南京大学

《数字电路与数字系统实验》实验报告

实验十： 音频输出实验

姓名： 毛彦杰

学号： 191220081

班级：数字电路与数字系统实验 2 班

院系： 计算机科学与技术系

邮箱： 1363818182@qq.com

实验时间： 2020/11/30

预习报告：

一、音频输出原理

人耳可以听到的声音的频率范围是 20-20kHz。音频设备如扬声器或耳机等所接收的音频信号一般是模拟信号，即时间上连续的信号。但是，由于数字器件只能以固定的时间间隔产生数字输出，我们需要通过数字/模拟转换将数字信号转换成模拟信号输出。根据采样定律，数字信号的采样率（每秒钟产生的数字样本数量）应不低于信号频率的两倍。所以，数字音频一般采用 44.1kHz(CD 音频) 或 48kHz 的采样率，以保证 20kHz 的信号不会失真。

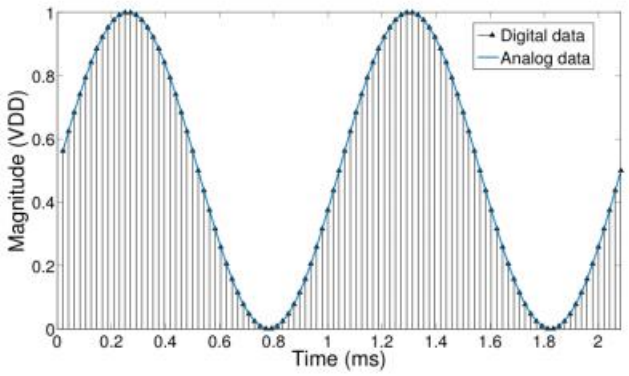

图 10-1: 数字信号到模拟信号的转换

图 10-1 显示了数字信号到模拟信号转换的基本原理。在 48kHz 的采样率下，我们每秒输出 48000 次，即每间隔 1/48000 秒（1/48 毫秒）的时间产生一个数字输出样本点，如图中黑色小三角所示。该输出经过平滑后，会产生一个对应的模拟信号。

假设我们需要产生一个人耳能听到的单频率 f = 960Hz 的正弦波信号，我们需要在合适的时间点上设置（或输出）合适的数字值来形成正弦波形。对于一个正弦波信号 s(t)，其数学表达式是：

$$s(t) = \sin(2\pi f t) \qquad (10-1)$$

其中 f 为频率，t 为时间。在数字信号中，我们用整数 n 来标记各个数字样本，样本编号顺序依次为 n = 0, 1, 2, 3, 4, ...。当采样率是 48kHz 时，每两个点之间的间隔是 1/48 毫秒。此时，我们可以将 t 改写成 t = n/48000 秒。这样式 (10-1)就变成：

$$s(n) = \sin(2\pi f n/48000) \qquad (10-2)$$

所以，对于整数 n 来说，每 50 个点对应正弦波的一个周期。仔细观察图 10-1 中的 960Hz 正弦波可知，其周期为 1/960 秒（1.042 毫秒），对应 50 个样本点。

在实际信号输出时，我们一般不采用浮点数而选用整数值来表示每个样本点的大小。这个过程称为量化 (Quantization)。假设我们用带符号的 16bit 整数（补码）来表示单个样本点，此时 32767 即对应输出的最大值（例如 +1V 电压），-32768 即对应输出的最小值（例如-1V 电压）。这时，我们就可以通过循环输出 50 个点的整数值 $\bar{s}(n) = \text{round}(s(n) \times 32767)$ 来产生一个 sin 波形，如表 10-1 所示：

表 10-1: 960Hz 数字信号示例

n	0	1	2	3	...	48	49	50	51
$s(n)$	0	0.125	0.249	0.368	...	-0.249	-0.125	0	0.125
$\bar{s}(n)$	0	4107	8149	12062	...	-8149	-4107	0	4107

预习任务

我们可以采用定点小数的方式，即用 16bit 来表示 k。其中前 10bit 是整数部分，用来查三角函数表，后面 6bit 是小数部分，用来提高精度。这时，如果将此 16bit 数看成是一个无符号整数的话，每个周期是 65536,而对应前 10 比特循环的周期是 1024 点。因此，n 每增加 1，我们需要 k 递增 960×65536/48000 = 1311，对应的小数值是 1311/64 = 20.4843。从整数的角度来看，n 变化 50 个样本点时 k 递增了 1311 × 50 = 65550，这个值略大于 65536 一些，会对周期带来一些小的误差，但是这样的误差对于人耳来说是可以容忍的。

1.根据频率 f 计算递增值 d = f×65536/48000

2. 在系统中维持一个 16bit 无符号整数计数器，每个样本点递增 d。

3. 根据 16 位无符号整数计数器的高 10 位来获取查表地址 k，并查找 1024 点的正弦函数表。

4. 使用查表结果作为当前的数字输出。

实验报告：

10.3 音频输出

一、实验目的

将之前实验实现的键盘与本实验的音频输出结合，实现一个简单的键盘电子琴功能。钢琴上的不同音高对应着不同的频率。我们可以根据按下的键的键值，决定播放的正弦的频率，从而实现电子琴的功能。

二、实验原理：音频输出原理

I2C 接口

模块 I2C_Controller 为开发板提供的 I2C 接口代码。调用一次该代码可以发送单个命令（将特定数据写入一个寄存器）。该接口通过 GO 信号指示进行发送（GO 需要在发送期间保持高电平），END 信号拉高说明发送结束。CLOCK 可以使用 10kHz 的时钟，I2C_DATA 是上层模块提供的待发送 24bit 数据，包含芯片地址、读写位、寄存器地址、寄存器值等。接口中可以添加信号 SD_COUNTER 和 SDO 用于测试模块内部的状态。ACK 信号返回发送后从节点的 3 个 ACK 比特。

```
module I2C_Controller (
        CLOCK,
        I2C_SCLK,//I2C CLOCK
        I2C_SDAT,//I2C DATA
        I2C_DATA,//DATA:[SLAVE_ADDR,SUB_ADDR,DATA]
        GO,      //GO transfor
        END,     //END transfor
```

```
        ACK,      //ACK
        RESET_N,
        //TEST
        //SD_COUNTER,
        //SDO
);
```

模块 I2C_Audio_Config 是一个简单的音频芯片配置示例。其在每次 reset 后根据预置的命令顺序配置音频芯片。该配置设置了正常音量（0dB),使用 48kHz 双声道每个声道 16bit 的 I2S 通信设置,并且将 I2S 的 XCLK 设置为 18.432MHz（48kHz 的 384 倍）。

```
module I2C_Audio_Config ( clk_i2c,
                          reset_n,
                          I2C_SCLK,
                          I2C_SDAT);
  parameter  total_cmd = 9; //sending 9 commands

  input clk_i2c;  //10k I2C clock
  input reset_n;
  output I2C_SCLK;
  inout  I2C_SDAT;

  reg [23:0] mi2c_data;
  reg  mi2c_go;
  wire mi2c_end;
  reg  [1:0] mi2c_state; //state 0: stop, state 1: send next;
                         //state 2: wait for finish, state 3:move index
  wire [2:0] mi2c_ack;
  wire [7:0] audio_addr;
  reg [3:0] cmd_count;
  reg [6:0] audio_reg [15:0]; //register to write
  reg [8:0] audio_cmd [15:0]; //register content
```

I2S 接口

在完成对 WM8731 的配置后，我们就可以通过 I^2S 接口来发送和接收数字音频信号了。I^2S 接口包括 AUD_XCK, AUD_BCLK, AUD_DACDAT, AUD_DACLRCK, AUD_ADCDAT, AUD_ADCLRCK 等多条信号线。其中 AUD_XCK 为音频信号的基准时钟，AUD_BCLK 为音频数据每个比特同步时钟，AUD_DACDAT 为输出数字信号数据，AUD_DACLRCK 用于输出的左右声道同步。AUD_ADCDAT, AUD_ADCLRCK 是用于输入音频信号的，只有在录音时需要，我们这里不用。

音频信号的基准时钟 AUD_XCK 一般设置为采样频率的 256 倍或者 384 倍。在我们的实验中，我们将采样频率设置为 48kHz，AUD_XCK 设为其 384 倍。因此，AUD_XCK 为 48000 × 384=18.432MHz。我们板上的标准时钟是 50MHz，如何能够产生 18.432MHz 的时钟呢？

这里我们需要调用 Quartus 提供的标准 IP 库来产生这类特殊的时钟。在 IP 库中选择 Library→Basic Functions→Clocks;PLLs and Resets→PLL→Altera PLL。

该 IP 核使用锁相环产生时钟，可以生成特殊频率。同其他 IP 核一样，设置 Verilog 文件名。请等待一些时间，直至 IP 核设置弹出，如图 10-6所示。我们选择 Fractional-N PLL，并使用 50MHz 时钟为输入，channel spacing 设为 1kHz，输出频率设置为 18.432M。这样就可以直接生成基准时钟 AUD_XCK 了。锁相环锁定信号 locked 可以不用处理。

在本实验中我们采用 I^2S 接口的 Left Justified（左对齐）的音频信号的传输模式，如图 10-7所示。在每一个样本点时间内，即 1/48 毫秒的时间内，我们需要传输 16bit 左声道数据和 16bit 右声道数据。因此，我们的 AUD_BCLK 需要的频率为 48000 × 32=1.536MHz，是基准频率 18.432MHz 的 12 分之一。所以 AUD_BCLK 可以用 AUD_XCK 计数分频获取。AUD_DACLRCK 确定当前传输的是左声道还是右声道，左声道为高电平，右声道为低电平。其频率为 48kHz，正好为 AUD_BCLK 的 32 分之一。此处需要注意的是，AUD_DACLRCK 和数据线 AUD_DACDAT 上的信号都是和 AUD_BCLK 的下降沿对齐的。

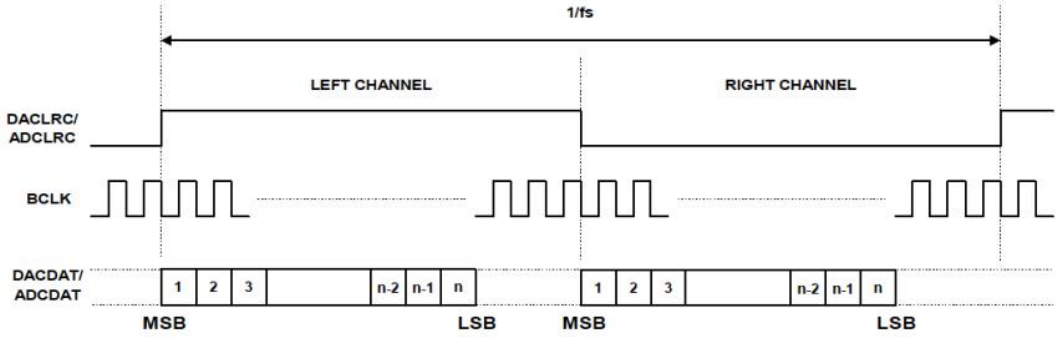

图 10-7: 左对齐的音频信号传输模式

在生成完时钟信号后，我们只需要按要求将每个样本点的 16bit 有符号整数数据按高位在前发送即可。对于我们的实验，可以将左右声道设置为一样的数据，实现单声道播放。

三、实验设备环境

硬件器材：FPGA 开发板、PS2 键盘、耳机

软件平台：Quartus 开发平台

四、实验步骤

设计思路：

通过上面的预习实验我已经基本了解了音频输出的工作原理。

此次实验我们可以分多个模块来实现：

首先我们得设计一个顶层实体，该实体里面的参数是生成音频信号需要的参数.

我们只需要用实例工程中的几个模块，生成 I2C 接口与 I2S 接口所需的变量。再加上我们的键盘来调整正弦查找表中的地址，就可以输出不同音调的音频了。可以得出如下设计代码：

设计代码：

顶层 sound_sample.v：

```
60  wire [15:0] frequency;
61
62
63  //=======================================================
64  //  Structural coding
65  //=======================================================
66
67  assign reset = ~KEY[0];
68
69  audio_clk u1(CLOCK_50, reset,AUD_XCK, LEDR[9]);
70
71  //mypart
72  keyboard k(CLOCK_50,SW[1],PS2_DAT,PS2_CLK,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,frequency);
73
74  //I2C part
75  clkgen #(10000) my_i2c_clk(CLOCK_50,reset,1'b1,clk_i2c);  //10k I2C clock
76
77
78  I2C_Audio_Config myconfig(KEY[3],clk_i2c, KEY[0],FPGA_I2C_SCLK,FPGA_I2C_SDAT,LEDR[2:0]);
79
80  I2S_Audio myaudio(AUD_XCK, KEY[0], AUD_BCLK, AUD_DACDAT, AUD_DACLRCK, audiodata);
81  //mypart
82  Sin_Generator sin_wave(AUD_DACLRCK, KEY[0], frequency, audiodata);//
83
84
85
86  endmodule
```

Sin_Generator.v：

```
1  module Sin_Generator(clk,
2                       reset_n,
3                       freq,
4                       dataout
5                       );
6   input clk;
7   input reset_n;
8   input [15:0] freq;
9   output reg [15:0] dataout;
10  (* ram_init_file = "sintable.mif" *) reg [15:0] sintable [1023:0]/*systhesis */;
11
12  reg [15:0] freq_counter; //16bit counter
13
14  always @(posedge clk) //change data at posedge of lrclk
15  begin
16          dataout <= sintable[freq_counter[15:6]]; // 10-bit address;
17  end
18
19
20  always @(posedge clk or negedge reset_n) //step counter
21  begin
22      if(!reset_n)
23         freq_counter <= 16'b0;
24      else
25         freq_counter <= freq_counter + freq;
26  end
27  endmodule
```

I2S_Audio.v:

```
module I2S_Audio(AUD_XCK,
                 reset_n,
                 AUD_BCK,
                 AUD_DATA,
                 AUD_LRCK,
                 audiodata);
 input AUD_XCK;
 input reset_n;
 output reg AUD_BCK;
 output AUD_DATA;
 output reg AUD_LRCK;
 input [15:0] audiodata;


 reg [7:0] bck_counter;
 reg [7:0] lr_counter;
 wire [7:0] bitaddr;

 //generate BCK 1.536MHz
 always @ (posedge AUD_XCK or negedge reset_n)
begin
       if(!reset_n)
       begin
          bck_counter <= 8'd0;
          AUD_BCK     <= 1'b0;
       end
       else
       begin
```

I2C_Audio_Config.v:

```
module I2C_Audio_Config ( change_vol,
                          clk_i2c,
                          reset_n,
                          I2C_SCLK,
                          I2C_SDAT,
                          testbit);
    parameter  total_cmd = 9;

    input clk_i2c;  //10k I2C clock
    input reset_n;
    input change_vol;

    output I2C_SCLK;
    output [2:0] testbit;
    inout  I2C_SDAT;

    reg [23:0] mi2c_data;
    reg  mi2c_go;
    wire mi2c_end;
    reg  [1:0] mi2c_state; //state 0: stop, state 1: sendnext;
                           //state 2: wait for finish, state 3:move index
    wire [2:0] mi2c_ack;
    wire [7:0] audio_addr;

    reg [3:0] cmd_count;
    reg [6:0] audio_reg [15:0]; //register to write
    reg [8:0] audio_cmd [15:0]; //register content
```

clkgen.v:

```
module clkgen(
    input clkin,
    input rst,
    input clken,
    output reg clkout
    );
    parameter clk_freq=1000;
    parameter countlimit=50000000/2/clk_freq; //
   //integer countlimit=8388;
   reg[31:0] clkcount;
   always @ (posedge clkin)
    if(rst)
     begin
        clkcount=0;
        clkout=1'b0;
     end
    else
    begin
     if(clken)
        begin
            clkcount=clkcount+1;
            if(clkcount>=countlimit)
             begin
                clkcount=32'd0;
                clkout=~clkout;
             end
            else
                clkout=clkout;
        end
```

ps2_keyboard. v:

```
module ps2_keyboard(clk,clrn,ps2_clk,ps2_data,data,ready,nextdata_n,overflow);
    input clk,clrn,ps2_clk,ps2_data;
    input nextdata_n;
    output [7:0] data;
    output reg ready;
    output reg overflow; // fifo overflow
    // internal signal, for test
    reg [9:0] buffer; // ps2_data bits
    reg [7:0] fifo[7:0]; // data fifo
    reg [2:0] w_ptr,r_ptr; // fifo write and read pointers
    reg [3:0] count; // count ps2_data bits
    // detect falling edge of ps2_clk
    reg [2:0] ps2_clk_sync;

    always @(posedge clk) begin
        ps2_clk_sync <= {ps2_clk_sync[1:0],ps2_clk};
    end

    wire sampling = ps2_clk_sync[2] & ~ps2_clk_sync[1];

    always @(posedge clk) begin
        if (clrn == 0) begin // reset
        count <= 0; w_ptr <= 0; r_ptr <= 0; overflow <= 0; ready<= 0;
        end
        else begin
            if ( ready ) begin // read to output next data
                if(nextdata_n == 1'b0) //read next data
                begin
                    r_ptr <= r_ptr + 3'b1;
```

keyboard. v:

```
keyboard.v | Compilation Report - sound_sample | sound_sample.SDC | sound_sample.v | Sin_Generator.v | I2S_Audi
            hex2 <= 7'b1111111;
            hex3 <= 7'b1111111;
            //mypart
            frequency = 0;
        end
        else begin
            hex0 <= int_to_seven(dout[3:0]);
            hex1 <= int_to_seven(dout[7:4]);
            //hex2 <= int_to_seven(asc[3:0]);
            //hex3 <= int_to_seven(asc[7:4]);
            //mypart
                case(dout)
                8'h1c: frequency = 179;
                8'h1b: frequency = 714;
                8'h23: frequency = 802;
                8'h2b: frequency = 900;
                8'h34: frequency = 954;
                8'h33: frequency = 1070;
                8'h3b: frequency = 1201;
                8'h42: frequency = 1349;
                default: frequency = 0;
                endcase
        end

        hex4 <= int_to_seven(count[3:0]);
        hex5 <= int_to_seven(count[7:4]);

    end
endmodule
```

在 keyboard 模块中根据键盘的扫描码设置查找表地址 frequency，从而得到相应的音频正弦信号。

激励代码：将使能端全部置为有效即可测试分频模块

引脚分配：

AUD_ADCDAT	Input	PIN_AJ29	5A	B5A_N0
AUD_ADCLRCK	Bidir	PIN_AH29	5A	B5A_N0
AUD_BCLK	Bidir	PIN_AF30	5A	B5A_N0
AUD_DACDAT	Output	PIN_AF29	5A	B5A_N0
AUD_DACLRCK	Bidir	PIN_AG30	5A	B5A_N0
AUD_XCK	Output	PIN_AH30	5A	B5A_N0
CLOCK2_50	Input	PIN_AA16	4A	B4A_N0
CLOCK3_50	Input	PIN_Y26	5B	B5B_N0
CLOCK4_50	Input	PIN_K14	8A	B8A_N0
CLOCK_50	Input	PIN_AF14	3B	B3B_N0
FPGA_I2C_SCLK	Output	PIN_Y24	5A	B5A_N0
FPGA_I2C_SDAT	Bidir	PIN_Y23	5A	B5A_N0
HEX0[6]	Output	PIN_AH18	4A	B4A_N0
HEX0[5]	Output	PIN_AG18	4A	B4A_N0
HEX0[4]	Output	PIN_AH17	4A	B4A_N0
HEX0[3]	Output	PIN_AG16	4A	B4A_N0
HEX0[2]	Output	PIN_AG17	4A	B4A_N0
HEX0[1]	Output	PIN_V18	4A	B4A_N0
HEX0[0]	Output	PIN_W17	4A	B4A_N0
HEX1[6]	Output	PIN_V17	4A	B4A_N0
HEX1[5]	Output	PIN_AE17	4A	B4A_N0
HEX1[4]	Output	PIN_AE18	4A	B4A_N0
HEX1[3]	Output	PIN_AD17	4A	B4A_N0
HEX1[2]	Output	PIN_AE16	4A	B4A_N0
HEX1[1]	Output	PIN_V16	4A	B4A_N0
HEX1[0]	Output	PIN_AF16	4A	B4A_N0
HEX2[6]	Output	PIN_W16	4A	B4A_N0

HEX2[5]	Output	PIN_AF18	4A	B4A_N0
HEX2[4]	Output	PIN_Y18	4A	B4A_N0
HEX2[3]	Output	PIN_Y17	4A	B4A_N0
HEX2[2]	Output	PIN_AA18	4A	B4A_N0
HEX2[1]	Output	PIN_AB17	4A	B4A_N0
HEX2[0]	Output	PIN_AA21	4A	B4A_N0
HEX3[6]	Output	PIN_AD20	4A	B4A_N0
HEX3[5]	Output	PIN_AA19	4A	B4A_N0
HEX3[4]	Output	PIN_AC20	4A	B4A_N0
HEX3[3]	Output	PIN_AA20	4A	B4A_N0
HEX3[2]	Output	PIN_AD19	4A	B4A_N0
HEX3[1]	Output	PIN_W19	4A	B4A_N0
HEX3[0]	Output	PIN_Y19	4A	B4A_N0
HEX4[6]	Output	PIN_AH22	4A	B4A_N0
HEX4[5]	Output	PIN_AF23	4A	B4A_N0
HEX4[4]	Output	PIN_AG23	4A	B4A_N0
HEX4[3]	Output	PIN_AE23	4A	B4A_N0
HEX4[2]	Output	PIN_AE22	4A	B4A_N0
HEX4[1]	Output	PIN_AG22	4A	B4A_N0
HEX4[0]	Output	PIN_AD21	4A	B4A_N0
HEX5[6]	Output	PIN_AB21	4A	B4A_N0
HEX5[5]	Output	PIN_AF19	4A	B4A_N0
HEX5[4]	Output	PIN_AE19	4A	B4A_N0
HEX5[3]	Output	PIN_AG20	4A	B4A_N0
HEX5[2]	Output	PIN_AF20	4A	B4A_N0
HEX5[1]	Output	PIN_AG21	4A	B4A_N0
HEX5[0]	Output	PIN_AF21	4A	B4A_N0
KEY[2]	Input	PIN_AA14	3B	B3B_N0
KEY[1]	Input	PIN_AK4	3B	B3B_N0
KEY[0]	Input	PIN_AJ4	3B	B3B_N0
LEDR[9]	Output	PIN_AC22	4A	B4A_N0
LEDR[8]	Output	PIN_AB22	5A	B5A_N0
LEDR[7]	Output	PIN_AF24	4A	B4A_N0
LEDR[6]	Output	PIN_AE24	4A	B4A_N0
LEDR[5]	Output	PIN_AF25	4A	B4A_N0
LEDR[4]	Output	PIN_AG25	4A	B4A_N0
LEDR[3]	Output	PIN_AD24	4A	B4A_N0
LEDR[2]	Output	PIN_AC23	4A	B4A_N0
LEDR[1]	Output	PIN_AB23	5A	B5A_N0
LEDR[0]	Output	PIN_AA24	5A	B5A_N0
PS2_CLK	Bidir	PIN_AB25	5A	B5A_N0
PS2_CLK2	Bidir	PIN_AC25	5A	B5A_N0
PS2_DAT	Bidir	PIN_AA25	5A	B5A_N0
PS2_DAT2	Bidir	PIN_AB26	5A	B5A_N0
SW[9]	Input	PIN_AA30	5B	B5B_N0
SW[8]	Input	PIN_AC29	5B	B5B_N0
SW[7]	Input	PIN_AD30	5B	B5B_N0
SW[6]	Input	PIN_AC28	5B	B5B_N0
SW[5]	Input	PIN_V25	5B	B5B_N0
SW[4]	Input	PIN_W25	5B	B5B_N0
SW[3]	Input	PIN_AC30	5B	B5B_N0
SW[2]	Input	PIN_AB28	5B	B5B_N0
SW[1]	Input	PIN_Y27	5B	B5B_N0
SW[0]	Input	PIN_AB30	5B	B5B_N0

五：实验过程中遇到的问题及解决

1.开始在 I2S 接口的信号分配上想了很久,不知道各个时钟信号的分配关系，后面得出结论：我们控制 CLOCK_50 即可。

2.忘记将 audio_cmd 高两位置零。

3.在调节音量的部分在 always 语块里使用两个 if 语句对 audio_cmd[3]和 audio_cmd[4]进行修改时编译报错，后来修改成一个语句内赋值就好了。另外在 I2C_Audio_Config 模块里设置 volume 变量会无法输出。应在顶层实体内设置变量 volume，再传入 I2C_Audio_Config 模块赋给 audio_cmd 这个 ram 就好了。

六、实验得到的启示

这次实验的难度又比之前的实验上升了许多，更具有挑战性了。

面对更大规模的项目，更需要我们坚持先规划好各个模块的大框架，再专门实现每一个小功能。自顶向下，逐步求精。并且在实现具体的小功能的时候考虑好使用的变量类型/宽度/个数，语句类型/函数/任务等等问题，用恰当的语句来实现适合的功能，构建数字电路建模的 first principle 思考体系。

机器总是对的。对于 Quatus 而言它自身不具备特别好的调试工具，所以应该巧妙运用更多自己的调试方法来对一个项目进行分模块的调试。

七、意见与建议

实验手册中的实例对学习 verilog 语句使用与 quartus 相关操作有巨大帮助,希望能在学习新知识的同时接触更多相关例子来加深对新知的理解与应用能力。

希望自己能加强对网上知识的搜索自学能力，这样可以迅速掌握新的知识。

非常感谢老师和主教哥哥们的指点！！谢谢!