2019-2020 春季《FPGA 设计》期末考察项目

乐曲硬件演奏电路设计报告

学号 ：2017141034　　姓名：梁辉鸿

日期：2020 年　5 月　15 日

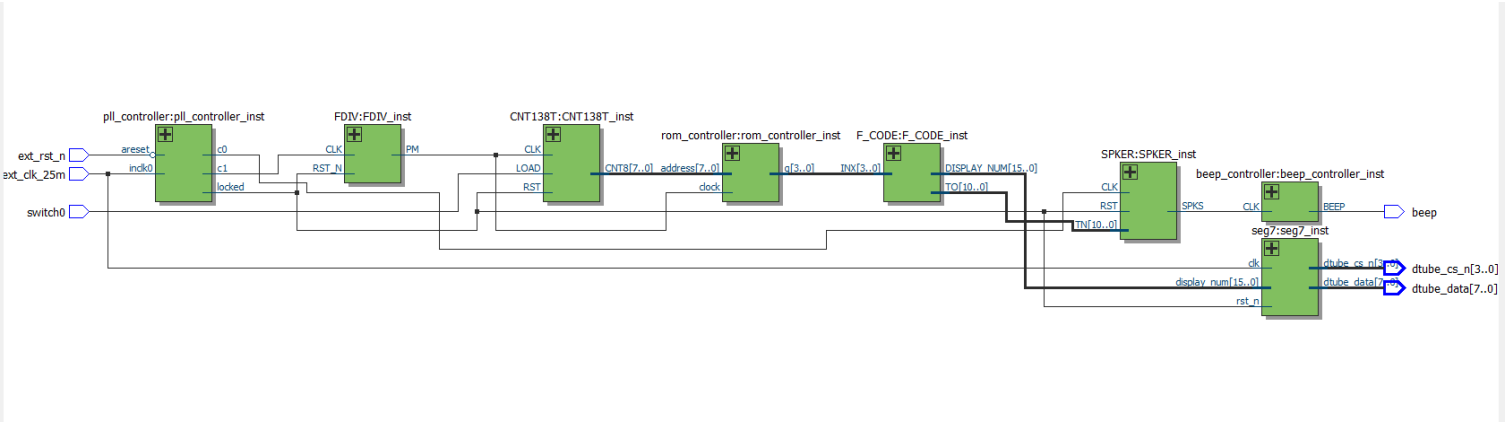
项目评分参考标准：

评分	参考分	实际得分
报告的完整性	5	
顶层 RTLView	5	
各模块说明	10	
各模块及联合测试模块的 RTLView	10	
各模块代码	10	
各模块仿真代码	10	
各模块仿真波形	10	
存储器内容测试	5	
演示视频	35	
选做题	25	
总分	125	

目录

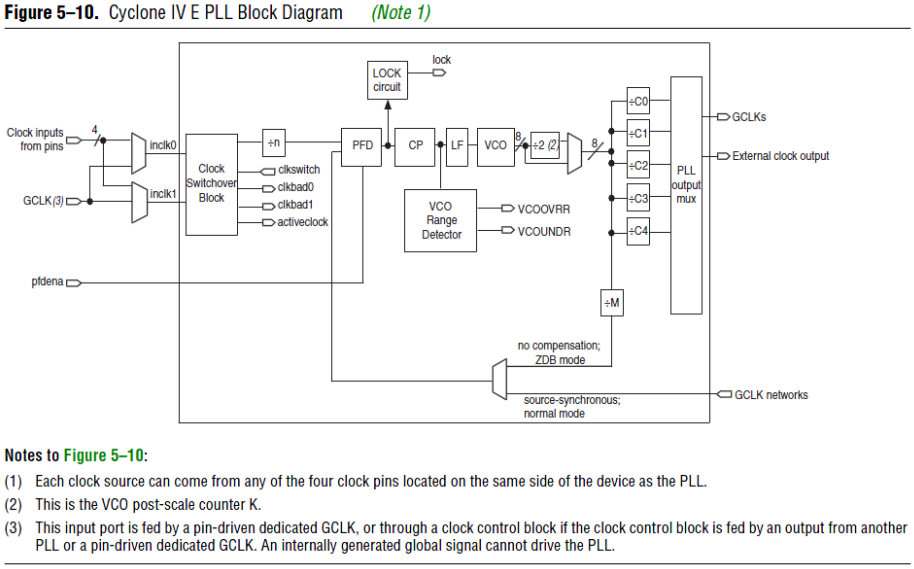
乐曲硬件演奏电路设计

1. 顶层模块 RTLView



2. IP 核—PLL 锁相环分频模块

2.1 原理说明



PLL(Phase Locked Loop)：为锁相回路或锁相环，属于集成的 IP 核，可直接调用并通过相应配置以产生不同的时钟分频或倍频，满足开发需求。在本实验中输入 25MHz 系统时钟并通过 PLL 产生 1MHz 和 2KHz 的时钟分频，分别用于 FDIV 分频电路模块和 SPKER 数控分频模块。

2.2 模块代码

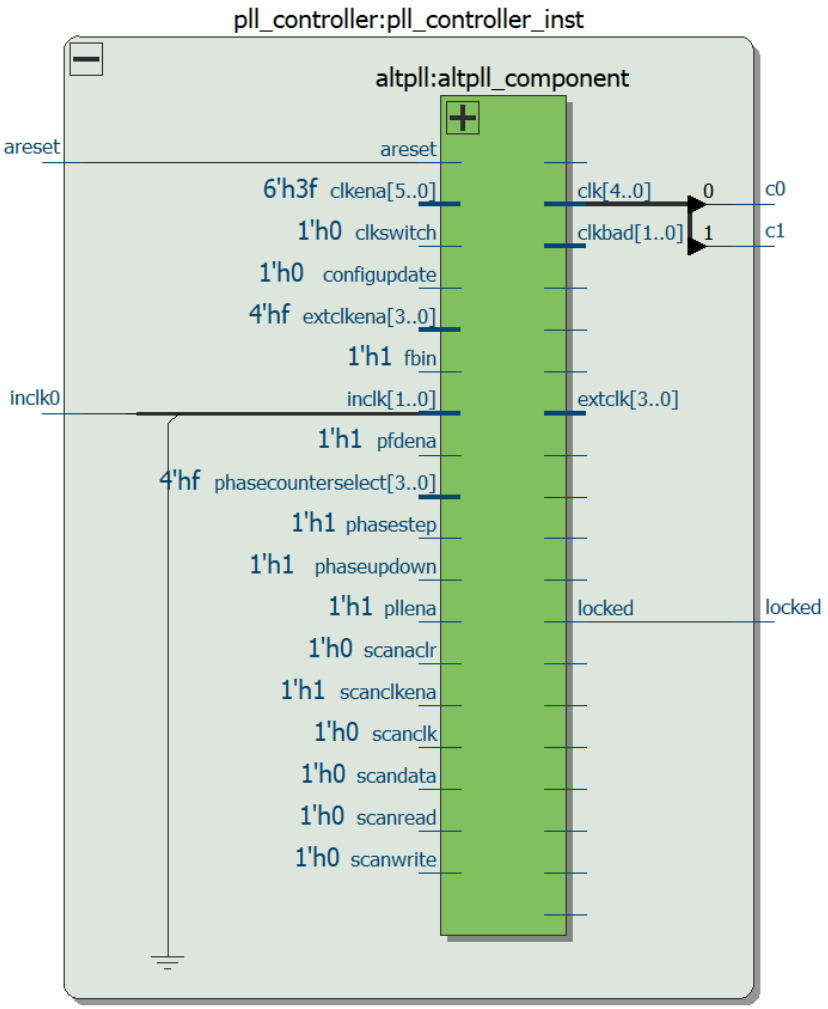
```
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module pll_controller (
    areset,
    inclk0,
    c0,
    c1,
    locked);

    input     areset;
    input     inclk0;
    output    c0;
    output    c1;
    output    locked;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri0      areset;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [4:0] sub_wire0;
    wire  sub_wire2;
    wire [0:0] sub_wire6 = 1'h0;
    wire [0:0] sub_wire3 = sub_wire0[0:0];
    wire [1:1] sub_wire1 = sub_wire0[1:1];
    wire  c1 = sub_wire1;
    wire  locked = sub_wire2;
    wire  c0 = sub_wire3;
    wire  sub_wire4 = inclk0;
    wire [1:0] sub_wire5 = {sub_wire6, sub_wire4};

    altpll  altpll_component (
                .areset (areset),
                .inclk (sub_wire5),
                .clk (sub_wire0),
                .locked (sub_wire2),
                .activeclock (),
                .clkbad (),
                .clkena ({6{1'b1}}),
```

```
                .clkloss (),
                .clkswitch (1'b0),
                .configupdate (1'b0),
                .enable0 (),
                .enable1 (),
                .extclk (),
                .extclkena ({4{1'b1}}),
                .fbin (1'b1),
                .fbmimicbidir (),
                .fbout (),
                .fref (),
                .icdrclk (),
                .pfdena (1'b1),
                .phasecounterselect ({4{1'b1}}),
                .phasedone (),
                .phasestep (1'b1),
                .phaseupdown (1'b1),
                .pllena (1'b1),
                .scanaclr (1'b0),
                .scanclk (1'b0),
                .scanclkena (1'b1),
                .scandata (1'b0),
                .scandataout (),
                .scandone (),
                .scanread (1'b0),
                .scanwrite (1'b0),
                .sclkout0 (),
                .sclkout1 (),
                .vcooverrange (),
                .vcounderrange ());
    defparam
        altpll_component.bandwidth_type = "AUTO",
        altpll_component.clk0_divide_by = 25,
        altpll_component.clk0_duty_cycle = 50,
        altpll_component.clk0_multiply_by = 1,
        altpll_component.clk0_phase_shift = "0",
        altpll_component.clk1_divide_by = 12500,
        altpll_component.clk1_duty_cycle = 50,
        altpll_component.clk1_multiply_by = 1,
        altpll_component.clk1_phase_shift = "0",
        altpll_component.compensate_clock = "CLK0",
        altpll_component.inclk0_input_frequency = 40000,
        altpll_component.intended_device_family = "Cyclone IV E",
```

```
        altpll_component.lpm_hint =
"CBX_MODULE_PREFIX=pll_controller",
        altpll_component.lpm_type = "altpll",
        altpll_component.operation_mode = "NORMAL",
        altpll_component.pll_type = "AUTO",
        altpll_component.port_activeclock = "PORT_UNUSED",
        altpll_component.port_areset = "PORT_USED",
        altpll_component.port_clkbad0 = "PORT_UNUSED",
        altpll_component.port_clkbad1 = "PORT_UNUSED",
        altpll_component.port_clkloss = "PORT_UNUSED",
        altpll_component.port_clkswitch = "PORT_UNUSED",
        altpll_component.port_configupdate = "PORT_UNUSED",
        altpll_component.port_fbin = "PORT_UNUSED",
        altpll_component.port_inclk0 = "PORT_USED",
        altpll_component.port_inclk1 = "PORT_UNUSED",
        altpll_component.port_locked = "PORT_USED",
        altpll_component.port_pfdena = "PORT_UNUSED",
        altpll_component.port_phasecounterselect = "PORT_UNUSED",
        altpll_component.port_phasedone = "PORT_UNUSED",
        altpll_component.port_phasestep = "PORT_UNUSED",
        altpll_component.port_phaseupdown = "PORT_UNUSED",
        altpll_component.port_pllena = "PORT_UNUSED",
        altpll_component.port_scanaclr = "PORT_UNUSED",
        altpll_component.port_scanclk = "PORT_UNUSED",
        altpll_component.port_scanclkena = "PORT_UNUSED",
        altpll_component.port_scandata = "PORT_UNUSED",
        altpll_component.port_scandataout = "PORT_UNUSED",
        altpll_component.port_scandone = "PORT_UNUSED",
        altpll_component.port_scanread = "PORT_UNUSED",
        altpll_component.port_scanwrite = "PORT_UNUSED",
        altpll_component.port_clk0 = "PORT_USED",
        altpll_component.port_clk1 = "PORT_USED",
        altpll_component.port_clk2 = "PORT_UNUSED",
        altpll_component.port_clk3 = "PORT_UNUSED",
        altpll_component.port_clk4 = "PORT_UNUSED",
        altpll_component.port_clk5 = "PORT_UNUSED",
        altpll_component.port_clkena0 = "PORT_UNUSED",
        altpll_component.port_clkena1 = "PORT_UNUSED",
        altpll_component.port_clkena2 = "PORT_UNUSED",
        altpll_component.port_clkena3 = "PORT_UNUSED",
        altpll_component.port_clkena4 = "PORT_UNUSED",
        altpll_component.port_clkena5 = "PORT_UNUSED",
        altpll_component.port_extclk0 = "PORT_UNUSED",
        altpll_component.port_extclk1 = "PORT_UNUSED",
```

```
        altpll_component.port_extclk2 = "PORT_UNUSED",
        altpll_component.port_extclk3 = "PORT_UNUSED",
        altpll_component.self_reset_on_loss_lock = "OFF",
        altpll_component.width_clock = 5;


endmodule
```

2.3 RTLView

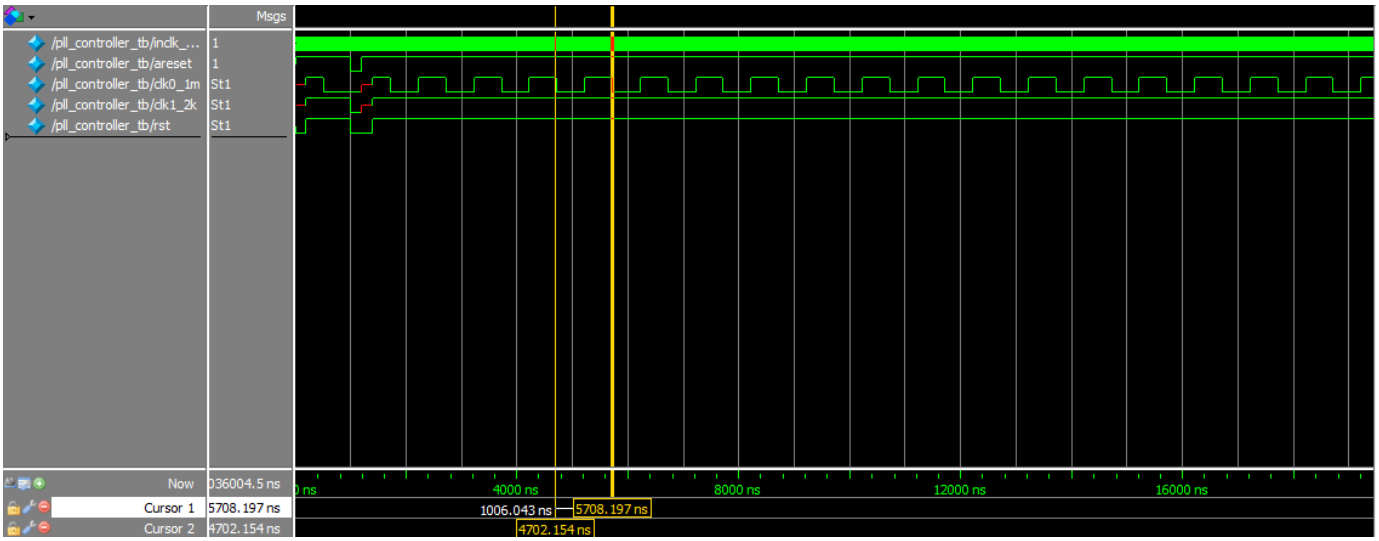


2.4 仿真代码

```
//pll 分频的测试模块
`timescale 1ns/1ns

module pll_controller_tb;
    reg inclk_25m;
```

```
    reg areset;
    wire clk0_1m;
    wire clk1_2k;
    wire rst;

    pll_controller
u1(.areset(!areset), .inclk0(inclk_25m), .c0(clk0_1m), .c1(clk1_2k)
, .locked(rst));
    initial
    begin
              inclk_25m=0;
              areset=1;
    #1000      areset=0;
    #200       areset=1;
    end

    always #20  inclk_25m=~inclk_25m;
endmodule
```

2.5 仿真波形



由波形可见，clk0_1m 端口的一个时钟周期为 1000ns，满足 1MHz 时钟分频的频率要求，而仿真中对于 2KHz 的时钟端口输出一直为高电平，多次排查找不出故障出在何处，但是在板级调试中 2KHz 分频的时钟输出是正常的。

3. FDIV—分频电路模块

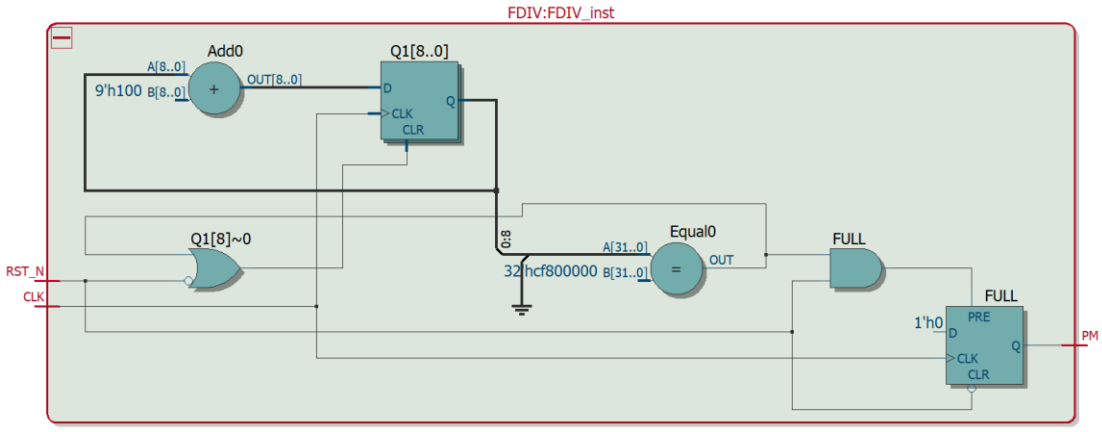
3.1 原理说明

此模块是一个分频模块，输入 2KHz 时钟，通过一个 9 位计数器实现分频得到一个频率为 4HZ 的分频时钟，满足后续设计乐曲中一拍子为 0.25 秒的要求。

3.2 模块代码

```
//分频电路模块——4Hz
module FDIV(CLK, PM, RST_N);
    input CLK;
    input  RST_N;
    output PM;
    reg[8:0] Q1;
    reg FULL;
    wire RST;

    always @(posedge CLK or posedge RST or negedge RST_N)
    begin
        if (!RST_N)
        begin
            Q1<=0; FULL<=0;
        end
        else if (RST)
        begin
            Q1<=0; FULL<=1;
        end
        else begin
            Q1<=Q1+1; FULL<=0;
        end
    end
    assign RST=(Q1==499);
    assign PM=FULL;

endmodule
```

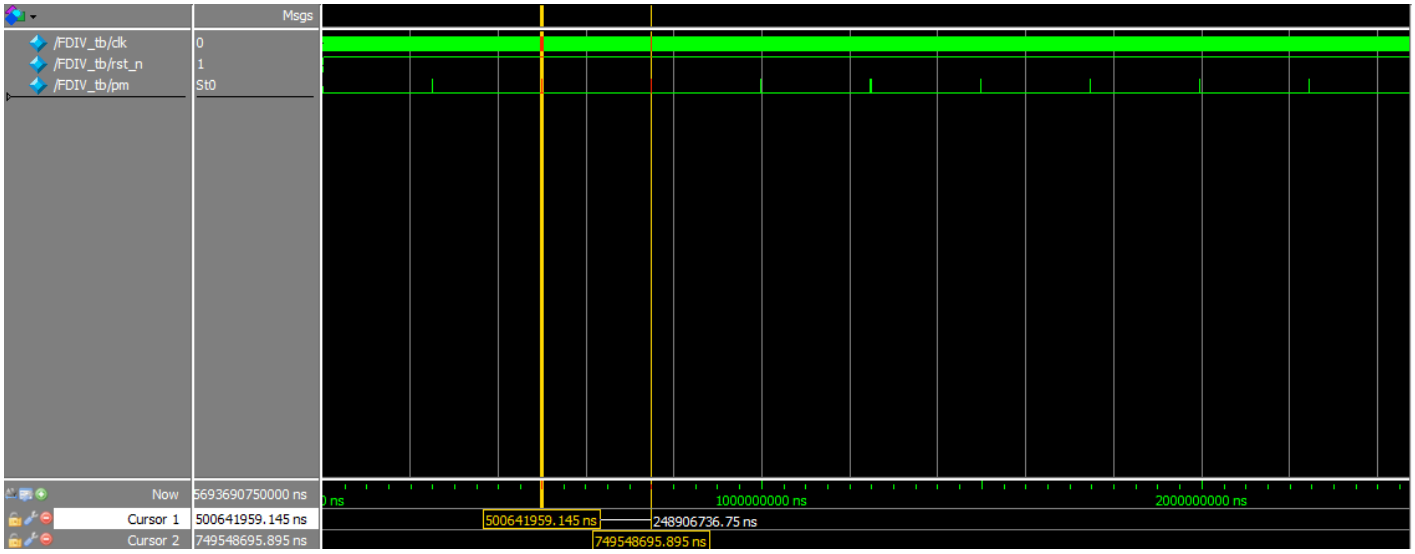
3.3 RTLView



3.4 仿真代码

```
//分频器的测试模块
`timescale 1ns/1ns

module FDIV_tb;
    reg clk;
    reg rst_n;
    wire pm;

    FDIV u1(.CLK(clk), .RST_N(rst_n), .PM(pm));
    initial
    begin
              clk=0;
              rst_n=1;
    #1000      rst_n=0;
    #1000      rst_n=1;
    end

    always #250000  clk=~clk;

endmodule
```

3.5 仿真波形



通过换算时钟周期可知分频后的时钟频率满足 4Hz 的要求。

4. CNT138T—乐曲长度控制计数器模块

4.1 原理说明

模块 CNT138T 是一个 8 位二进制计数器,内部设置计数最大值为 139，作为音符数据 ROM 的地址发生器。这个计数器的计数频率即为 4Hz，即每一计数值的停留时间为 0.25 秒，恰为当全音符设为 1 秒时，四四拍的 4 分音符持续时间。例如，“梁祝”乐曲的第一个音符为“3”，此音在逻辑中停留了 4 个时钟节拍，即 1 秒时间，相应地，所对应的“3”音符分频预置值为 11’ H40C，在 SPKER 的输入端停留了 1 秒。随着计数器 CNT138T 按 4Hz 的时钟速率作加法计数时，即随地址值递增时,音符数据 ROM 模块 MUSIC 中的音符数据将从 ROM 中通过 q[3:0]端口输向 F_CODE 模块，“梁祝”乐曲就开始连续自然地演奏起来了。CNT138T 的节拍是 139，正好等于 ROM 中的简谱码数，所以可以确保循环演奏。对于其他乐曲，此技术最大值要根据情况更改。

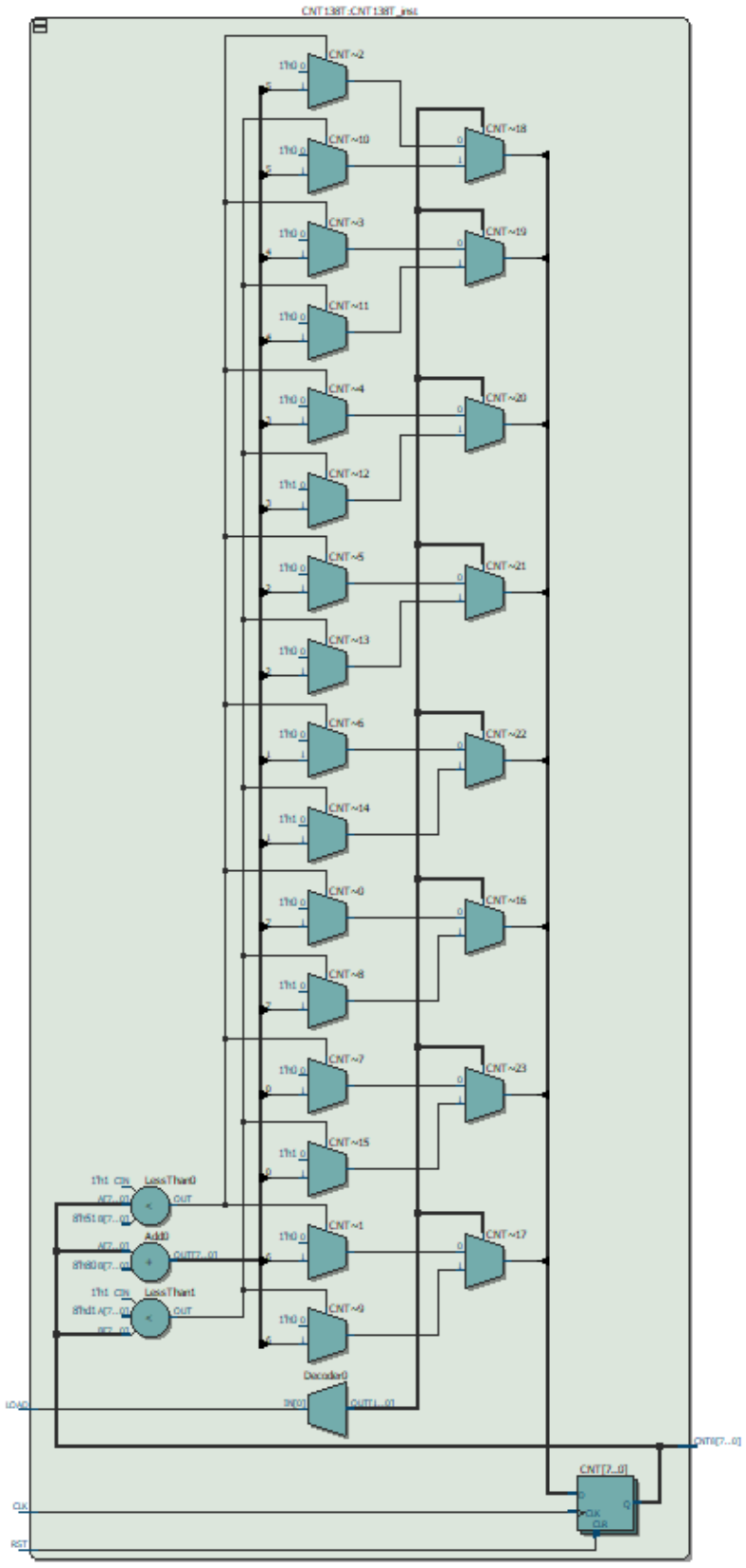
CNT138T 有一个 LOAD 端口，由拨码开关控制，若 LOAD=0，CNT 会从 0 加到 138，然后复 0 循环，即为播放第一首歌的过程；若 LOAD=1，CNT 会从 139 加到 255，然后回到 139 循环，即为选播第二首歌的过程。

4.2 模块代码

```
//乐曲长度控制计数器模块--音符数据 ROM 的地址发生器
module CNT138T(CLK, RST, LOAD, CNT8);
    input CLK;
```

```
    input RST;
    input LOAD;
    output[7:0] CNT8;
    reg[7:0] CNT;

    always @(posedge CLK or negedge RST)
    begin
        if (!RST) CNT<=8'b00000000;
        else
        begin
            case (LOAD)
                0:
                begin
                    if (CNT<=138) CNT<=CNT+1;
                    else CNT=0;
                end
                1:
                begin
                    if (CNT>=139&&CNT<=256) CNT<=CNT+1;
                    else CNT=139;
                end
            endcase
        end
    end
    assign CNT8=CNT;

endmodule
```

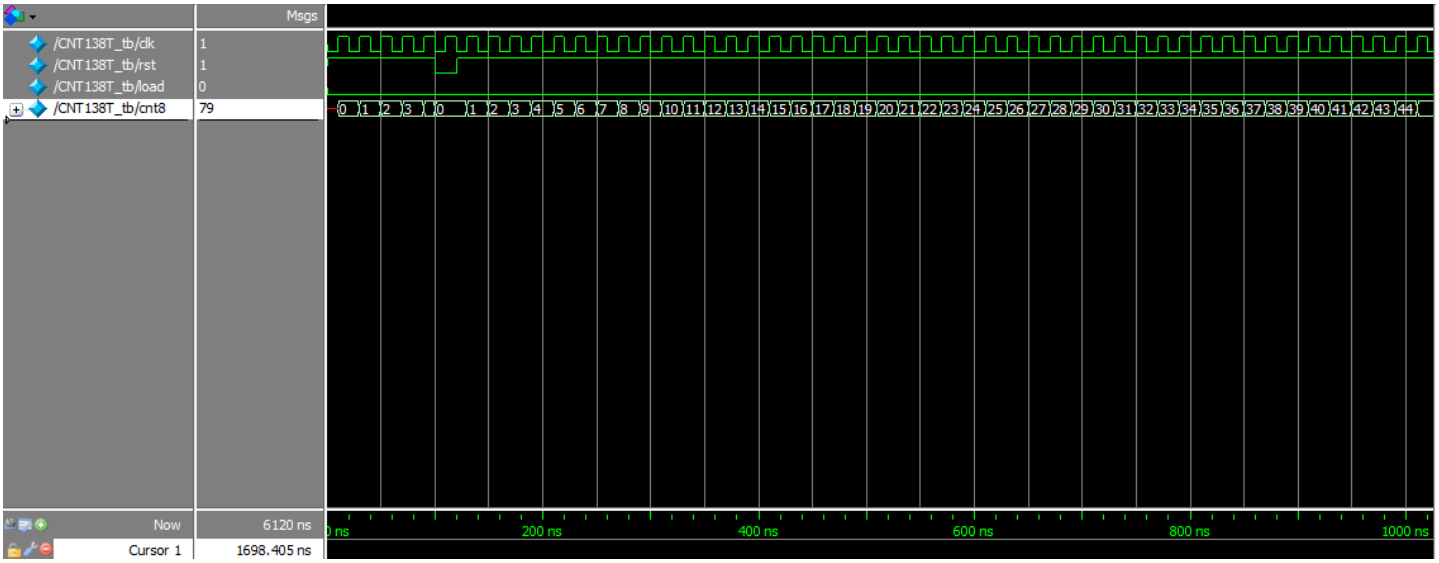
4.3 RTLView

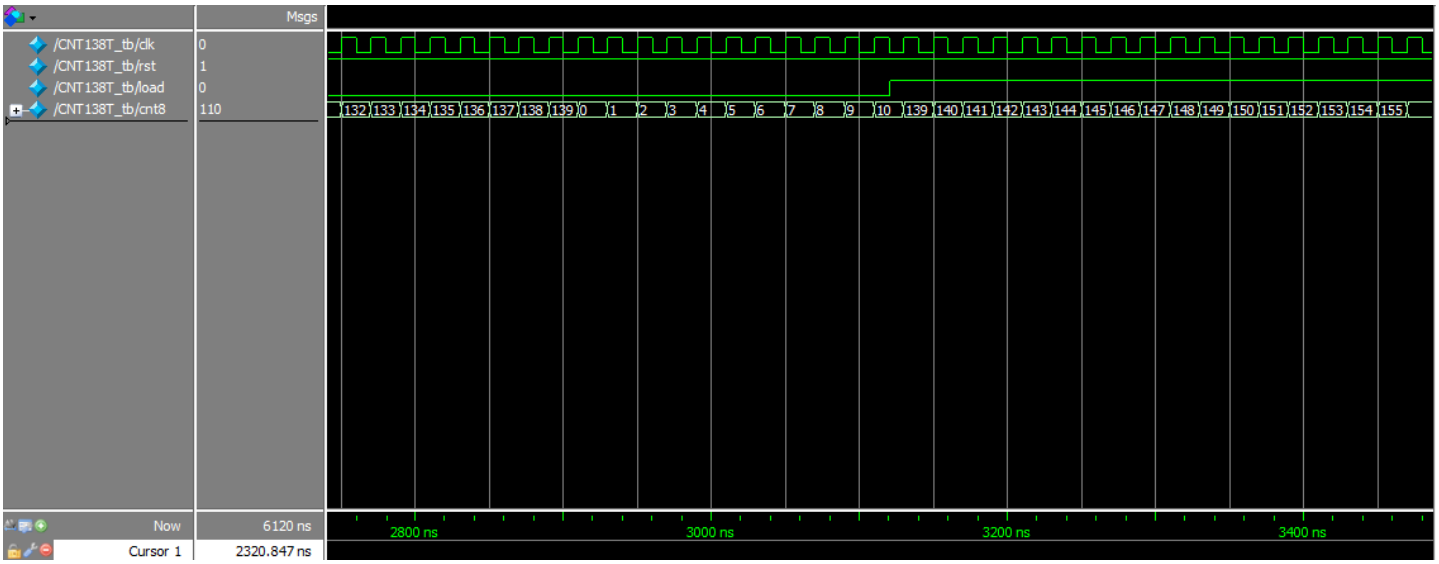

CNT138T:CNT138T_inst
CNT~2
CNT~10
CNT~3
CNT~11
CNT~4
CNT~12
CNT~5
CNT~13
CNT~6
CNT~14
CNT~0
CNT~8
CNT~7
CNT~15
CNT~1
CNT~9
CNT~18
CNT~19
CNT~20
CNT~21
CNT~22
CNT~16
CNT~23
CNT~17
LessThan0
Add0
LessThan1
Decoder0
CNT[7..0]
CLK
RST

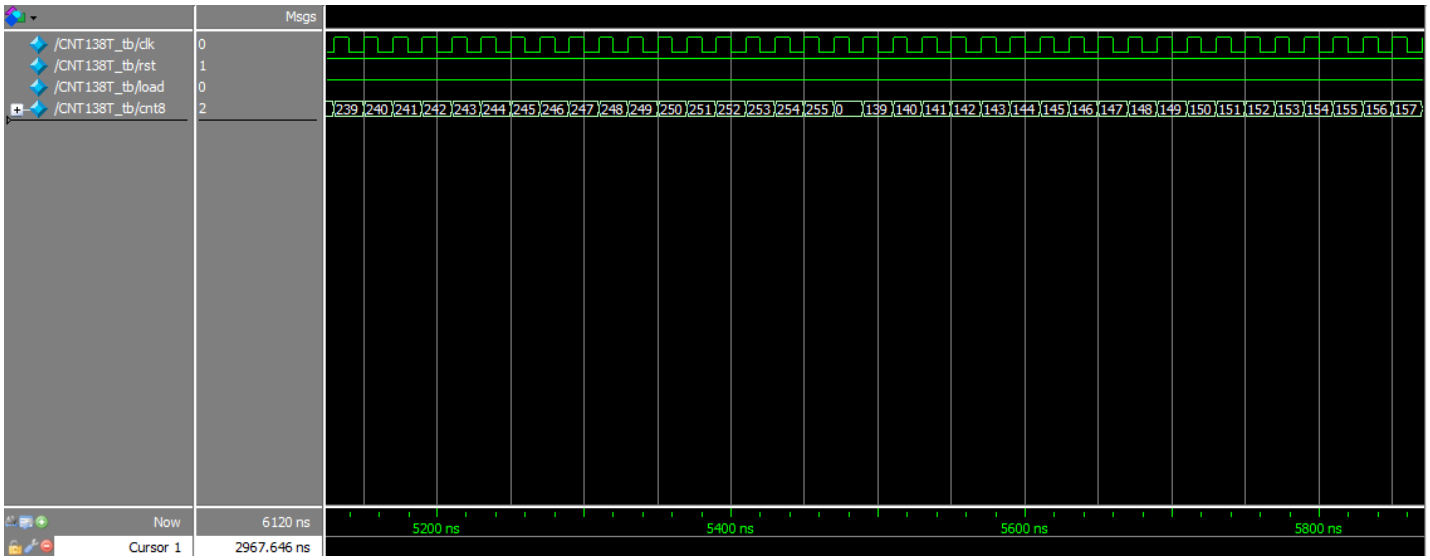
4.4 仿真代码

```
//ROM 地址发生器的测试模块
`timescale 1ns/1ns

module CNT138T_tb;
    reg clk;
    reg rst;
    reg load;
    wire[7:0] cnt8;

    CNT138T u1(.CLK(clk), .RST(rst), .LOAD(load), .CNT8(cnt8));
    initial
    begin
              rst=1;
              clk=0;
              load=0;
    #100       rst=0;
    #20     rst=1;
    #3000       load=1;
    #3000       $stop;
    end

    always #10  clk=~clk;

endmodule
```

4.5 仿真波形


Msgs
/CNT138T_tb/clk
/CNT138T_tb/rst
/CNT138T_tb/load
/CNT138T_tb/cnt8
1
1
0
79
Now
6120 ns
Cursor 1
1698.405 ns
200 ns
400 ns
600 ns
800 ns
1000 ns


Msgs
/CNT138T_tb/clk
/CNT138T_tb/rst
/CNT138T_tb/load
/CNT138T_tb/cnt8
0
1
0
110
Now
6120 ns
Cursor 1
2320.847 ns
2800 ns
3000 ns
3200 ns
3400 ns


Msgs
/CNT138T_tb/clk
/CNT138T_tb/rst
/CNT138T_tb/load
/CNT138T_tb/cnt8
0
1
0
2
Now
6120 ns
Cursor 1
2967.646 ns
5200 ns
5400 ns
5600 ns
5800 ns

根据波形可知 CNT138T 计数器可根据 load 端输入有选择地实现计数递增，并且达到最大值可复位循环计数，因此可满足音乐循环演奏和切换歌曲的要求。

5. IP 核—乐谱码 ROM 模块

5.1 原理说明

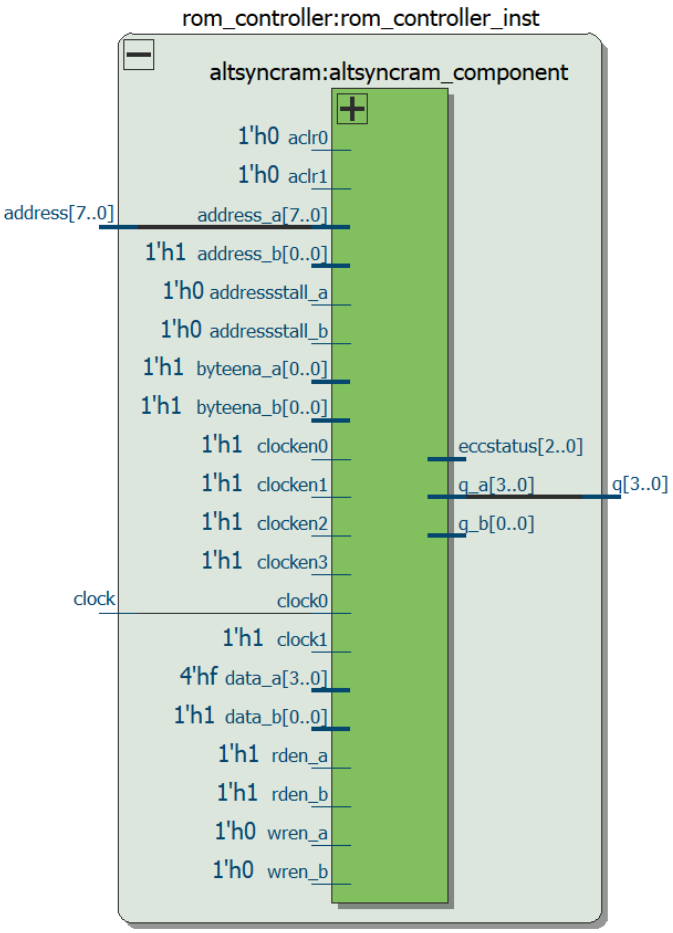
ROM 属于集成的 IP 核，可直接调用并存储所需的信息，可通过预置.mif 文件而向内存写入信息。在本实验中，Music ROM 存储数据宽度为 4 位，内存深度为 256，输入为频率 4Hz 的时钟，由 CNT138T 模块输出的计数结果作为 ROM 的地址，随着地址的递增相应地输出存储在 ROM 中的乐谱简码，并且作为后面 F_CODE 模块的预置数查表输入。

5.2 模块代码

```
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module rom_controller (
    address,
    clock,
    q);

    input   [7:0]  address;
    input     clock;
    output  [3:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1      clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [3:0] sub_wire0;
    wire [3:0] q = sub_wire0[3:0];

    altsyncram  altsyncram_component (
                .address_a (address),
                .clock0 (clock),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
```

```
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_a ({4{1'b1}}),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_a (1'b0),
                .wren_b (1'b0));
    defparam
        altsyncram_component.address_aclr_a = "NONE",
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file =
"../../source_code/rom_init.mif",
        altsyncram_component.intended_device_family = "Cyclone IV
E",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 256,
        altsyncram_component.operation_mode = "ROM",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "CLOCK0",
        altsyncram_component.ram_block_type = "M9K",
        altsyncram_component.widthad_a = 8,
        altsyncram_component.width_a = 4,
        altsyncram_component.width_byteena_a = 1;


endmodule
```

5.3 RTLView

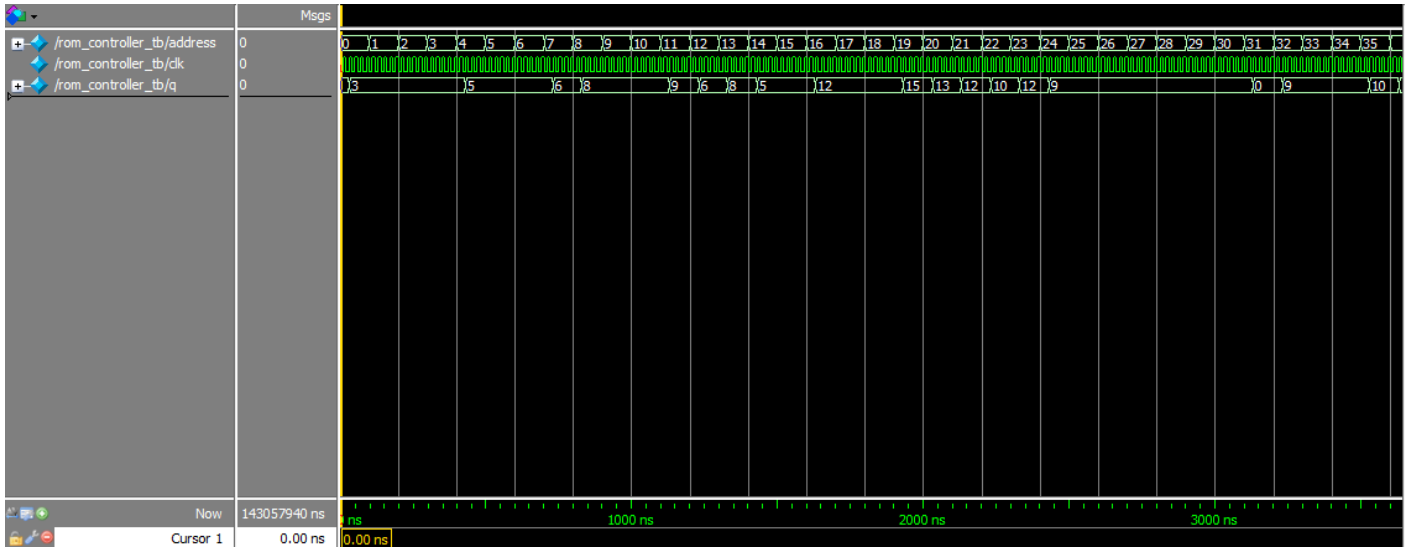


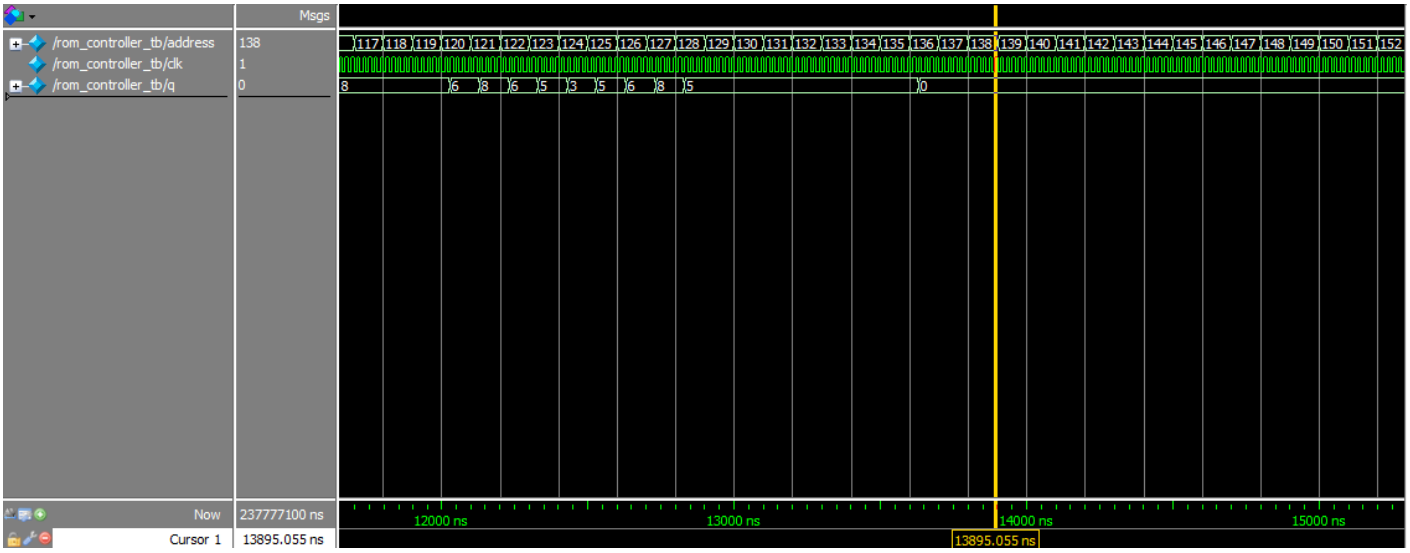
5.4 仿真代码

```
//存储乐谱码 ROM 的测试模块
`timescale 1ns/1ns

module rom_controller_tb;
    reg[7:0] address;
    reg clk;
    wire[3:0] q;

    rom_controller  u1(.address(address), .clock(clk), .q(q));
    initial
    begin
           clk=0;
           address=0;
    end
```

```
    always #10  clk=~clk;
    always #100 address=address+1;

endmodule
```

5.5 仿真波形





由仿真结果可知，ROM 内所存储的乐谱简码与所要求写入的简码一致。（要注意，ROM 出现新地址时，输出对应的数据要延时一个时钟周期才会出现）

6. F_CODE—分频预置数查表电路模块

6.1 原理说明

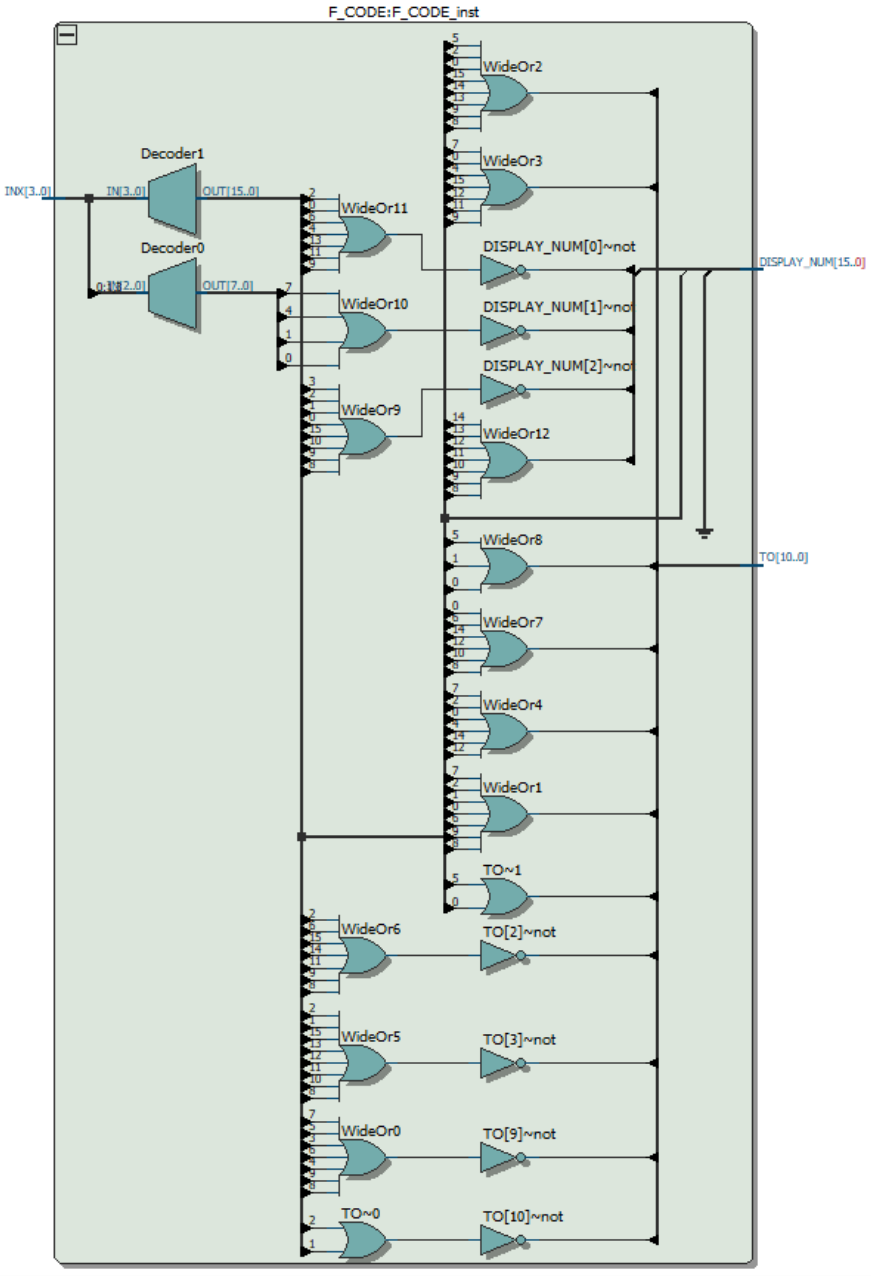
音符的持续时间需根据乐曲的速度及每个音符的节拍数来确定，模块F_CODE的功能首先是为模块SPKER（11位分频器）提供决定所发音符的分频预置数，而此数在SPKER输入口停留的时间即为此音符的节拍周期。模块F_CODE是乐曲简谱码对应的分频预置数查表电路，程序中设置了“梁祝”乐曲全部音符所对应的分频预置数，共14个，每一音符的停留时间则由音乐节拍和音调发生查表模块MUSIC ROM中简谱码和工作时钟inclock的频率决定，在此为4Hz，由分频模块FDIV得到。而模块F_CODE的14个值的输出由对应于MUSIC ROM模块输出的q[3:0]即4位输入值INX[3:0]确定，而INX[3:0]最多有16种可选值。输向模块F_CODE中INX[3:0]的值在SPKER中对应的输出频率值与持续的时间由模块MUSIC ROM决定。

6.2 模块代码

```
//乐曲简谱码对应的分频预置数查表电路模块
module F_CODE(INX, DISPLAY_NUM, TO);
    input[3:0] INX;
    output[15:0] DISPLAY_NUM; //数码管显示数据：[15:12]数码管千位--“H”（低中高音标识符），[3:0]数码管个位--“CODE”（乐谱音符简码）
    output[10:0] TO;
    reg[10:0] TO;
    reg[11:0] CODE;
    reg[3:0] H;

    always @(INX)
    begin
    case (INX)
        0: begin TO<=11'H7FF; CODE<=0; H<=0; end
        1: begin TO<=11'H305; CODE<=1; H<=0; end
        2: begin TO<=11'H390; CODE<=2; H<=0; end
        3: begin TO<=11'H40C; CODE<=3; H<=0; end
        4: begin TO<=11'H45C; CODE<=4; H<=0; end
        5: begin TO<=11'H4AD; CODE<=5; H<=0; end
        6: begin TO<=11'H50A; CODE<=6; H<=0; end
        7: begin TO<=11'H55C; CODE<=7; H<=0; end
        8: begin TO<=11'H582; CODE<=1; H<=1; end
        9: begin TO<=11'H5C8; CODE<=2; H<=1; end
        10: begin TO<=11'H606; CODE<=3; H<=1; end
        11: begin TO<=11'H640; CODE<=4; H<=1; end
```

```
        12: begin TO<=11'H656; CODE<=5; H<=1; end
        13: begin TO<=11'H684; CODE<=6; H<=1; end
        14: begin TO<=11'H69A; CODE<=7; H<=1; end
        15: begin TO<=11'H6C0; CODE<=1; H<=2; end
       default: begin TO<=11'H6C0; CODE<=1; H<=2; end
    endcase
    end
    assign DISPLAY_NUM={H,CODE};

endmodule
```

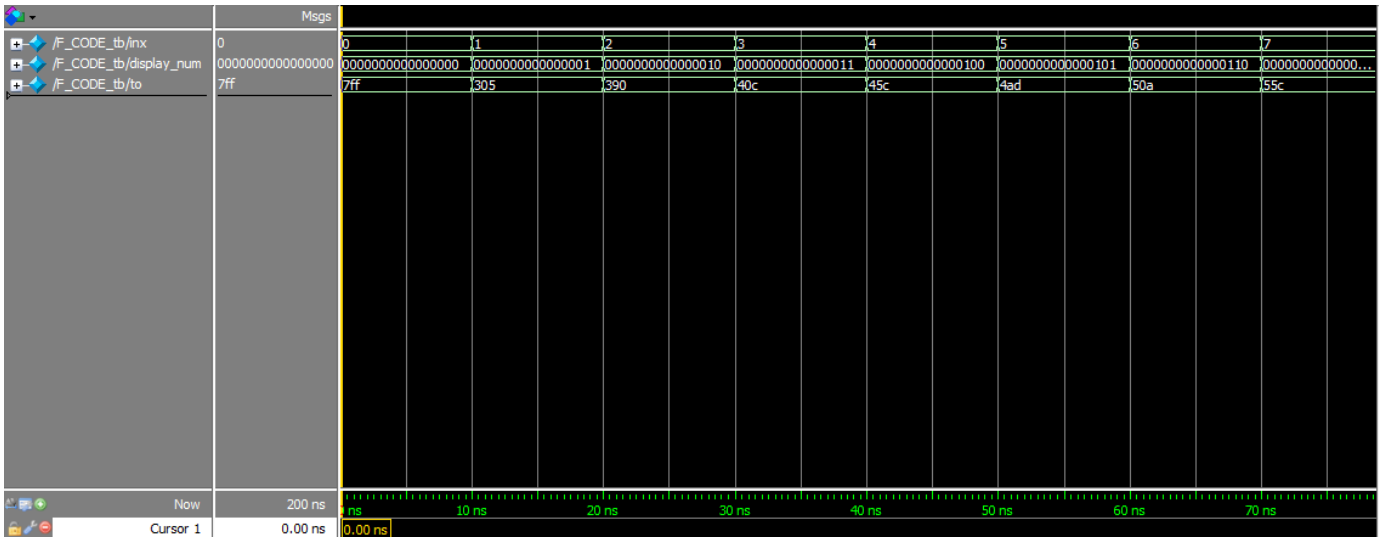
6.3 RTLView

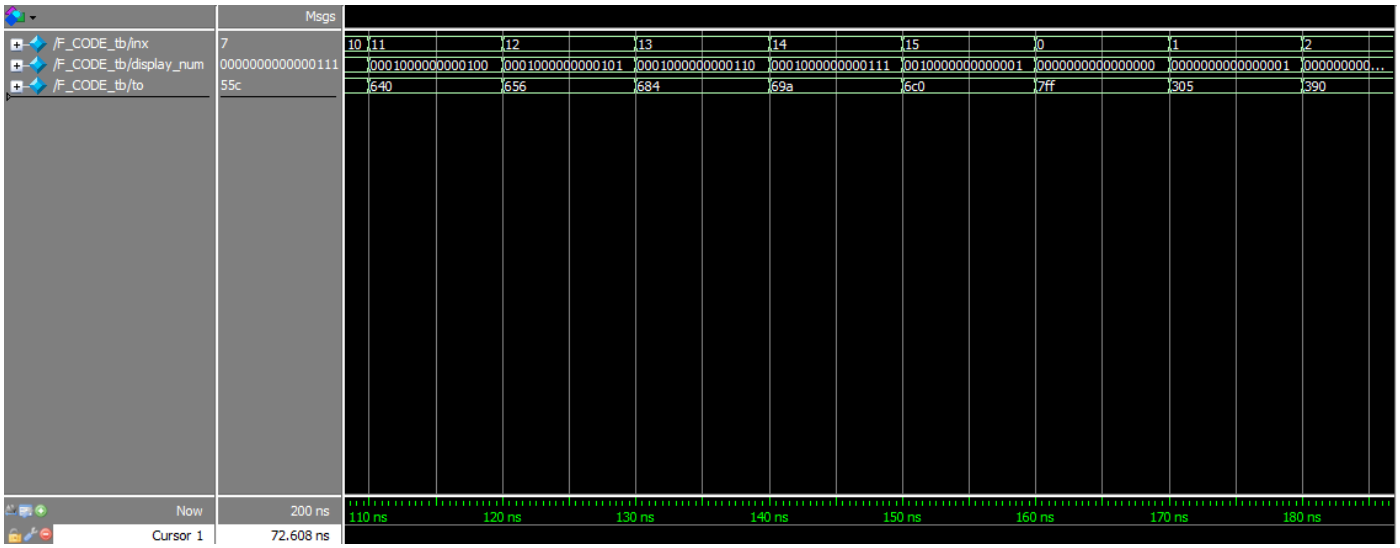


6.4 仿真代码

```
//分频预置数查表电路的测试模块
`timescale 1ns/1ns

module F_CODE_tb;
    reg [3:0] inx;
    wire [15:0] display_num;
    wire [10:0] to;

    F_CODE u1(.INX(inx), .DISPLAY_NUM(display_num), .TO(to));
    initial
    begin
              inx=0;
    #200       $stop;
    end

    always #10 inx=inx+1;

endmodule
```

6.5 仿真波形





可发现，对于不同的 inx 即乐谱简码，F_CODE 模块都可输出相应的正确的预置数和数码管显示数据。（对于数码管显示的数据，其个位即 display_num[3:0]表示相应的音号，千位即 display_num[15:12]表示相应的音高（0、1、2））。

7. SPKER—数控分频模块

7.1 原理说明

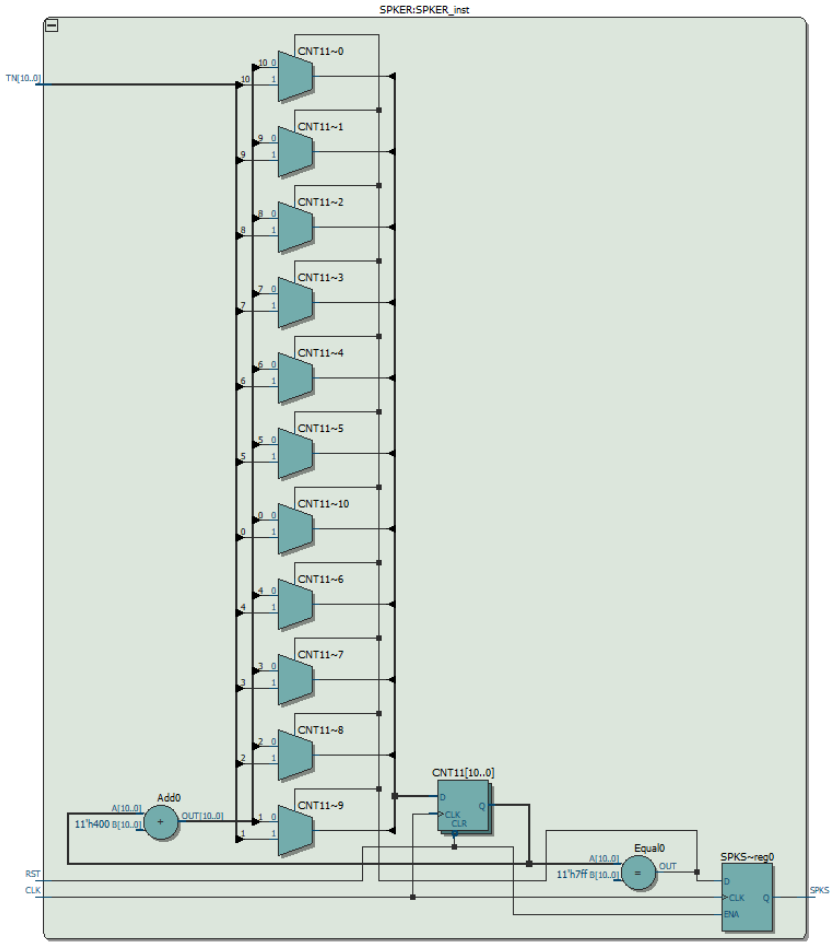
SPKER 模块是一个数控分频器，由其 CLK 端输入一具有较高频率（1MHz）的时钟，通过 SPKER 分频后，经过 D 触发器构成的分频电路，由 SPK_KX 口输出。由于直接从分频器中出来的输出信号是脉宽级窄的信号，为了有利于驱动扬声器，需另加一个 D 触发器分频以均衡其占空比，但这时的频率是原来的 1/2。SPKER 对 CLK 输入信号的分频比由输入的 11 位预置数 TN[10:0]决定。SPKS_KS 口的输出频率将决定每一音符的音调，这样，分频计数器的预置值 TN[10:0]与输出频率就有了对应关系，而输出的频率又与音乐音符的发声有对应关系。例如在 F_CODE 模块中若取 TN[10:0]=11'h40C ，将由 SPK_KX 发出音符位“3”音点信号频率。

7.2 模块代码

```
//数控分频模块——按预置数发声演奏
module SPKER(CLK, RST, TN, SPKS);
    input CLK;
    input RST;
    input[10:0] TN;
    output SPKS;
    reg SPKS;
    reg[10:0] CNT11;
```

```
    always @(posedge CLK or negedge RST)
    begin
    // CNT11B_LOAD: 11 位可预置计数器
        if (!RST) CNT11<=0;
        else if (CNT11==11'h7FF)
        begin
            CNT11=TN;
            SPKS<=1'b1;
        end
        else
        begin
            CNT11=CNT11+1;
            SPKS<=1'b0;
        end
    end

endmodule
```

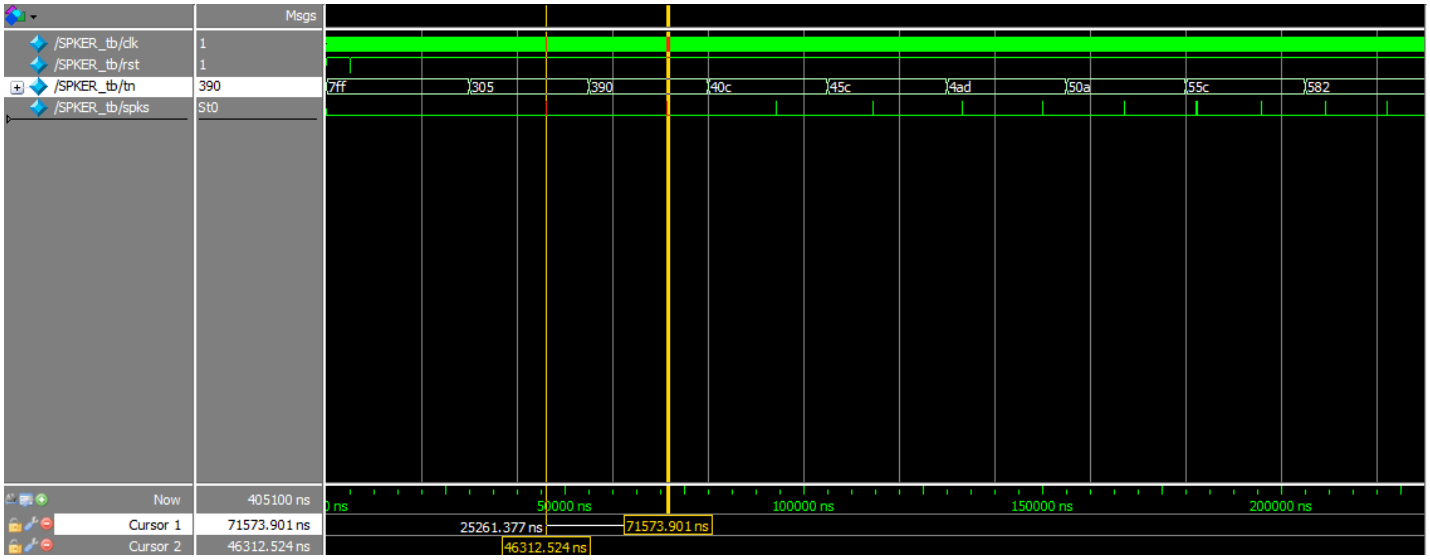
7.3 RTLView



7.4 仿真代码

```
//数控分频模块的测试模块
`timescale 1ns/1ns

module SPKER_tb;
    reg clk;
    reg rst;
    reg[10:0] tn;
    wire spks;

    SPKER u1(.CLK(clk), .RST(rst), .TN(tn), .SPKS(spks));
    initial
    begin
              clk=0;
              rst=1;
              tn=11'h7ff; //简谱"0"
    #5000      rst=0;
    #100       rst=1;
    #25000  tn=11'h305; //简谱"1"
    #25000  tn=11'h390; //简谱"2"
    #25000  tn=11'h40c; //简谱"3"
    #25000  tn=11'h45c; //简谱"4"
    #25000  tn=11'h4ad; //简谱"5"
    #25000  tn=11'h50a; //简谱"6"
    #25000  tn=11'h55c; //简谱"7"
    #25000  tn=11'h582; //简谱"8"
    #25000  tn=11'h5c8; //简谱"9"
    #25000  tn=11'h606; //简谱"10"
    #25000  tn=11'h640; //简谱"11"
    #25000  tn=11'h656; //简谱"12"
    #25000  tn=11'h684; //简谱"13"
    #25000  tn=11'h69a; //简谱"14"
    #25000  tn=11'h6c0; //简谱"15"
    #25000  $stop;
    end

    always #10 clk=~clk;

endmodule
```

7.5 仿真波形



通过换算 spks 脉冲间的时间距离可知根据不同预置数所输出的每段频率均满足乐曲演奏的实际逻辑要求。

8. BEEP_CONTROLLER—蜂鸣器驱动电路模块

8.1 原理说明

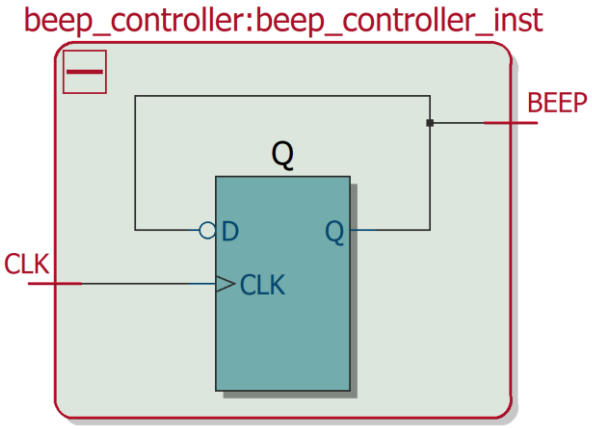
由于直接从 SPKER 分频器中出来的输出信号是脉宽极窄的信号（可从 SPKER 模块的仿真波形看出），为了有利于驱动扬声器，需另加一个 D 触发器分频以均衡其占空比，但这时的频率是原来的 1/2。而蜂鸣器驱动电路模块便是由一个设计的 D 触发器组成，spks 口输出的窄带脉宽经驱动电路分频后输出一个可实际用于蜂鸣器发声演奏的二分频。

8.2 模块代码

```
//蜂鸣器驱动电路模块
module beep_controller(CLK, BEEP);
    input CLK;
    output BEEP;
    reg Q=0;

    always @(posedge CLK)
    begin
        Q=~Q;
    end
```

```
    assign BEEP=Q;

endmodule
```

8.3 RTLView

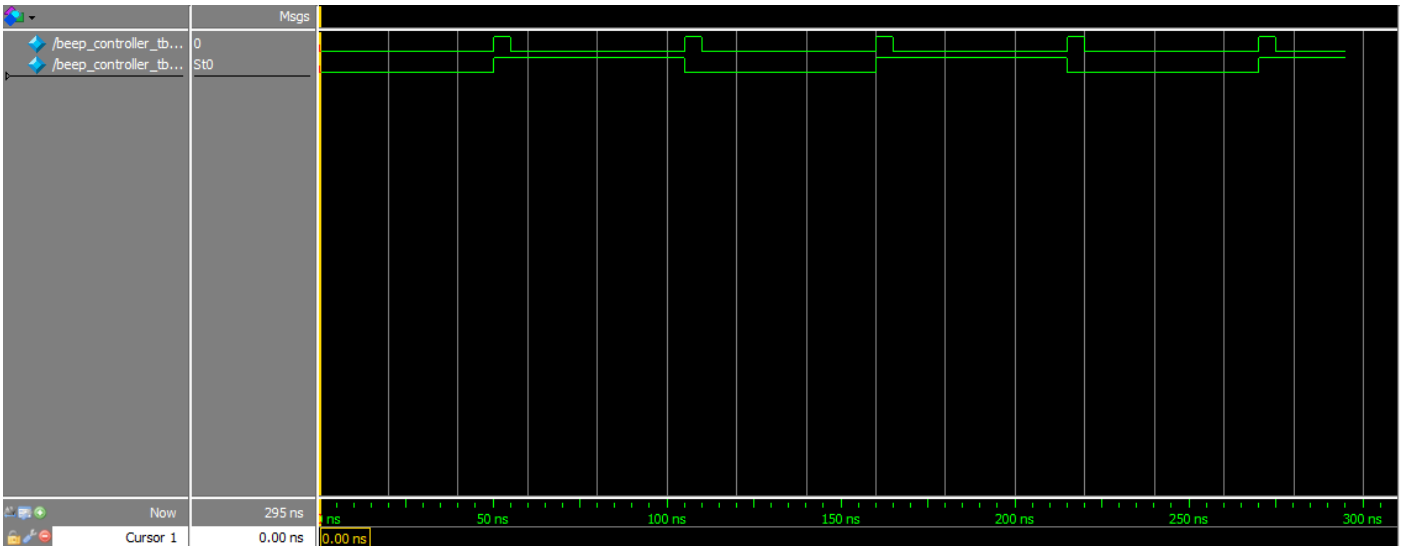


8.4 仿真代码

```
//蜂鸣器驱动电路的测试模块
`timescale 1ns/1ns

module beep_controller_tb;
    reg clk;
    wire beep;

    beep_controller u1(.CLK(clk), .BEEP(beep));
    initial
    begin
           clk=0;
    #50 clk=~clk;
    #5     clk=~clk;
    #50 clk=~clk;
    #5     clk=~clk;
    #50 clk=~clk;
    #5     clk=~clk;
    #50 clk=~clk;
    #5     clk=~clk;
    #50 clk=~clk;
    #5     clk=~clk;
    #20 $stop;
    end
```

```
endmodule
```

8.5 仿真波形



可以看出，原本脉宽极窄的信号经 D 触发器分频后输出一个占空比为 50%的可实际用于蜂鸣器发声演奏的二分频，其频率为原始的 1/2。

9. SEG7—数码管显示驱动模块

9.1 原理说明

此模块为数码管显示驱动模块，可将要显示的数据 display_num 译码转换成开发板上二极管可显示的实际数据，并在二极管上展示出来。其中在逻辑的译码实现上，会分为二极管段选信号和位选信号，通过分时计数来分别显示个位、十位、百位、千位。而在本实验中，我们要显示的数据个位为实际乐谱音号，千位为音高。

9.2 模块代码

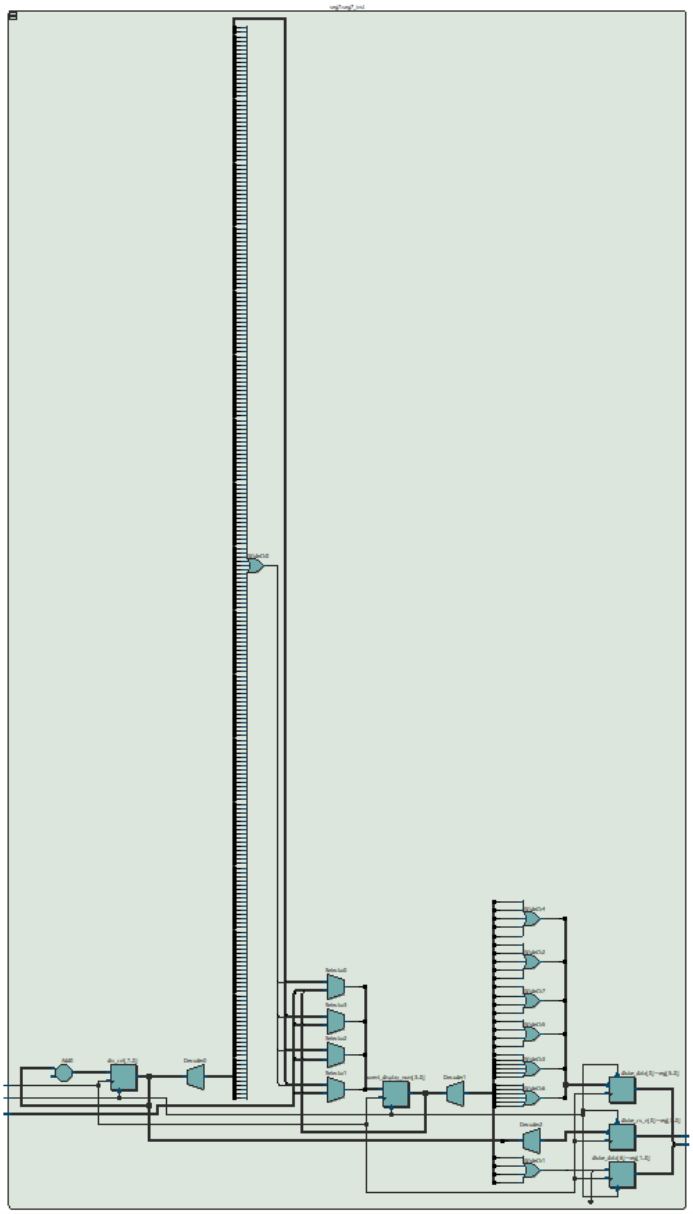
```
//数码管显示驱动模块
module seg7(
          input clk,      //时钟信号，25MHz
          input rst_n,    //复位信号，低电平有效
          input[15:0] display_num,    //数码管显示数据，[15:12]--数码管千位，[11:8]--数码管百位，[7:4]--数码管十位，[3:0]--数码管个位
          output reg[3:0] dtube_cs_n, //7 段数码管位选信号
```

```
        output reg[7:0] dtube_data  //7 段数码管段选信号（包括小数点为
8 段）
    );

//-------------------------------------------------
//参数定义

//数码管显示 0~F 对应段选输出
parameter   NUM0    = 8'h3f,//c0,
          NUM1    = 8'h06,//f9,
          NUM2    = 8'h5b,//a4,
          NUM3    = 8'h4f,//b0,
          NUM4    = 8'h66,//99,
          NUM5    = 8'h6d,//92,
          NUM6    = 8'h7d,//82,
          NUM7    = 8'h07,//F8,
          NUM8    = 8'h7f,//80,
          NUM9    = 8'h6f,//90,
          NUMA    = 8'h77,//88,
          NUMB    = 8'h7c,//83,
          NUMC    = 8'h39,//c6,
          NUMD    = 8'h5e,//a1,
          NUME    = 8'h79,//86,
          NUMF    = 8'h71,//8e;
          NDOT    = 8'h80;    //小数点显示

//数码管位选 0~3 对应输出
parameter   CSN     = 4'b1111,
          CS0     = 4'b1110,
          CS1     = 4'b1101,
          CS2     = 4'b1011,
          CS3     = 4'b0111;

//-------------------------------------------------
//分时显示数据控制单元
reg[3:0] current_display_num;   //当前显示数据
reg[7:0] div_cnt;   //分时计数器

    //分时计数器
always @(posedge clk or negedge rst_n)
    if(!rst_n) div_cnt <= 8'd0;
    else div_cnt <= div_cnt+1'b1;

    //显示数据
```

```
always @(posedge clk or negedge rst_n)
    if(!rst_n) current_display_num <= 4'h0;
    else begin
        case(div_cnt)
            8'hff: current_display_num <= display_num[3:0];
            8'h3f: current_display_num <= display_num[7:4];
            8'h7f: current_display_num <= display_num[11:8];
            8'hbf: current_display_num <= display_num[15:12];
            default: ;
        endcase
    end

    //段选数据译码
always @(posedge clk or negedge rst_n)
    if(!rst_n) dtube_data <= NUM0;
    else begin
        case(current_display_num)
            4'h0: dtube_data <= NUM0;
            4'h1: dtube_data <= NUM1;
            4'h2: dtube_data <= NUM2;
            4'h3: dtube_data <= NUM3;
            4'h4: dtube_data <= NUM4;
            4'h5: dtube_data <= NUM5;
            4'h6: dtube_data <= NUM6;
            4'h7: dtube_data <= NUM7;
            4'h8: dtube_data <= NUM8;
            4'h9: dtube_data <= NUM9;
            4'ha: dtube_data <= NUMA;
            4'hb: dtube_data <= NUMB;
            4'hc: dtube_data <= NUMC;
            4'hd: dtube_data <= NUMD;
            4'he: dtube_data <= NUME;
            4'hf: dtube_data <= NUMF;
            default: ;
        endcase
    end

    //位选译码
always @(posedge clk or negedge rst_n)
    if(!rst_n) dtube_cs_n <= CSN;
    else begin
        case(div_cnt[7:6])
            2'b00: dtube_cs_n <= CS0;
            2'b01: dtube_cs_n <= CS1;
```

```
            2'b10: dtube_cs_n <= CS2;
            2'b11: dtube_cs_n <= CS3;
            default:  dtube_cs_n <= CSN;
        endcase
    end


endmodule
```

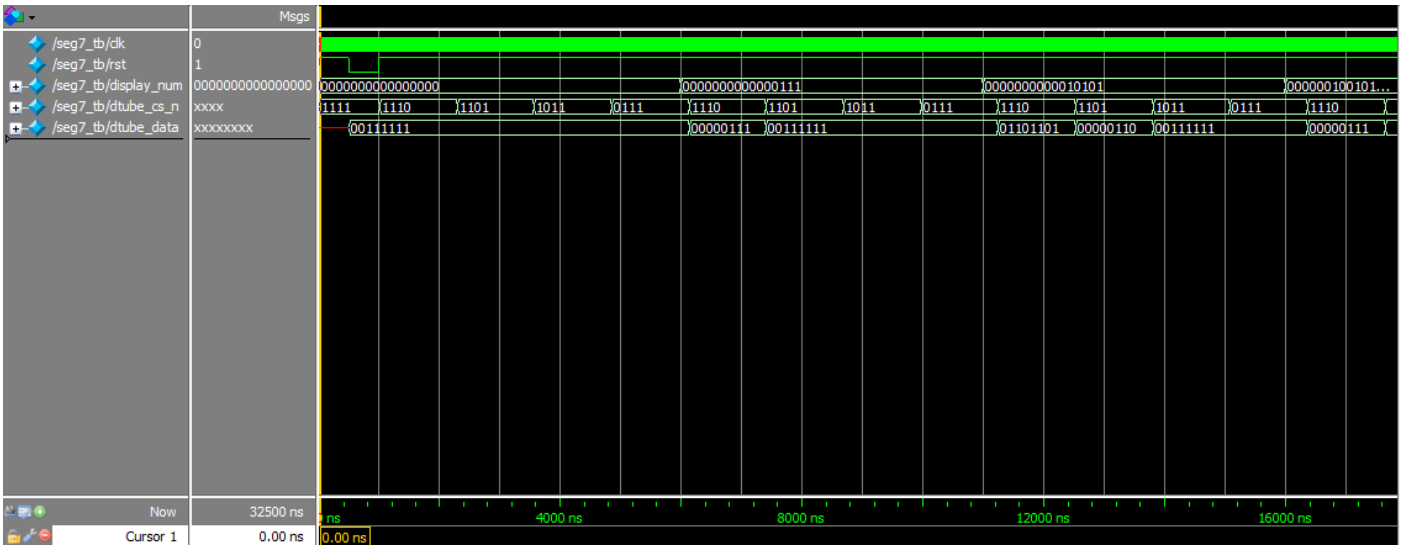
9.3 RTLView

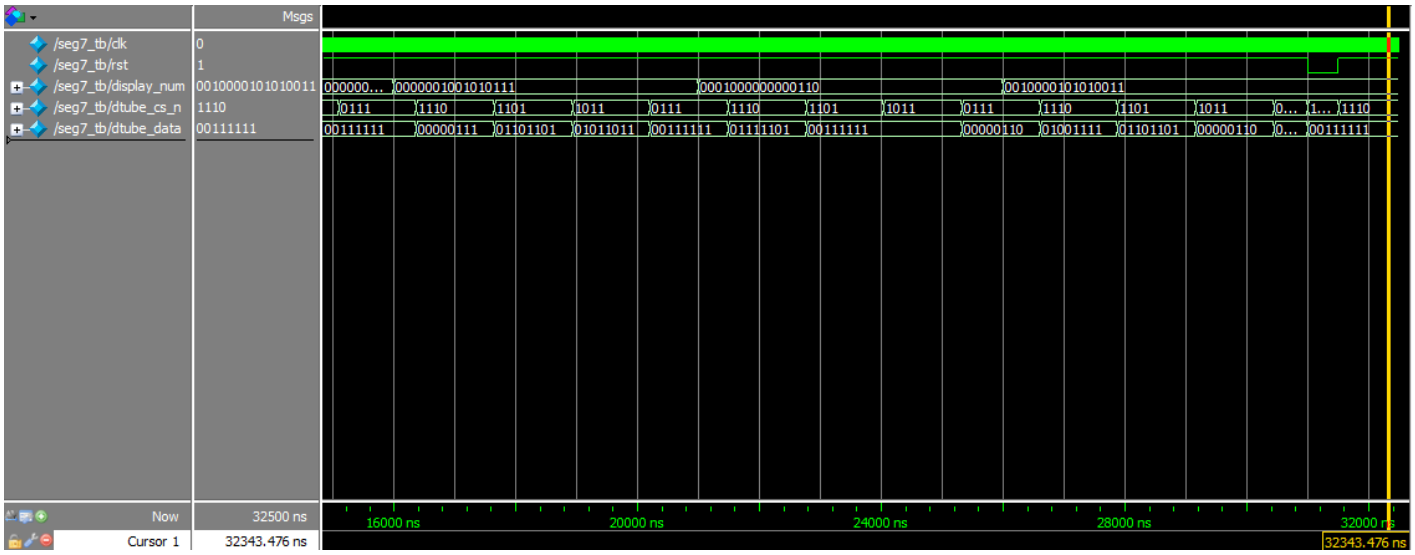


9.4 仿真代码

```
//数码管显示驱动的测试模块
`timescale 1ns/1ns

module seg7_tb;
    reg clk;
    reg rst;
    reg[15:0] display_num;
    wire[3:0] dtube_cs_n;
    wire[7:0] dtube_data;

    seg7
u1(.clk(clk), .rst_n(rst), .display_num(display_num), .dtube_cs_n(d
tube_cs_n), .dtube_data(dtube_data));
    initial
    begin
              clk=0;
              rst=1;
              display_num=0;
    #500       rst=0;
    #500       rst=1;
    #5000      display_num=16'b0000_0000_0000_0111;   //显示数据“7”
    #5000      display_num=16'b0000_0000_0001_0101;    //显示数据“15”
    #5000      display_num=16'b0000_0010_0101_0111;    //显示数据“257”
    #5000      display_num=16'b0001_0000_0000_0110;    //显示数据
“1006”
    #5000      display_num=16'b0010_0001_0101_0011;    //显示数据
“2153”
    #5000      rst=0;
    #500       rst=1;
    #1000      $stop;
    end

    always #10  clk=~clk;

endmodule
```

9.5 仿真波形





结合波形中的段选信号 dtube_data 和位选信号 dtube_cs_n 以及所要显示的数据 display_num，对照数码管的译码逻辑表，可知显示正确。

10.M_PLAYER—顶层模块

10.1 原理说明

顶层模块的功能是将上述所有子模块连接集成到一起，定义输入输出端口，从而实现整个乐谱演奏系统的功能，功能实现过程为系统时钟经 pll 模块分频后输出 1MHz 和 2KHz 的分频时钟，分别作用于 FDIV 模块和 SPKER 模块，FDIV 模块分频输出一个 4Hz 的时钟到 CNT138T 模块，以相应的频率输出递增的地址数据

作用到 Music ROM 模块，输出存于其中的乐谱简码，继而在 F_CODE 模块中查表输出相应的预置数，并把对应的音号和音高输出到 seg7 模块在数码管显示，而预置数则输入到 SPKER 模块中经分频后输出发声频率信号，经 BEEP 驱动模块分频后得到可实际用于演奏的音乐频率输送到蜂鸣器发声演奏，因此存储在 Music ROM 中的歌曲即可按照特定音高频率和节拍循环演奏，并且通过数码管可看到当前演奏的音号和音高。并且通过拨码开关控制 load 端可实现切换播放存储在 ROM 中的两首歌曲。

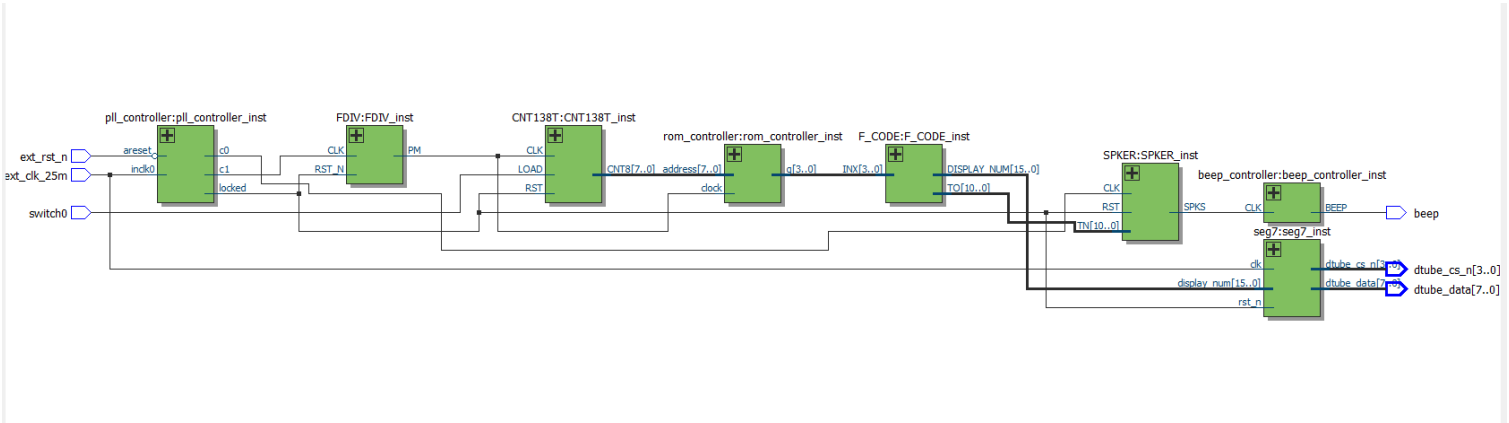
10.2 模块代码

```
//乐曲演奏电路设计---实现"梁祝"乐曲的循环演奏
module M_Player(
          input ext_clk_25m, //外部输入 25MHz 时钟信号
          input ext_rst_n, //外部输入复位信号，低电平有效
          input switch0, //通过拨码开关控制 CNT138T 模块，手动选择切换歌曲"梁祝"和"欢乐颂"
          output[3:0] dtube_cs_n, //7 段数码管位选信号
          output[7:0] dtube_data, //7 段数码管段选信号（包括小数点为 8 段）
          output beep     //蜂鸣器控制信号，1--响，0--不响
       );

//----------------------------------------
//PLL 例化
wire clk_2k;    //PLL 输出 2KHz 时钟
wire clk_1m;    //PLL 输出 1MHz 时钟
wire sys_rst_n; //PLL 输出的 locked 信号，作为 FPGA 内部的复位信号，低电平复位，高电平正常工作

pll_controller  pll_controller_inst (
    .areset ( !ext_rst_n ),
    .inclk0 ( ext_clk_25m ),
    .c0 ( clk_1m ),
    .c1 ( clk_2k ),
    .locked ( sys_rst_n )
    );

//------------------------------------------------------------------------
//2KHz 时钟进行分频，产生一个 4Hz 频率的时钟使能，即满足乐曲 0.25 秒一个拍子的要求
wire clk_4;     //分频模块输出 4Hz 时钟

FDIV    FDIV_inst (
```

```
    .CLK( clk_2k ),
    .RST_N( sys_rst_n ),
    .PM( clk_4 )
    );

//-------------------------------------------------
//4Hz 控制乐曲节拍，通过 CNT138T 实现 MUSIC ROM 读取地址递增
wire [7:0] rom_address; //计数器模块输出 ROM 地址

CNT138T CNT138T_inst (
    .CLK( clk_4 ),
    .RST( sys_rst_n ),
    .LOAD( switch0 ),
    .CNT8( rom_address )
    );

//-------------------------------------------------
//ROM 例化，乐谱码按地址存放于 ROM 中
wire [3:0] inx; //ROM 模块输出乐谱简码

rom_controller  rom_controller_inst (
    .address ( rom_address ),
    .clock ( clk_4 ),
    .q ( inx )
    );

//-------------------------------------------------
//预置数查表模块根据输入的乐谱简码输出相应的分频预置数
wire [15:0] display_num;    //输出乐谱音符到数码管显示
wire [10:0] tn; //输出预置数

F_CODE  F_CODE_inst (
    .INX( inx ),
    .DISPLAY_NUM( display_num ),
    .TO( tn )
    );

//----------------------------------
//数控分频模块——按预置数发声演奏
wire spks;

SPKER   SPKER_inst  (
    .CLK( clk_1m ),
    .RST( sys_rst_n ),
```

```
    .TN( tn ),
    .SPKS( spks )
    );

//-----------------------------------
//蜂鸣器发声驱动

beep_controller beep_controller_inst(
    .CLK( spks ),
    .BEEP( beep )
    );

//-------------------------------------------------------------------
-----------------------------
//4 位数码管显示驱动：[15:12]数码管千位--"H"（低中高音标识符），[3:0]数码管个
位--"CODE"（乐谱音符简码）

seg7    seg7_inst(
        .clk( ext_clk_25m ),    //时钟信号
        .rst_n( sys_rst_n ),    //复位信号，低电平有效
        .display_num( display_num ),        //显示数据
        .dtube_cs_n( dtube_cs_n ),  //7 段数码管位选信号
        .dtube_data( dtube_data )       //7 段数码管段选信号（包括小数点为
8 段）
        );


endmodule
```

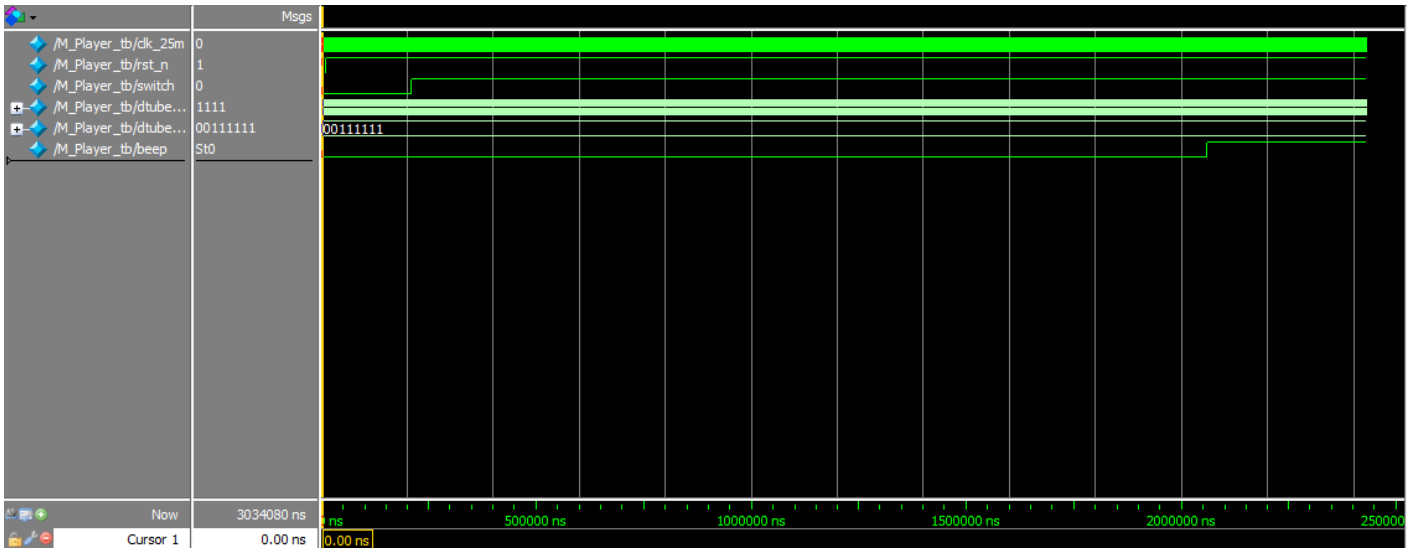
10.3 RTLView



10.4 仿真代码

```
//顶层模块（整个乐曲演奏电路）的测试模块
`timescale 1ns/1ns

module M_Player_tb;
    reg clk_25m;
    reg rst_n;
    reg switch;
    wire[3:0] dtube_cs_n;
    wire[7:0] dtube_data;
    wire beep;

    M_Player
u1(.ext_clk_25m(clk_25m), .ext_rst_n(rst_n), .switch0(switch), .dtu
be_cs_n(dtube_cs_n), .dtube_data(dtube_data), .beep(beep));
    initial
    begin
              clk_25m=0;
              rst_n=1;
              switch=0;
    #10000  rst_n=0;
    #100       rst_n=1;
    #200000 switch=1;
    end

    always #20 clk_25m=~clk_25m;

endmodule
```

10.5 仿真波形



从波形中可看到，数码管显示一直为 0000，蜂鸣器发声情况也不正常，经分析前面每个子模块都仿真正确，而顶层模块仿真出错原因应该是 pll 模块分频所得 2KHz 频率部分在仿真中无波形，导致没有 2KHz 的时钟信号输入到 FDIV、F_CODE 等模块，因此也没有乐曲数据输出，顶层模块仿真中数码管显示保持为 0，蜂鸣器不发声。但在板级调试中，可正常发声演奏。

11.联合测试仿真

11.1 原理说明

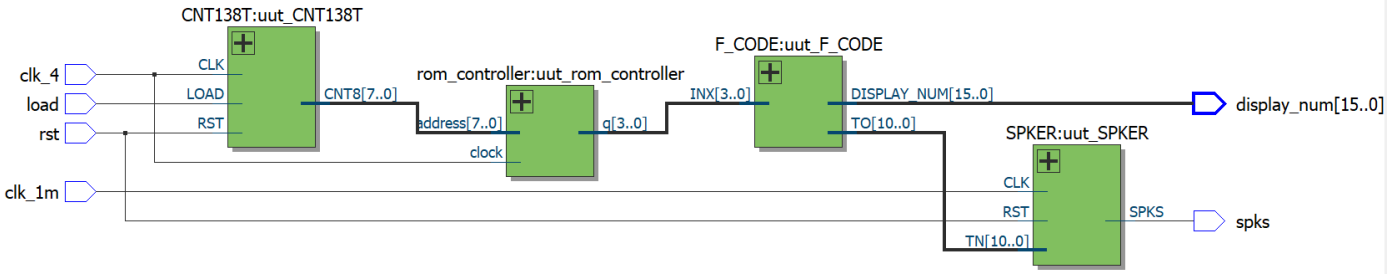
通过编写一个顶层模块联合测试模块 CNT138T、Music ROM、F_CODE 和 SPKER，进一步确认 F_CODE 中的音符预置数的精确性，因为这些数据决定了音准。

11.2 模块代码

```
//联合测试仿真的顶层模块
module test(
        input clk_4,
        input clk_1m,
        input rst,
        input load,
        output[15:0] display_num,
        output spks
        );
```

```
wire [7:0] rom_address;

CNT138T uut_CNT138T (
    .CLK( clk_4 ),
    .RST( rst ),
    .LOAD(load),
    .CNT8( rom_address )
    );

wire [3:0] inx;

rom_controller  uut_rom_controller (
    .address ( rom_address ),
    .clock ( clk_4 ),
    .q ( inx )
    );

wire [10:0] tn;

F_CODE  uut_F_CODE (
    .INX( inx ),
    .DISPLAY_NUM( display_num ),
    .TO( tn )
    );


SPKER   uut_SPKER   (
    .CLK( clk_1m ),
    .RST( rst ),
    .TN( tn ),
    .SPKS( spks )
    );

endmodule
```

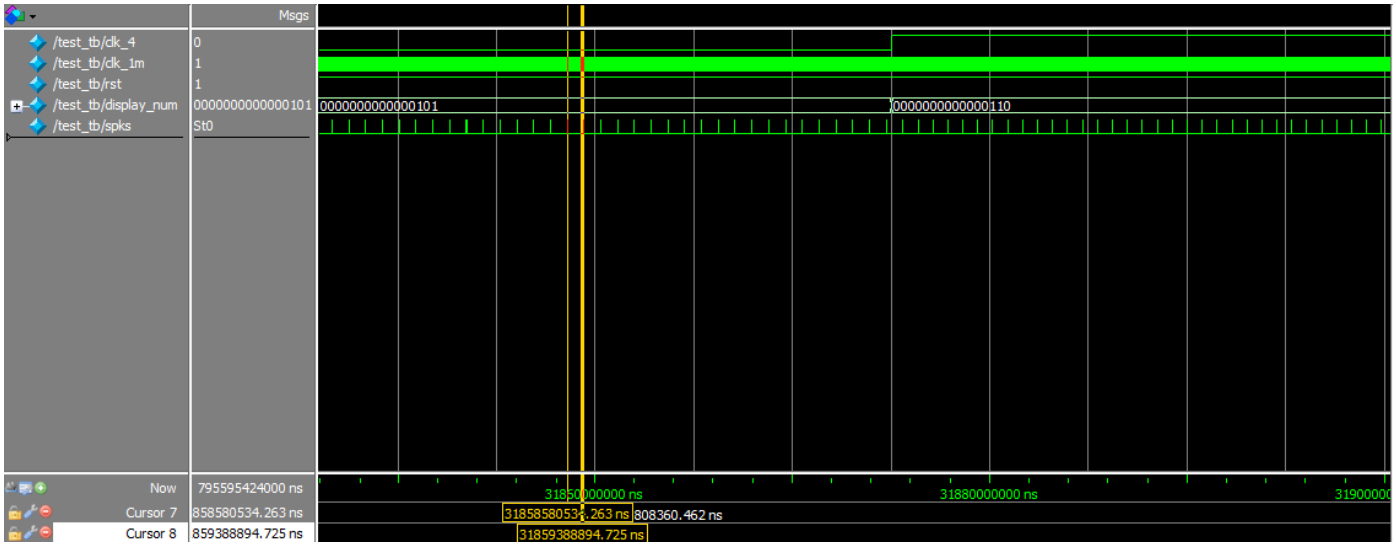
11.3 RTLView

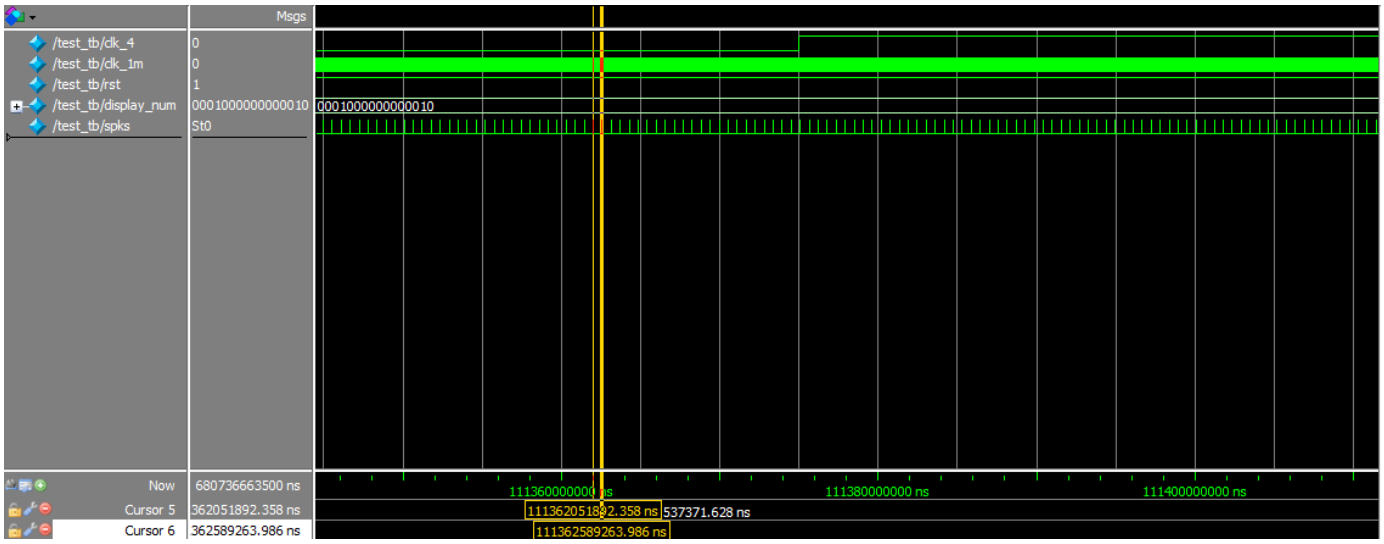


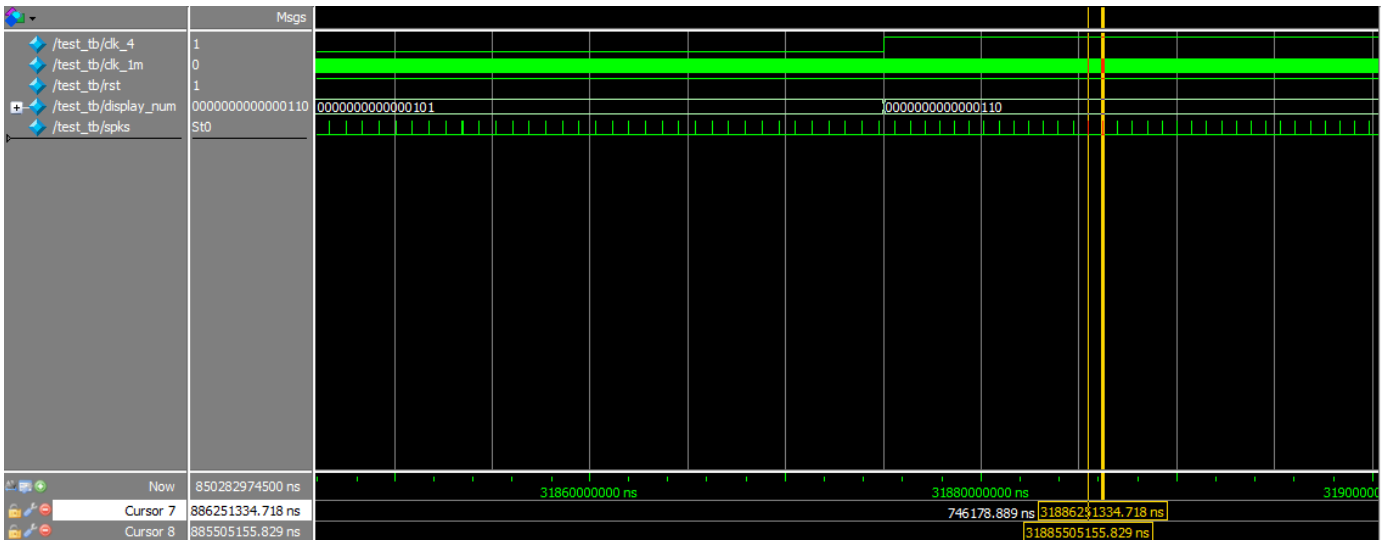
11.4 仿真代码

```
//联合测试的测试仿真模块
`timescale 1ns/1ns

module test_tb;
    reg clk_4;
    reg clk_1m;
    reg rst;
    reg load;
    wire[15:0] display_num;
    wire spks;

    test
u1(.clk_4(clk_4), .clk_1m(clk_1m), .rst(rst), .load(load), .display
_num(display_num), .spks(spks));
    initial
    begin
              clk_4=0;
              clk_1m=0;
              rst=1;
              load=0;
    #1000      rst=0;
    #1000      rst=1;
    end

    always #125000000 clk_4=~clk_4;
    always #500        clk_1m=~clk_1m;

endmodule
```

11.5 仿真波形


Msgs
/test_tb/clk_4
/test_tb/clk_1m
/test_tb/rst
/test_tb/display_num
/test_tb/spks
0
1
1
0000000000000101
St0
0000000000000101
0000000000000110
Now
795595424000 ns
Cursor 7
858580534.263 ns
Cursor 8
859388894.725 ns
808360.462 ns
31859388894.725 ns

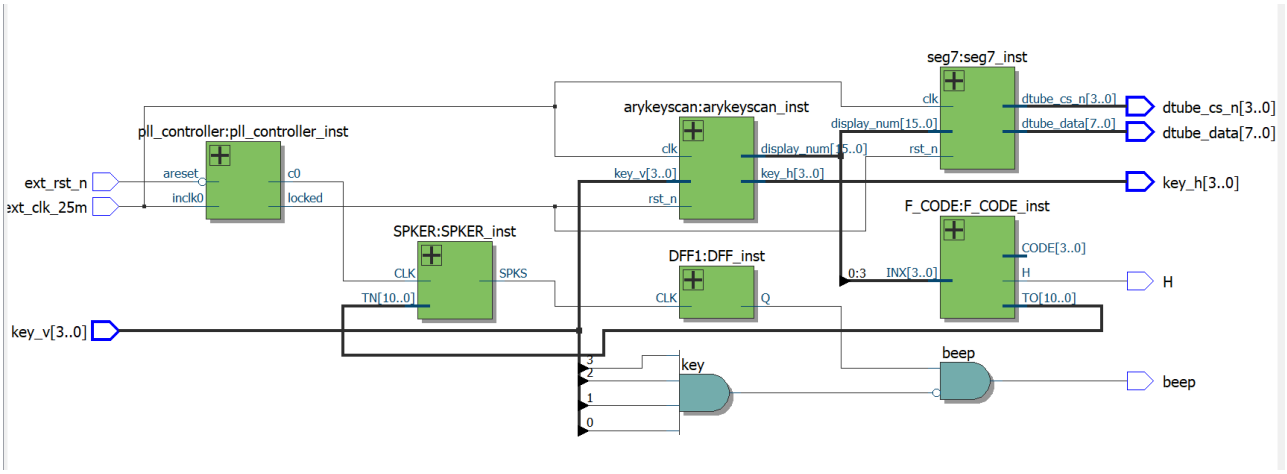

Msgs
/test_tb/clk_4
/test_tb/clk_1m
/test_tb/rst
/test_tb/display_num
/test_tb/spks
0
0
1
0001000000000010
St0
0001000000000010
Now
680736663500 ns
Cursor 5
362051892.358 ns
Cursor 6
362589263.986 ns
537371.628 ns
111362589263.986 ns
111380000000 ns
111400000000 ns


Msgs
/test_tb/clk_4
/test_tb/clk_1m
/test_tb/rst
/test_tb/display_num
/test_tb/spks
1
0
1
0000000000000110
St0
0000000000000101
0000000000000110
Now
850282974500 ns
Cursor 7
886251334.718 ns
Cursor 8
885505155.829 ns
31860000000 ns
31880000000 ns
746178.889 ns
31886251334.718 ns
31885505155.829 ns

从图示波形中，可看到各模块均可正常工作。通过将 spks 的脉冲时间距离换算成频率，和电子琴音阶基频图对照，发现有些音号的频率不对，因此 F_CODE 中的预置数也有问题，为保证正确演奏，需根据电子琴音阶基频图重新计算预置数并在 F_CODE 中进行一定的修改。

12. 电子琴设计

12.1 顶层模块 RTLView



12.2 各模块代码

顶层模块--

```
//采集 4X4 矩阵按键的键值，输出到数码管的末位，数码管每新输入一位数据，都会将原有数据右移一位
module piano(
          input ext_clk_25m,  //外部输入 25MHz 时钟信号
          input ext_rst_n,    //外部输入复位信号，低电平有效
          input[3:0] key_v,   //4 个列按键输入，未按下为高电平，按下后为低电平
          output[3:0] key_h,  //4 个行按键输出
          output[3:0] dtube_cs_n, //7 段数码管位选信号
          output[7:0] dtube_data, //7 段数码管段选信号（包括小数点为 8 段）
        output beep,//蜂鸣器
          output H//高音 led 指示
          );

wire key;   //所有按键值相与的结果，用于按键触发判断
assign key = key_v[0] & key_v[1] & key_v[2] & key_v[3];
```

```
//-------------------------------------
//键值采集，产生数码管显示数据
wire[15:0] display_num; //数码管显示数据，[15:12]--数码管千位，[11:8]--
数码管百位，[7:4]--数码管十位，[3:0]--数码管个位

arykeyscan     arykeyscan_inst(
                 .clk(ext_clk_25m),      //时钟信号
                 .rst_n(sys_rst_n),  //复位信号，低电平有效
                 .key_v(key_v),  //4 个按键输入，未按下为高电平，按下后为
低电平
                 .key_h(key_h),  //4 个行按键输出
                 .display_num(display_num)
              );


//-------------------------------------
//4 位数码管显示驱动

seg7       seg7_inst(
              .clk(ext_clk_25m),      //时钟信号
              .rst_n(sys_rst_n),  //复位信号，低电平有效
              .display_num(display_num),
              .dtube_cs_n(dtube_cs_n),    //7 段数码管位选信号
              .dtube_data(dtube_data)     //7 段数码管段选信号（包括小数点
为 8 段）
       );

//-------------------------------------------------------------------
-------------
//PLL 例化
wire clk_1m;    //PLL 输出 1MHz 时钟
wire sys_rst_n; //PLL 输出的 locked 信号，作为 FPGA 内部的复位信号，低电平复
位，高电平正常工作

pll_controller  pll_controller_inst (
    .areset ( !ext_rst_n ),
    .inclk0 ( ext_clk_25m ),
    .c0 ( clk_1m ),
    .locked ( sys_rst_n )
    );

//-------------------------------------------------------------------
-----------
```

```
wire[3:0] CODE;
wire[10:0] TO;

F_CODE F_CODE_inst(
      .INX(display_num[3:0]),
      .CODE(CODE),
      .H(H),
      .TO(TO)
   );

//---------------------------------------
wire SPKS;

SPKER SPKER_inst(
     .CLK(clk_1m),
     .TN(TO),
     .SPKS(SPKS)
   );

//-----------------------------------------------
wire Q;

DFF1 DFF_inst(
      .CLK(SPKS),
      .Q(Q)
  );

assign beep=(~key)&Q;

endmodule
```

sigkeyscan.v

```
module sigkeyscan(
          input clk,  //外部输入 25MHz 时钟信号
          input rst_n,    //外部输入复位信号，低电平有效
          input[3:0] key_v,   //4 个列按键输入，未按下为高电平，按下后为低
电平
          output[3:0] keyv_value      //列按键按下键值，高电平有效
          );

//-------------------------------------
//按键抖动判断逻辑
wire key;   //所有按键值相与的结果，用于按键触发判断
reg[3:0] keyr;  //按键值 key 的缓存寄存器
```

```
assign key = key_v[0] & key_v[1] & key_v[2] & key_v[3];

always @(posedge clk or negedge rst_n)
    if (!rst_n) keyr <= 4'b1111;
    else keyr <= {keyr[2:0],key};

wire key_neg = ~keyr[2] & keyr[3];  //有按键被按下
wire key_pos = keyr[2] & ~keyr[3];  //有按键被释放

//-------------------------------------
//定时计数逻辑，用于对按键的消抖判断
reg[19:0]  cnt;

    //按键消抖定时计数器
always @ (posedge clk or negedge rst_n)
   if (!rst_n) cnt <= 20'd0;
    else if(key_pos || key_neg) cnt <= 20'd0;
    else if(cnt < 20'd999_999) cnt <= cnt + 1'b1;
    else cnt <= 20'd0;

reg[3:0] key_value[1:0];

    //定时采集按键值
always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
        key_value[0] <= 4'b1111;
        key_value[1] <= 4'b1111;
    end
    else begin
        key_value[1] <= key_value[0];
        if(cnt == 20'd999_999) key_value[0] <= key_v;   //定时键值采集
        else ;
    end

assign keyv_value = key_value[1] & ~key_value[0];       //消抖后按键值
变化标志位


endmodule
```

arykeyscan.v

```
module arykeyscan(
          input clk,  //外部输入 25MHz 时钟信号
```

```
            input rst_n,    //外部输入复位信号，低电平有效
            input[3:0] key_v,   //4 个按键输入，未按下为高电平，按下后为低电
平
            output reg[3:0] key_h,  //4 个行按键输出
            output reg[15:0] display_num    //数码管显示数据，[15:12]--
数码管千位，[11:8]--数码管百位，
                                        //[7:4]--数码管十位，[3:0]--数码管
个位
            );

//--------------------------------------
//列按键键值采样
wire[3:0] keyv_value;       //列按键按下键值，高电平有效

sigkeyscan      sigkeyscan_inst(
                  .clk(clk),  //外部输入 25MHz 时钟信号
                  .rst_n(rst_n),  //外部输入复位信号，低电平有效
                  .key_v(key_v),  //4 个独立按键输入，未按下为高电平，按下
后为低电平
                  .keyv_value(keyv_value)     //列按键按下键值，高电平有
效
              );

//--------------------------------------
//状态机采样键值
reg[3:0] nstate,cstate;
parameter K_IDLE = 4'd0;    //空闲状态，等待
parameter K_H1OL = 4'd1;    //key_h[0]拉低
parameter K_H2OL = 4'd2;    //key_h[1]拉低
parameter K_H3OL = 4'd3;    //key_h[2]拉低
parameter K_H4OL = 4'd4;    //key_h[3]拉低
parameter K_CHCK = 4'd5;

    //状态切换
always @(posedge clk or negedge rst_n)
    if(!rst_n) cstate <= K_IDLE;
    else cstate <= nstate;

always @(cstate or keyv_value or key_v)
    case(cstate)
        K_IDLE: if(keyv_value != 4'b0000) nstate <= K_H1OL;
                else nstate <= K_IDLE;
        K_H1OL: nstate <= K_H2OL;
        K_H2OL: if(key_v != 4'b1111) nstate <= K_IDLE;
```

```
                else nstate <= K_H3OL;
        K_H3OL: if(key_v != 4'b1111) nstate <= K_IDLE;
                else nstate <= K_H4OL;
        K_H4OL: if(key_v != 4'b1111) nstate <= K_IDLE;
                else nstate <= K_CHCK;
        K_CHCK: nstate <= K_IDLE;
        default: ;
    endcase

//-------------------------------------
//采样键值
reg[3:0] new_value; //新采样数据
reg new_rdy;    //新采样数据有效

always @(posedge clk or negedge rst_n)
    if(!rst_n) begin
        key_h <= 4'b0000;
        new_value <= 4'd0;
        new_rdy <= 1'b0;
    end
    else begin
        case(cstate)
            K_IDLE: begin
                key_h <= 4'b0000;
                new_value <= 4'd0;
                new_rdy <= 1'b0;
            end
            K_H1OL: begin
                key_h <= 4'b1110;
                new_value <= 4'd0;
                new_rdy <= 1'b0;
            end
            K_H2OL: begin
                case(key_v)
                    4'b1110: begin
                            key_h <= 4'b0000;
                            new_value <= 4'd0;
                            new_rdy <= 1'b1;
                        end
                    4'b1101: begin
                            key_h <= 4'b0000;
                            new_value <= 4'd1;
                            new_rdy <= 1'b1;
                        end
```

```
                4'b1011: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd2;
                        new_rdy <= 1'b1;
                    end
                4'b0111: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd3;
                        new_rdy <= 1'b1;
                    end
                default: begin
                        key_h <= 4'b1101;
                        new_value <= 4'd0;
                        new_rdy <= 1'b0;
                    end
            endcase
        end
        K_H3OL: begin
            case(key_v)
                4'b1110: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd4;
                        new_rdy <= 1'b1;
                    end
                4'b1101: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd5;
                        new_rdy <= 1'b1;
                    end
                4'b1011: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd6;
                        new_rdy <= 1'b1;
                    end
                4'b0111: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd7;
                        new_rdy <= 1'b1;
                    end
                default: begin
                        key_h <= 4'b1011;
                        new_value <= 4'd0;
                        new_rdy <= 1'b0;
                    end
```

```
            endcase
        end
        K_H4OL: begin
            case(key_v)
                4'b1110: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd8;
                        new_rdy <= 1'b1;
                    end
                4'b1101: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd9;
                        new_rdy <= 1'b1;
                    end
                4'b1011: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd10;
                        new_rdy <= 1'b1;
                    end
                4'b0111: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd11;
                        new_rdy <= 1'b1;
                    end
                default: begin
                        key_h <= 4'b0111;
                        new_value <= 4'd0;
                        new_rdy <= 1'b0;
                    end
            endcase
        end
        K_CHCK: begin
            case(key_v)
                4'b1110: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd12;
                        new_rdy <= 1'b1;
                    end
                4'b1101: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd13;
                        new_rdy <= 1'b1;
                    end
                4'b1011: begin
```

```
                        key_h <= 4'b0000;
                        new_value <= 4'd14;
                        new_rdy <= 1'b1;
                    end
                4'b0111: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd15;
                        new_rdy <= 1'b1;
                    end
                default: begin
                        key_h <= 4'b0000;
                        new_value <= 4'd0;
                        new_rdy <= 1'b0;
                    end
            endcase
        end
        default: ;
    endcase
end

//--------------------------------------
//产生最新键值

always @(posedge clk or negedge rst_n)
    if(!rst_n) display_num <= 16'h0000;
    else if(new_rdy) display_num <= {display_num[11:0],new_value};



endmodule
```

FCODE.v

```
module F_CODE(INX,CODE,H,TO);
  input[3:0] INX;
  output[3:0] CODE;
  output H;
  output[10:0] TO;
  reg[10:0] TO;
  reg[3:0] CODE;
  reg H;

  always@(INX)
  begin
    case(INX) //译码电路，查表方式
```

```
   0 : begin TO <= 11'H7FF; CODE<=0; H<=0; end
    1 : begin TO <= 11'H089; CODE<=1; H<=0; end
    2 : begin TO <= 11'H159; CODE<=2; H<=0; end
    3 : begin TO <= 11'H213; CODE<=3; H<=0; end
    4 : begin TO <= 11'H268; CODE<=4; H<=0; end
    5 : begin TO <= 11'H305; CODE<=5; H<=0; end
    6 : begin TO <= 11'H390; CODE<=6; H<=0; end
    7 : begin TO <= 11'H40C; CODE<=7; H<=0; end
    8 : begin TO <= 11'H444; CODE<=1; H<=1; end
    9 : begin TO <= 11'H4AD; CODE<=2; H<=1; end
   10 : begin TO <= 11'H50A; CODE<=3; H<=1; end
   11 : begin TO <= 11'H534; CODE<=4; H<=1; end
   12 : begin TO <= 11'H582; CODE<=5; H<=1; end
   13 : begin TO <= 11'H5C8; CODE<=6; H<=1; end
   14 : begin TO <= 11'H606; CODE<=7; H<=1; end
   15 : begin TO <= 11'H622; CODE<=1; H<=1; end
   default: begin TO <= 11'H622; CODE<=1; H<=1; end
   endcase
 end
endmodule
```

SPKER.v

```
module SPKER(CLK,TN,SPKS);
  input CLK; input[10:0] TN; output SPKS;
  reg SPKS; reg[10:0] CNT11;
  always@(posedge CLK) begin : CNT11B_LOAD//11 位可预置计数器
    if (CNT11==11'h7FF) begin CNT11=TN; SPKS<=1'b1; end
     else begin CNT11=CNT11+1; SPKS<=1'b0; end
     end
endmodule
```

DFF1.v

```
module DFF1(CLK,Q);
  output Q;
  input CLK;
  reg Q;
  always@(posedge CLK)
  Q <=~Q;
endmodule
```

seg7.v

```
module seg7(
          input clk,      //时钟信号，25MHz
          input rst_n,    //复位信号，低电平有效
```

```
          input[15:0] display_num,    //数码管显示数据，[15:12]--数码管千位，[11:8]--数码管百位，[7:4]--数码管十位，[3:0]--数码管个位
          output reg[3:0] dtube_cs_n, //7 段数码管位选信号
          output reg[7:0] dtube_data  //7 段数码管段选信号（包括小数点为 8 段）
       );

//----------------------------------------------
//参数定义

//数码管显示 0~F 对应段选输出
parameter   NUM0    = 8'h3f,//c0,
          NUM1    = 8'h06,//f9,
          NUM2    = 8'h5b,//a4,
          NUM3    = 8'h4f,//b0,
          NUM4    = 8'h66,//99,
          NUM5    = 8'h6d,//92,
          NUM6    = 8'h7d,//82,
          NUM7    = 8'h07,//F8,
          NUM8    = 8'h7f,//80,
          NUM9    = 8'h6f,//90,
          NUMA    = 8'h77,//88,
          NUMB    = 8'h7c,//83,
          NUMC    = 8'h39,//c6,
          NUMD    = 8'h5e,//a1,
          NUME    = 8'h79,//86,
          NUMF    = 8'h71,//8e;
          NDOT    = 8'h80;    //小数点显示

//数码管位选 0~3 对应输出
parameter   CSN     = 4'b1111,
          CS0     = 4'b1110,
          CS1     = 4'b1101,
          CS2     = 4'b1011,
          CS3     = 4'b0111;

//----------------------------------------------
//分时显示数据控制单元
reg[3:0] current_display_num;   //当前显示数据
reg[7:0] div_cnt;   //分时计数器

    //分时计数器
always @(posedge clk or negedge rst_n)
    if(!rst_n) div_cnt <= 8'd0;
```

```
    else div_cnt <= div_cnt+1'b1;

    //显示数据
always @(posedge clk or negedge rst_n)
    if(!rst_n) current_display_num <= 4'h0;
    else begin
        case(div_cnt)
            8'hff: current_display_num <= display_num[3:0];
            8'h3f: current_display_num <= display_num[7:4];
            8'h7f: current_display_num <= display_num[11:8];
            8'hbf: current_display_num <= display_num[15:12];
            default: ;
        endcase
    end

    //段选数据译码
always @(posedge clk or negedge rst_n)
    if(!rst_n) dtube_data <= NUM0;
    else begin
        case(current_display_num)
            4'h0: dtube_data <= NUM0;
            4'h1: dtube_data <= NUM1;
            4'h2: dtube_data <= NUM2;
            4'h3: dtube_data <= NUM3;
            4'h4: dtube_data <= NUM4;
            4'h5: dtube_data <= NUM5;
            4'h6: dtube_data <= NUM6;
            4'h7: dtube_data <= NUM7;
            4'h8: dtube_data <= NUM8;
            4'h9: dtube_data <= NUM9;
            4'ha: dtube_data <= NUMA;
            4'hb: dtube_data <= NUMB;
            4'hc: dtube_data <= NUMC;
            4'hd: dtube_data <= NUMD;
            4'he: dtube_data <= NUME;
            4'hf: dtube_data <= NUMF;
            default: ;
        endcase
    end

    //位选译码
always @(posedge clk or negedge rst_n)
    if(!rst_n) dtube_cs_n <= CSN;
    else begin
```

```
        case(div_cnt[7:6])
            2'b00: dtube_cs_n <= CS0;
            2'b01: dtube_cs_n <= CS1;
            2'b10: dtube_cs_n <= CS2;
            2'b11: dtube_cs_n <= CS3;
            default:  dtube_cs_n <= CSN;
        endcase
    end


endmodule
```

12.3 电子琴各模块功能和电路功能描述

sigkeyscan	输出消除抖动后的列按键按下键值
arykeyscan	根据 sigkeyscan 模块提供的列按下键值扫描键盘得出具体的按键位置
pll	产生 1MHz 时钟供 SPKER 模块分频
F_CODE	根据输入的音调(按键位置数值)产生分频预置数
SPKER	根据预置数分频，产生不同频率信号驱动蜂鸣器
DFF1	根据 spker 提供的信号驱动蜂鸣器
seg7	用数码管显示按键位置数值

电子琴可实现识别开发板上 4x4 键盘所输入的键值，16 个按键依次对应音谱简码“0-15”，同时通过蜂鸣器将对应音码发声演奏出来，同时数码管可依次显示刚按下的琴键所对应的音码，可通过复位键清零。

13.实验过程问题及解决描述

整体来说，这个实验的复杂程度比较高，自己独立完成还是比较吃力。同时书写报告等工作也较繁琐，但是通过查找资料和向同学求助可以很大程度上解决一些问题。本次实验中遇到的问题一是蜂鸣器发声频率和歌曲不对应，通过重新比对和计算预置数基本解决了。问题二是电子琴设计部分比较复杂，本人对开发板上 4x4 键盘的原理还不是很理解，所以在设计中很难将多个模块正确的连接，通过向同学请教和查找相关资料摸清了 4x4 按键的原理，梳理清楚电路的逻辑，也基本上将电子琴设计出来了。问题三是 pll 模块的仿真中遇到了 2KHz 分频端口无脉冲，而板级调试一切正常的情况，经过多次排查都无法发现错误原因，目

前还未能解决。

14.演示步骤说明

14.1 乐曲演奏电路演示：

1. “梁祝乐曲”循环播放，在数码管显示当前播放的音号，个位为对应音号“0~7”，千位为对应音高“0~2”。
2. 通过拨码开关切换另一首歌曲“欢乐颂”。
3. 按下复位键，可实现数码管清零和当前歌曲复位，重新播放。
4. 通过拨码开关可重新切换到“梁祝”歌曲。

14.2 电子琴演示：

1. 依次按下 4x4 键盘上的所有按键，实现依次弹奏“0~15”的音号，蜂鸣器音调越来越高，数码管个位会依次显示当前弹奏的音号。（此开发板的 S15 键是坏的，因此按下 S15 时无法发声，数码管无法显示）
2. 按下复位键，实现数码管清零。