Fudan University

专用集成电路设计

人体反应测试仪——课程设计报告

姓名：王文忠

学号：21307130032

2024 年 6 月 12 日

1 设计规划

1.1 设计要求

设计一个人体反应测试仪，用 4 个 7 段 LED 作为显示，2 个按键分别表示启动（START）和反应按键（REACT）。Start 键表示开始测试，内部计数器开始计数，四个 7 段显示为 $----$，接着随机时间（500ms 到 5S 不等）变为 1111 被测试的人看到后快速按反应按键，测试仪显示从开始变为 1111 到被测试人按反应按键的时间，即为被测试人的反应时间。该反应时间的分辨率为 1ms，所以最大可测试时间为 9999ms，但是如果此时间为负或小于人体最小反应时间 (100ms)，则测试为 Fail(显示 FaiL)。

1.2 设计思路

此次设计过程严格遵循 Top-Down 设计方法，根据设计需求，首先从明确电路系统的输入输出要求和性能指标入手，随后逐步深入到系统级、寄存器传输级、逻辑门级直至物理器件级的详细设计与验证工作。通过分析人体反应测试仪的各个阶段的状态转换，初步得到系统级设计方案。在此基础上，进一步开展了 RTL 级设计与验证工作，逐一编写符合要求的各个承担内部逻辑信号转换或输入输出转换不同功能的子模块，最后统一在顶层实现实例化。为确保设计的正确性与可靠性，编写了不同适应模块测试需求的 testbench 文件，进行了一系列测试工作，包括对各个子模块的功能仿真以及最后综合时序仿真，得到理想波形。最终，利用型号为 xc7a35tcpg236-1 的 FPGA 开发板完成了设计的物理实现。

根据需求，设定了以下子模块以满足各个功能：

1. 按键消抖、传输启动模块 buffet_elimination——此模块的设计是为了延长 start 和 react 按键的高电平信息，以便在后续信号传输中识别到已”start” 或者已”react” 的信息。同时加入消抖识别，以防止不理想的突刺信号影响结果。

2. 随机时间计时模块 random_count——此模块的设计是为了得到（伪）随机时间，同时计时，以便随机时间计时完成传递计时完成信号 random_finish 信号给到后续模块来响应。

3. 反应时间计时模块 react_time_count——此模块的设计是为了计算人体的反应时间，传递两个结果，一个是是否超出 9999ms 的最大反应时间的限制的信号 react_exceed，另一个是反应时间的结果 t_react，传递给 display 模块达到显示时间的目的。

4. 结果显示模块 display——此模块的设计是为了最终在 4 位 7 段 LED 上显示结果，接受各种影响显示结果的信号，通过真值表和内部逻辑的判定最终以数码管自己的编码方式显示 5 种结果，即不显示、显示”− − −−””1111””FAIL” 和反应时间的十进制表示。

5. 分频模块 fre_divide——此模块的设计是考虑了上板需求，因为目标 FPGA 的频率为 100MHz，而本设计的频率要求为 1000Hz（分辨率为 1ms），因此添加分频器，以达到频率转换的目的。

1.3 设计流程图

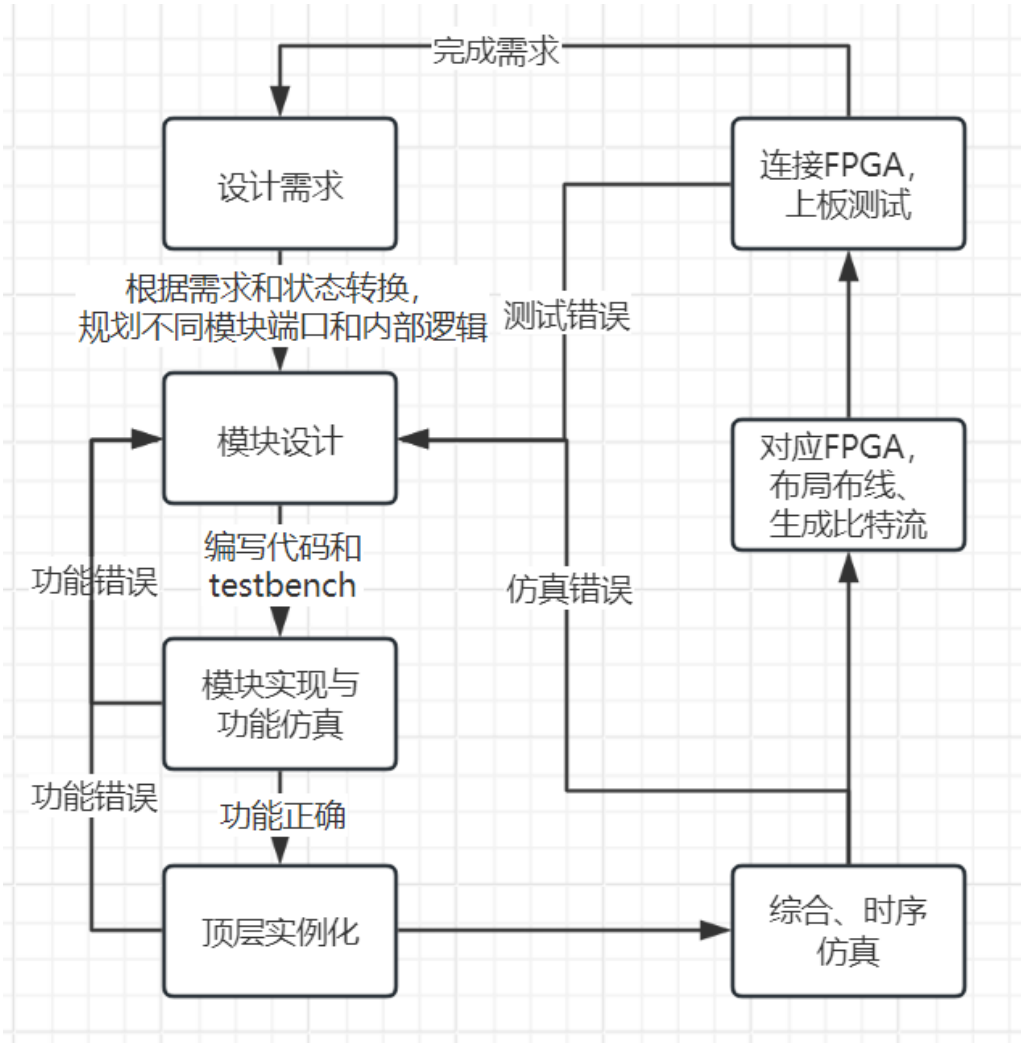

图 1: 设计流程图

1.4 设计平台

本设计采用 Vivado 平台完成模块的编写、仿真测试和综合。在 FPGA 的开发过程中，一个合适的开发环境是至关重要的。Vivado 作为主流的开发工具之一，其功能性强大，支持多样化的数据输入手段。此外，Vivado 还内置了综合器和仿真器，这些工具可以协助开发者从新建工程开始，依次完成设计输入、分析综合、约束输入以及设计实现等关键步骤。最终，Vivado 能够生成 bitstream 文件，使得用户能够顺利地将设计下载到 FPGA 中，从而完成整个开发流程。

2 设计实现

2.1 框图介绍

为满足需求，先做出状态转换图理清信号转换关系。如下图，将 LED 数码管的显示结果与输入和中间信号联系，具体真值表在各模块设计中的结果显示模块 display 中有详细说明。

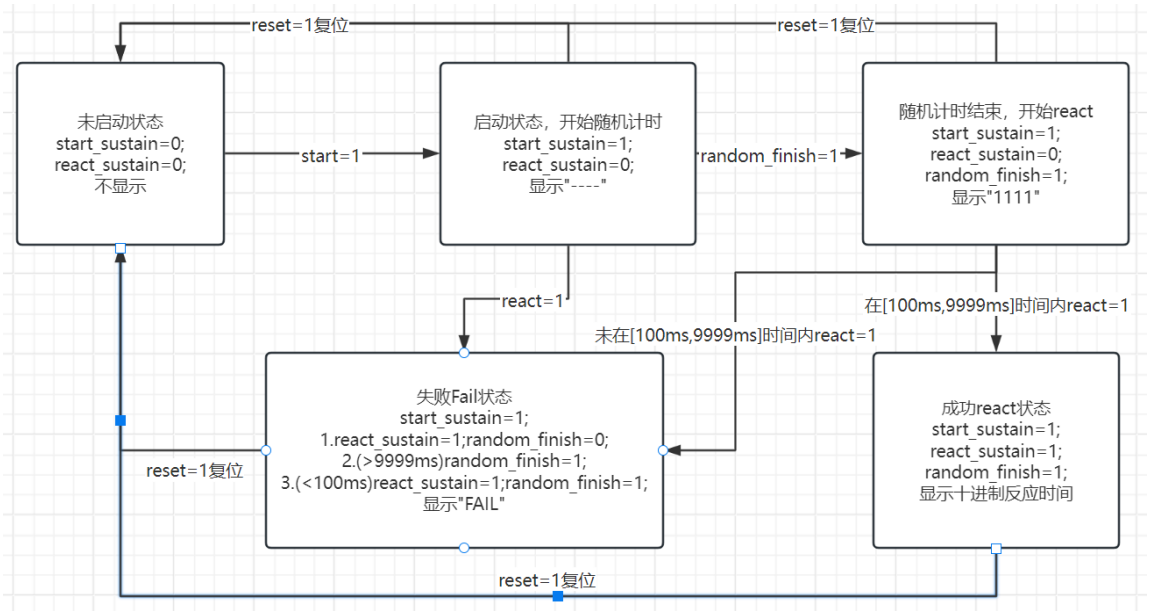

图 2: 状态转换图

本设计最终的电路图如下，依据前面所阐述的几个模块，首先时钟分频模块 fre_divide 先得到本设计需求的 1000Hz 的时钟频率。然后设计按键消抖、传输启动模块 buffet_elimination，检测是否抖动并且延长 start 和 react 的信号高电平以利于后续逻辑判断。启动 start 按键

后，随机计时模块 random_count 启动，伪随机数的确定在按下 start 键后确定，并且在计时到随机时间后输出完成随机时间计时信号 random_finish 信号为高电平。反应时间计时模块在接收到 random_finish 被置为高电平后启动开始计时，当 react 按键后，停止计时并输出反应时间 t_react，同时检测计时是否超过了 9999ms，输出是否超时信号 react_exceed，以利于后续 FAIL 状态判断。然后将所有信号传输至结果显示模块 display，在这里根据各个信号和结果显示的逻辑关系，给出最终要显示在 4 位 7 段 LED 数码管的编码，具体情况见各模块设计中的结果显示模块 display。

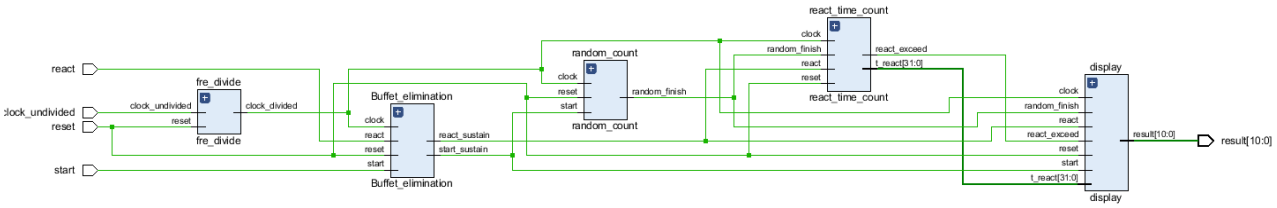

图 3: 电路原理图

2.2 各模块设计和验证（仿真激励和结果说明）

2.2.1 按键消抖、传输启动模块 buffet_elimination

本模块的设计是为了延长接收到的瞬变信号 start 和 react 信号在整个测试流程的高电平信息，同时加入消除抖动功能，即忽略毛刺信号 start 或者 react，只有真正遍历一定周期（本设计是未加入分频器前 4 个周期，即保持 4ms 的高电平信号），才视为真正按下 start 或者 react 键，然后延长其高电平至整个测试流程结束（reset 被置为 1）。仿真激励输入如下：

```
initial begin
    reset = 0;
    start = 0;
    react = 0;
    #5 start = 1; react = 1;
    #2 start = 0; react = 0; reset = 1;
    #1 reset = 0;
    #3 start = 1; react = 1;
    #5 start = 0; react = 0;
    #8 reset = 1;
    #1 reset = 0;
end
```

图 4: 按键消抖、传输启动模块仿真激励

输出波形结果如下，可以看到第一阶段 start 和 react 信号仅持续 2ms，被视为毛刺信号，因此输出的持续测试响应 start_sustain 和 react_sustain 信号均为 0。在第二阶段，持续 5ms

的 start 和 react 的置 1，可以看到在第 4ms 时就已经输出 start_sustain 和 react_sustain 信号均为 1，并一直持续直到复位信号 reset 被置为 1。

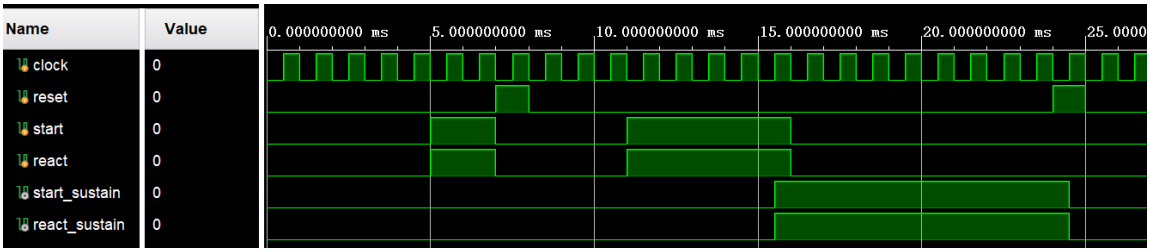

图 5: 按键消抖、传输启动模块仿真结果

代码如下：

```
module Buffet_elimination(
    input clock,
    input reset,
    input start,
    input react,
    output reg start_sustain,
    output reg react_sustain
    );

    reg [3:0] start_count;
    reg [3:0] react_count;

    initial begin
        start_sustain = 0;
        react_sustain = 0;
        start_count = 0;
        react_count = 0;
    end

    always @(posedge clock or posedge reset)begin
        if(reset)begin
```

```
            start_sustain <= 0;
            react_sustain <= 0;
            start_count <= 0;
            react_count <= 0;
        end
        else begin
        if(start && start_count < 4)begin
            start_count <= start_count + 1;
        end
        if(start_count >= 4)begin
            start_sustain <= 1;
        end
        if(react && react_count < 4)begin
            react_count <= react_count + 1;
        end
        if(react_count >= 4)begin
            react_sustain <= 1;
        end
        end
        end
    end
endmodule
```

2.2.2 随机时间计时模块 random_count

本模块接受 start 信号和 clock 信号，在 start 信号 =1 时生成随机时间 randomtime，并在 start=1 后等待随机时间后输出随机时间结束信号 random_finish=1。

如下图 2.2.1.1，在 start=0（即 start 尚未置为 1），可以看到随机时间在每一周期都在不断迭代更新，随机数生成原理运用线性反馈移位寄存器（LFSR），反馈信号 feedback 通过异或操作生成。$lfsr <= \{lfsr[14:0], feedback\}$ 语句保持随机信号的不断更新，在 start 尚未置为 1 前产生足够的伪随机样本，同时为了满足随机时间大于 500ms，小于 5s 的要求，运用取余语句得出每个随机数对应的符合规定的随机时间：$randomtime <=$

$MIN\_COUNT + (lfsr\%(MAX\_COUNT - MIN\_COUNT))$，这里设置 MIN_COUNT 为 500，MAX_COUNT 为 5000。以下图 2.2.1 为例，以十六进制表示，可以看到每个周期的随机时间（以 ms 为单位）：832，3129，3222，1445……

图 6: 随机时间的产生与更新

当 start 被置为 1 时，随即停止随机数的更新，并开始周期计数，仿真阶段已设置一个周期为 1ms，因此每个周期设置计数器 counter 自加一，当到达确定好的随机时间 randomtime 后，即将随机时间结束信号 random_finishzhi 置为 1 输出。如下图 2.2.1.2 和图 2.2.1.3，start 信号在 testbench 设置为 1000ms 时被置为高电平，此时随机时间是 4731ms（十六进制为 0x127b），因此可以看到图 2.2.1.3 上经历 4731.5ms 后，即在 5731.5ms，得到输出的 random_finish 被置为高电平，多出来的 0.5ms 是由于在 clock 上升沿触发的缘故。

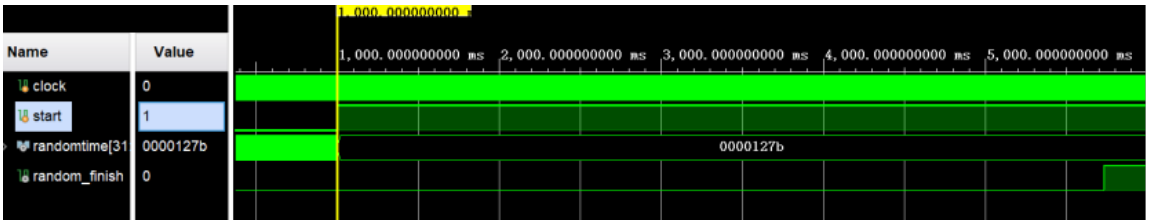

图 7: 随机时间开始计时

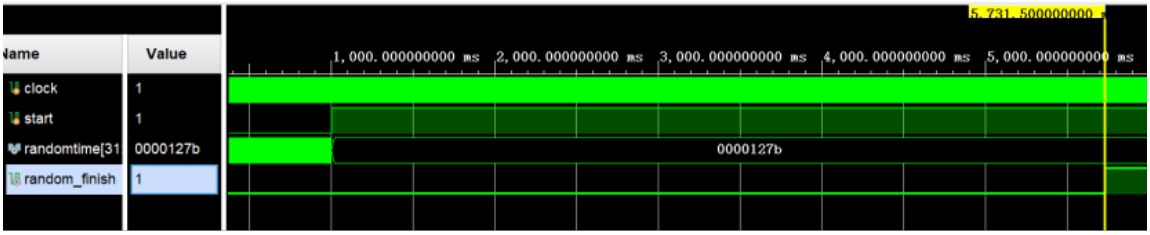

图 8: 随机时间结束计时

代码如下：

```
module random_count(
    input clock,
    input reset,
    input start,
    output reg [31:0] randomtime,
    output reg random_finish
);

    // Parameters to define the range of random time (500ms to 5s)
    // Assuming clock frequency is 1000Hz
    // 500ms => 500 clock cycles
    // 5s => 5,000 clock cycles
    localparam MIN_COUNT = 500;
    localparam MAX_COUNT = 5000;

    reg [31:0] counter;
    reg [15:0] lfsr;
    wire feedback;

    initial begin
        random_finish = 0;
        counter = 0;
        randomtime = 0;
        lfsr = 16'hACE1;  // Seed for the LFSR
    end

    // LFSR logic to generate pseudo-random numbers
    assign feedback = lfsr[15] ^ lfsr[13] ^ lfsr[12] ^ lfsr[10];

    always @(posedge clock or reset) begin
```

```
        if (reset) begin
            // Reset all state variables to initial values
            random_finish <= 0;
            counter <= 0;
            randomtime <= 0;
            lfsr <= 16'hACE1;  // Seed for the LFSR
        end
        else if (!start) begin
            // Generate and update random number using LFSR
            lfsr <= {lfsr[14:0], feedback};
            randomtime <= MIN_COUNT + (lfsr % (MAX_COUNT - MIN_COUNT));
            // Reset counter and random_finish
            counter <= 0;
            random_finish <= 0;
        end
        else begin
            if (counter < randomtime) begin
                // Traversaling set randomtime
                counter <= counter + 1;
            end
            else begin
                random_finish <= 1;
            end
        end
    end
endmodule
```

2.2.3 反应时间计时模块 react_time_count

本模块接受 clock 信号、react 信号和 random_finish 信号，在接收到 random_finish 信号被置为高电平后开始计时，一直持续到 react 信号被置为 1 后，即被测者做出反应，停止

计时，并输出这段反应时间 t_react。若超出 9999ms 后 react 信号仍为 0，则表示反应测试失败，输出超出反应测试响应时间信号 react_exceed 被置为 1。

仿真激励如下：

```
initial begin
    clock = 0;
    reset = 0;
    react = 0;
    random_finish = 0;
    #10 random_finish = 1;
    #500 react = 1;
    #500 reset = 1;
    random_finish = 0;
    react = 0;
    #10 random_finish = 1;
    reset = 0;
    #10200;
end
```

图 9: 反应时间计时模块仿真激励

可知，第一阶段反应输入 react 在 500ms 延迟后被置为 1，十六进制是 0x01f4，仿真结果如下，可验证其正确性。第二阶段，先清零置为后，在随机时间结束信号 random_finish 在 1.01s 左右被置为 1 后，延迟 10200ms（超过规定的最大值 9999ms），可以在仿真结果上看到在 11.0205s 左右，输出超出反应测试响应时间信号 react_exceed 被置为 1。

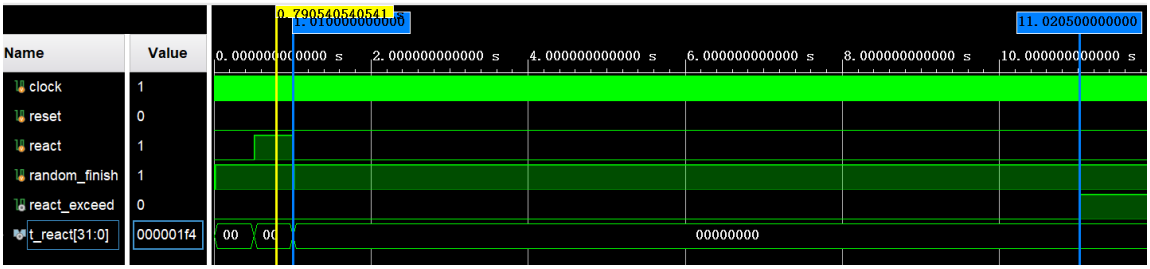

图 10: 反应时间计时模块仿真结果

代码如下：

```
module react_time_count(
    input clock,
    input reset,
```

```
    input react,
    input random_finish,
    output reg react_exceed,
    output reg [31:0] t_react // Save reaction time by using 32-bit register
    );

    reg [31:0] cc_counter; // Counter clock cycles
    reg counting; // Flag to detect whether to count time
    reg prev_react; // Save last state of react to detect edge

    initial begin
        t_react = 0;
        cc_counter = 0;
        counting = 0;
        prev_react = 0;
        react_exceed = 0;
    end

    always @(posedge clock or posedge) begin
        if(reset == 1)begin
            t_react <= 0;
            cc_counter <= 0;
            counting <= 0;
            prev_react <= 0;
            react_exceed <= 0;
        end
        else begin
        prev_react <= react;
        if (random_finish) begin
            if (!counting) begin
                // Start timing
```

```
            counting <= 1;
            cc_counter <= 1;
        end else if (counting && !prev_react && react && !react_exceed) begin
            // Stop timing if react turns to 1 from 0
            t_react <= cc_counter;
            counting <= 0; // Stop timing
        end else if (counting) begin
            // Continue to time, and every cycle represents 1ms
            cc_counter <= cc_counter + 1;
            if(cc_counter >= 10000)begin
                react_exceed <= 1;
            end
        end
    end
    else begin
        // Reset state if random_finish = 0
        counting <= 0;
        cc_counter <= 0;
        t_react <= 0;
        prev_react <= 0;
    end
    end
  end
endmodule
```

2.2.4 结果显示模块 display

本模块接受 clock 信号、reset 信号、start 信号、random_finish 信号、react 信号、react_exceed 信号和 t_react 输入，输出 4 位 7 段 LED 数码管的 11 位显示信号 result。由于本设计的物理实现选择的是 xc7a35tcpg236-1，它的数码管显示是动态刷新的，每个周期只选通一个数码管并显示，因此输出的 11 位 result 中的高四位用来控制哪个数码管在此刻周

期被选通，从左至右的选通编码分别表示为 0111，1011，1101，1110（低电平有效）；低七位用来表示单个 7 段数码管的显示（低电平有效），其显示对应的编码见表 1，CP 接口显示小数点，本例用不到，所以忽略，仅考虑 7 段显示。

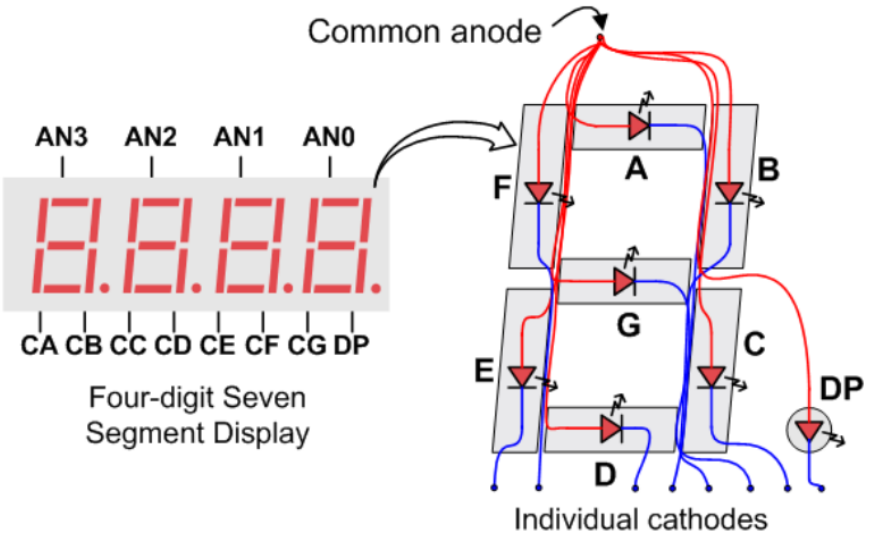

图 11: FPGA 数码管显示原理

关于代码的编写思路，首先将输入的反应时间 t_react 转换为 4 位”BCD” 码，但在这里使用数码管的显示方式编码，编码如上表，每一位通过 case 语句分别转换为 7 位的 7 段数码管的显示码，并将结果分别暂存在 28 位的 t_react_converted 的低 7 位，[13:7]，[20:14] 和高 7 位中。这里注意要将千位的数字 0 改为不显示，因为能正常显示结果就规定了 react 时间要大于等于 100ms，因此不考虑百位为 0 的情况，而当千位为 0 时，就需要改为更符合常理认知的不显示。

关于输出结果的逻辑产生，当 start 为 0 时显然为不显示状态，而考虑在 start 信号为高电平时，根据不同的输入信号组合设置 result 的值，以决定显示内容。具体见表 2 真值表。

同时还设置了寄存器 whether_fail，用以区分 random_fail 未被置为 1 而 react 已被置为 1 的 fail 状态，和后续 random_fail 被置为 1 从而得到错误的反应时间状态。因此，此寄存器 whether_fail 用以判断状态是否已经转变为 Fail 从而阻止其他信号引起的状态发生变化。代码中设置了 4 个状态常量，分别是 DISPLAY_NONE、DISPLAY_DASH、DISPLAY_TEST 和 DISPLAY_FAIL，分别对应不显示、显示”− − −−”、显示”1111” 和显示”FA1L” 四种状态的数码管编码，在不同的组合逻辑下将 result 赋值，最后将 8 位分割分配给四位数码管显示 s0-s3。

字符	从 CA 到 CG 的编码情况
不显示	1111111
-	1111110
F	0111100
A	0001000
L	1111001
0	0000001
1	1001111
2	0010110
3	0000110
4	1001100
5	0100100
6	0100000
7	0001111
8	0000000
9	0000100

表 1: 数码管对应字符编码表

start	random_finish	react	react_exceed	display
0	X	X	X	Nothing
1	0	0	X	− − − −
1	0	1	X	Fail
1	1	0	0	1111
1	1	1	X	t_react or Fail
1	1	0	1	Fail

对四种不同情况进行了仿真测试，分别是正常的人体反应流程（本例反应 120ms）、提前于随机时间结束按键 react 的 Fail 结果、随机时间结束后立即按键 react（本例 16ms）的 Fail 结果和超过 9999ms 按键 react 的 Fail 结果。

1. 第一个是正常的人体反应流程，从下图可以看到数码显示管按下 start 键前后从动态刷新的不显示：千位 3ff（0111 1111111，不显示）、百位 5ff（1011 1111111，不显示）、十

位 6ff（1101 1111111，不显示）和个位 77f（1110 1111111，不显示），到显示”−−−−”：千位 3fe（0111 1111110，显示-）、百位 5fe（1011 1111110，显示-）、十位 6fe（1101 1111110，显示-）和个位 77e（1110 1111110，显示-）。

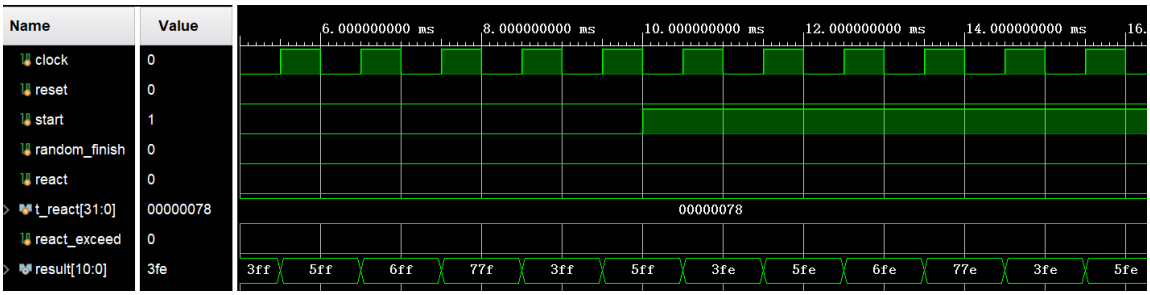

图 12: 按下 start 键前后显示变化图

经历随机时间结束后 d 动态刷新显示”1111”：千位 3cf（0111 1001111，显示 1）、百位 5cf（1011 1001111，显示 1）、十位 6cf（1101 1001111，显示 1）和个位 74f（1110 1001111，显示 1）。

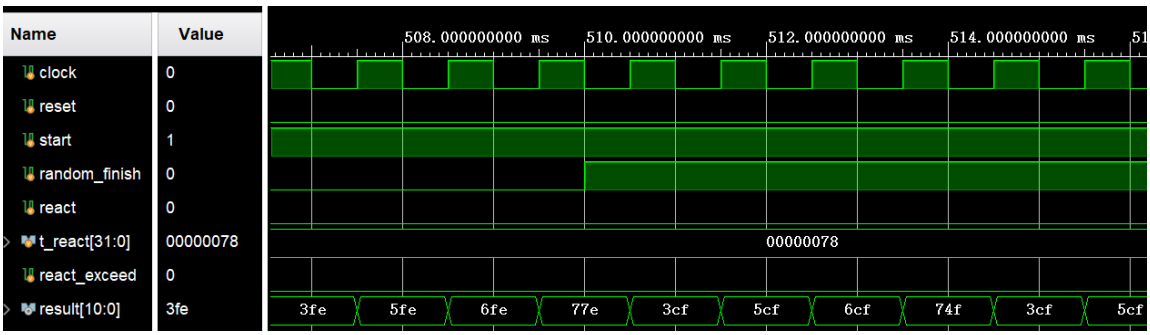

图 13: 随机时间结束前后的显示变化图

再经过仿真测试中给定的延迟（本例为 120ms）后，得到正确的输出结果，即显示 500（千位不显示）：千位 3ff（0111 1111111，不显示）、百位 5cf（1011 1001111，显示 1）、十位 692（1101 0010010，显示 2）和个位 701（1110 0000001，显示 0）。

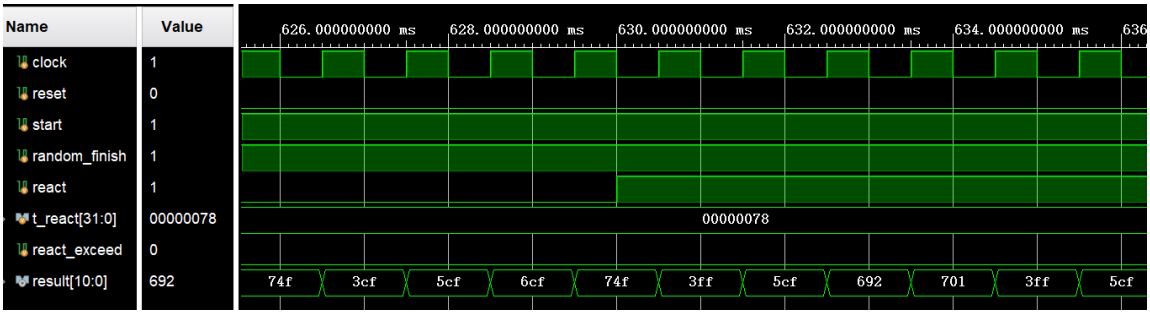

图 14: 正常反应 120ms 前后的显示变化图

2. 第二个是提前于随机时间结束按键 react 的 Fail 结果，如下图，这里可以看到 react 的被置为 1 变化提前于 random_finish，因此直接看到输出结果为 Fail：千位 3b8（0111 0111000，显示 F）、百位 588（1011 0001000，显示 A）、十位 6cf（1101 1001111，显示 1）和个位 771（1110 1110001，显示 L），在 random_finish 被置为 1 后仍然为 Fail，直到 reset 信号重置。

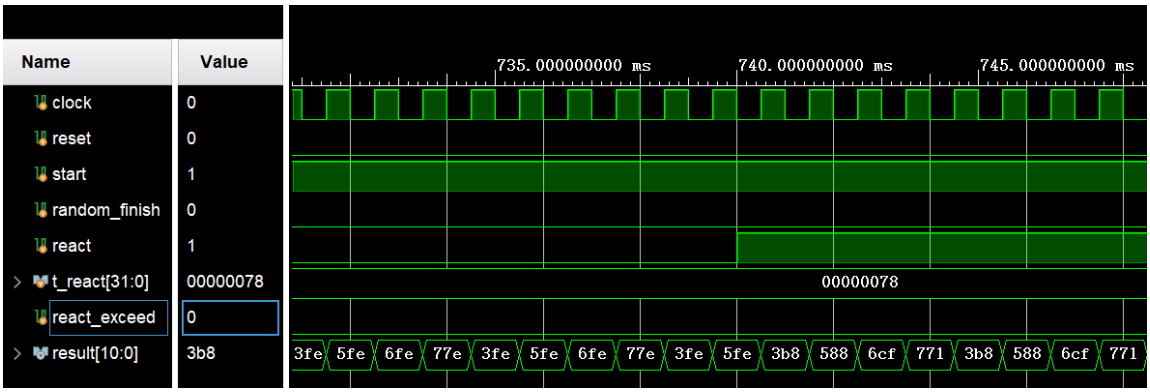

图 15: 提前于随机时间结束按下 react 键的显示变化图

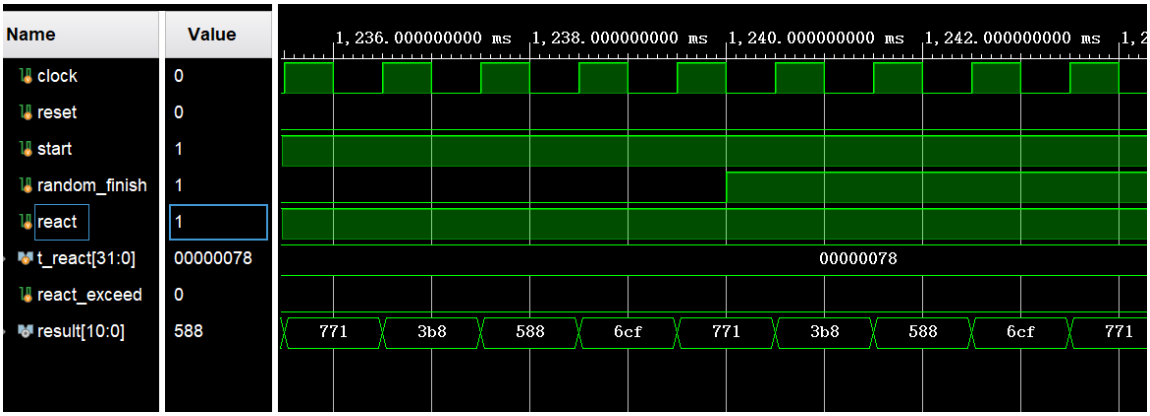

图 16: 提前按下 react 后随机时间结束的显示变化图

3. 第三个是随机时间结束后立即按键 react（本例 16ms）的 Fail 结果，如下图，这里可以看到 random_finish 在整个 testbench 内第 1800ms 被置为 1，在 1816msreact 信号也被置为 1，反应时间仅 16ms，判定为 Fail 状态，在第 1816ms 时输出信号由显示”1111”变化到”Fail”，具体每位数码管的显示情况如上所述。

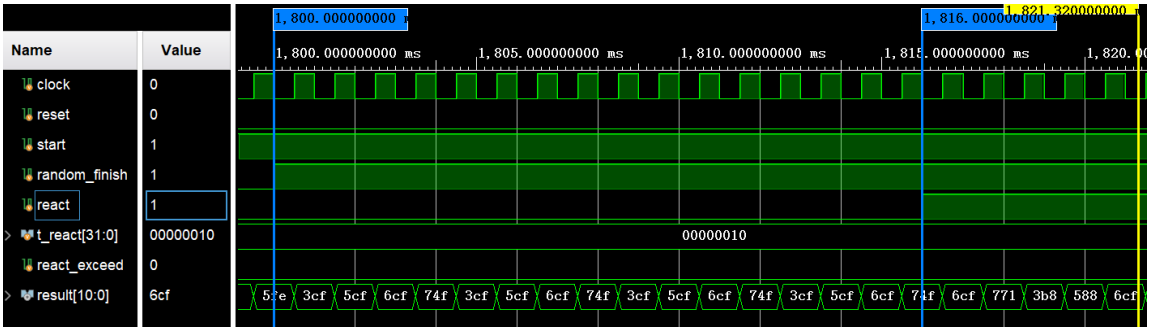

图 17: 随机时间结束后 16ms 内 react 的显示变化图

4. 第四个是超过 9999ms 按键 react 的 Fail 结果，如下图，这里由反应时间计时模块给到的超过 9999ms 反应时间的 react_exceed 被置为 1，可以看到输出信号由显示”1111”变化到”Fail”。

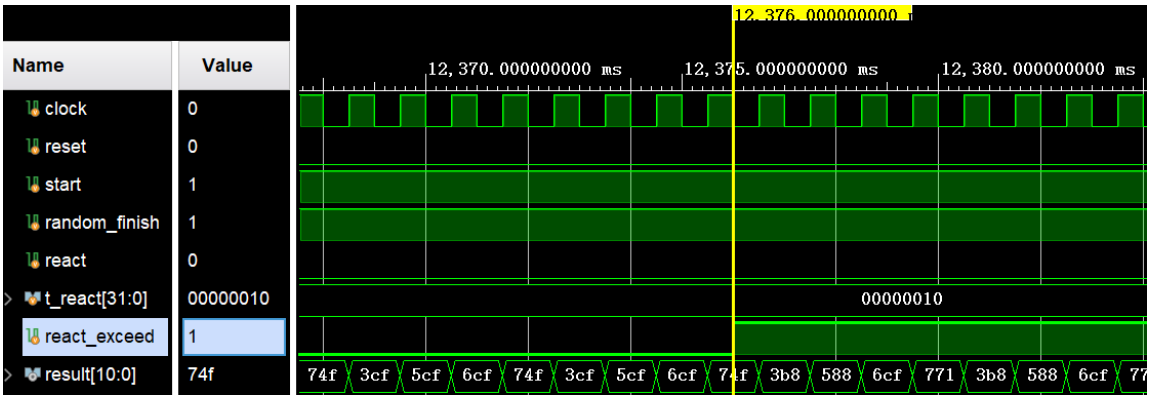

图 18: 超过 9999ms 不按键 react 的显示变化图

模块代码如下：

```
module display(
    input clock,
    input reset,
    input start,
```

```
    input random_finish,
    input react,
    input [31:0] t_react,
    input react_exceed,
    output reg [10:0] result
    );

    reg [27:0] t_react_converted; // Convert t_react to be four-bit DEC for LED
    reg whether_fail; // Judge whether failuer has happened
    reg whether_test; // Judge whether test has begun
    reg [1:0] counter1;
    reg [1:0] counter2;
    reg [1:0] counter3;
    reg [1:0] counter4;
    reg [1:0] counter5;
    reg [1:0] counter6;
    reg [1:0] counter7;

    // Define charater constant
    parameter DISPLAY_NONE = 7'b1111111; // Display nothing
    parameter DISPLAY_DASH = 7'b1111110; // Display "-"
    parameter DISPLAY_F = 11'b01110111000; // Display "F"
    parameter DISPLAY_A = 11'b10110001000; // Display "A"
    parameter DISPLAY_I = 11'b11011001111; // Display "I"
    parameter DISPLAY_L = 11'b11101110001; // Display "L"
    parameter DISPLAY_0 = 7'b0000001; // Display "0"
    parameter DISPLAY_1 = 7'b1001111; // Display "1"
    parameter DISPLAY_2 = 7'b0010010; // Display "2"
    parameter DISPLAY_3 = 7'b0000110; // Display "3"
    parameter DISPLAY_4 = 7'b1001100; // Display "4"
    parameter DISPLAY_5 = 7'b0100100; // Display "5"
```

```
36      parameter DISPLAY_6 = 7'b0100000; // Display "6"
37      parameter DISPLAY_7 = 7'b0001111; // Display "7"
38      parameter DISPLAY_8 = 7'b0000000; // Display "8"
39      parameter DISPLAY_9 = 7'b0000100; // Display "9"
40
41      initial begin
42          counter1 <= 0;
43          counter2 <= 0;
44          counter3 <= 0;
45          counter4 <= 0;
46          counter5 <= 0;
47          counter6 <= 0;
48          counter7 <= 0;
49      end
50
51      always @(posedge clock or posedge reset) begin
52          // Convert HEX reaction time to four-bit DEC
53          case(t_react%10)
54              0: t_react_converted[6:0] <= DISPLAY_0;
55              1: t_react_converted[6:0] <= DISPLAY_1;
56              2: t_react_converted[6:0] <= DISPLAY_2;
57              3: t_react_converted[6:0] <= DISPLAY_3;
58              4: t_react_converted[6:0] <= DISPLAY_4;
59              5: t_react_converted[6:0] <= DISPLAY_5;
60              6: t_react_converted[6:0] <= DISPLAY_6;
61              7: t_react_converted[6:0] <= DISPLAY_7;
62              8: t_react_converted[6:0] <= DISPLAY_8;
63              9: t_react_converted[6:0] <= DISPLAY_9;
64          endcase
65          case((t_react/10)%10)
66              0: t_react_converted[13:7] <= DISPLAY_0;
```

```
67              1: t_react_converted[13:7] <= DISPLAY_1;
68              2: t_react_converted[13:7] <= DISPLAY_2;
69              3: t_react_converted[13:7] <= DISPLAY_3;
70              4: t_react_converted[13:7] <= DISPLAY_4;
71              5: t_react_converted[13:7] <= DISPLAY_5;
72              6: t_react_converted[13:7] <= DISPLAY_6;
73              7: t_react_converted[13:7] <= DISPLAY_7;
74              8: t_react_converted[13:7] <= DISPLAY_8;
75              9: t_react_converted[13:7] <= DISPLAY_9;
76          endcase
77          case((t_react/100)%10)
78              0: t_react_converted[20:14] <= DISPLAY_0;
79              1: t_react_converted[20:14] <= DISPLAY_1;
80              2: t_react_converted[20:14] <= DISPLAY_2;
81              3: t_react_converted[20:14] <= DISPLAY_3;
82              4: t_react_converted[20:14] <= DISPLAY_4;
83              5: t_react_converted[20:14] <= DISPLAY_5;
84              6: t_react_converted[20:14] <= DISPLAY_6;
85              7: t_react_converted[20:14] <= DISPLAY_7;
86              8: t_react_converted[20:14] <= DISPLAY_8;
87              9: t_react_converted[20:14] <= DISPLAY_9;
88          endcase
89          case(t_react/1000)
90              0: t_react_converted[27:21] <= DISPLAY_NONE; // Display nothing if hundreds and thou
91              1: t_react_converted[27:21] <= DISPLAY_1;
92              2: t_react_converted[27:21] <= DISPLAY_2;
93              3: t_react_converted[27:21] <= DISPLAY_3;
94              4: t_react_converted[27:21] <= DISPLAY_4;
95              5: t_react_converted[27:21] <= DISPLAY_5;
96              6: t_react_converted[27:21] <= DISPLAY_6;
97              7: t_react_converted[27:21] <= DISPLAY_7;
```

```
98              8: t_react_converted[27:21] <= DISPLAY_8;
99              9: t_react_converted[27:21] <= DISPLAY_9;
100         endcase
101
102         if(reset)begin
103             result[6:0] <= DISPLAY_NONE; // Reset, no display
104             if(counter1 == 0)begin
105                 result[10:7] <= 4'b0111;
106                 counter1 <= 1;
107             end
108             if(counter1 == 1)begin
109                 result[10:7] <= 4'b1011;
110                 counter1 <= 2;
111             end
112             if(counter1 == 2)begin
113                 result[10:7] <= 4'b1101;
114                 counter1 <= 3;
115             end
116             if(counter1 == 3)begin
117                 result[10:7] <= 4'b1110;
118                 counter1 <= 0;
119             end
120             whether_test <= 0;
121             whether_fail <= 0;
122         end
123         else if (start) begin
124             if(!random_finish && !react)begin
125                 result[6:0] <= DISPLAY_DASH; // Display "----", if start
126                 if(counter2 == 0)begin
127                     result[10:7] <= 4'b0111;
128                     counter2 <= 1;
```

```
129             end
130             if(counter2 == 1)begin
131                 result[10:7] <= 4'b1011;
132                 counter2 <= 2;
133             end
134             if(counter2 == 2)begin
135                 result[10:7] <= 4'b1101;
136                 counter2 <= 3;
137             end
138             if(counter2 == 3)begin
139                 result[10:7] <= 4'b1110;
140                 counter2 <= 0;
141             end
142             whether_test <= 0;
143             whether_fail <= 0;
144         end
145         else if(!random_finish && react)begin
146             // Display "Fail", react before random time finish
147             if(counter3 == 0)begin
148                 result <= DISPLAY_F;
149                 counter3 <= 1;
150             end
151             if(counter3 == 1)begin
152                 result <= DISPLAY_A;
153                 counter3 <= 2;
154             end
155             if(counter3 == 2)begin
156                 result <= DISPLAY_I;
157                 counter3 <= 3;
158             end
159             if(counter3 == 3)begin
```

```
160                 result <= DISPLAY_L;
161                 counter3 <= 0;
162             end
163             whether_test <= 0;
164             whether_fail <= 1;
165         end
166         else if(random_finish && !react && !react_exceed && !whether_fail)begin
167             result[6:0] <= DISPLAY_1; // Display "1111", random time finish, it should be to react
168             if(counter4 == 0)begin
169                 result[10:7] <= 4'b0111;
170                 counter4 <= 1;
171             end
172             if(counter4 == 1)begin
173                 result[10:7] <= 4'b1011;
174                 counter4 <= 2;
175             end
176             if(counter4 == 2)begin
177                 result[10:7] <= 4'b1101;
178                 counter4 <= 3;
179             end
180             if(counter4 == 3)begin
181                 result[10:7] <= 4'b1110;
182                 counter4 <= 0;
183             end
184             whether_test <= 1;
185         end
186         else if(random_finish && react && t_react >= 100 && whether_test) begin
187             // Display t_react, react successfully
188             if(counter7 == 0)begin
189                 result[10:7] <= 4'b0111;
190                 result[6:0] <= t_react_converted[27:21];
```

```
191                 counter7 <= 1;
192             end
193             if(counter7 == 1)begin
194                 result[10:7] <= 4'b1011;
195                 result[6:0] <= t_react_converted[20:14];
196                 counter7 <= 2;
197             end
198             if(counter7 == 2)begin
199                 result[10:7] <= 4'b1101;
200                 result[6:0] <= t_react_converted[13:7];
201                 counter7 <= 3;
202             end
203             if(counter7 == 3)begin
204                 result[10:7] <= 4'b1110;
205                 result[6:0] <= t_react_converted[6:0];
206                 counter7 <= 0;
207             end
208         end
209         else begin
210             // Display "Fail", react failed
211             if(counter5 == 0)begin
212                 result <= DISPLAY_F;
213                 counter5 <= 1;
214             end
215             if(counter5 == 1)begin
216                 result <= DISPLAY_A;
217                 counter5 <= 2;
218             end
219             if(counter5 == 2)begin
220                 result <= DISPLAY_I;
221                 counter5 <= 3;
```

```
222                 end
223                 if(counter5 == 3)begin
224                     result <= DISPLAY_L;
225                     counter5 <= 0;
226                 end
227                 whether_fail <= 1;
228             end
229         end
230         else begin
231                 result[6:0] <= DISPLAY_NONE; // Display Nothing, for else conditions
232                 if(counter6 == 0)begin
233                     result[10:7] <= 4'b0111;
234                     counter6 <= 1;
235                 end
236                 if(counter6 == 1)begin
237                     result[10:7] <= 4'b1011;
238                     counter6 <= 2;
239                 end
240                 if(counter6 == 2)begin
241                     result[10:7] <= 4'b1101;
242                     counter6 <= 3;
243                 end
244                 if(counter6 == 3)begin
245                     result[10:7] <= 4'b1110;
246                     counter6 <= 0;
247                 end
248                 whether_fail <= 0;
249         end
250     end
251 endmodule
```

testbench 代码如下：

```
module display_tb();
    reg clock;
    reg reset;
    reg start;
    reg random_finish;
    reg react;
    reg [31:0] t_react;
    reg react_exceed;
    wire [10:0] result;

display display(
    .clock(clock),
    .reset(reset),
    .start(start),
    .random_finish(random_finish),
    .react(react),
    .t_react(t_react),
    .react_exceed(react_exceed),
    .result(result)
    );

initial begin
    clock = 0;
    forever #0.5 clock = ~clock;
end

initial begin
    reset = 0;
    start = 0;
```

```
30      random_finish = 0;
31      react = 0;
32      react_exceed = 0;
33      t_react = 32'h00000078; // Reaction time for 120ms
34
35      #10 start = 1;
36      #500 random_finish = 1;
37      #120 react = 1; // Show react time
38
39      #50 reset = 1; // Manually reset for not connecting with other modules
40      start = 0;
41      random_finish = 0;
42      react = 0;
43
44      #10 reset = 0;
45      start = 1;
46      #50 react = 1;
47      #500 random_finish = 1; // Show failure under the condition of reacting before random time finis
48
49      #50 reset = 1; // Manually reset for not connecting with other modules
50      start = 0;
51      random_finish = 0;
52      react = 0;
53
54      #10 reset = 0;
55      start = 1;
56      t_react = 32'h00000010; // Reaction time for 16ms, which is under the minimum of scale
57      #500 random_finish = 1;
58      #16 react = 1; // Show failure under the condition of reaction time is under the minimum of scale
59
60      #50 reset = 1; // Manually reset for not connecting with other modules
```

```
61      start = 0;
62      random_finish = 0;
63      react = 0;
64
65      #10 reset = 0;
66      start = 1;
67      #500 random_finish = 1;
68      #10000 react_exceed = 1; // It should be a great delay, but here is set for convenience to show //
69      // Show failure under the condition of reaction time exceeds the maximum of scale
70
71  end
72  endmodule
```

2.2.5 分频模块 fre_divide

本模块是由于上板需要，目标 FPGA 支持 100MHz, 因此需要将 100000 个周期合并在一起成一个大周期以满足 1000Hz，即本设计所要求的分辨率为 1ms，即一个周期 1ms。代码如下，考虑当计数器达到合并周期的一半时将时钟翻转：

```
1   module fre_divide(
2       input clock_undivided,
3       input reset,
4       output reg clock_divided
5   );
6
7       reg [15:0] counter;
8       parameter DIVISOR = 50000;
9
10      initial begin
11          counter <= 0;
12          clock_divided <= 0;
```

```
13      end
14
15      always @(posedge clock_undivided or posedge reset) begin
16          if (reset) begin
17              counter <= 16'b0;
18              clock_divided <= 1'b0;
19          end else begin
20              if (counter == (DIVISOR - 1)) begin
21                  counter <= 16'b0;
22                  clock_divided <= ~clock_divided;
23              end else begin
24                  counter <= counter + 1;
25              end
26          end
27      end
28
29  endmodule
```

仿真结果如下所示，可以看到分频后的时钟周期达到了所要求的 1ms。

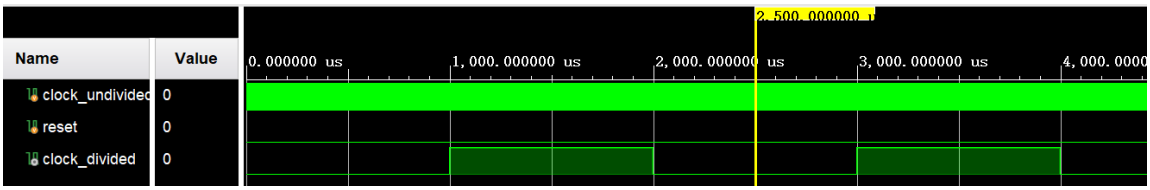

图 19: 分频模块仿真结果

2.3 综合实现（Synthesis & Implement）

将上述模块逐一互连，得到以下电路原理图。但本电路图无法用于功能仿真和时序仿真，首先时钟频率过快，100MHz 的目标 FPGA 要求每个周期 20ns，而流程至少需求 15000ms（考虑随机时间和反应时间最大值之和），在实际仿真中导致扫描效率特别慢。因此在不上板的仿真中，设计去除分频器，手动改写 timescale 最小单位为 1ms，设置每 0.5ms 将时钟翻

转以模拟 1000Hz 的效果。同时，关于 react 信号如何给定，由于上板可以看到数码管的显示由”− − −−” 到”1111”，可以即时地给予输入激励 react=1，但仿真必须在开始前规定好输入激励地给定方式，因此，本设计在仿真层面使用 react_simulation 模拟了不同情况的输入激励，从而达到理想的仿真结果。

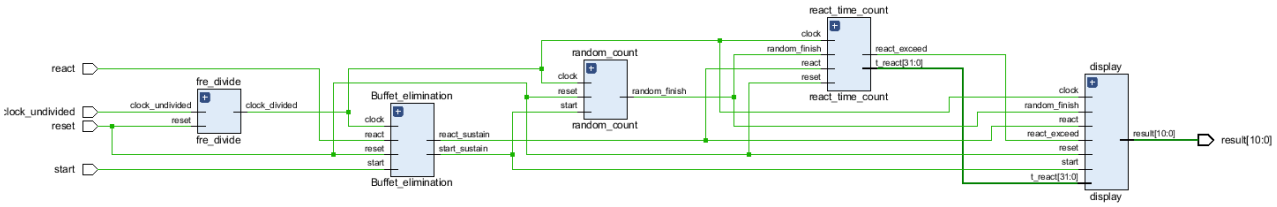

图 20: 电路原理图

2.3.1 完整的成功反应测试的时序仿真

首先考虑正常和准确的测试流程，即人体看到”1111” 后在大于 100ms，小于 9999ms 的时间范围内按下 react 键，得到自己的反应时间输出。设置输入激励方式如下，即当接收到 random_finish 被置为高电平的信号后，延迟 589 个周期响应 react_simulation 被置为 1。

```
always @(posedge clock) begin
    if(random_finish)begin
        tmp <= tmp + 1;
        if(tmp >= 589)begin
            react_simulation <= 1; // Simulate reaction after 589ms
        end
    end
    else begin
        react_simulation <= 0;
    end
end
```

但考虑按键消抖模块的连续四个周期检测，以及两次在下一级时钟上升沿触发，因此实际的延迟时间是 589+6=595 个周期，即 595ms。见下图的仿真结果，可见人体反应的延迟

时间就是 595ms：千位 3ff（0111 1111111，不显示）、百位 5a4（1011 0100100，显示 5）、十位 684（1101 0000100，显示 9）和个位 724（1110 0100100，显示 5）。

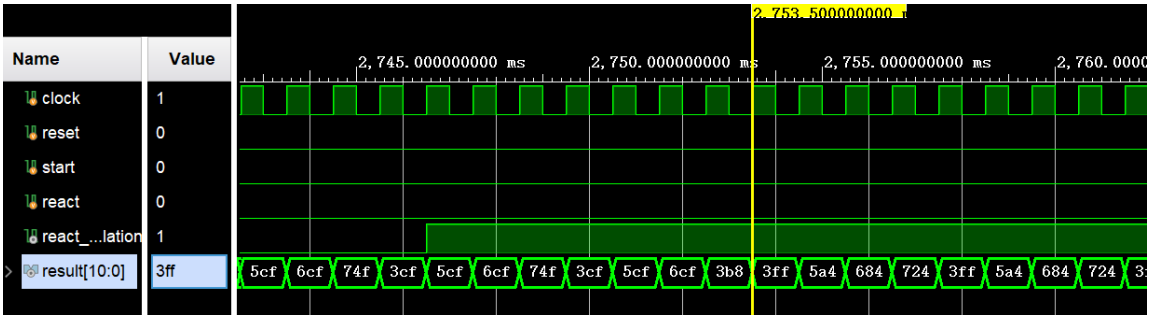

图 21: 综合时序仿真——成功检测反应时间

2.3.2 提前于随机时间结束按下 react 的 Fail 时序仿真

第二种情况是在随机时间未结束前就直接按下 react 键导致测试失败，即 random_finish 尚未置为 1，react_simulation 就已置为 1。输入激励考虑检测到开始持续信号 start_sustain 键置为 1 后，就将 react_simulation 置为 1，以达到在随机时间结束前就 react 的效果，代码如下：

```
    always @(posedge clock) begin
// Fail for react before random_finish
        if(start_sustain)begin
            react_simulation <= 1;
        end
    end
```

仿真结果如下，可以看到在数码管显示从”----” 尚未变到随机时间结束后的”1111”：千位 3cf（0111 1001111，显示 1）、百位 5cf（1011 1001111，显示 1）、十位 6cf（1101 1001111，显示 1）和个位 74f（1110 1001111，显示 1），就已经是 Fail 状态：千位 3b8（0111 0111000，显示 F）、百位 588（1011 0001000，显示 A）、十位 6cf（1101 1001111，显示 1）和个位 771（1110 1110001，显示 L）。

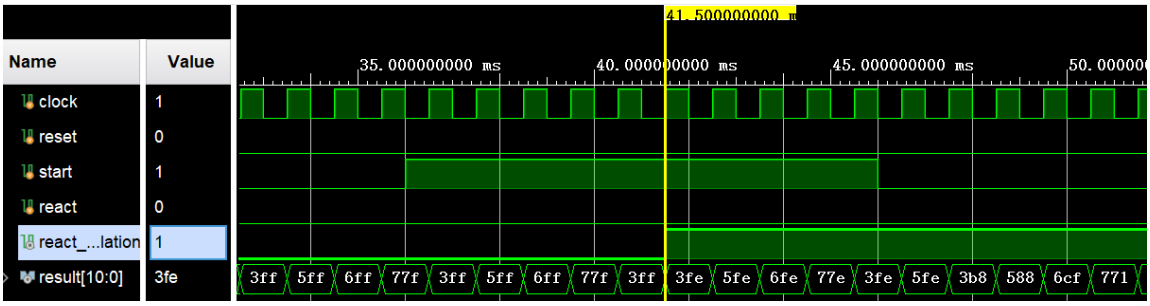

图 22: 综合时序仿真——随机时间结束前 react 的 Fail 状态

2.3.3 随机时间结束后极快按下 react 的 Fail 状态

第三种情况是在随机时间结束后极快速按下 react 键，导致不在人体最小反应时间范围内（即小于 100ms），本例设置 random_finish 被置为 1 后，经过 63ms 后将 react_simulation 置为 1，以达到在随机时间结束后极快速按下 react 键的效果。代码如下：

```
    always @(posedge clock) begin
// Fail for reaction only in 63ms after random_finish
        if(random_finish)begin
            tmp <= tmp + 1;
            if(tmp >= 63)begin
                react_simulation <= 1; // Simulate reaction after 63ms
            end
        end
        else begin
            react_simulation <= 0;
        end
    end
```

仿真结果如下，可以看到，随机时间刚结束，数码管显示”1111”：千位 3cf（0111 1001111，显示 1）、百位 5cf（1011 1001111，显示 1）、十位 6cf（1101 1001111，显示 1）和个位 74f（1110 1001111，显示 1），随机时间结束状态仅维持 69ms，比 63ms 多出来的 6 个周期在完整的测试流程仿真已阐述过，由于极短时间内按下 react 键导致未判定为多于 100ms，显示 Fail：千位 3b8（0111 0111000，显示 F）、百位 588（1011 0001000，显示 A）、十位 6cf（1101

1001111，显示 1）和个位 771（1110 1110001，显示 L）。

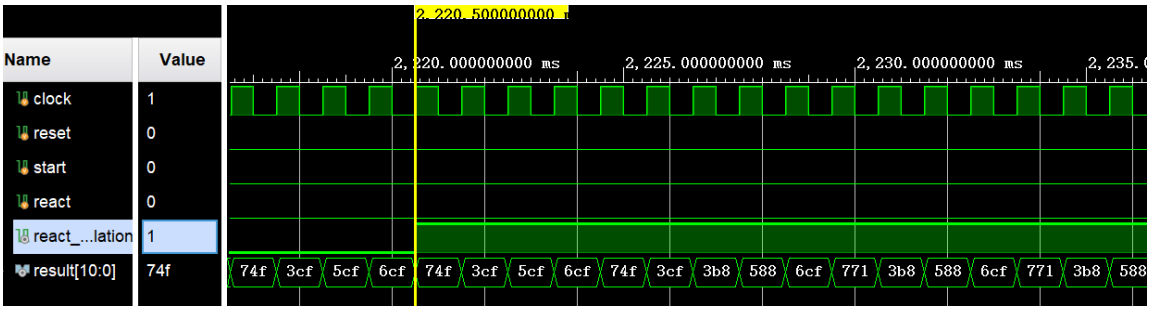

图 23: 综合时序仿真——随机时间结束后极快 react 的 Fail 状态

2.3.4 随机时间结束后 10s 内不 react 的 Fail 状态

第四种情况是在在随机时间结束后 10s 内不按下 react 键，导致超过人体最大反应时间（即大于 9999ms），本例设置仿真持续 17000ms，在 random_finish 被置为 1 后，在 10s 内不做任何额外设置以达到在随机时间结束后 10s 内不 react 键的效果。为了验证设计准确性，设置在 12s 时按下 react 键以检验仍显示 Fail 状态。代码如下：

```
    always @(posedge clock) begin
// Fail for no reaction in 10s after random_finish
        if(random_finish)begin
            tmp <= tmp + 1;
            if(tmp >= 12000)begin
                react_simulation <= 1; // Simulate reaction after 12s
            end
        end
        else begin
            react_simulation <= 0;
        end
    end
```

仿真结果如下所示，可以看到随机时间结束后，数码管显示了 10s 左右的”1111” 状态：千位 3cf（0111 1001111，显示 1）、百位 5cf（1011 1001111，显示 1）、十位 6cf（1101 1001111，显示 1）和个位 74f（1110 1001111，显示 1），随后就转换至 Fail 状态：千位 3b8（0111

0111000，显示 F）、百位 588（1011 0001000，显示 A）、十位 6cf（1101 1001111，显示 1）和个位 771（1110 1110001，显示 L）。在后面约 14s 的位置即使按下 react，也依然无法改变数码管的 Fail 状态显示。

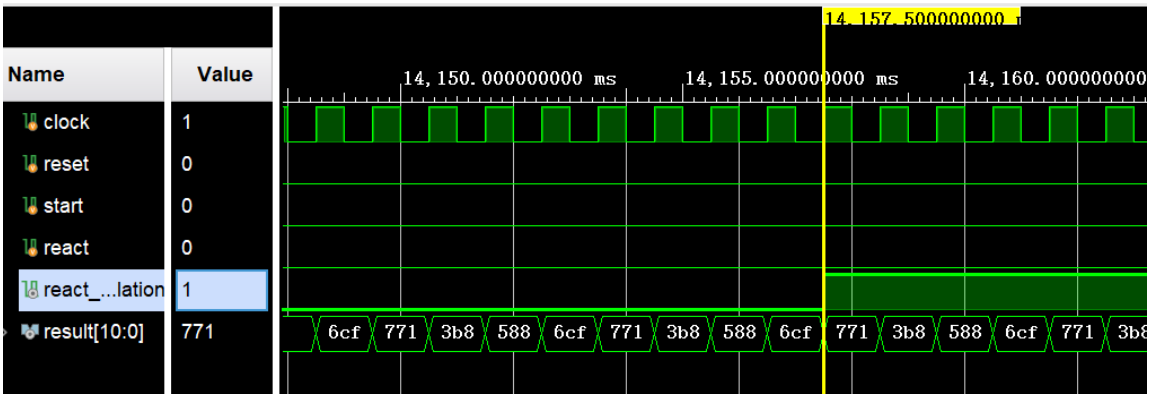

图 24: 综合时序仿真——在随机时间结束后 10s 内不 react 的 Fail 状态

3 物理实现

本次物理实现采用型号为 xc7a35tcpg236-1 的 FPGA 板，在进行了综合和实现后，将接口与 top 模块的端口相对应，对应关系如下图。然后连接板子进行了测试，附录有本次上板的操作视频，以及综合报告、布局布线报告和时序验证报告。视频中的操作展示了 4 种所有情况的其中 3 种，因为在随机时间结束后在 100ms 内快速按下 react 键难以达到，因此仅实现了仿真层面。

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco
All ports (15)							
result (11)	OUT			☑	34	LVCMOS33*	3.300
result[10]	OUT		W4	☑	34	LVCMOS33*	3.300
result[9]	OUT		V4	☑	34	LVCMOS33*	3.300
result[8]	OUT		U4	☑	34	LVCMOS33*	3.300
result[7]	OUT		U2	☑	34	LVCMOS33*	3.300
result[6]	OUT		W7	☑	34	LVCMOS33*	3.300
result[5]	OUT		W6	☑	34	LVCMOS33*	3.300
result[4]	OUT		U8	☑	34	LVCMOS33*	3.300
result[3]	OUT		V8	☑	34	LVCMOS33*	3.300
result[2]	OUT		U5	☑	34	LVCMOS33*	3.300
result[1]	OUT		V5	☑	34	LVCMOS33*	3.300
result[0]	OUT		U7	☑	34	LVCMOS33*	3.300
Scalar ports (4)							
clock_un	IN		W5	☑	34	LVCMOS33*	3.300
react	IN		T18	☑	14	LVCMOS33*	3.300
reset	IN		W19	☑	14	LVCMOS33*	3.300
start	IN		U17	☑	14	LVCMOS33*	3.300

图 25: IO_Planing 对应图

4 设计总结

4.1 注意事项与编程技巧

本设计使用 Verilog 硬件描述语言进行编程，平台仍位于 Vivado。由于之前做过 RISC 流水线处理器的课程设计，但尚对 Verilog 编程了解不深，特别是在给寄存器赋值这里，会忽略不写位数的寄存器默认只有 1 位，因此只能表示 1 和 0，在之后的 debug 过程中也是多次发现。Verilog 语言的整体代码逻辑类似 C 语言，有 C 语言的基础上手还是很快的。

在计数模块，考虑设置计数器 counter，但由于时钟上升沿触发，实际经过的计数周期要多一个，所以设置的计数目标要通常比实际目标少一个。在随机数生成模块，由于没有任何一个程序能生成真正的随机数，所以考虑随机性唯一发生的事件，就是按键 start 的时刻是不确定，所以从 reset 复位后，就一直不断通过对随机数的种子某些特定位的异或，来构造每个周期伪随机数的生成。在结果显示模块，也是通过仿真后得出信号之间的冲突冒险——即在随机时间结束前按下 react 就显示的 Fail 状态会由于后续的随机时间结束 random_finish 被置为高电平，从而状态又转变为随机时间计时完成的”1111” 状态，因此设置标志变量来指

示是否已经显示 Fail 状态的 whether_fail 和显示是否正处于测试状态 whether_test，以此来避免信号冲突导致结果不理想。

4.2 总结与体会心得

在设计之初，画完状态转换图后，为了急于验证思路，并没有遵循由小到大的设计方法，直接由 top 模块开始粗浅的写起，要实例化某个模块就开始写某个模块，导致最后跑仿真是显示高阻态，调试 debug 也很难从局部下手，因为大模块设计很多端口。因此后来静下心来从一个个子模块开始写起，先明确要写什么样的子模块，在本设计中承担了什么样的角色，功能有什么，输入输出分别是什么，这样我按照人体反应测试流程的思路，从一开始的按键消抖、传输启动模块 buffet_elimination，明确信号 start 和 react 毕竟在实际的测试中是按键的形式，是不延续的突刺激励信号，因此有必要为了后续的逻辑判断而将其信号的高电平延长得到新的可以用于逻辑判断的信号 start_sustain 和 react_sustain。

然后遵循测试流程，按键 start 后是随机时间开始计时，因此设计一个能具有生成随机数和计时功能的模块 random_count，一开始确实是没有随机数的生成的思路，因此上网查阅资料发现有线性反馈移位寄存器（LFSR）的随机数生成原理，而确定随机数的那个瞬间就是 start 信号被置为高电平的时刻，因此才顺利写出模块，实现仿真。接着的反应计时模块 react_time_count 就比较顺利的写出来，但一开始没有考虑到超出 9999ms 的情况要即时的显示 Fail 状态，因此后续才添加了超时的判断输出 react_exceed。接着是结果显示模块 display，这里涉及的逻辑判断较为复杂，因此我先画出真值表，将输入信号和结果显示的逻辑关系先表示清楚，然后在仿真过程中发现信号冲突的问题，react 先比 random_finish 变 1，随后 random_finish 变 1 时，状态会从 Fail 状态变成随机时间结束状态”1111”，后来加入标识变量以解决问题，这也是提醒我要注意在今后的设计中要时时注意信号冲突的问题。

然后由于先不考虑上板，因此跳过分频模块，直接在顶层实例化、综合和时序仿真，这次就能一气呵成得出想要的理想的结果，这样的自上而下 Top-down 的思路考虑，自小而大的设计流程是让我颇有收获的。然后上板后，由于看到板子的使用说明后，发现是动态刷新的数码管显示，和我预想的 4 位接入同时显示不同，因此关于 display 模块的设计要大部分增改，还是沿用计数器的思想，在每一周期从左到右选通数码管并显示。中间也是对板子的接口（按键）是高电平有效还是低电平有效没有明确规范，后来看到说明手册上的电路图，看到数码管用 PMOS 连接高电平，5 个 push 按键二极管开关连接高电平，因此根据电学知识推得数码管接口低电平有效，push 按键高电平有效，最后的测试也是符合我的设想，还是十分愉快的。

最后感谢俞老师课上的指导！我想我会继续以这样严谨、自小而大、布大局重小项的态度继续我的学习生活！

4.3 压缩包附文件

1. 工程项目
   (a) Buffet_elimination
   (b) display
   (c) FPGAtest（连接上板的代码）
   (d) fre_divide
   (e) project_1（用于仿真的代码）
   (f) random_count
   (g) react_time_count
2. 报告
   (a) 布局布线报告
   (b) 综合报告
3. 源代码压缩包
4. 上板操作演示视频