数字逻辑电路实验报告

实验题目：序列检测器

实验日期：2025 年 11 月 09 日

实验者姓名：丁毅　2023K8009908031

1　实验目的

1、掌握有限状态机的实现原理和方法。

2、掌握序列检测的方法。

2　实验原理

1、有限状态机 (Finite State Machine, FSM)

有限状态机，常称为状态机 (State Machine) ，是一种用于描述有限数量状态 (State) 以及在状态之间进行转移和执行动作等行为的数学模型。在时序逻辑电路中，有限状态机的输出取决于过去的输入部分与当前输入部分，通常还包含一组具备记忆功能的状态寄存器 (State Register) ，用于记录其内部状态。

有限状态机的下一状态不仅受输入信号影响，还与当前状态相关。其内部组合逻辑为次态逻辑和输出逻辑，分别用于确定下一个状态和输出结果。

根据输出方程的不同，有限状态机可分为米利型 (Mealy Machine) ——某时刻的输出是该时刻输入和当前状态的函数，以及穆尔型 (Moore Machine) ——某时刻的输出仅取决于当前电路状态。

2、序列检测器 (Sequence Detector)

该模块的主要功能是从数字码流中识别出指定的序列。

3　实验仪器和设备

EGO1 开发板，以及搭载 Xilinx Artix XC7A35T-1CSG324C 芯片的 FGPA 开发板

4　实验方法和步骤

1、设计采用有限状态机作为序列检测手段的序列检测器

1) 确定待测序列，根据待测序列确定出状态机的状态转移图，根据状态转移图完成有限状态机序列检测模块的硬件语言编写。

本次实验，由于我的学号为 2023K8009908031，最后两位 (31)_10 = (11111)_2，四位二进制码不足以完整表示，不妨直接截断 (取后四位)，得到待测序列为：**10_1111**。

根据待测序列 **10_1111**，绘制出状态转移图如下：

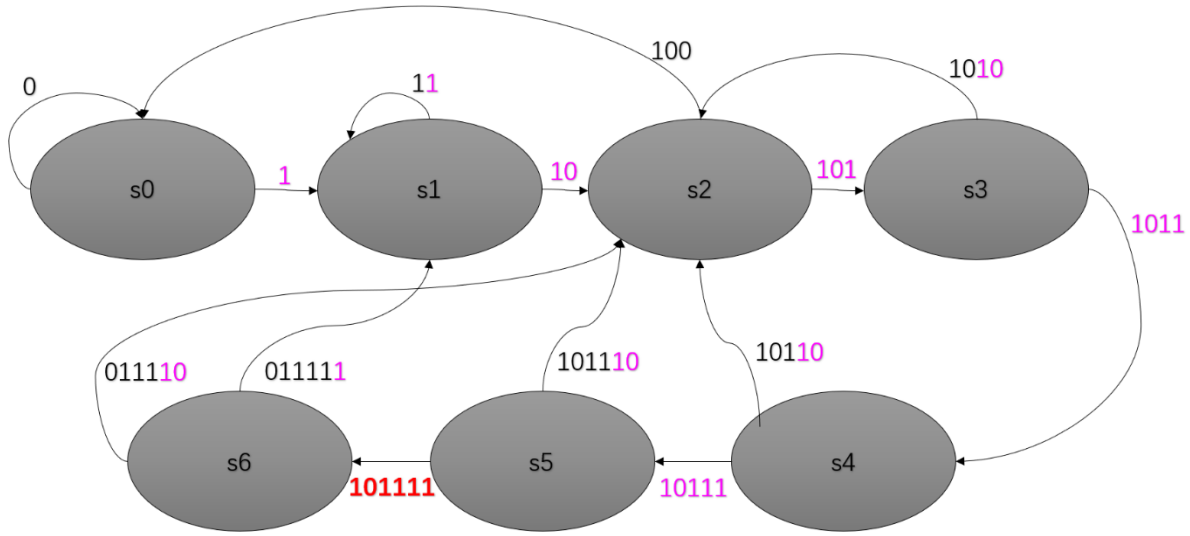


图 1. 目标序列为 101111 的状态转移图

据此写出状态机的代码：

```
// sequence detector using state machine
// 2025.11.06 by YiDingg (https://www.zhihu.com/people/YiDingg)

module fsm(
    input clk,      // tell the machine when we should start
    input din,      // the signal we input
    input rst,      // restart
    output count,   // tell us the result
    output reg [2:0] st_cur
    );

    reg count_store;  // store the count

    parameter s_0 = 3'd0;   // 0
    parameter s_1 = 3'd1;   // 1
    parameter s_2 = 3'd2;   // 10
    parameter s_3 = 3'd3;   // 100
    parameter s_4 = 3'd4;   // 1001
    parameter s_5 = 3'd5;   // 10011
    parameter s_6 = 3'd6;   // 100110

    reg [2:0] st_next;
```

```
// step 1:state transfer
always @(posedge clk or posedge rst) begin
    if(rst)begin
        st_cur <= 0;    // restart
    end else
    begin
        st_cur <= st_next;
    end
end

// target sequence: 101111
// four possible preceding bits:
// (00 101111)_2 = (47)_10
// (01 101111)_2 = (111)_10
// (10 101111)_2 = (175)_10
// (11 101111)_2 = (239)_10

// step 2:state switch,using block assignment for combination_logic
always @(*)begin
    case(st_cur)
        s_0:
            if(din)begin
                st_next = s_1;
            end else
            begin
                st_next = s_0;
            end

        s_1:
            if(din)begin
                st_next = s_1;
            end else
            begin
                st_next = s_2;
            end

        s_2:
            if(din)begin
                st_next = s_3;
            end else
            begin
                st_next = s_0;
            end

        s_3:
```

```
            if(din)begin
                st_next = s_4;
            end else
            begin
                st_next = s_2;
            end

        s_4:
            if(din)begin
                st_next = s_5;
            end else
            begin
                st_next = s_2;
            end

        s_5:
            if(din)begin
                st_next = s_6;
            end else
            begin
                st_next = s_2;
            end

        s_6:
            if(din)begin
                st_next = s_1;
            end else
            begin
                st_next = s_2;
            end
    endcase
end

// step 3: output logic,using non_block assignment
always @(posedge clk or posedge rst)begin
    if(rst)begin
        count_store <= 0;
    end else
    begin
        if(st_next == s_6)begin
            count_store <= 1;
        end else
        begin
            count_store <= 0;
        end
    end
```

```
    end

    assign count = count_store;
endmodule
```

代码采用了经典三段式状态机架构，将状态转移、状态切换逻辑和输出逻辑分离。

首先，模块定义了七个状态参数，每个状态对应序列检测过程中的一个特定阶段：s_0 表示初始空闲状态，s_1 表示已接收到第一个 '1'，s_2 表示接收到 "10"，s_3 表示 "100"，s_4 表示 "1001"，s_5 表示 "10011"，最终 s_6 状态表示成功检测到完整的目标序列 "100110"。这种状态划分清晰地反映了序列检测的渐进过程。

代码的核心采用三段式状态机设计。第一段是状态转移逻辑，在时钟上升沿或复位信号有效时更新当前状态。当复位信号 rst 为高电平时，状态机回到初始状态 s_0；否则，当前状态 st_cur 更新为下一状态 st_next。第二段是组合逻辑的状态切换部分，使用阻塞赋值，通过 case 语句根据当前状态和输入信号 din 决定状态转移路径。例如，在 s_0 状态下，如果输入为 '1' 则转移到 s_1 状态，否则保持在 s_0 状态；在 s_5 状态下，如果输入为 '1' 则转移到表示序列匹配的 s_6 状态，否则回退到 s_2 状态重新开始检测。

第三段是输出逻辑，采用时序逻辑的非阻塞赋值。当检测到下一状态将进入 s_6 时，count_store 信号在下一个时钟周期变为高电平，表示成功检测到目标序列；否则输出保持为低电平。这种设计确保了输出信号与时钟同步，避免了组合逻辑可能产生的毛刺。最终，count_store 的值通过连续赋值语句输出到 count 端口。整个状态机通过这种清晰的分层设计，实现了对 "100110" 序列的可靠检测，同时在检测到完整序列后能够自动回到适当的初始状态继续后续的检测任务。

2) 同时还需实现使用数码管显示当前检测结果，1 表示检测有效，0 表示检测无效，可调用已写好的模块。利用译码模块，将状态机的检测输出 dout 输入至译码模块即可。当检测有效时就会显示 1，否则为 0。

3) 使用 LD2 中的 LED[7:0]表示输入序列，使用 LD1 中的 LED[7:5]表示当前状态。

将状态机的状态和八位 reg 变量 array_now 输出，并将其对应到相应的 LED 管脚，即可实现用 LED 显示输入序列和当前状态。

4) 利用开关实现串行输入、异步复位、输入脉冲，并加入防抖模块。

防抖模块可直接调用。最后的顶层模块代码如下：

```
module top_fsm( // top module of the FSM sequence detector
    input clk,  // clock of sysytem
    input s0,   // signal press
    input rst,  // signal:restart
    input SW0,  // signal:din
    input  [3:0] ctrl, // control signal for anode
    output [7:0] on,   // output of reg
    output [7:0] sseg, // output reflects on the led
    output wire [3:0] an,
    output wire [2:0] st,
    output wire debug
    );

    assign an = ctrl;
    wire s0_o;
    wire rst_o;
    wire count_o;

    db db_fsm_clk(
    .clk(clk),
    .sw(s0),
    .db(s0_o)
    );

    db db_fsm_rst(
    .clk(clk),
    .sw(!rst),
    .db(rst_o)
    );

    LD LD(
    .clk(s0_o),
    .rst(rst_o),
    .din(SW0),
    .on(on)
    );

    fsm fsm(
    .clk(s0_o),
    .rst(rst_o),
    .din(SW0),
    .count(count_o),
    .st_cur(st)
    );
```

```
    hex_7seg hex_7seg(
    .hex(count_o),
    .dp(0),
    .sseg(sseg)
    );

    assign debug = s0;
endmodule
```

输入信号利用 SW0 开关控制高低，按下 S2 (R15) 执行输入，输出由三位二进制数表示的当前状态，8 位二进制数表示的当前输入内容，用于数码管的八位二进制输出和一个数码管片选信号。其中按键输入需要消抖，消抖之后将其输入至状态机模块产生结果，最后输出至译码器模块使得检测结果在数码管上显示。

为方便后面的结果展示，我们在这里先给出行为级仿真的 testbench 代码，下一小节便可以直接展示仿真结果：

```
`timescale 1ns / 1ps

module tb_fsm();

    reg clk;
    reg rst;
    reg din;
    wire dout;
    wire [2:0] state;
    wire [7:0] array_now;

    fsm uut (
        .clk(clk),
        .rst(rst),
        .din(din),
        .dout(dout),
        .state(state),
        .array_now(array_now)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        rst = 1;
        din = 0;
```

```
        // Reset
        #20;
        rst = 0;
        #10;

        $display("Time\tValue\tBinary\t\tState\tdout\tMatch");
        $display("-------------------------------------------------");

        // Test specific target values
        test_value(8'd47);   // 00101111 - should match
        test_value(8'd111);  // 01101111 - should match
        test_value(8'd175);  // 10101111 - should match
        test_value(8'd239);  // 11101111 - should match

        #100;
        $display("=== Test Complete ===");
        $finish;
    end

    task test_value;
        input [7:0] value;
        integer i;
        begin
            $display("--- Testing value %d (%b) ---", value, value);

            // Send 8 bits MSB first
            for (i = 7; i >= 0; i = i - 1) begin
                din = value[i];
                @(posedge clk);
                #1;
                $display("%0t\t-\t-\t\t%d\t%b", $time, state, dout);
            end

            // Check result
            @(posedge clk);
            #1;
            if (dout) begin
                $display("*** SUCCESS: Sequence detected for value %d ***", value);
            end else begin
                $display("*** FAIL: Sequence not detected for value %d ***", value);
            end

            // Wait between tests
            repeat(4) @(posedge clk);
        end
    endtask
```

```
    endmodule
```

2、设计采用移位寄存器作为序列检测手段的序列检测器

直接利用移位寄存器来检测时，其结构非常简单，代码如下所示：

```
module shift_reg(
    input clk,
    input din,
    input rst,
    output reg [7:0] on,
    output reg [3:0] out
    );

    reg count = 0;

    always @(posedge clk or posedge rst)begin
        if(rst)begin
            on <= 0;
            out<= 0;
        end else
        begin
            on[7:0] <= {on[6:0],din};
            case({on[4:0],din})
                6'b101111: out <= 1;
                default:   out <= 0;
            endcase
        end
    end
endmodule
```

模块的核心是一个 8 位的移位寄存器 on[7:0]，它在每个时钟周期的上升沿执行移位操作。当复位信号 rst 有效时，整个系统被清零，移位寄存器 on 和输出信号 out 都被重置为 0。在正常工作时，输入信号 din 被移位到寄存器的最低位，而原有数据依次向高位移动，形成 {on[6:0], din} 的移位模式。这种移位操作使得寄存器始终保持着最近输入的 8 位数据历史，为序列检测提供了必要的数据基础。

顶层 top 部分与前面是类似的：

```
module top_sr(
    input clk,//clock of sysytem
    input s0,//signal press
    input rst,//signal:restart
```

```
    input SW0,//signal:din
    input [3:0] ctrl,
    output [7:0] on,//output of reg
    output [7:0] sseg,//output reflects on the led
    output wire[3:0] an
    );

    assign an = ctrl;
    wire s0_o;
    wire rst_o;
    wire [3:0]out_o;

    db db_fsm_clk(
    .clk(clk),
    .sw(s0),
    .db(s0_o)
    );

    db db_fsm_rst(
    .clk(clk),
    .sw(!rst),
    .db(rst_o)
    );

    shift_reg shift_reg(
    .clk(s0_o),
    .rst(rst_o),
    .din(SW0),
    .out(out_o),
    .on(on)
    );

    hex_7seg hex_7seg(
    .hex(out_o),
    .dp(0),
    .sseg(sseg)
    );


endmodule
```

类似地，为方便后面的结果展示，我们在这里先给出行为级仿真的 testbench 代码，下一小节便可以直接展示仿真结果：

```
// tb_sr.v
```

```
`timescale 1ns / 1ps

module tb_sr();

    reg clk;
    reg rst;
    reg din;
    wire [7:0] on;
    wire [3:0] out;

    shift_reg uut (
        .clk(clk),
        .rst(rst),
        .din(din),
        .on(on),
        .out(out)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        rst = 1;
        din = 0;

        // Reset
        #20;
        rst = 0;
        #10;

        $display("Time\tValue\tBinary\t\ton_reg\tout\tMatch");
        $display("-------------------------------------------------------");

        // Test specific target values that contain sequence "101111"
        test_value(8'd47);   // 00101111 - should match
        test_value(8'd111);  // 01101111 - should match
        test_value(8'd175);  // 10101111 - should match
        test_value(8'd239);  // 11101111 - should match

        // Test some values that should NOT match
        test_value(8'd46);   // 00101110 - should NOT match
        test_value(8'd63);   // 00111111 - should NOT match
        test_value(8'd191);  // 10111111 - should NOT match

        #100;
        $display("=== Test Complete ===");
```

```
        $finish;
    end

    task test_value;
        input [7:0] value;
        integer i;
        begin
            $display("--- Testing value %d (%b) ---", value, value);

            // Send 8 bits MSB first
            for (i = 7; i >= 0; i = i - 1) begin
                din = value[i];
                @(posedge clk);
                #1;
                $display("%0t\t-\t-\t\t%b\t%b", $time, on, out);
            end

            // Check result after complete byte is shifted in
            @(posedge clk);
            #1;
            if (out == 1) begin
                $display("*** SUCCESS: Sequence detected for value %d ***", value);
            end else begin
                $display("*** Sequence not detected for value %d ***", value);
            end

            // Wait between tests to observe
            repeat(4) @(posedge clk);
        end
    endtask

    endmodule
```

5 实验内容和结果

1、利用状态机实现的序列检测器行为级仿真结果如下：

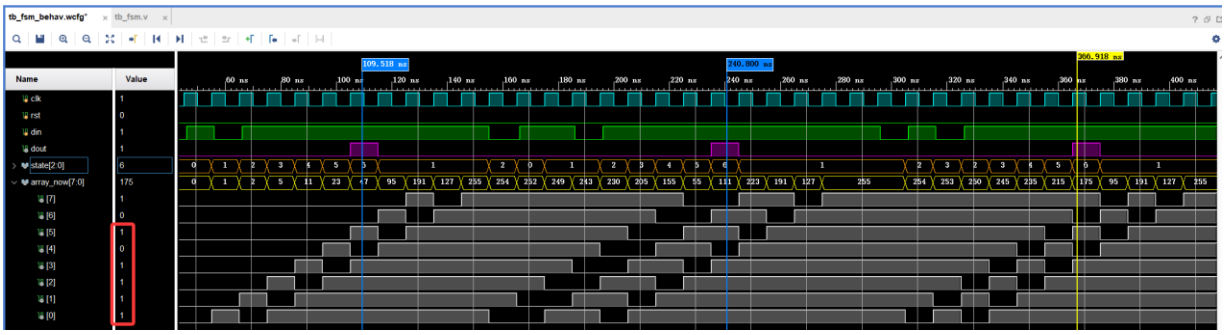


图 2. 状态机序列检测器的行为级仿真结果

通过 array_now 可以很清楚的看到此时的输入序列，根据输入序列不同，状态机的 state 也在对应的变化。当输入序列完整地输入了 **101111** 时可以看到在输入的同时检测结果也输出了高电平。具体而言，八位序列检测器会在下面四种情况给出“满足条件”的高电平：

```
// target sequence: 101111
// four possible preceding bits:
// (00 101111)_2 = (47)_10
// (01 101111)_2 = (111)_10
// (10 101111)_2 = (175)_10
// (11 101111)_2 = (239)_10
```

图中可以看到，我们的序列检测器在 (47_10), (111)_10 和 (175)_10 都能正确识别，这验证了模块的正确性。

对移位寄存器也做类似的仿真测试，结果如下：

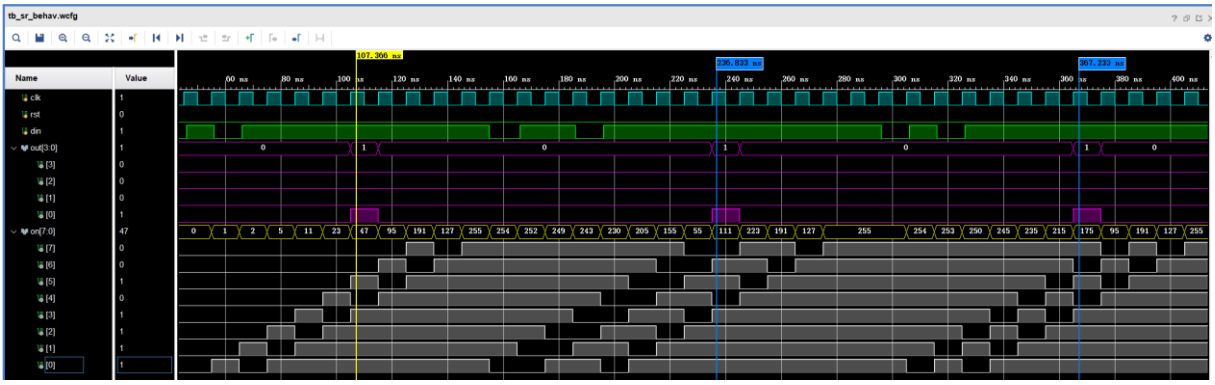


图 3. 移位寄存器序列检测器的行为级仿真结果

可以看到，模块正常运行，在输入为 (47_10), (111)_10 和 (175)_10 时都能正确识别目标序列 **101111**。

2、将程序烧录至板上，检查是否能正常运行。

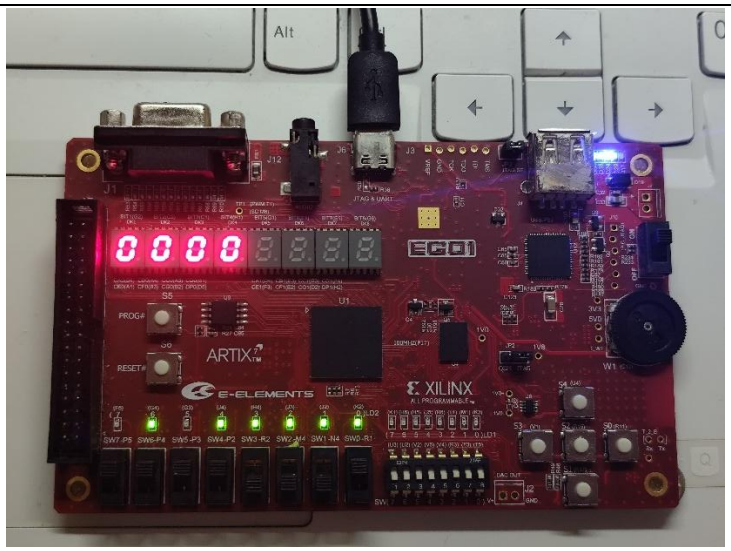
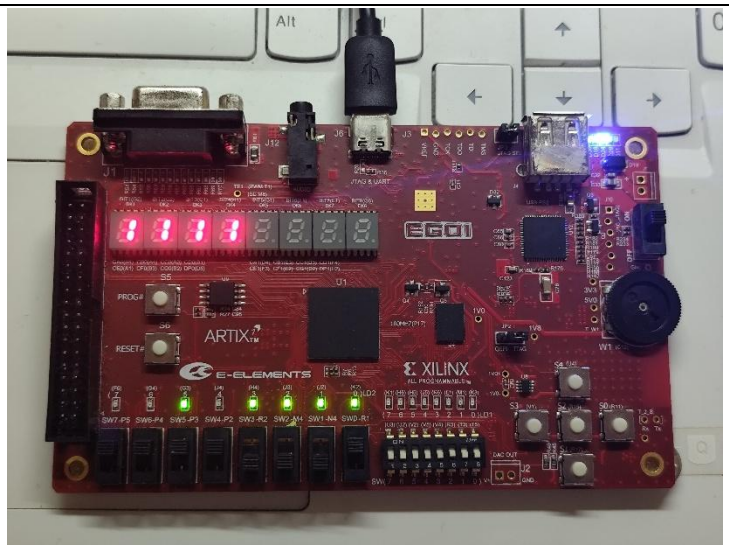
设置 constraints 文件如下：

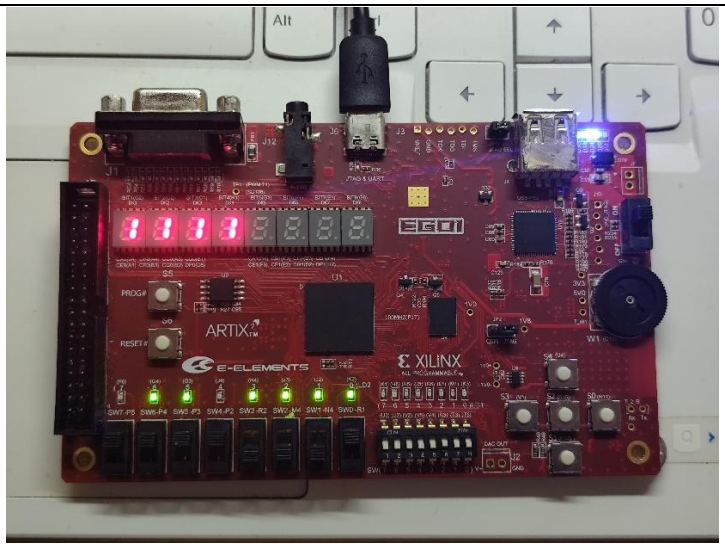
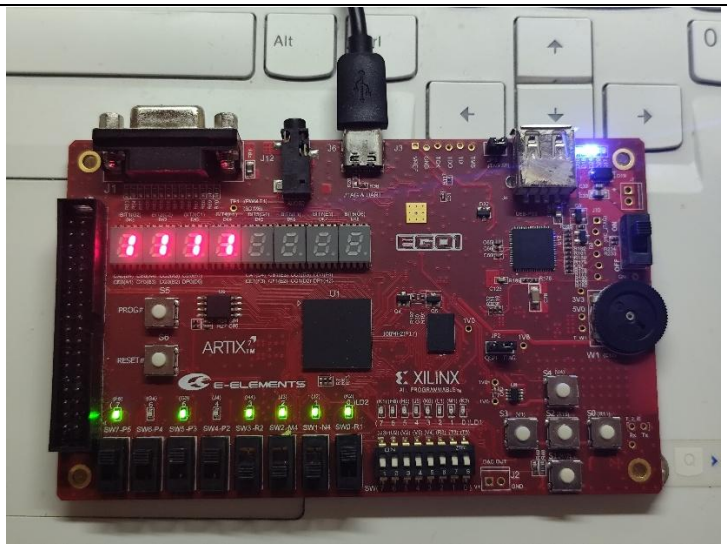
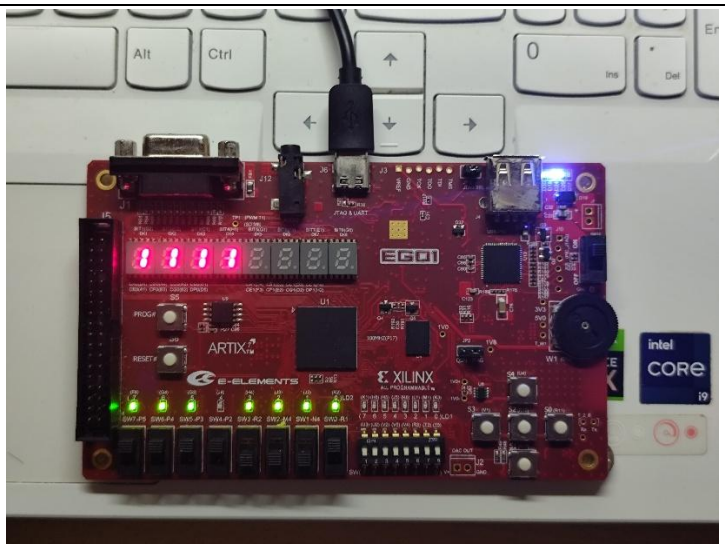
```
# top_fsm.xdc

# Clock, Button & Reset
set_property -dict {PACKAGE_PIN P17 IOSTANDARD LVCMOS33} [get_ports clk]
set_property -dict {PACKAGE_PIN R15 IOSTANDARD LVCMOS33} [get_ports s0]
set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports rst]
# SW0 For input
set_property -dict {PACKAGE_PIN R1 IOSTANDARD LVCMOS33} [get_ports SW0]
# Select Number
set_property -dict {PACKAGE_PIN G2 IOSTANDARD LVCMOS33} [get_ports {an[3]}]
set_property -dict {PACKAGE_PIN C2 IOSTANDARD LVCMOS33} [get_ports {an[2]}]
set_property -dict {PACKAGE_PIN C1 IOSTANDARD LVCMOS33} [get_ports {an[1]}]
set_property -dict {PACKAGE_PIN H1 IOSTANDARD LVCMOS33} [get_ports {an[0]}]
# Select Segment
set_property -dict {PACKAGE_PIN B4 IOSTANDARD LVCMOS33} [get_ports {sseg[6]}]
set_property -dict {PACKAGE_PIN A4 IOSTANDARD LVCMOS33} [get_ports {sseg[5]}]
set_property -dict {PACKAGE_PIN A3 IOSTANDARD LVCMOS33} [get_ports {sseg[4]}]
set_property -dict {PACKAGE_PIN B1 IOSTANDARD LVCMOS33} [get_ports {sseg[3]}]
set_property -dict {PACKAGE_PIN A1 IOSTANDARD LVCMOS33} [get_ports {sseg[2]}]
set_property -dict {PACKAGE_PIN B3 IOSTANDARD LVCMOS33} [get_ports {sseg[1]}]
set_property -dict {PACKAGE_PIN B2 IOSTANDARD LVCMOS33} [get_ports {sseg[0]}]
set_property -dict {PACKAGE_PIN D5 IOSTANDARD LVCMOS33} [get_ports {sseg[7]}]

# Green LEDs 1
set_property -dict {PACKAGE_PIN L1 IOSTANDARD LVCMOS33} [get_ports {str_cur[2]}]
set_property -dict {PACKAGE_PIN M1 IOSTANDARD LVCMOS33} [get_ports {str_cur[1]}]
set_property -dict {PACKAGE_PIN K3 IOSTANDARD LVCMOS33} [get_ports {str_cur[0]}]

# Green LEDs 2
set_property -dict {PACKAGE_PIN F6 IOSTANDARD LVCMOS33} [get_ports {on[7]}]
set_property -dict {PACKAGE_PIN G4 IOSTANDARD LVCMOS33} [get_ports {on[6]}]
set_property -dict {PACKAGE_PIN G3 IOSTANDARD LVCMOS33} [get_ports {on[5]}]
set_property -dict {PACKAGE_PIN J4 IOSTANDARD LVCMOS33} [get_ports {on[4]}]
set_property -dict {PACKAGE_PIN H4 IOSTANDARD LVCMOS33} [get_ports {on[3]}]
set_property -dict {PACKAGE_PIN J3 IOSTANDARD LVCMOS33} [get_ports {on[2]}]
set_property -dict {PACKAGE_PIN J2 IOSTANDARD LVCMOS33} [get_ports {on[1]}]
set_property -dict {PACKAGE_PIN K2 IOSTANDARD LVCMOS33} [get_ports {on[0]}]

#for ctrl
set_property -dict {PACKAGE_PIN P5 IOSTANDARD LVCMOS33} [get_ports {ctrl[3]}]
set_property -dict {PACKAGE_PIN P4 IOSTANDARD LVCMOS33} [get_ports {ctrl[2]}]
set_property -dict {PACKAGE_PIN P3 IOSTANDARD LVCMOS33} [get_ports {ctrl[1]}]
    set_property -dict {PACKAGE_PIN P2 IOSTANDARD LVCMOS33} [get_ports {ctrl[0]}]
```

依次完成 synthesis, implementation 并生成 bitstream 文件，烧录到板子中进行实验。

(1) 有限状态机：

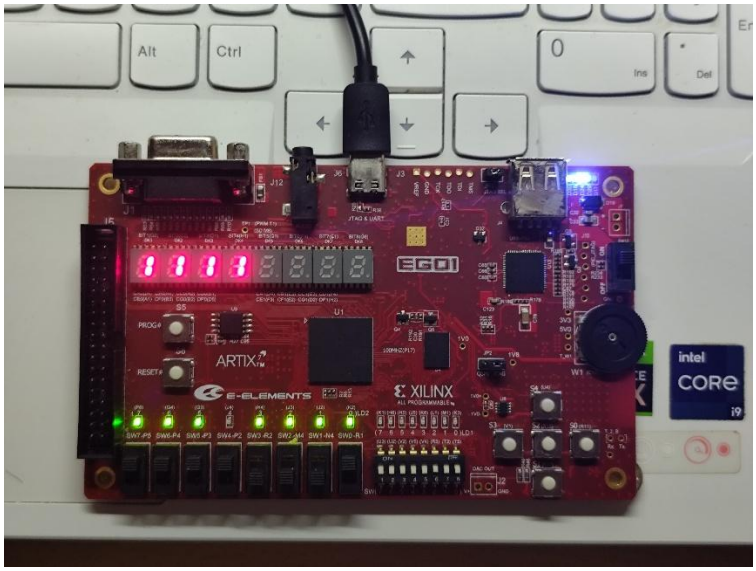
打开开关 SW7 以显示数码管，利用 SW0 控制逻辑高低，S2 执行输入，在板上输入不同序列，观察得到的结果：

输入序列 01011111，后六位为 011111，不满足目标序列，数码管输出零	 图 4. 状态机序列检测器实测效果 (1)
输入序列 00101111，后六位为 101111，满足目标序列，数码管输出一	 图 5. 状态机序列检测器实测效果 (2)

输入序列 01101111，后六位为 101111，满足目标序列，数码管输出一	 图 6. 状态机序列检测器实测效果 (3)
输入序列 10101111，后六位为 101111，满足目标序列，数码管输出一	 图 7. 状态机序列检测器实测效果 (4)
输入序列 11101111，后六位为 101111，满足目标序列，数码管输出一	 图 8. 状态机序列检测器实测效果 (5)

(2) 移位寄存器

类似地，生成 bitstream 文件后烧录至板子中，输入不同序列进行测试，这里直接给出其中一个示例图，其余情况不再重复列出：

移位寄存器构建的序列检测器，输入序列 11101111，后六位为 101111，满足目标序列，数码管输出一	 图 9. 移位寄存器序列检测器实测效果

以上序列检测的两种实现方式，功能均正常。

6 思考题及结论

(1) 用状态机和移位寄存器来实现序列检测，各自的优缺点是什么？

状态机方法在序列检测中的主要优势在于其极高的灵活性和可扩展性。状态机能够优雅地处理复杂的序列模式，包括重叠序列检测、部分匹配时的智能状态回退以及多模式序列的并行检测。并且，当序列长度增加时，状态数目通常呈线性而非指数增长。此外，状态机的设计具有很好的可维护性，状态转移图提供了直观的设计文档，便于后续的调试和功能扩展。

相比之下，移位寄存器方法更为简洁和高效，但可维护性较低。这种实现方式硬件结构简单直接，通过连续的移位操作和并行比较来完成检测，避免了复杂的状态转移逻辑设计。在时序性能方面，移位寄存器通常能够达到更高的工作频率，因为其关键路径较短，特别适合对时序要求严格的高速应用。