hour_h       ), //output reg [3:0]hour_h_o,
41          .hour_l_o         ( hour_l       ), //output reg [3:0]hour_l_o,
42          .minutes_h_o     ( minutes_h     ), //output reg [3:0]minutes_h_o,
43          .minutes_l_o     ( minutes_l     ), //output reg [3:0]minutes_l_o,
44          .second_led      ( led          ), //output reg second_led,
45          .state_flag      ( dig_ctl      ) //output reg [2:0]state_flag
46      );
47
48      display_ctl #(
49          .CLK_FRE         ( CLK_FRE )//parameter CLK_FRE = 26'd12_000_000
50      )
51      (
52          .clk             ( clk          ), //input          clk,
53          .dig_ctl         ( dig_ctl      ), //input          [2:0] dig_ctl,
54          .hour_h          ( hour_h       ), //input          [3:0] hour_h,
55          .hour_l          ( hour_l       ), //input          [3:0] hour_l,
56          .minutes_h       ( minutes_h     ), //input          [3:0] minutes_h,
57          .minutes_l       ( minutes_l     ), //input [3:0]          minutes_l,
58          .sec_en          ( led          ), //input          sec_en,
59
60          .dig             ( dig          ), //output reg [3:0] dig,
61          .smg            ( smg          ), //output reg [7:0] smg
62      );
63
64      endmodule
65

```



## 10.4.2 Design of clock timing and control module

### 10.4.2.1 Top level design of timing and control

We will realize 2 functions described above in this module: timing and control.

The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
rstn	1	Input	External input reset signal
key	3	Input	Clock calibration signal input (soft touch key)
hour_h_o	4	Output	Reading of hours at high-order digit
hour_l_o	4	Output	Reading of hours at low-order digit
minutes_h_o	4	Output	Reading of minutes at high-order digit
minutes_l_o	4	Output	Reading of minutes at low-order digit
second_led	1	Output	Display of seconds with twinkles (a twinkle of LED is 1s)
state_flag	3	Output	Output of the numeric display segment selection

Key points of the module design are as follows (for the whole module please see the source file):

1. Clock calibration control (an example of low-order digit of minute reading. Control of other digits are similar, and attention should be paid to carrying of numbers.)

```

1      always @(posedge clk)
2      begin
3          if(key_cnt!=3'd1)                // Make the low-order digit of minute reading the
same as the output value before calibration
4              minutes_1_fix <= `UD minutes_1;
5          else if(key_cnt==3'd1)          // Conduct corresponding adjustment in calibration
state of low-order digit of minute reading
6              begin
7                  if(up_pluse)              // Press on "+" key to add the calibration value
by 1;
8                      begin
9                          if(minutes_1_fix == 4'd9) // If the calibration value is 9 now, it will become 0
10                             minutes_1_fix <= `UD 4'd0; // "+"
11                          else
12                             minutes_1_fix <= `UD minutes_1_fix + 1'b1; // "+"
13                      end
14                  else if(down_pluse)      // Press on "-" key to subtract the calibration value
by 1;
15                      begin
16                          if(minutes_1_fix == 4'd0) // If the calibration value is 0 now, it will become 9
17                             minutes_1_fix <= `UD 4'd9;
18                          else
19                             minutes_1_fix <= `UD minutes_1_fix - 1'b1;
20                      end
21                  else
22                     minutes_1_fix <= `UD minutes_1_fix;
23              end
24          else
25             minutes_1_fix <= `UD minutes_1_fix;
26      end

```

- Control of carrying of the clock (an example of low-order digit of minute reading. The carrying signal of the previous level triggers accumulation, and the carrying signal of itself triggers return to zero)

```

1      wire min_1_carry;
2      assign min_1_carry = (sec_carry == 1'b1) && (minutes_1==4'd9);
3      //minutes_1_gen
4      always @(posedge clk)
5      begin
6          if(!rstn) // Initial value of 0
7              minutes_1 <= `UD 4'd0;
8          else if(key_cnt==3'd1) // The low-order digit of minute reading is the calibration value
during the calibration
9              minutes_1 <= `UD minutes_1_fix;
10         else if(min_1_carry) // Carrying occurs at 9 minutes and 59 seconds, and 0 is assigned
to the low-order digit
11             minutes_1 <= `UD 4'd0;
12         else if(sec_carry) // Carrying to low-order digit of minute reading at 60 seconds
13             minutes_1 <= `UD minutes_1 +1'b1;
14     end
15

```



### 10.4.2.2 Design of key control module

The main function of key control module is to receive key input signal and trigger calibration and adjustment of clock reading. KEY0 adjusts the calibration digit, KEY1 controls “+”, and KEY2 controls “-”. Key points of the module are as follows (key\_out is key-vibration elimination output):

```
1      /*=====
2          //key[0] -> k1 ; is used for calibration marking; key_cnt = 3'd0 is used for normal display
3          key_cnt = 3'd1 is used for calibration of low-order digit of minute reading; 3'd2 is used
4          for calibration of high-order digit of minute reading
5          key_cnt = 3'd3 is used for calibration of low-order digit of hour reading; 3'd4 is used for
6          calibration of high-order digit of hour reading
7          =====*/
8      reg [2:0]key_out_reg=3'd0;
9      always @(posedge clk)
10     begin
11         key_out_reg <= `UD key_out;
12     end
13
14     reg [2:0]key_cnt=3'd0;
15     always @(posedge clk)
16     begin
17         if(key_cnt==3'd4 && (!key_out[0] && key_out_reg[0]))
18             key_cnt <= `UD 3'd0;
19         else if(!key_out[0] && key_out_reg[0])
20             key_cnt <= `UD key_cnt + 1'b1;
21     end
22
23     assign dig_ctl = key_cnt; // Output to other modules
24     /*=====
25         key[1] is used for "+"; key[2] is used for "-"
26         =====*/
27     always @(posedge clk)
28     begin
29         up_pluse    <= `UD !key_out[1] && key_out_reg[1];
30         down_pluse <= `UD !key_out[2] && key_out_reg[2];
31     end
```

### 10.4.3 Design of the numeric display module

Compared with the previous experiment, a function should be added to the numeric display module. The calibration digit of the numeric display should twinkle in calibration mode, so a signal with a cycle of 1s should be introduced. The numeric display should be on for 0.5 s and off for 0.5 s alternately. A key should be introduced to the digit of twinkle to control the output dig\_ctl signal (which is described in previous code).

The design of the twinkle control module is as follows:



```
1  always @(*)
2  begin
3      case(control_dig) // control_dig is the key-controlled calibration digit signal, which
4      corresponds to dig_ctl of the key
5          4'd0: // Normal display
6              case(sel)
7                  2'd0:dig = ~(4'b0001);
8                  2'd1:dig = ~(4'b0010);
9                  2'd2:dig = ~(4'b0100);
10                 2'd3:dig = ~(4'b1000);
11                 default:dig = ~(4'b0000);
12             endcase
13         4'd4: // Calibration of high-order digit of hour reading
14             case(sel) // sec_en is a square wave with a cycle of 1s and a duty ratio of 50%
15                 2'd0:begin if(sec_en) dig = ~(4'b0001 );else dig = ~(4'b0000); end
16                 2'd1:dig = ~(4'b0010);
17                 2'd2:dig = ~(4'b0100);
18                 2'd3:dig = ~(4'b1000);
19                 default:dig = ~(4'b0000);
20             endcase
21         4'd3: // Calibration of low-order digit of hour reading
22             case(sel)
23                 2'd0:dig = ~(4'b0001);
24                 2'd1:begin if(sec_en) dig = ~(4'b0010 );else dig = ~(4'b0000); end
25                 2'd2:dig = ~(4'b0100);
26                 2'd3:dig = ~(4'b1000);
27                 default:dig = ~(4'b0000);
28             endcase
29         4'd2: // Calibration of high-order digit of hour reading
30             case(sel)
31                 2'd0:dig = ~(4'b0001);
32                 2'd1:dig = ~(4'b0010);
33                 2'd2:begin if(sec_en) dig = ~(4'b0100 );else dig = ~(4'b0000); end
34                 2'd3:dig = ~(4'b1000);
35                 default:dig = ~(4'b0000);
36             endcase
37         4'd1: // Calibration of low-order digit of minute reading
38             case(sel)
39                 2'd0:dig = ~(4'b0001);
40                 2'd1:dig = ~(4'b0010);
41                 2'd2:dig = ~(4'b0100);
42                 2'd3:begin if(sec_en) dig = ~(4'b1000 );else dig = ~(4'b0000); end
43                 default:dig = ~(4'b0000);
44             endcase
45         default:dig = ~(4'b0000);
46     endcase
47 end
```

## 10.5 Result of the experiment

The result after uploading: the numeric display begins its display from 00: 00, and LED1 twinkles with a cycle of 1s.

Press down the soft touch KEY0 to enter calibration mode. When KEY0 is pressed down for the

first time, calibration of the low-order digit of minute reading begins. Each time KEY0 is pressed down again, the calibration digit will move leftward by a digit, till to the high-order digit of hour reading, where another press on KEY0 will exit calibration mode and enter the mode of normal timing.

In calibration mode, each time the soft touch KEY1 is pressed down, the corresponding calibration digit will be added by 1, and the digit will return to 0 after reaching the maximum value of counting.

In calibration mode, each time the soft touch KEY2 is pressed down, the corresponding calibration digit will be subtracted by 1, and the digit will become the maximum value of counting after reaching 0.

## 11 Frequency meter

### 11.1 Purpose of the experiment

To measure the frequency of a square wave signal.

### 11.2 Requirements of the experiment

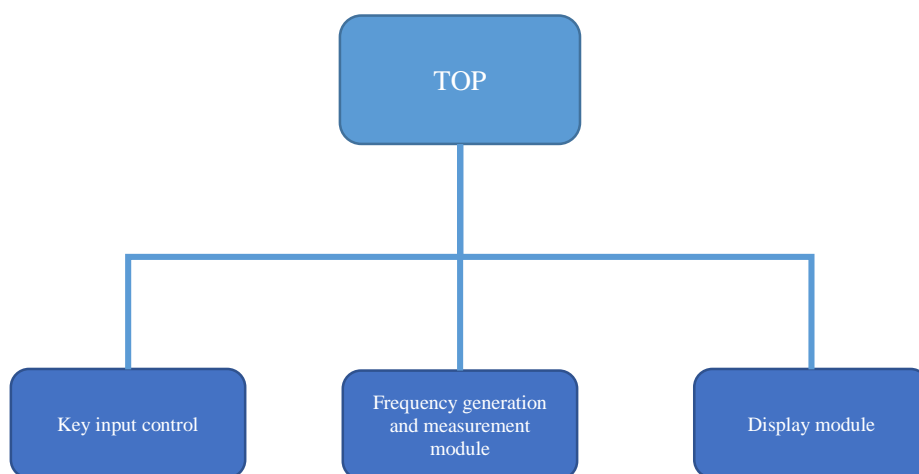
Frequency division of the 50MHz system clock signal to get a low-frequency signal, for which 16 different frequencies can be set via KEY0, and measurement of the frequency of the low-frequency signal with the frequency meter we design. The frequency is measured each 4s. 1s is used for measurement, and 3s for display. The reading will change during the measurement, and the result will be displayed for 3s after the measurement; then the next measurement will begin. 9999 is displayed when the frequency measured is higher than 9999Hz.

### 11.3 Principle of the experiment

Based on the above requirements, the input and output signals of the top level modules should be as follows:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
key	1	Input	Switching signal input from the signal source (soft touch key)
dig	4	Output	Output of the numeric display digit selection
smg	8	Output	Output of the numeric display segment selection

Division of the whole functional module can be designed as shown in the figure below:



#### 11.3.1 Key input control module

Key-vibration elimination and key counting. The clock frequency to be tested is controlled via the

number of key input (with a range of 0-15), and 15 frequency signals are set to be tested.

The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
key	1	Input	Switching signal input from the signal source (soft touch key)
key_times	4	Output	Output of serial number for the signal to be tested

### 11.3.2 Frequency generation and measurement module

Frequency measurement method: statistics of the number of the clock's rising edges within 1s (attention should be paid to overflow in the decimal system). After the result of statistics is got, the measurement result will be displayed for 3s, after which the next set of frequencies can be tested.

The input and output signals are shown in the table below:

Signal	Bit width	Direction	Description
clk	1	Input	External input clock, input clock of 12MHz on the Runber board
key_times	4	Input	Input of serial number for the signal to be tested
seg0	4	Output	Output at ones place of the frequency measurement (Hz)
seg1	4	Output	Output at tens place of the frequency measurement (Hz)
seg2	4	Output	Output at hundreds place of the frequency measurement (Hz)
seg3	4	Output	Output at thousands place of the frequency measurement (Hz)

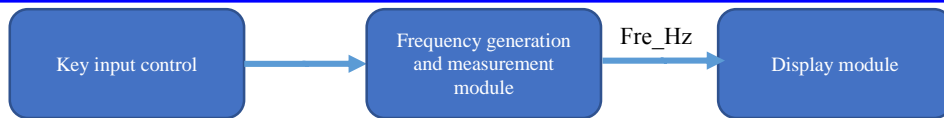
### 11.3.3 Control of display on the numeric display

Similar to the numeric display control above, every digit of the value to be displayed should be transmitted to the module.

## 11.4 Source code of the experiment

### 11.4.1 Modules at top level

The main function of the top level module is to connect the 3 modules. The connection between the signals is as follows (for detail of the code please see the source code file):



Among them key\_times is the times the key is pressed down, and Fre\_Hz is the measured clock frequency (in Hz). There are 4 sets of signals in total, which are ones place, tens place, hundreds place and thousands place.

#### 11.4.2 Key input control module

The main function of this module is to count the number of times the key is pressed down and send it to the frequency measurement module. The code is relatively simple, which is not described here; for details please see the source code file.

#### 11.4.3 Frequency generation and measurement module

##### 11.4.3.1 Generation of low-frequency clock

We have only 4 numeric displays to display the 16 different frequencies of up to 9999Hz. The frequency of the input clock on the Runber board is 12MHz. When the counting of clock cycles reaches 24'd1200000 and returns to zero, a counter with a cycle of 1s (1Hz) is generated. When the counting of clock cycles reaches 11'd1200 and returns to zero, a counter with a cycle of 100us (10KHz) is generated. To generate a low-frequency clock, we use a counter with a bit width of 24 bits. A certain bit of the counter is used as the clock to be tested. When bit 24 is used, the frequency is 0.715Hz; when bit 23 is used, the frequency is 1.43Hz; when bit 12 is used, the frequency is 2929.69Hz; when bit 10 is used, the frequency is 11718.75Hz. The method above will result in 2 sets of frequencies out of the scope, and the signal generation method is as follows:

```
1 reg [23:0] clk_gen;
2 always @(posedge clk)
3 begin
4     clk_gen <= `UD clk_gen + 1'b1;
5 end
6
7 wire freq_gen;
8 assign freq_gen = clk_gen[23-key_times];
```

##### 11.4.3.2 Counting of frequency measurement

As described in the principle, we get the frequency of the signal by counting the number of rising edges of the signal to be tested within a period of 1s. So we need first to get the rising edges of the signals to be tested and the pulse signals with a time width of 1s.

The rising edges of the signals to be tested are obtained as follows:



```
1 reg freq_gen_reg;
2 always @(posedge clk)
3 begin
4     freq_gen_reg <= `UD freq_gen;
5 end
6
7 wire freq_risedge;
8 assign freq_risedge = !freq_gen_reg && freq_gen;
```

The pulse signals with a width of 1s are generated as follows (There are several similar methods in previous experiments to generate the signal with a cycle of 1s, which will not be described here):

```
1 wire test_flag;
2 reg [1:0]flag_cnt=2'd0;
3 always @(posedge clk_1hz)
4 begin
5     flag_cnt <= `UD flag_cnt + 1'b1;
6 end
7
8 assign test_flag = (flag_cnt==2'd0);
```

The method of counting is shown in the code below (examples of ones place and tens place. The method is similar for hundreds place and thousands place. For details please see the source code file):



```
1 wire seg0_carry; // Overflow at ones place
2 wire seg1_carry; // Overflow at tens place
3 wire seg2_carry; // Overflow at hundreds place
4 wire seg3_carry; // Overflow at thousands place
5 // test_flag is a test enable signal with a pulse of 1s, and freq_risedge is the rising edge of the
  signal to be tested
6 assign seg0_carry = (seg0 == 4'd9) && freq_risedge && test_flag;
7 assign seg1_carry = (seg1 == 4'd9) && seg0_carry;
8 assign seg2_carry = (seg2 == 4'd9) && seg1_carry;
9 assign seg3_carry = (seg3 == 4'd9) && seg2_carry;
10
11 always @(posedge clk) // Ones place of the frequency
12 begin
13     if(seg3_carry) //9 is assigned when the measurement reaches the maximum
14         seg0 <= `UD 4'd9;
15     else if(seg0_carry) // Overflow
16         seg0 <= `UD 4'd0;
17     else if(freq_risedge && test_flag)//
18         seg0 <= `UD seg0 + 1'b1;
19     else if(test_start) //0 is assigned to the numeric display before each measurement
20         seg0 <= `UD 4'd0;
21 end
22
23 always @(posedge clk) // Tens place of the frequency
24 begin
25     if(seg3_carry) //9 is assigned when the measurement reaches the maximum
26         seg1 <= `UD 4'd9;
27     else if(seg1_carry) // Processing of overflow after the counting of the current place reaches 9
28         seg1 <= `UD 4'd0;
29     else if(seg0_carry) // Signal of carrying from a low place triggers addition to the counting of
  the current place by 1
30         seg1 <= `UD seg1 + 1'b1;
31     else if(test_start) // 0 is assigned to the numeric display before each measurement
32         seg1 <= `UD 4'd0;
33 end
34
```

### 11.5 Result of the experiment

Result after uploading: display on the numeric display from 0000, and then the cyclical variation of 1s for measurement and 3s for display alternately.

Press down the soft touch KEY0 to adjust the frequency of the signal to be tested, which will increase with the number of times the key is pressed down. The frequency will become the lowest when the key is pressed down 16 times.

## 12 Reaction time tester

### 12.1 Purpose of the experiment

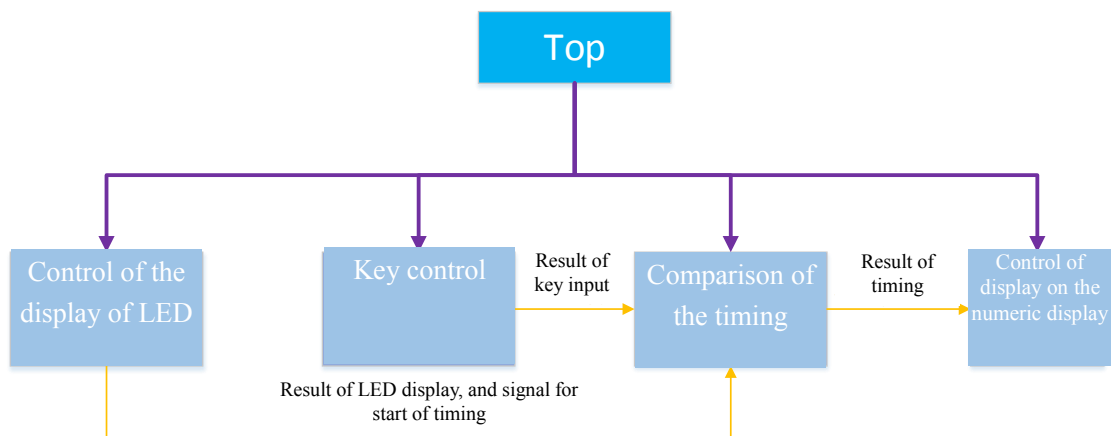
To measure the interval from the time when the human eye recognizes the LED is on to when the key is pressed down to confirm the judgment, so as to get the human reaction time.

### 12.2 Requirements of the experiment

One of the 8 LEDs (LED0-LED7) gets on randomly, and the corresponding key is pressed down right away upon seeing the LED is on. The time interval (in ms) from when the LED is on to when the key is pressed down is measured, which is then displayed in decimal system on the numeric display.

### 12.3 Principle of the experiment

1. A random LED state is generated, and a shifting flowing light is defined. A fixed state of the led being on is fetched 3s after the start is triggered, a set of display states of led is latched, and start of timing is triggered.
2. Timing in ms is conducted. The timer is reset with the trigger start signal. Counting is conducted via accumulation at the ones place, tens place, hundreds place and thousands place of the timer respectively.
3. The display on the numeric display returns to zero as triggered by the re-measurement signal.
4. The comparison module compares the key triggers, and exits if the triggers are the same.



### 12.4 Source code of the experiment

#### 12.4.1 Top level design

For external input and output signals, the experiment can be completed with the clock, keys, LEDs and numeric displays.

```

1  module lock_top(
2      input          clk, // Input clock of 50MHz
3      input  [7:0]   key, // 8 keys for input
4      output [7:0]   led, // 8 LEDs for output
5      output [7:0]   smg, // 8-digit numeric display segment selection
6      output [3:0]   dig  //4-digit numeric display digit selection
7  );
8
9      //=====
10     wire  [7:0] btn_deb;
11     wire          restart;
12     wire          det_start;
13     wire          det_end;
14     wire [15:0] ctrl;
15     //=====
16     btn_deb#( // Key-vibration elimination
17         .BTN_WIDTH      ( 4'd8      ) //parameter  BTN_WIDTH = 4'd8 // Bit width of keys
18     ) U_btn_deb
19     (
20         .clk             ( clk        ), //input  clk, // Input processing clock of 50MHz
21         .btn_in          ( key        ), //input [BTN_WIDTH-1:0] btn_in, // Input key signal
22         .btn_deb         ( btn_deb    ) //output reg [BTN_WIDTH-1:0] btn_deb // Output key-vibration elimination signal
23     );
24
25     assign restart = (~btn_deb[0])&(~btn_deb[7]);
26
27     led_ctl led_ctl(
28         .clk             ( clk        ), //input  clk, // Input processing clock of 50MHz
29         .restart          ( restart    ), //input restart, // Input restart signal, valid for high level
30         .det_start       ( det_start  ), //output reg det_start, // Output detection start signal
31         .led             ( led        ) //output reg [7:0] led // Output detection conditions, triggered with LED vision
32     );
33
34     compare compare(
35         .clk             ( clk        ), //input clk, // Input processing clock of 50MHz
36         .det_start       ( det_start  ), //input det_start, // Timing start signal, valid for high level
37         .restart          ( restart    ), //input restart, // Input restart signal, valid for high level
38         .btn_deb         ( btn_deb    ), //input [ 7:0] btn_deb, // Input key signal
39         .bit_sel         ( led        ), //input [ 7:0] bit_sel, // Input detection conditions
40         .det_end         ( det_end    ), //output det_end, // Output timing stop signal
41         .ctrl            ( ctrl       ) //output reg [15:0] ctrl // Output result of timing
42     );
43
44     seq_display seq_display(
45         .clk             ( clk        ), //input clk, // Input processing clock of 50MHz
46         .restart          ( restart    ), //input restart, // Restart signal, valid for high level
47         .det_end         ( det_end    ), //input det_end, // Input timing stop signal
48         .ctrl            ( ctrl       ), //input [15:0] ctrl, // Result of timing
49
50         .smg             ( smg        ), //output reg [7:0] smg, //8-digit numeric display segment
selection
51         .dig             ( dig        ) //output reg [3:0] dig //4-digit numeric display digit selection
52     );
53
54     endmodule
55

```

### 12.4.2 Key-vibration elimination

This module eliminates the vibration of the input key, which can set the key width and alter it as necessary. This is described in previous chapters and not repeated here.

### 12.4.3 LED display control module

There is a cyclic shift register (with an initial value of 1111\_1110) in the module. 3s after power on the shift register is triggered to latch a value to the LED display register as a test signal for the reaction test, and a trigger signal for start of timing is generated and sent to the comparison and timing module for reaction test.

```
1  module led_ctl(  
2      input                clk,          // Input processing clock of 50MHz  
3      input                restart,     // Input restart signal, valid for high level  
4      output reg          det_start,    // Output detection start signal  
5      output reg [7:0]    led          // Output detection conditions, triggered with LED vision  
6  );  
7  
8  
9  //=====  
10 =  
11 reg [25:0] time_1s_cnt = 26'd0; //0~49_999_999 Cycle of 1ms  
12 always @(posedge clk)  
13 begin  
14     if(time_1s_cnt == 26'd4999_9999)  
15         time_1s_cnt <= `UD 26'd0;  
16     else  
17         time_1s_cnt <= `UD time_1s_cnt + 26'd1;  
18 end  
19  
20 reg [1:0] second_cnt = 2'd0;  
21 always @(posedge clk)  
22 begin  
23     if(restart)  
24         second_cnt <= `UD 2'd0;  
25     else if(time_1s_cnt == 26'd4999_9999)  
26         begin  
27             if(second_cnt == 2'd3)  
28                 second_cnt <= `UD second_cnt;  
29             else  
30                 second_cnt <= `UD second_cnt + 2'd1;  
31         end  
32     end  
33 end
```



```
32 //=====
33 // Cyclic shift register
34 reg [7:0] led_temp = 8'b1111_1110;
35 reg [9:0] time_led_cnt=10'd0;
36 always @(posedge clk)
37 begin
38     if(time_led_cnt == 10'd579)
39         time_led_cnt <= `UD 10'd0;
40     else
41         time_led_cnt <= `UD time_led_cnt + 10'd1;
42 end
43
44 always @(posedge clk)
45 begin
46     if(time_led_cnt == 10'd579)
47         led_temp <= `UD {led_temp[0],led_temp[7:1]};
48 end
49
50 //=====
51 reg [8:0] time_cnt=9'd0; // Fetch random LED state
52 always @(posedge clk)
53 begin
54     time_cnt <= `UD time_cnt + 10'd1;
55 end
56
57 //=====
58 reg start_cnt=1'b0; // Start counting
59 always @ (posedge clk)
60 begin
61     if(second_cnt == 2'd3 && start_cnt == 1'b0 && time_cnt == 10'd375)
62         det_start <= `UD 1'b1;
63     else
64         det_start <= `UD 1'b0;
65 end
66
67 always @(posedge clk)
68 begin
69     if(restart)
70         start_cnt <= `UD 1'b0;
71     else if(second_cnt == 2'd3 && start_cnt == 1'b0 && time_cnt == 10'd375)
72         start_cnt <= `UD start_cnt + 1'b1;
73 end
74
75 always @(posedge clk)
76 begin
77     if(restart)
78         led <= `UD 8'b1111_1111;
79     else if(second_cnt == 2'd3 && start_cnt == 1'b0 && time_cnt == 10'd375)
80         led <= `UD led_temp;
81 end
82
83 endmodule
84
```

#### 12.4.4 Timing comparison module

The main function of this module is to compare whether the key input is the same as the test signal and record the time used. The timing begins when the test start signal is triggered. The counting of ms stops when the results in the comparison are the same or time is over, and the result of counting is output. There are 4 counters of ones, tens, hundreds and thousands, each of which has a range of 0-9. A signal of carrying is generated after 9 is reached, which triggers the addition to the counting at the high-order place by 1. Attention should be paid to overflow of counting (time-out, 9999ms for 4 digits in decimal system). Timing and counting signals are output (falling edges of the timing enable signal are fetched).

```
1  module compare(
2      input          clk,          // Input processing clock of 50MHz
3      input          det_start,    // Input timing start signal for a clock cycle, valid for high level
4      input          restart,     // Restart signal
5      input [ 7:0]   btn_deb,     // Input key signal
6      input [ 7:0]   bit_sel,     // Input detection conditions
7
8      output         det_end,     // Output timing stop signal
9      output reg [15:0] ctrl      // Output result of timing
10 );
11
12 //=====
13 // ms counter
14 reg [15:0] time_ms_cnt= 16'd0; //0~49999   Cycle of 1ms
15 always@(posedge clk)
16 begin
17     if(det_start)
18         time_ms_cnt <= `UD 16'd0;
19     else if(time_ms_cnt == 16'd49999)
20         time_ms_cnt <= `UD 16'd0;
21     else
22         time_ms_cnt <= `UD time_ms_cnt + 16'd1;
23 end
24
25 //=====
26 reg    counter_en = 1'b0;
27 reg    flow = 1'b0;
28 reg    counter_en_1d = 1'b0;
29 always @(posedge clk)
30 begin
31     if(det_start)
32         counter_en <= `UD 1'b1;
33     else if((btn_deb == bit_sel) || flow || restart)
34         counter_en <= `UD 1'b0;
35 end
36
37 always @(posedge clk)
38 begin
39     counter_en_1d <= `UD counter_en;
40 end
41
42 assign det_end = (~counter_en & counter_en_1d) & (~restart);
43
```



```
44 //=====  
45 // Statistics of reaction time  
46 wire [3:0] dec_single,dec_ten,dec_hundred,dec_thousand;  
47 wire      dec_sigle_trg;  
48 wire      carry1,carry2,carry3,carry4;// Carrying at 4 digits  
49  
50 assign dec_sigle_trg = counter_en & (time_ms_cnt == 16'd49999);  
51  
52 always @(posedge clk)  
53 begin  
54     if(det_start)  
55         flow <= `UD 1'b0;  
56     else if(carry4)  
57         flow <= `UD 1'b1;  
58 end  
59  
60 assign ctrl={dec_thousand,dec_hundred,dec_ten,dec_single};  
61  
62 dec_counter single(  
63     .clk          ( clk          ),//input          clk,  
64     .det_start    ( det_start    ),//input          det_start,  
65     .trig         ( dec_sigle_trg ),//input          trig,  
66     .flow         ( flow         ),//input          flow,  
67     .carry        ( carry1       ),//output reg     carry,  
68     .dec          ( dec_single   ) //output reg [3:0] dec  
69 );  
70  
71 dec_counter ten(  
72     .clk          ( clk          ),//input          clk,  
73     .det_start    ( det_start    ),//input          det_start,  
74     .trig         ( carry1       ),//input          trig,  
75     .flow         ( flow         ),//input          flow,  
76     .carry        ( carry2       ),//output reg     carry,  
77     .dec          ( dec_ten      ) //output reg [3:0] dec  
78 );  
79  
80 dec_counter hundred(  
81     .clk          ( clk          ),//input          clk,  
82     .det_start    ( det_start    ),//input          det_start,  
83     .trig         ( carry2       ),//input          trig,  
84     .flow         ( flow         ),//input          flow,  
85     .carry        ( carry3       ),//output reg     carry,  
86     .dec          ( dec_hundred  ) //output reg [3:0] dec  
87 );  
88  
89 dec_counter thousand(  
90     .clk          ( clk          ),//input          clk,  
91     .det_start    ( det_start    ),//input          det_start,  
92     .trig         ( carry3       ),//input          trig,  
93     .flow         ( flow         ),//input          flow,  
94     .carry        ( carry4       ),//output reg     carry,  
95     .dec          ( dec_thousand ) //output reg [3:0] dec  
96 );  
97  
98 endmodule  
99
```



If a trigger signal comes when the counting reaches 9, the module processing carrying of the counting will generate a carrying signal as a trigger signal for the next digit, and reset the counting as 0. If a trigger signal comes when the counting is less than 9, the module will add the counting by 1.

```
1  module dec_counter(  
2      input      clk,  
3      input      det_start,  
4      input      trig,  
5      input      flow,  
6  
7      output reg  carry,  
8      output reg [3:0] dec  
9  );  
10  
11  always @(posedge clk)  
12  begin  
13      if(det_start) // Reset  
14          dec <= `UD 4'd0;  
15      else if(flow) // Overflow processing  
16          dec <= `UD 4'd10;  
17      else if(trig)  
18          begin  
19              if(dec == 4'd9)  
20                  dec <= `UD 4'd0;  
21              else if(dec != 4'd10)  
22                  dec <= `UD dec + 1'b1;  
23          end  
24      end  
25  
26  always @(posedge clk)  
27  begin  
28      if(trig & dec == 4'd9)  
29          carry <= `UD 1'b1;  
30      else  
31          carry <= `UD 1'b0;  
32      end  
33  
34  endmodule  
35
```

#### 12.4.5 Display module of the numeric display

The function of this module is to display the result of timing. The 4 numeric displays will display 0000 upon receiving restart signal, and display the result of timing at the corresponding digits upon receiving timing stop signal. 0-9 is displayed for normal timing, and HHHH for time-out. This module is used several times in previous experiments. What is different here is the differences in setting and result display.

#### 12.5 Result of the experiment

Human reaction time can be calculated with an accuracy of 1ms, with the maximum value being 9999ms.

### 13 Servo Experiment

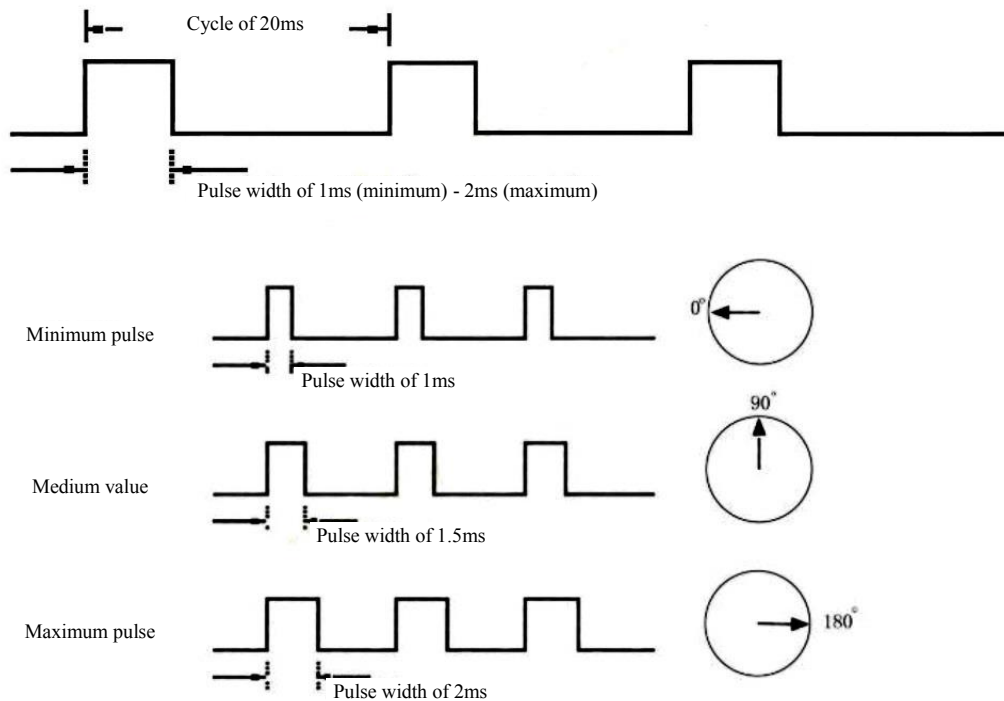
#### 13.1 Purpose of the experiment

To control the rotation of servo by a certain angle.

#### 13.2 Requirements of the experiment

Convert the angle of the rotation of the servo into the number of clock cycles of the PWM signal at high level. The angle by which the servo rotates is adjusted with 2 keys: each time key1 is pressed down, the time of PWM signal at high level decreases and the angle decreases by  $5^\circ$ ; each time key2 is pressed down, the time of PWM signal at high level increases and the angle increases by  $5^\circ$ .

#### 13.3 Principle of the experiment



Control pulse width of SG90S servo is 0.5ms-2.5ms.

The frequency of the onboard crystal oscillator is 12MHz, and 240000 clock cycles (Tclk) are needed for timing of 20ms.

- (1) Design an 18-bit counter, which returns to zero when the counting reaches Tclk - 1.
- (2) Convert the input angle into the number of clock cycles.

When driving SG90S, a variation of 2ms in pulse width can control  $180^\circ$  of the rotation of the servo. So an adjustment of  $1^\circ$  needs a variation of  $1/90$ ms in pulse width. There will be representation error when a clock of 12MHz is used, and additional logical resources are needed to control the error.

- (3) PWM signal is obtained by comparing the threshold value with the counting of the counter.

- (4) Adjust the angle of variation to  $n^\circ$  of rotation per cycle of PWM wave (the accuracy set as  $n^\circ$ ), and save the current angle in the register.

## 13.4 Source code of the experiment

### 13.4.1 Modules at top level

```
1  module servo_ctl(  
2      input    clk,  
3      input [1:0] key,  
4      output   pwm  
5  );  
6  
7      wire [7:0] ctrl;  
8  
9      key_ctl key_ctl(  
10         .clk      ( clk      ),//input      clk,  
11         .key      ( key      ),//input      key,  
12         .ctrl     ( ctrl     )//output     [1:0] ctrl  
13     );  
14  
15     pwm_servo #(  
16         .CLK_FRE      ( 24'd12_000_000 ),//parameter  
17         .PWM_CYCLE_TIME ( 4'd20          )//parameter  
18     ) pwm_servo (   
19         .clk          ( clk          ),//input      clk,  
20         .cfg          ( ctrl          ),//input [7:0] cfg,  
21         .pwm          ( pwm          )//output     pwm  
22     );  
23  
24     endmodule  
25
```

### 13.4.2 Key-vibration elimination module

This module is similar to the key-vibration elimination experiment. Please see the experiment in section 3.

### 13.4.3 Module for angle adjustment via keys

The angle is adjusted with 2 keys, which increases or decreases it by  $5^\circ$ . The keys of Runber are at high level by default. After vibration elimination, a change from high level to low level is considered a key being pressed down, and the angle increases or decreases within a scope. The angle is thus adjusted via the keys.



```
1     reg[1:0] btn_deb_1d;
2     always @(posedge clk)
3     begin
4         btn_deb_1d <= `UD btn_deb;
5     end
6     // set angle
7     reg [7:0] key_push_cnt=8'b0000_0000;
8     always @(posedge clk)
9     begin
10        if(~btn_deb[0] & btn_deb_1d[0])
11        begin
12            if(key_push_cnt <= 8'd5)
13                key_push_cnt <= `UD key_push_cnt;
14            else
15                key_push_cnt <= `UD key_push_cnt -8'd5;
16        end
17        else if(~btn_deb[1] & btn_deb_1d[1])
18        begin
19            if(key_push_cnt >= 8'd175)
20                key_push_cnt <= `UD key_push_cnt;
21            else
22                key_push_cnt <= `UD key_push_cnt + 8'd5;
23        end
24    end
25    assign ctrl = key_push_cnt;
```

#### 13.4.4 PWM output module

The cycle of PWM is set as 20ms, and the minimum time at high level is 0.5ms. The time needed for the adjustment of the angle is added to the aforesaid time. PWM\_CYCLE is the number of clock cycles.

```
1     // PWM cycle control
2     always @(posedge clk)
3     begin
4         if(pwm_cnt == PWM_CYCLE - 1'b1)
5             pwm_cnt <= `UD 16'd0;
6         else
7             pwm_cnt <= `UD pwm_cnt + 1'd1;
8     end
9
10    // PWM signal control
11    always @(posedge clk)
12    begin
13        if(pwm_cnt == 16'd0)
14            pwm_clk <= `UD 1'b1;
15        else if(pwm_cnt == pwm_set)
16            pwm_clk <= `UD 1'b0;
17    end
18
```

A variation of 40 clock cycles in the time at high level corresponds to a variation of  $0.3^\circ$  in the angle. Within 10 PWM cycles, the time at high level changes 3 times with a gradient of 40 clock cycles (a variation of about  $1^\circ$  of the corresponding angle), until the angle reaches the set value.



```
1  reg [3:0] angle_cnt=0; // Measurement of the current angle.
2  // 40 times counter -> 0.3 A variation of about 0.3° in time width for 40 clock cycles
3  always @(posedge clk)
4  begin
5      if(angle != cfg && pwm_cnt == PWM_CYCLE - 1'b1)
6          begin
7              if(angle_cnt == 4'd9)
8                  angle_cnt <= `UD 4'd0;
9              else
10                 angle_cnt <= `UD angle_cnt + 1'b1;
11            end
12        else if(angle == cfg)
13            angle_cnt <= `UD 4'd0;
14        else
15            angle_cnt <= `UD angle_cnt;
16    end
17
18    // cfg set the servo angle
19    // angle Record of current angle in the module
20    always @(posedge clk)
21    begin
22        if(angle < cfg)
23            begin
24                if(pwm_cnt == PWM_CYCLE - 1'b1)
25                    begin
26                        if((angle_cnt == 4'd2||angle_cnt == 4'd6||angle_cnt == 4'd9))
27                            angle <= `UD angle + 1'b1;
28                        else
29                            angle <= `UD angle;
30                    end
31                end
32            else if(angle > cfg)
33                begin
34                    if(pwm_cnt == PWM_CYCLE - 1'b1)
35                        begin
36                            if((angle_cnt == 4'd2||angle_cnt == 4'd6||angle_cnt == 4'd9))
37                                angle <= `UD angle - 1'b1;
38                            else
39                                angle <= `UD angle;
40                        end
41                    end
42                else
43                    angle <= `UD cfg;
44            end
45    end
```

PWM output is on the basis of whether the set value of the angel is reached. If the set value is not reached, the time at high level is changed at a gradient of 40 clock cycles per PWM cycle; if the set value is reached, the time of each PWM at high level is kept unchanged.

```
1 // angle to counter Value of angle converted to threshold of the counter
2 always @(posedge clk)
3 begin
4     if(pwm_cnt == PWM_CYCLE - 1'b1)
5         begin
6             if(angle == cfg)
7                 pwm_set <= `UD pwm_set;
8             else if(angle < cfg)
9                 pwm_set <= `UD pwm_set + 6'd40;
10            else
11                pwm_set <= `UD pwm_set - 6'd40;
12        end
13    else
14        pwm_set <= `UD pwm_set;
15 end
16
```

### 13.5 Result of the experiment

Result of the experiment: when key1 is pressed down, the angle of rotation of the Servo is reduced by 5°; when key2 is pressed down, the angle of rotation of the Servo is increased by 5°.

## 14 Ultrasonic ranging

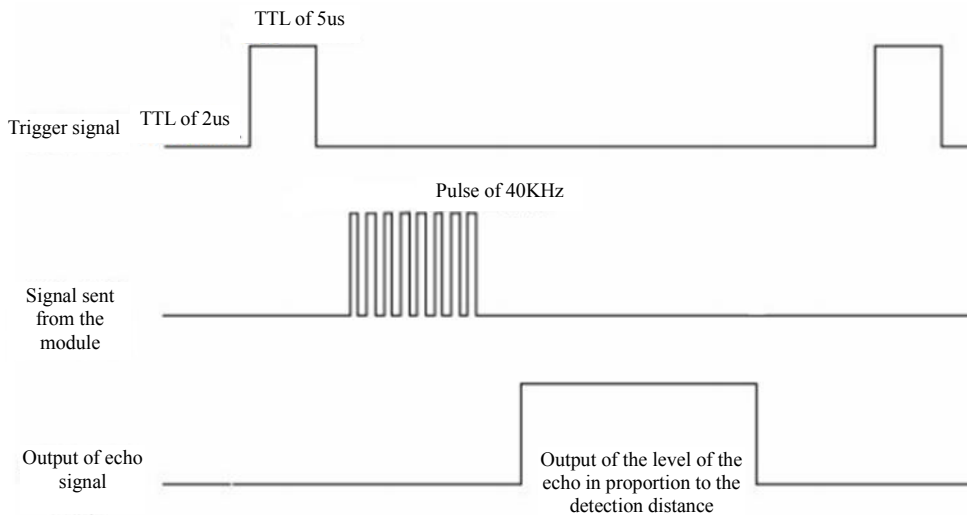
### 14.1 Purpose of the experiment

To drive the ultrasonic ranging sensor with FPGA development board to realize ranging with ultrasonic wave module

### 14.2 Requirements of the experiment

To design a cycle counter of 1s which sends a trigger signal to the ultrasonic sensor every second, detects the echo signal of the ultrasonic sensor and calculates the time of the echo signal at high level, calculates the distance based on the time of the echo signal, and displays the distance (in centimeters) on the numeric display.

### 14.3 Principle of the experiment



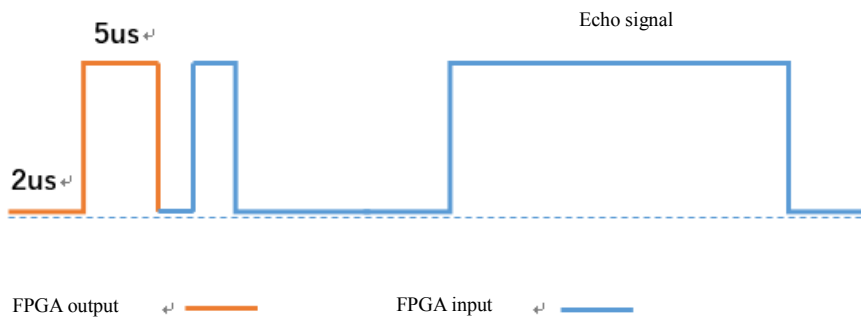
After the ultrasonic ranging module receives the trigger signal, it sends a pulse signal of 40KHz to drive the ultrasonic wave sending sensor to transmit a ultrasonic signal. After the ultrasonic signal is transmitted, the low level of the echo signal output from the ultrasonic ranging module changes to high level; after the ultrasonic wave receiving sensor receives the reflected signal, the high level of the echo signal changes to low level. It is assumed that the velocity of propagation of acoustic wave in the air remains constant and the distance between the ultrasonic sensor and the object remains the same during the ranging, and the distance can be calculated by calculating the time of the echo signal at high level, that is, the total time of the propagation of the acoustic wave back and forth.

#### 14.3.1 Input/output control

As ultrasonic ranging is designed to repeat the measurement every second, the control module needs to output 2us of low level + 5us of high level trigger signal to repeat the measurement 1s after the trigger. The control state switches at fixed time. The states of the sig pins of FPGA are shown in the table below:

Time	Input/output state	Level
2us	Output	Low
5us	Output	High
1s	Input	Echo signal

After the trigger signal ends and the output state of fpga pins changes to input state, a high level signal appears first. The time of the signal at low level is first read in the input state lest the signal be taken as echo signal. If the time at low level is long enough, the time of the echo signal at high level calculated subsequently is valid.



Some important parameters needed for the control function are listed below:

Name of parameter	Time	Count of clock cycles	Meaning
TRIG_L	2us	24	Trigger signal at low level for 2us
TRIG_H	5us	60	Trigger signal at high level for 5us
PERIOD	1s	12000000	Measurement every second
COUNT_CM	58us	695	Measurement of time for every 1cm

In terms of the code, a counter is designed to control the input/output state. When the count is less than TRIG\_L, low level is output; when the count is TRIG\_L - TRIG\_H, high level is output. The 1s afterward is in input state. When TRIG\_L+TRIG\_H+PERIOD is reached, the counter returns to the counting of 0 cycle, and the measurement is repeated.

### 14.3.2 Calculation of the distance

The time of the echo signal at high level is used to calculate the distance. The formula used is  $t1/58$ .



When the frequency of the system clock is 12MHz, 12 clock cycles constitute 1us. The measurement accuracy of 1cm corresponds to a time of 58us, that is, 696 clock cycles. The number of clock cycles needed for measurement of a distance of 1cm is recorded via the counter. If the counting reaches 696 clock cycles, the distance is added by the minimum accuracy (1cm), and the final distance measured is calculated. The counting is stopped upon the end of the echo signal. As numeric displays are used for display, multi-stage trigger counter is designed during the measurement to assign the result of measurement to the ones place, tens place and hundreds place.

### 14.3.3 Control of display on the numeric display

Similar to the numeric display control above, every digit of the value to be displayed should be transmitted to the module.

## 14.4 Source code of the experiment

### 14.4.1 Modules at top level

The modules at top level connect the input/output control and calculation module with the display module.

The input/output control and calculation module determines sending of trigger signal and receiving of echo signal, calculates the time of echo signal, and converts the time into distance; the display module displays the numbers on the numeric display.

```
1  module top_ult
2  (
3      input          clk, //12MHZ
4      input          rstn,
5      Inout         sig,
6      output reg [3:0] dig,
7      output reg [7:0] smg
8  );
9  /* Counting of distance */
10 wire [3:0] count_one;
11 wire [3:0] count_ten;
12 wire [3:0] count_hundred;
13 wire [3:0] count_thousand;
14 cnt cnt
15 (
16     .clk          ( clk          ),
17     .sig          ( sig          ),
18     .rstn         ( rstn         ),
19     .count_one    ( count_one    ),
20     .count_ten    ( count_ten    ),
21     .count_hundred ( count_hundred ),
22     .count_thousand ( count_thousand )
23 );
24
```



```
25  /*=====
26  Clock frequency division
27  =====*/
28  wire clk_100khz;
29  div_clk div_clk
30  (
31  .clk          ( clk          ),
32  .clk_100khz  ( clk_100khz  )
33  );
34  /*=====
35  Display on the numeric display
36  =====*/
37  reg [1:0]sel=0;
38  wire [3:0]dig0;
39  wire [7:0]smg0;
40
41  always @(posedge clk_100khz)
42  begin
43      sel <= `UD sel+1'b1;
44  end
45
46  seq_control seq_control_0
47  (
48  .sel      ( 2'd3      ),
49  .key      ( count_one ),
50  .dig      ( dig0      ),
51  .smg      ( smg0      )
52  );
53
54  wire [3:0]dig1;
55  wire [7:0]smg1;
56  seq_control seq_control_1
57  (
58  .sel      ( 2'd2      ),
59  .key      ( count_ten ),
60  .dig      ( dig1      ),
61  .smg      ( smg1      )
62  );
63  always @(posedge clk_100khz)
64  begin
65      if(sel==2'b00)
66          smg <= `UD smg0;
67      else if(sel==2'b01)
68          smg <= `UD smg1;
69      else if(sel==2'b10)
70          smg <= `UD smg2;
71      else if(sel==2'b11)
72          smg <= `UD smg3;
73  end
74  endmodule
75
```

#### 14.4.2 Input/output control

Counting is carried out in every measurement cycle, 2us at low level and 5us at high level is output,



and the state is converted to input to detect echo signal.

As the sig pin is reused by the trig and echo signals of the ultrasonic wave module, it should be defined as an inout port. In the design of FPGA control module, a whole measurement cycle is divided into 3 stages: the first stage is to output a trigger signal of 2us at low level, the second stage is to output a trigger signal of 5us at high level, and the third stage is to input an echo signal and calculate the time of valid echo signal at high level.

```
1  /*=====
2      Timing of cycles in each state, to control the current state at different time
3      Output of 2us at low level, output of 5us at high level, and input of timing at high level
4  =====*/
5  reg[23:0] count_period =24'b0;
6  always @ (posedge clk)
7  begin
8      if(!rstn||(count_period==(TRIG_L+TRIG_H+PERIOD)))
9          count_period<=`UD 24'b0;
10     else
11         count_period<=`UD count_period+1'd1;
12     End
13
14  /*=====
15      Trig signal of 2us at low level + 5us at high level
16  =====*/
17  reg trig; // Mark of output trigger signal
18  always @ (posedge clk)
19  begin
20      if(!rstn)
21          trig <=`UD 1'd1;
22      else if(count_period<=(TRIG_L+TRIG_H))
23          trig <=`UD 1'd1;
24      else
25          trig <=`UD 1'd0;
26     End
27
28  reg sig_out; // Output trigger signal of 2us at low level + 5us at high level
29  always @ (posedge clk)
30  begin
31      if(!rstn)
32          sig_out <=`UD 1'd0;
33      else if((count_period>TRIG_L)&&(count_period<=(TRIG_L+TRIG_H)))
34          sig_out <=`UD 1'd1;
35      else
36          sig_out <=`UD 1'd0;
37     end
38
```

```

39     assign sig = (trig==1'b0) ? 1'bz : sig_out; // Input when trig is 0, and output when trig is 1
40     reg in; // The input signal is delayed for a cycle in the input state, and reg buffer is used for
the input signal of inout
41     always @ (posedge clk)
42     begin
43         if(!trig)
44             in <= `UD sig; // Delayed for a cycle
45         else
46             in <= `UD 1'b0;
47     end
48

```

#### 14.4.3 Calculation of velocity

Whether the input signal of FPGA is a valid echo signal is determined.

```

1  /*=====
2  The signal is determined as a valid echo signal
3  =====*/
4  reg [17:0]low_cnt=18'b0; //200us == 12*200 = 2400;
5  always @ (posedge clk)
6  begin
7      if(trig == 1'b0)
8      begin
9          if(~in)
10         begin
11             if(low_cnt == 18'd2400)
12                 low_cnt <= `UD low_cnt;
13             else
14                 low_cnt <= `UD low_cnt + 18'b1; // counting when in=0, and reset when in=1
15         end
16     else
17         low_cnt <= `UD 18'b0;
18     end
19     else
20         low_cnt <= `UD 18'b0;
21 end
22
23 reg flag=1'b0; // Input mark for valid high level signal
24 always @ (posedge clk)
25 begin
26     if(!rstn||trig==1'b1)
27         flag <= `UD 1'b0;
28     else if(low_cnt == 18'd1200) // A mark of validity of 1 for 1200 clock cycles at
high level
29         flag <= `UD ~flag;
30     else
31         flag <= `UD flag;
32 end
33

```

2 counters are designed in input state. Counter 1 performs timing when the input signal is at low level, and the echo signal is valid only if the low level is maintained for enough time. The counter keeps counting in cycle when the input is at high level. When the counting reaches COUNT\_CM,



the distance increases by 1cm and the counting returns to 0 and begins again.

Each time the counting reaches 695, the ones place of the distance increases by 1; each time the ones place increases after it reaches 9, the ones place returns to 0 and the tens place increases by 1; each time the tens place increases after it reaches 9, the tens place returns to 0 and the hundreds place increases by 1.

```
1     reg [12:0] count_echo=13'd0;
2     always @ (posedge clk)
3     begin
4         if(!rstn||trig==1'b1||count_echo==COUNT_CM&&!flag)
5             count_echo <= `UD 13'd0;
6         else if((trig==0)&&(in==1'b1)&&(count_echo<COUNT_CM)&&flag)
7             count_echo <= `UD count_echo+1'd1; // Counting for 1cm
8         else
9             count_echo <= `UD count_echo;
10    end
11
12    always @ (posedge clk)
13    begin
14        if(!rstn||trig==1'b1||count_one==4'd10)
15            count_one<= `UD 1'd0;
16        else if((count_echo==COUNT_CM)&&(count_one<=4'd9))// Ones place
17            count_one<= `UD count_one+1'd1;
18        else
19            count_one<= `UD count_one;
20    end
21
22    always @ (posedge clk)
23    begin
24        if(!rstn||trig==1'b1||count_ten==4'd10)
25            count_ten<= `UD 1'd0;
26        else if((count_one==4'd10)&&(count_ten<=4'd9))// Tens place
27            count_ten<= `UD count_ten+1'd1;
28        else
29            count_ten<= `UD count_ten;
30    end
31
32    always @ (posedge clk)
33    begin
34        if(!rstn||trig==1'b1||count_hundred==4'd10)
35            count_hundred<= `UD 1'd0;
36        else if((count_ten==4'd10)&&(count_hundred<=4'd9))// Hundreds place
37            count_hundred<= `UD count_hundred+1'd1;
38        else
39            count_hundred<= `UD count_hundred;
40    end
41
```

```
42 always @ (posedge clk)
43 begin
44     if(!rstn||trig==1'b1||count_thousand==4'd10)
45         count_thousand<= `UD 1'd0;
46     else if((count_hundred==4'd10)&&(count_thousand<=4'd9)) / Thousands place
47         count_thousand<= `UD count_thousand+1'd1;
48     else
49         count_thousand<= `UD count_thousand;
50 end
```

### 14.5 Result of the experiment



Result of the experiment: the ultrasonic ranging can achieve accurate measurement of distance and the display in cm on the numeric display.

## 15 Measurement of temperature and humidity

### 15.1 Purpose of the experiment

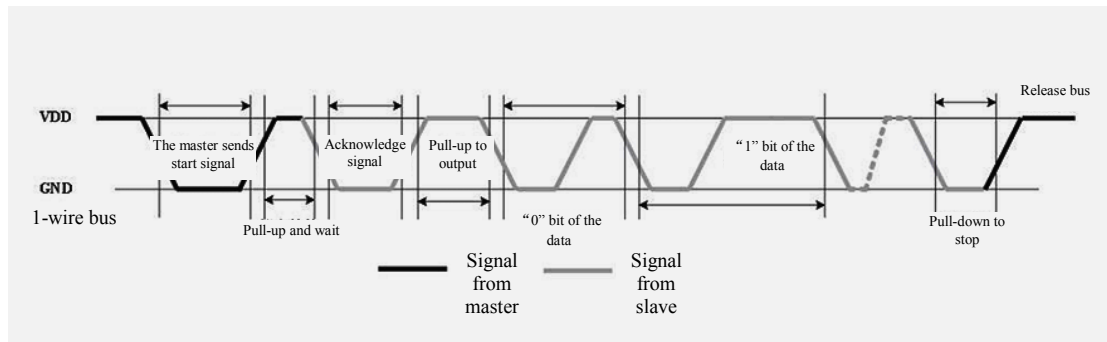
Use of FPGA development Board to drive temperature and humidity sensor to measure temperature and humidity.

### 15.2 Requirements of the experiment

During the measurement of temperature and humidity, a trigger signal of 18ms at low level is sent at first. If an acknowledge signal is received, detection and saving of the subsequent 40 bits of data begins. If the data passes the verification, the received data is converted to values of temperature and humidity in decimal system, the humidity is displayed on the first 2 digits of the numeric display, and the temperature is displayed on the last 2 digits of the numeric display (integer part only).

### 15.3 Principle of the experiment

#### 15.3.1 Input/output control



A trigger signal of 18ms at low level is first output to the temperature and humidity sensor. It then changes into input state to receive acknowledge signal of 80us at low level + 80us at high level, and begins to read the temperature and humidity values. 50Us at low level + 70us at high level signifies “1”, and 50us at low level + 26-28us at high level signifies “0”.

#### 15.3.2 Data conversion

Example 1: the 40-bit data received is:

<u>0011 0101</u>	<u>0000 0000</u>	<u>0001 1000</u>	<u>0000 0000</u>	<u>0100 1101</u>
High-order 8 bits of humidity	Low-order 8 bits of humidity	High-order 8 bits of temperature	Low-order 8 bits of temperature	Parity bits

Calculation:

$$0011\ 0101 + 0000\ 0000 + 0001\ 1000 + 0000\ 0000 = 0100\ 1101$$

The received data is correct:

Humidity:  $0011\ 0101 = 35H = 53\%RH$

Temperature: 0001 1000 = 18H = 24°C

### 15.3.3 Control of display on the numeric display

Similar to the numeric display control above, every digit of the value to be displayed should be transmitted to the module.

## 15.4 Source code of the experiment

### 15.4.1 Modules at top level

First, the input/output control and data reading module sends trigger signal, receives acknowledge signal and reads the data, and the data conversion module then converts the data into values of temperature and humidity in decimal system. Finally, the display module displays the values of temperature and humidity on the numeric display.





```
1  `timescale 1ns / 1ps
2  `define UD #1
3  module top_dht
4  (
5      input clk,
6      input rstn,
7      inout in_out,
8      output reg [3:0]dig,
9      output reg [7:0]smg
10 );
11 /*=====
12                      Trigger and data reading
13 =====*/
14 wire [31:0]information;
15 dht dht(
16     .clk      (clk      ),//input clk,
17     .rstn     (rstn     ),//input rstn,
18     .in_out   (in_out   ),//inout in_out,
19     .information (information) //output reg [31:0]information
20 );
21
22 wire [3:0]humidity_one;// Humidity
23 wire [3:0]humidity_ten;
24 wire [3:0]humidity_decimal;
25 wire [3:0]temp_one;// Temperature
26 wire [3:0]temp_ten;
27 wire [3:0]temp_decimal;
28 /*=====
29                      Conversion of data format
30 =====*/
31 change change(
32     .clk      (clk      ),
33     .rstn     (rstn     ),
34     .information (information ),
35     .humidity_one (humidity_one ),
36     .humidity_ten (humidity_ten ),
37     .humidity_decimal(humidity_decimal),
38     .temp_one     (temp_one     ),
39     .temp_ten     (temp_ten     ),
40     .temp_decimal (temp_decimal ) );
41 /*=====
42                      Clock frequency division
43 =====*/
44 wire clk_100khz;
45 div_clk div_clk
46 (
47     .clk      (clk),
48     .clk_100khz (clk_100khz)
49 );
50
```

```
51  /*=====
52  Display of temperature on the numeric display
53  =====*/
54  reg [1:0]sel=0;
55  wire [3:0]dig0;
56  wire [7:0]smg0;
57  always @(posedge clk_100khz)
58  begin
59      sel <= `UD sel+1'b1;
60  end
61  seq_control seq_control_0
62  (
63      .sel(2'd3),
64      .key(temp_one),
65      .dig(dig0),
66      .smg(smg0)
67  );
68  wire [3:0]dig1;
69  wire [7:0]smg1;
70  seq_control seq_control_1
71  (
72      .sel(2'd2),
73      .key(temp_ten),
74      .dig(dig1),
75      .smg(smg1)
76  );
77  wire [3:0]dig2;
78  wire [7:0]smg2;
79  seq_control seq_control_2
80  (
81      .sel(2'd1),
82      .key(humidity_one),
83      .dig(dig2),
84      .smg(smg2)
85  );
86  wire [3:0]dig3;
87  wire [7:0]smg3;
88  seq_control seq_control_3
89  (
90      .sel(2'd0),
91      .key(humidity_ten),
92      .dig(dig3),
93      .smg(smg3)
94  );
95
```

```

96     always @(posedge clk_100khz)
97     begin
98     if(sel==2'b00)
99         dig <= `UD dig0;
100    else if(sel==2'b01)
101        dig <= `UD dig1;
102    else if(sel==2'b10)
103        dig <= `UD dig2;
104    else if(sel==2'b11)
105        dig <= `UD dig3;
106    end
107    always @(posedge clk_100khz)
108    begin
109        if(sel==2'b00)
110            smg <= `UD smg0;
111        else if(sel==2'b01)
112            smg <= `UD smg1;
113        else if(sel==2'b10)
114            smg <= `UD smg2;
115        else if(sel==2'b11)
116            smg <= `UD smg3;
117    end
118
119 endmodule
120

```

### 15.4.2 Input/output control

First, the sensor of DHT11 needs power on for 1s, the in\_out pin of fpga outputs trigger signal of 1s at high level first and then trigger signal of 18ms at low level. The output of the in\_out is then finished, which should convert into input state. All the data sent back from DHT11 starts from low level, and the type of the signal can be determined based on the time at high level.

In writing the code, several key parameters should first be set. The frequency of the system clock is 12MHz, the number of clock cycles needed for output of 1s at high level DELAY = 24'd1200000, and the number of clock cycles needed for output of 18ms at low level OUT = 24'd216000. When the measurement starts, a counter is designed to begin the counting. Control of the states is shown in the table below:

Time	Input/output	Level
0-1s	Output	High
1s to 1s+18ms	Output	Low
After 1s+18ms	Input	



```
1  /*=====
2      Input/output control of the inout pin
3  =====*/
4      parameter DELAY = 24'd12000000; // Delay of 1s after the key is pressed down
5      parameter OUT = 24'd216000; // Output of trigger signal of 18ms at low level
6      18*12*1000=216000
7      reg [23:0]count_ctl=24'd0; // Timing of output signal
8      always @ (posedge clk) // Timing of 1s+18ms
9      begin
10         if(!rstn)
11             count_ctl <= `UD 24'd0;
12         else if(count_ctl < DELAY + OUT)
13             count_ctl <= `UD count_ctl + 24'b1;
14         else
15             count_ctl <= `UD count_ctl;
16     end
17     reg control=1'b0;
18     always @ (posedge clk) // Control signal for input/output signals
19     begin
20         if(!rstn)
21             control <= `UD 1'b0; // Condition for output
22         else if(count_ctl==OUT + DELAY) // Input state after output of 1s at high level + 18ms
23             at low level
24             control <= `UD 1'b1; // Condition for input
25         else
26             control <= `UD control;
27     end
28     reg out_reg=1'b0;
29     always @ (posedge clk) // Output signal
30     begin
31         if(!rstn)
32             out_reg <= `UD 1'b1; // First, output of 1s at high level after the key is pressed down
33         else if(count_ctl==DELAY) // Then, output of trigger signal of 18ms at low level 1s after
34             the key is pressed down
35             out_reg <= `UD 1'b0;
36         else if(count_ctl==OUT + DELAY)
37             out_reg <= `UD 1'b1;
38         else
39             out_reg <= `UD out_reg;
40     end
41     // Input/output control of inout: first, output of the 1s at high level, and then output
42     // of 18ms at low level
43     // High resistance Z during the input, with control being "0"
44     // Control is "1" 1s+18ms afterwards, in a input state of high resistance
45     assign in_out = (control == 1'b0)? out_reg:1'bz;
46
```

### 15.4.3 Data reading

After in\_out enters input state, it waits for acknowledge signal and data signal from the DHT11 sensor. A counter is designed, which is reset when the input of in\_out is at low level. When the input

of in\_out is at high level, the counter begins counting from 0 till the falling edge of input signal comes.

The type of the signal is determined based on the counting of the cnt during the time at high level, as shown in the table below. A marking signal data\_en is designed, which is 1 after the acknowledge signal comes. The data received afterwards is valid data and is stored. The design of the condition for determination of the signal type should allow a wide scope for the determination, provided that the signals can be distinguished.

Signal type	Time at high level	Number of clock cycles	Condition
Acknowledge signal	80us	960	>900
“0” bit of the data	26-28us	312-336	<360
“1” bit of the data	70us	840	>800

```

1      always @ (posedge clk) // Timing begins when the input signal is at high level, which is 0
      when the signal is at low level
2      begin
3          if(!rstn)
4              cnt <= `UD 12'b0;
5          else if(in == 1'b1)
6              cnt <= `UD cnt + 12'b1;
7          else
8              cnt <= `UD 12'b0;
9      end
10
11 // Delay for a cycle, which is used to determine whether the input signal changes from high level to
      low level
12      always @ (posedge clk)
13      begin
14          cnt1 <= `UD cnt;
15      end
16
17 // The counter for timing of high level is delayed for a clock cycle again for assignment to cnt2,
      one of the conditions for assignment for data info      always @ (posedge clk)
18      begin
19          cnt2 <= `UD cnt1;
20      end
21
22
    
```

```

23  /*=====
24      The signal for determination is ack, "0" or "1"
25  =====*/
26  reg data_en=1'b0; // The signal mark is read after the acknowledge signal comes
27  reg data=1'b0; // The current data is "1" or "0"
28  always @ (posedge clk)
29  begin
30      if(!rstn)
31          data_en <= `UD 1'b0;
32      else if(cnt1 > cnt) // Mark of transition of the input signal from high level to low level
33      begin
34          if( cnt1>900) // Acknowledge signal at high level for 80us: 80*12=960clk
35              data_en <= `UD 1'b1;
36      end
37      else
38          data_en <= `UD data_en;
39  end
40  always @ (posedge clk)
41  begin
42      if(cnt1 > cnt && data_en) // Transition of the input signal from high level to low level
43      // after the acknowledge signal comes
44      begin
45          if(cnt1<360) // Data "0" at high level for 26-28us,
46              data <= `UD 1'b0;
47          if( cnt1>800) // Data "1" at high level for 70us,
48              data <= `UD 1'b1;
49      end
50      else
51          data <= `UD 1'b0;
52  end

```

**Process of storage of the 40-bit data**

The high-order bit is output first when DHT11 outputs data. The low-order 39th bit in the 40-bit data info is first shifted left to the high-order 39th bit, the current data is stored in the lowest bit of the info, and then the next data is assigned to the lowest bit after the info data is shifted left.

No.	39	38	37	.....	.....	2	1	0		Data
Data	0	0	0	0	0	0	0	1	←	1
	↙							↙		
	0	0	0	0	0	0	1	0	←	0
	↙							↙		
	0	0	0	0	0	1	0	1	←	1

```

1  /*=====
2      Assignment of each bit of the data to info[40] one by one
3  =====*/
4      reg [7:0]data_cnt = 8'd40; // Data counting of the sensor
5      always @ (posedge clk)
6      begin
7          if(!rstn)
8              data_cnt <= `UD 8'd40;
9          else if(cnt1 > cnt && data_en)
10             data_cnt <= `UD data_cnt - 8'b1;
11         else if(data_cnt ==8'd0) // Mark of counting of the 40 bits to the maximum
12             data_cnt <= `UD data_cnt;
13     end
14
15     reg [39:0]info =40'b0; // The sensor signal is first output from the high-order bit, and the
assignment begins from the low-order bit and shifts to the high-order bit gradually.
16     always @ (posedge clk)
17     begin
18         if(!rstn)
19             info <= `UD 40'b0;
20         else if(cnt2 > cnt1 && data_en)
21             info <= `UD {info[38:0],data};
22     end
23

```

The low-order 8 bits of the 40-bit data are the parity bit. If the low-order 8 bits of the sum of the 4 sets of 8-bit data is the same as the parity bit, the data passes the parity check.

```

1  /*=====
2      Calculation of the parity bit, and assignment of the data to the output after the parity check
is passed
3  =====*/
4      reg [7:0]check = 8'd0; // The parity bit equals the low-order 8 bits of the sum of the 4 40-bit data
5      always @ (posedge clk)
6      begin
7          if(data_cnt ==8'd0)
8              check <= `UD info[39:32]+info[31:24]+info[23:16]+info[15:8];
9      end
10     always @ (posedge clk)
11     begin
12         if(!rstn)
13             information[31:0] <= `UD 32'h0;
14         else if(data_cnt ==8'd0)
15             begin
16                 if(check[7:0]==info[7:0])
17                     information[31:0] <= `UD info[39:8]; // Assignment after the parity check is
passed
18             end
19     end

```

#### 15.4.4 Data conversion

After the data is received, the values of temperature and humidity correspond to a 8-bit data in

binary system, which needs to be converted to data in decimal system to be displayed on the numeric display; the ones place and tens place of the data in decimal system should be stored in 2 registers.

As the sensor has a measurement accuracy of 1, the fractional part can be ignored; the scope of the measurement is within 100. The temperature and humidity can each be displayed on 2 numeric displays. The number on the tens place can be determined with the scope of the data, and the number on the ones place can be determined by subtracting ten times of the number on the tens place from the data.

Scope of the data	0-9	10-19	.....	80-89	90-99
Tens place	0	1	.....	8	9
Ones place	integer-0	integer-10	.....	integer-80	integer-90

```

1  module change(
2      input clk,
3      input rstn,
4      input  [31:0]information, // Read the high-order 32 bits of the 40-bit data
5      output reg [3:0]humidity_one=0, // Humidity
6      output reg [3:0]humidity_ten=0,
7      output reg [3:0]humidity_decimal,
8      output reg [3:0]temp_one=0, // Temperature
9      output reg [3:0]temp_ten=0,
10     output reg [3:0]temp_decimal
11 );
12
13 reg [7:0]humidity_integer=0;
14 reg [7:0]temp_integer=0;
15
16 always @ (posedge clk)
17 begin
18     humidity_integer[7:0] <= `UD information[31:24]; // Integer part of the humidity
19 end
20 always @ (posedge clk)
21 begin
22     humidity_decimal[3:0] <= `UD information[19:16]; // Fractional part of the humidity
23 end
24

```





```
25     always @(posedge clk)
26     begin
27         if(humidity_integer < 8'd10)
28             begin
29                 humidity_one <= `UD humidity_integer[3:0];
30                 humidity_ten <= `UD 4'd0;
31             end
32         else if(humidity_integer < 8'd20)
33             begin
34                 humidity_one <= `UD humidity_integer - 8'd10;
35                 humidity_ten <= `UD 4'd1;
36             end
37         else if(humidity_integer < 8'd30)
38             begin
39                 humidity_one <= `UD humidity_integer - 8'd20;
40                 humidity_ten <= `UD 4'd2;
41             end
42         else if(humidity_integer < 8'd40)
43             begin
44                 humidity_one <= `UD humidity_integer - 8'd30;
45                 humidity_ten <= `UD 4'd3;
46             end
47         else if(humidity_integer < 8'd50)
48             begin
49                 humidity_one <= `UD humidity_integer - 8'd40;
50                 humidity_ten <= `UD 4'd4;
51             end
52         else if(humidity_integer < 8'd60)
53             begin
54                 humidity_one <= `UD humidity_integer - 8'd50;
55                 humidity_ten <= `UD 4'd5;
56             end
57         else if(humidity_integer < 8'd70)
58             begin
59                 humidity_one <= `UD humidity_integer - 8'd60;
60                 humidity_ten <= `UD 4'd6;
61             end
62         else if(humidity_integer < 8'd80)
63             begin
64                 humidity_one <= `UD humidity_integer - 8'd70;
65                 humidity_ten <= `UD 4'd7;
66             end
67         else if(humidity_integer < 8'd90)
68             begin
69                 humidity_one <= `UD humidity_integer - 8'd80;
70                 humidity_ten <= `UD 4'd8;
71             end
72         else if(humidity_integer < 8'd100)
73             begin
74                 humidity_one <= `UD humidity_integer - 8'd90;
75                 humidity_ten <= `UD 4'd9;
76             end
77     end
```



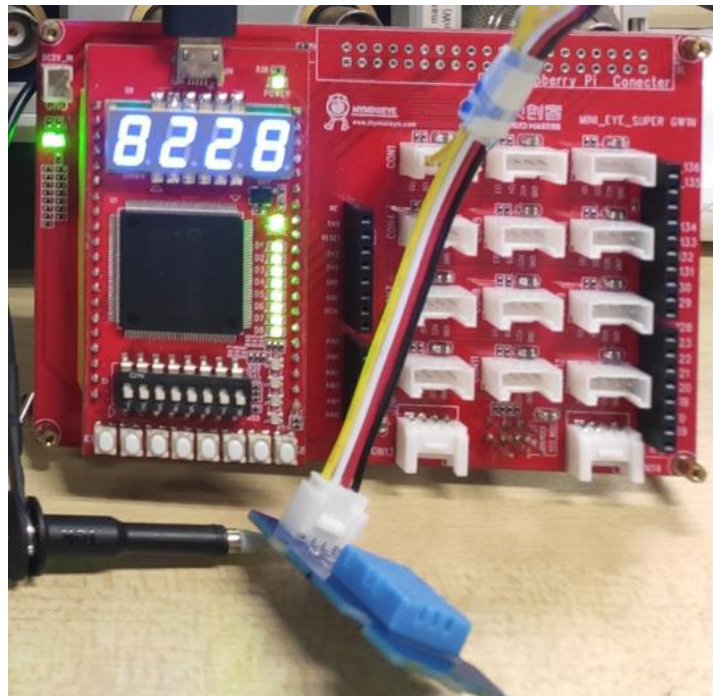
```
78     always @ (posedge clk)
79     begin
80         temp_integer[7:0] <= `UD information[15:8]; // Integer part of the temperature
81     end
82     always @ (posedge clk)
83     begin
84         temp_decimal[3:0] <= `UD information[3:0]; // Fractional part of the temperature
85     end
86
87     always @(posedge clk)
88     begin
89         if(temp_integer < 8'd10)
90         begin
91             temp_one <= `UD temp_integer[3:0];
92             temp_ten <= `UD 4'd0;
93         end
94         else if(temp_integer < 8'd20)
95         begin
96             temp_one <= `UD temp_integer - 8'd10;
97             temp_ten <= `UD 4'd1;
98         end
99         else if(temp_integer < 8'd30)
100        begin
101            temp_one <= `UD temp_integer - 8'd20;
102            temp_ten <= `UD 4'd2;
103        end
104        else if(temp_integer < 8'd40)
105        begin
106            temp_one <= `UD temp_integer - 8'd30;
107            temp_ten <= `UD 4'd3;
108        end
109        else if(temp_integer < 8'd50)
110        begin
111            temp_one <= `UD temp_integer - 8'd40;
112            temp_ten <= `UD 4'd4;
113        end
114        else if(temp_integer < 8'd60)
115        begin
116            temp_one <= `UD temp_integer - 8'd50;
117            temp_ten <= `UD 4'd5;
118        end
119        else if(temp_integer < 8'd70)
120        begin
121            temp_one <= `UD temp_integer - 8'd60;
122            temp_ten <= `UD 4'd6;
123        end
124        else if(temp_integer < 8'd80)
125        begin
126            temp_one <= `UD temp_integer - 8'd70;
127            temp_ten <= `UD 4'd7;
128        end
129    end
```

```

129     else if(temp_integer < 8'd90)
130     begin
131         temp_one <= `UD temp_integer - 8'd80;
132         temp_ten<=`UD 4'd8;
133     end
134     else if(temp_integer < 8'd100)
135     begin
136         temp_one <= `UD temp_integer - 8'd90;
137         temp_ten<=`UD 4'd9;
138     end
139 end
140
141 endmodule
142

```

### 15.5 Result of the experiment



Result of the experiment: the system can drive the temperature and humidity sensor effectively, and the measured values of temperature and humidity are displayed on the numeric display.