数电 Verilog HDL 作业

6.2 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

```
module mux4_1(in0, in1, in2, in3, sel, out);
    input in0, in1, in2, in3;
    input[1:0] sel; // 选择信号
    output out;

    // 根据选择信号的值选出对应输入作为输出
    assign out = (sel == 0) ? in0 : (sel == 1) ? in1 : (sel ==2)? in2 : in3;
endmodule
```

② 仿真：

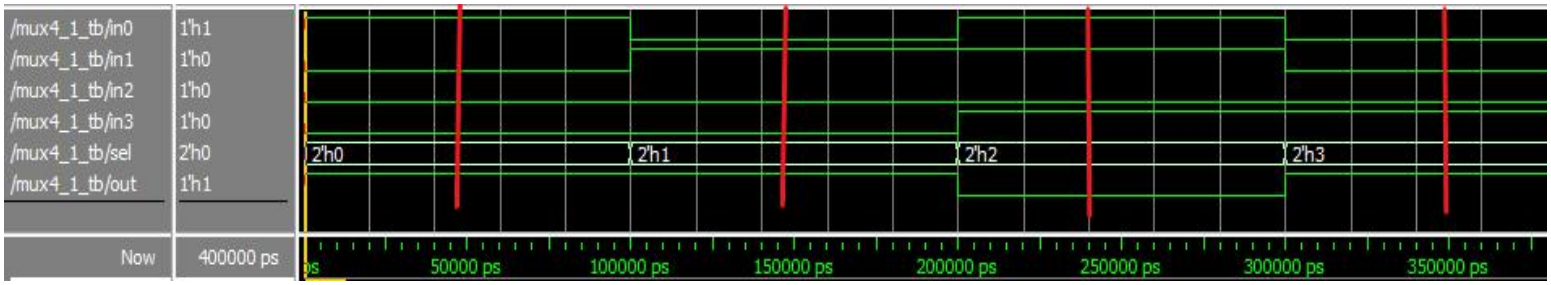

图 6.2 四选一数据选择器仿真结果

如图，当 sel = 0 时，输出 out = in0 = 1；当 sel = 1 时，输出 out = in1 = 1；当 sel = 2 时，输出 out = in2 = 0；当 sel = 3 时，输出 out = in3 = 1，符合设计要求。

6.3 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

```
module counter_8bits(clk, reset, q);
      input clk, reset;
      output reg [7:0] q;

      always @ (posedge clk) // 时钟上升沿触发
      begin
        if (reset == 1) q = 8'b0; // 同步复位
        else
```

```
        begin
          if (q == 8'hff) q = 0; // 溢出则清 0
        else q = q + 1; // 计数
        end
      end
endmodule
```

② 仿真：

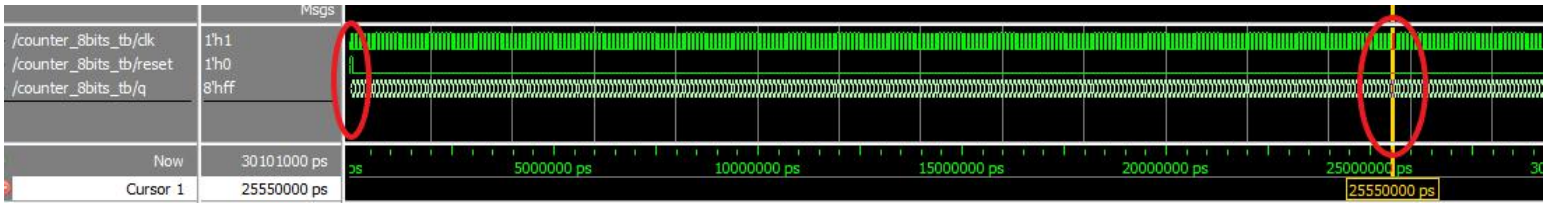

图 6.3.1 八位计数器仿真结果

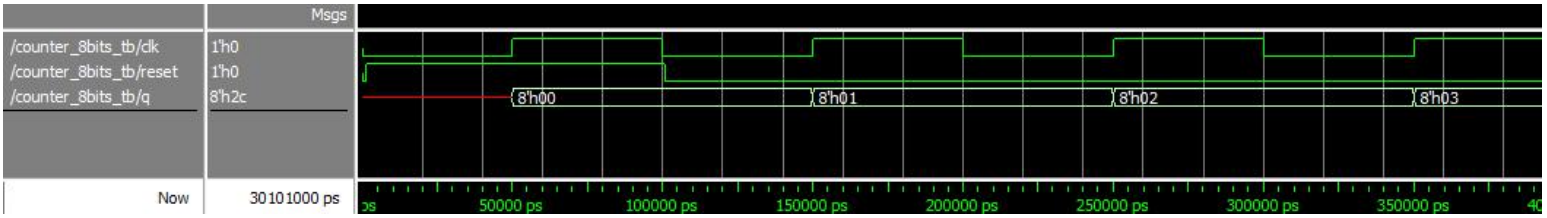

图 6.3.2 八位计数器仿真结果局部放大图(reset 置零)

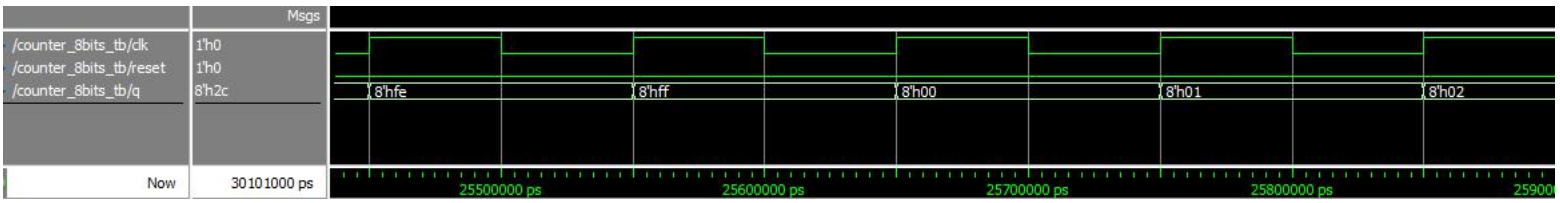

图 6.3.3 八位计数器仿真结果局部放大图(溢出置零)

如图，当时钟上升沿到来时，计数器的计数值加 1，当计数结果为 8'hff 时，计数器将自动从 0 开始重新计数。若时钟上升沿时 reset = 1，则计数结果将置 0，符合设计要求。

6.4 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

```
module shift_reg_4bits(clk, reset, in, out);
    input clk, reset, in;
    output reg[3:0] out = 4'b0;

    always @ (posedge clk)
    begin
```

```
        if (reset) out = 4'b0; // 同步清零，高电平有效
        else
        begin
            out <= out<<1; // 输出信号左移一位(并行输出)
            out[0] <= in; // 输入信号补充到最低位
        end
    end
endmodule
```

② 仿真：

图 6.4 四位移位寄存器仿真结果

如图，若时钟上升沿时 reset = 1，则四位并行输出将清 0；在其余情况下，移位寄存器会将保存数值左移一位，并将输入补充到最低位。以图中为例，时钟上升沿到来时，若 out = 4’b0000，in = 1，则此时输出为 out = 4’b0001；若 out = 4’b0001，in = 0，则此时输出为 out = 4’b0010。由此可知，设计符合要求。

6.9 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

```
module comp_8bits(in, out);
    input[7:0] in;
    output[7:0] out;

    // 根据首位判断处理方式，若首位为 0，则补码等于原码，否则除首位外求反再加 1
    assign out = (in[7] == 1'b0) ? in : {in[7], (~in[6:0] + 1)};
endmodule
```

② 仿真：

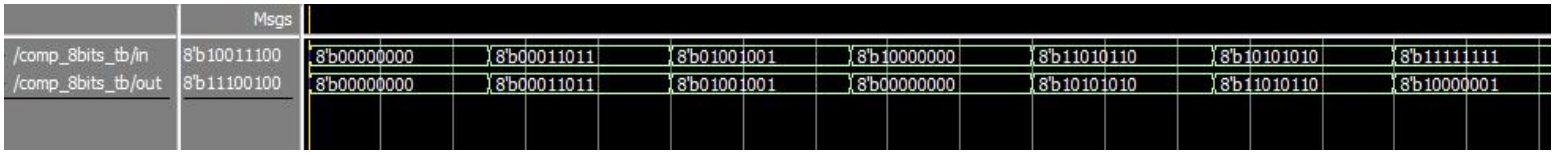

图 6.9 求八位二进制符号数补码仿真结果

如图，正数的补码为其本身，符合要求；对负数，8’b10000000 的补码为 8’b00000000，8’b11010110 的补码为 8’b10101010，等等，计算结果均正确，故设计符合要求。

7.3 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

i) 结构描述：

```
module JK_trigger(clk, J, K, q, qn);
    input J, K, clk; // 输入
    output q, qn; // 输出
    wire clk_0, x1, y1, q1, q1n, x2, y2;

    nand(x1, J, qn, clk);
    nand(y1, K, q, clk);
    not(clk_0, clk);
    nand(q1, x1, q1n);
    nand(q1n, y1, q1);
    nand(x2, q1, clk_0);
    nand(y2, q1n, clk_0);
    nand(q, x2, qn);
    nand(qn, y2, q);
endmodule
```

ii) 行为描述：

```
module JK_trigger(clk, J, K, q, qn);
    input J, K, clk; // 输入
    output reg q;
    output wire qn; // 输出
```

```
        always @ (posedge clk)
        begin
            if (J == 1 && K == 1) q = ~q; // 翻转
            else if (J == 1 && K == 0) q = 1; // 置 1
            else if (J == 0 && K == 1) q = 0; // 置 0
            else q = q; // 保持
        end
        assign qn = ~q;
    endmodule
```

② 综合：

i) 结构描述：

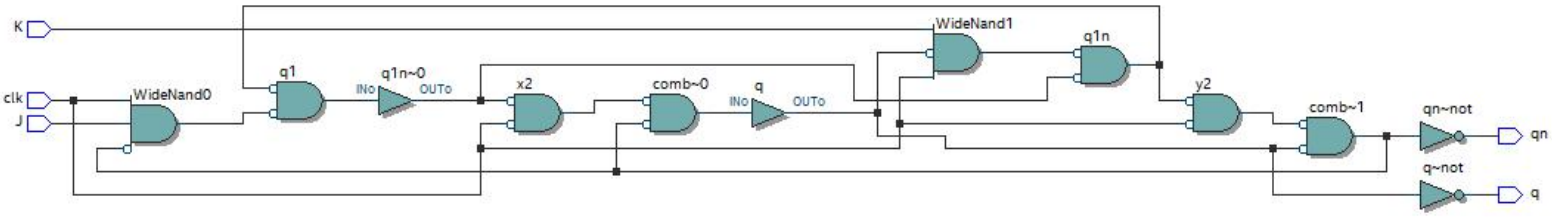

图 7.3.1 JK 触发器综合图(结构描述)

ii) 行为描述：

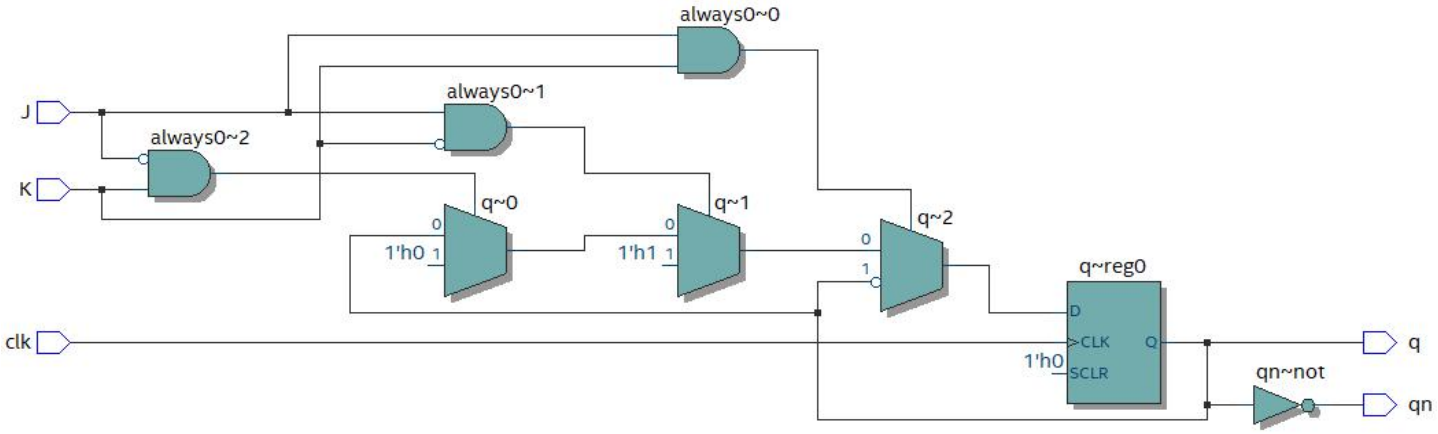

图 7.3.2 JK 触发器综合图(行为描述)

③ 仿真：

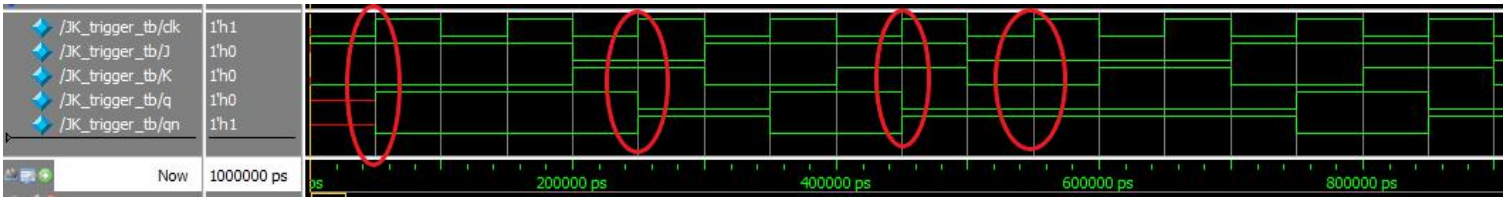

图 7.3.3 JK 触发器仿真结果

如图，在时钟上升沿，当 J = 1, K = 0 时，q = 1；当 J = 0, K = 1 时，q = 0；当 J = 1, K = 1 时，q 翻转；当 J = 0, K = 0 时，q 保持，且上述过程中 q 始终与 qn 相反，符合设计要求。

7.6 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

```
module pal_se_4bits(clk, reset, load, in, out);
    input clk, reset, load;
    input[3:0] in;
    output reg out;
    reg[3:0] buff; // 保存串行数据

    always @ (posedge clk)
    begin
        if (reset) buff = 4'b0; // 同步清零，高电平有效
        else if (load) buff = in; // 载入数据，高电平有效
        // 时钟上升沿输出一位
        out = buff[3];
        buff <= buff<<1; // 左移
    end
endmodule
```

② 仿真：

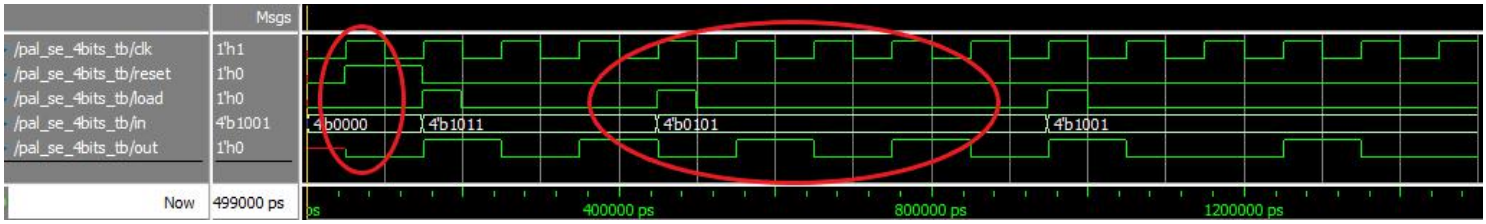

图 7.6 四位并串转换器

如图，在时钟上升沿，若 reset = 1，则输出 out = 0；当时钟上升沿 load = 1 时，转换器将预置数存入寄存器，并每个时钟周期输出一位，例如，当预置数为 4’b0101 时，转换器的输出依次为 0、1、0、1，之后若未置数，则输出为 0，其余情况类似。综上，设计符合要求。

7.9 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

```
module light(clk, reset, q);
    input clk, reset;
    output reg[7:0] q;  // 控制花灯，某一位为 1，则对应花灯亮起
    reg[3:0] state;
    parameter s0 = 4'b0000,
              s1 = 4'b0001,
              s2 = 4'b0011,
              s3 = 4'b0010,
              s4 = 4'b0110,
              s5 = 4'b0111,
              s6 = 4'b0101,
              s7 = 4'b0100,
              s8 = 4'b1100,
              s9 = 4'b1101,
              s10 = 4'b1111,
              s11 = 4'b1110; // 状态编码

    always @ (posedge clk)
    begin
        if (reset)
        begin
            q = 8'b0; // 同步清零，高电平有效
            state = s0;
        end
        else
        begin
            case (state)
                s0: begin state = s1; q = 8'h0; end // 8 路彩灯同时亮灭
                s1: begin state = s2; q = 8'hff; end
                s2: begin state = s3; q = 8'h01; end // 从左至右逐个亮起
                s3: begin state = s4; q = 8'h02; end
                s4: begin state = s5; q = 8'h04; end
```

```
                s5: begin state = s6; q = 8'h08; end
                s6: begin state = s7; q = 8'h10; end
                s7: begin state = s8; q = 8'h20; end
                s8: begin state = s9; q = 8'h40; end
                s9: begin state = s10; q = 8'h80; end
                s10: begin state = s11; q = 8'b01010101; end // 4亮4灭
                s11: begin state = s0; q = 8'b10101010; end
                default: begin state = s0; q = 8'b0; end
            endcase
        end
    end
endmodule
```

② 仿真：

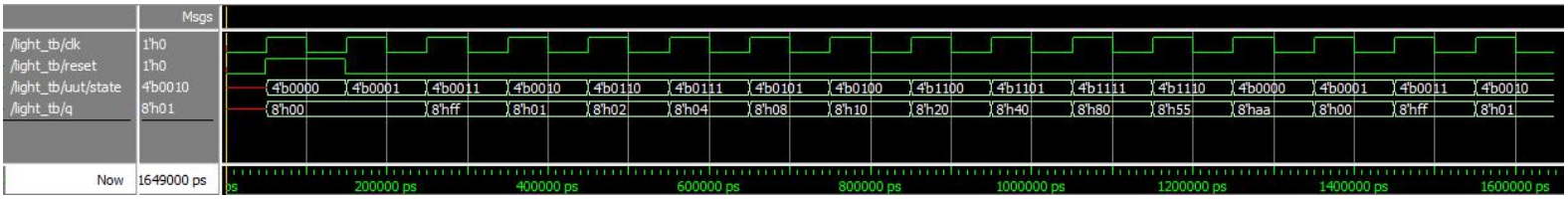

图 7.9 彩灯控制程序

如图，以输出 q 的每一位取值表示某一盏灯的亮灭，1 表示亮，0 表示灭，则随着时钟信号，输出 q 依次为 8’b00000000、8’b11111111、8’b00001111、8’b00000010、8’b00000100、8’b00001000、8’b00010000、8’b00100000、8’b01000000、8’b10000000、8’b01010101、8’b10101010（十六进制格式见仿真结果图），彩灯亮灭符合要求，且状态循环输出，故程序正确。

9.3 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

```
module seq1101(clk, reset, in, out);
    input clk, reset, in;
    output reg out;
    reg[2:0] state;
    parameter s0 = 3'b000,
              s1 = 3'b001,
              s2 = 3'b011,
```

```
          s3 = 3'b010,
          s4 = 3'b110; // 状态编码

always @ (posedge clk)
begin
    if (reset) // 同步复位，s0 为起始状态，输出为 0
    begin
        state = s0;
        out = 0;
    end
    else
    begin
        case (state)
            s0: begin if (in) state = s1;
                      else state = s0; end // 检测到第一个'1'
            s1: begin if (in) state = s2;
                      else state = s0; end // 检测到第二个'1'
            s2: begin if (!in) state = s3; // 检测到'0'
                      else state = s2; end // 否则检测到'1'，返回 s2
            s3: begin if (in) state = s4;
                      else state = s0; end // 检测到第三个'1'，即检测到该序列
            s4: begin if (in) state = s2;
                      else state = s0; end // 新的开始
            default: state = s0;
        endcase
    end
end
always @ (*)
begin
    case (state)
        s4: out = 1;
        default: out = 0;
    endcase
```

```
    end
endmodule
```

② 仿真：

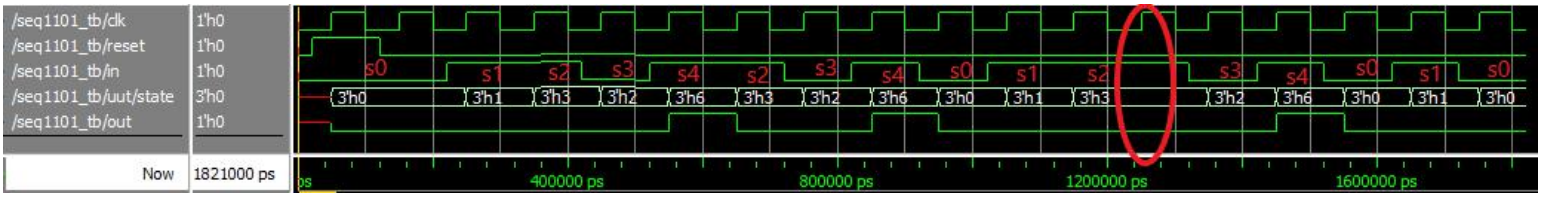

图 9.3 1101 序列检测器仿真结果

如图，各状态已标记在图中。开始工作后，在时钟上升沿，若检测器检测到第一个 1，则状态机状态由 s0 跳到 s1，若继续检测到 1、0、1，则状态机由 s1 -> s2 -> s3 -> s4，并输出 out = 1。此时若继续在时钟上升沿检测到 1，由于前一个数字为 1，所以状态机直接跳到 s2，即检测到第二个 1，然后继续工作。若在检测序列时，有与 1101 顺序不同的输入出现，则会进行对应的状态转换，例如，若已检测到两个 1，当继续输入 1 时，state 状态保持为 s2；若已检测到第一个 1，当下一个输入为 0 时，state 状态为 s0。综上，设计符合要求。

9.5 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

i) 顶层设计

```
module music_player(clk_10m, reset, speaker);
    input clk_10m, reset;
    output speaker;

    wire[5:0] q; // 计时
    wire[3:0] high, med, low; // 音乐信息
    wire clk_4, clk_5m;

    // 地址计数器，得到当前乐符位置
    counter_n #(.n(63), .counter_bits(6)) cnt0(.clk(clk_4), .en(1), .r(reset),
                                              .q(q), .co());

    // 由 10MHz 分频得到 5MHz 时钟
    counter_n #(.n(2), .counter_bits(1)) cnt1(.clk(clk_10m), .en(1), .r(reset),
```

```
                                                    .q(), .co(clk_5m));

    // 由 5MHz 分频得到 4Hz 时钟
    counter_n #(.n(1250000), .counter_bits(21)) cnt2(.clk(clk_5m), .en(1),
                                                    .r(reset), .q(), .co(clk_4));

    // 只读存储器
    song_rom rom0(.clk(clk_4), .dout({high, med, low}), .addr(q));

    // 分频器及输出
    divider d0(.clk_5m(clk_5m), .clk_4(clk_4), .din({high, med, low}),
               .speaker(speaker));
endmodule
```

ii) 计数器(分频器)

```
module counter_n(clk, en, r, q, co);
    parameter n = 2; // 计数的模(分频比)
    parameter counter_bits = 1; // 状态位数
    input clk, en, r ;
    output co; // 进位输出
    output reg[counter_bits-1:0] q = 0;

    assign co = (q == (n-1)) && en; //进位

    always @(posedge clk)
    begin
      if (r) q = 0;
      else if(en)  // 使能高电平计数
      begin
        if (q == (n-1)) q = 0 ; // 清 0, 实现循环播放
        else q = q + 1; // 计数
      end
      else q = q; // 否则保持
```

```
    end
endmodule
```

iii) 分频器及输出

```
module divider(clk_5m, clk_4, din, speaker);
    input clk_5m, clk_4;
    input[11:0] din; // 音乐信息
    output reg speaker = 0;

    reg[13:0] origin = 0, divide = 0; // 分频比
    reg carry = 0;

    always @ (posedge clk_5m) // 通过置数，改变分频比(carry)
    begin
        if (origin)
        begin
            if (divide == 16383)
            begin
                carry = 1;
                divide = origin;
            end
            else
            begin
                divide = divide + 1;
                carry = 0;
            end
        end
        else divide = 0;
    end

    always @ (posedge carry) // 输出
    begin
        speaker = ~speaker;
```

```
    end

    always @ (posedge clk_4) // 频率
    begin
        case (din)
            'h001: origin <= 4915;
            'h002: origin <= 6168;
            'h003: origin <= 7281;
            'h004: origin <= 7792;
            'h005: origin <= 8730;
            'h006: origin <= 9565;
            'h007: origin <= 10310;
            'h010: origin <= 10647;
            'h020: origin <= 11272;
            'h030: origin <= 11831;
            'h040: origin <= 12094;
            'h050: origin <= 12556;
            'h060: origin <= 12974;
            'h070: origin <= 13346;
            'h100: origin <= 13516;
            'h200: origin <= 13829;
            'h300: origin <= 14109;
            'h400: origin <= 14235;
            'h500: origin <= 14470;
            'h600: origin <= 14678;
            'h700: origin <= 14864;
            'h000: origin <= 16383;
        endcase
    end
endmodule
```

iv) 音乐存储 ROM，代码较长，详见文件夹中文件 song_rom.v

② 仿真：

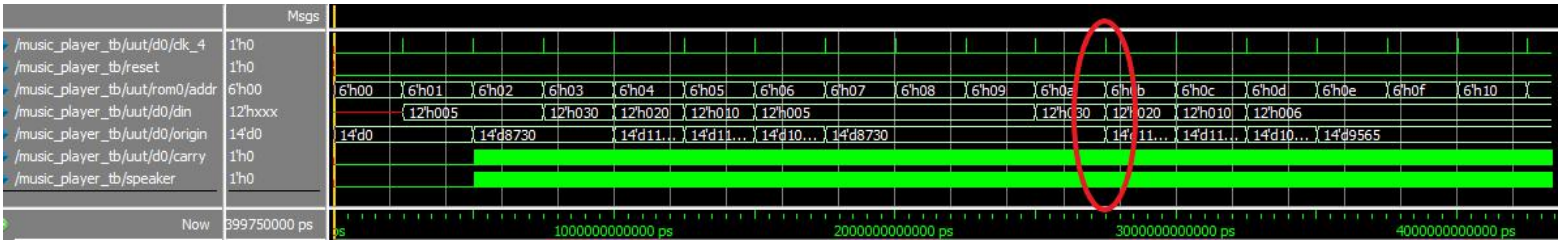

图 9.5.1 音乐播放器仿真结果

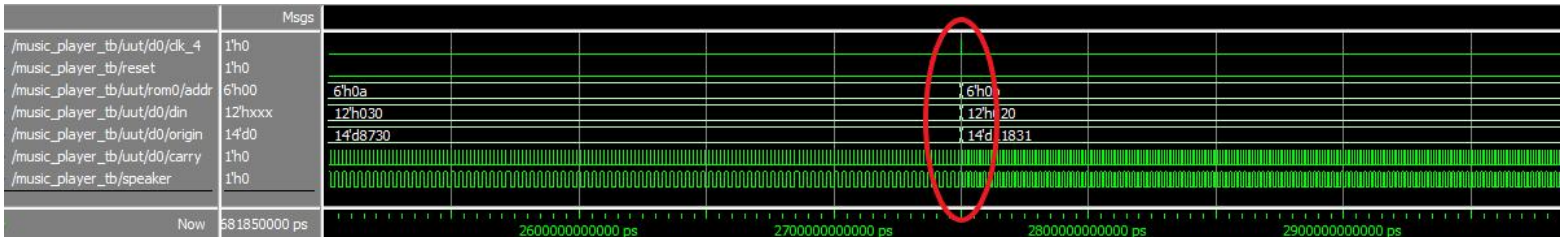

图 9.5.2 音乐播放器仿真结果(局部放大图)

如图，程序以方波信号 speaker 作为输出，其频率及该频率持续时间作为音符和音长。经过分频后，clk_4 的周期为 0.25s，符合预期。din 为每拍的音符，通过其选择相对应的初始频率 origin，计数 origin 到休止符频率 16383，从而将 5MHz 时钟分频，由此得到的 carry 信号将控制 speaker 上跳及下跳，即控制了 speaker 的频率。通过从 ROM 中循环读取音符，可以在每一拍输出对应频率的方波，进而控制扬声器播放音乐。由图 9.5.2 可见，在 clk_4 上升沿两侧，speaker 方波的频率存在差异，这是因为 origin 即两拍之间的音符不同，符合预期。综上，电路设计正确。

9.6 解：

① 设计代码（测试代码详见文件夹_tb.v 文件）：

i) 顶层设计

```
module jukebox(clk_10m, reset, song, speaker);
    input clk_10m, reset;
    input[1:0] song; // 四首歌
    output speaker;

    wire[5:0] q; // 计时
    wire[3:0] high, med, low; // 音乐信息
    wire clk_4, clk_5m;

    // 地址计数器，得到当前乐符位置(每首歌最多 64 拍，不足用休止符补充)
```

```
    counter_n #(.n(64), .counter_bits(6)) cnt0(.clk(clk_4), .en(1), .r(reset),
                                                .song(song), .q(q), .co());

    // 由 10MHz 分频得到 5MHz 时钟
    counter_n #(.n(2), .counter_bits(1)) cnt1(.clk(clk_10m), .en(1), .r(reset),
                                                .song(0), .q(), .co(clk_5m));

    // 由 5MHz 分频得到 4Hz 时钟
    counter_n #(.n(1250000), .counter_bits(21)) cnt2(.clk(clk_5m), .en(1),
                                            .r(reset), .song(0), .q(), .co(clk_4));

    // 只读存储器
    songs_rom rom0(.clk(clk_4), .dout({high, med, low}), .addr({song, q}));

    // 分频器及输出
    divider d0(.clk_5m(clk_5m), .clk_4(clk_4), .din({high, med, low}),
                .speaker(speaker));
endmodule
```

ii) 计数器(分频器)

```
module counter_n(clk, en, r, song, q, co);
    parameter n = 2; // 计数的模(分频比)
    parameter counter_bits = 1; // 状态位数
    input clk, en, r;
    input wire[1:0] song;
    output co; // 进位输出
    output reg[counter_bits-1:0] q = 0;

    assign co = (q == (n-1)) && en; //进位
    if (song == 2'bxx)
        assign song = 0; // 默认播放第一首歌

    always @(posedge clk)
```

```
    begin
      if (r) q = 0;
      else if(en)  // 使能高电平计数
      begin
        if (q == (n-1)) q = 0 ; // 清0
        else q = q + 1; // 计数
      end
      else q = q; // 否则保持
    end

    always @ (song) q = 0; // 只要乐曲变化，就从第一拍开始播放
endmodule
```

iii) 分频器及输出

```
module divider(clk_5m, clk_4, din, speaker);
    input clk_5m, clk_4;
    input[11:0] din; // 音乐信息
    output reg speaker = 0;

    reg[13:0] origin = 0, divide = 0; // 分频比
    reg carry = 0;

    always @ (posedge clk_5m) // 通过置数，改变分频比(carry)
    begin
        if (origin)
        begin
            if (divide == 16383)
            begin
                carry = 1;
                divide = origin;
            end
            else
            begin
```

```
            divide = divide + 1;
            carry = 0;
        end
    end
    else divide = 0;
end

always @ (posedge carry) // 输出
begin
    speaker = ~speaker;
end

always @ (posedge clk_4) // 频率
begin
    case (din)
        'h001: origin <= 4915;
        'h002: origin <= 6168;
        'h003: origin <= 7281;
        'h004: origin <= 7792;
        'h005: origin <= 8730;
        'h006: origin <= 9565;
        'h007: origin <= 10310;
        'h010: origin <= 10647;
        'h020: origin <= 11272;
        'h030: origin <= 11831;
        'h040: origin <= 12094;
        'h050: origin <= 12556;
        'h060: origin <= 12974;
        'h070: origin <= 13346;
        'h100: origin <= 13516;
        'h200: origin <= 13829;
        'h300: origin <= 14109;
        'h400: origin <= 14235;
```

```
            'h500: origin <= 14470;
            'h600: origin <= 14678;
            'h700: origin <= 14864;
            'h000: origin <= 16383;
        endcase
    end
endmodule
```

iv) 音乐存储 ROM，代码较长，每个音符的位置分为两部分，第一部分为乐曲序号，第二部分为该音符在乐曲中的位置，详见文件夹中文件 songs_rom.v

② 仿真：

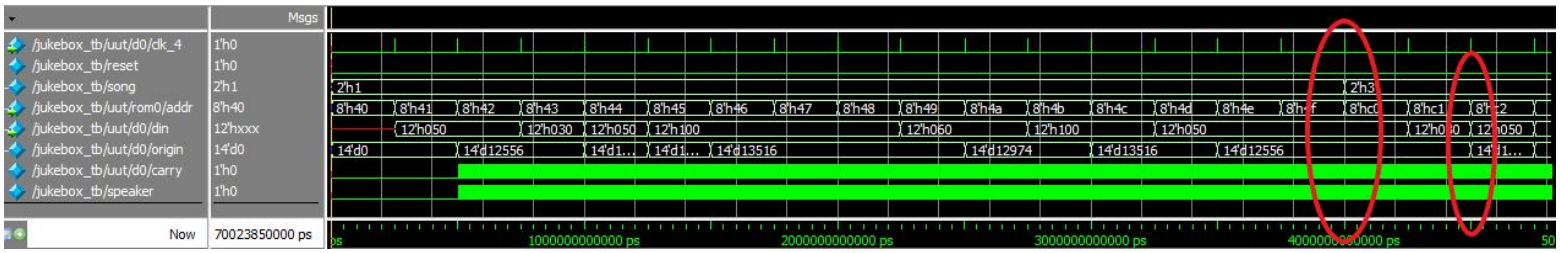

图 9.6.1 点播器仿真结果

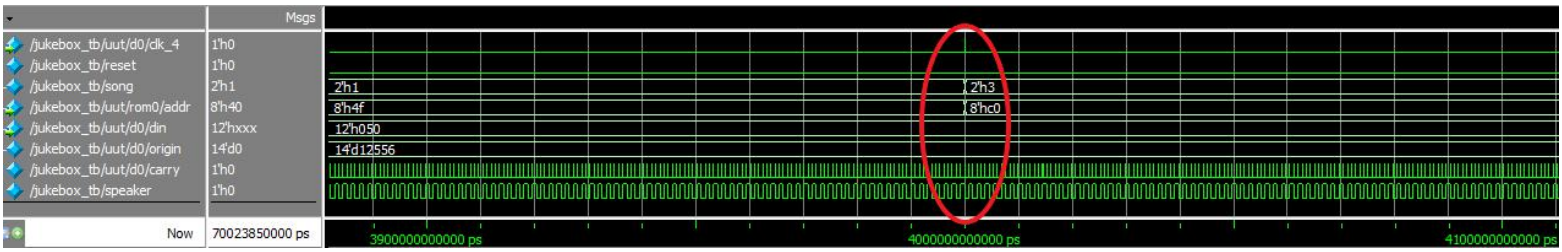

图 9.6.2 点播器仿真结果(切歌)

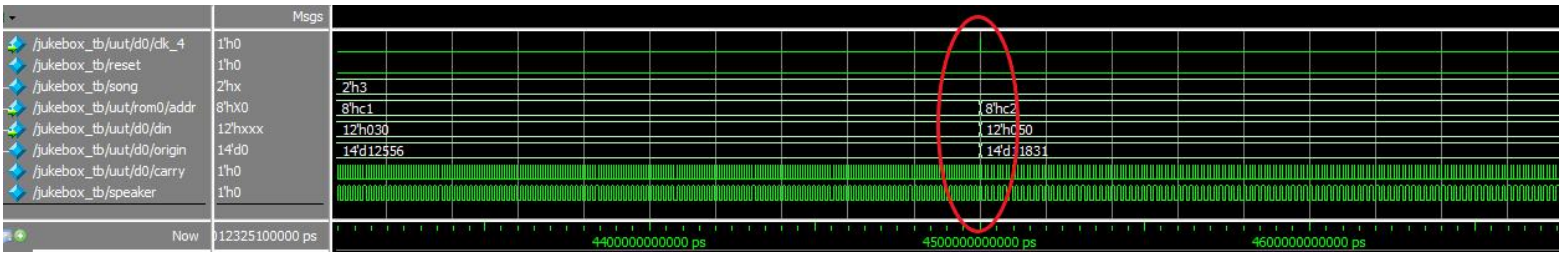

图 9.6.3 点播器仿真结果(局部放大图)

如图，此点播器每首歌曲的播放原理与 9.5 类似，不同之处在于乐曲的选择。因此，程序添加了输入变量 song 作为乐曲序号，其与 songs_rom 输入 addr 的前两位相匹配，用于唯一选择对应音符。当一首歌播放完毕后，由于 song 未改变，因此会自动循环播放，符合要求；若手动切歌，以图 9.6.1 与 9.6.2 为例，song 由 1 变为 3，与此同时下一个时钟到来时 din(即音符信息)会从

第四首歌(song = 0 对应第一首歌)的第一个音符开始输出(详见 songs_rom)，符合手动选择歌曲播放要求。除此之外，由图 9.6.3 可见，clk_4 上升沿两侧 speaker 方波的频率不同，即音符频率(origin)不同，故程序可正确从 ROM 中读取音符信息并输出。综上，此点唱机设计正确。