浙江大学

本科实验报告

课程名称：计算机组成

姓　　名：应周骏

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3200103894

指导教师：马德

2022 年 3 月 17 日

浙江大学实验报告

课程名称：计算机组成 实验类型：综合

实验项目名称：ALU 和有限状态设计

学生姓名：应周骏 专业：计算机科学与技术 学号：3200103894

同组学生姓名：无 指导老师：马德

实验地点：东 4-509 实验日期：2022 年 3 月 17 日

一、实验目的和要求

Part Ⅰ

1. 复习寄存器传输控制技术；
2. 掌握 CPU 的核心组成：数据通路与控制器；
3. 设计数据通路的功能部件；
4. 进一步了解计算机系统的基本结构；
5. 熟练掌握 IP 核的使用方法；

Part Ⅱ

1. 复习有限状态机的基本概念；
2. 掌握有限状态机的两种模型；
3. 设计有限状态机解决实际问题；

二、实验内容和原理

内容：

1. 设计实现数据通路部件 ALU；
2. 设计实现数据通路部件 Register Files；
3. 设计有限状态机完成序列检测器并测试；

原理：

1. 实验一：ALU 设计

实现 5 个基本运算，要求整理逻辑实验的 ALU，实现 verilog 输入并仿真，并拓展 ALU 的功能。

表 2.1.1 ALU 功能

ALU Control Lines	Function	note
000	**And**	**兼容**
001	**Or**	**兼容**
010	**Add**	**兼容**
110	**Sub**	**兼容**
111	**Set on less than**	--
100	**nor**	**扩展**
101	**srl**	**扩展**
011	**xor**	**扩展**

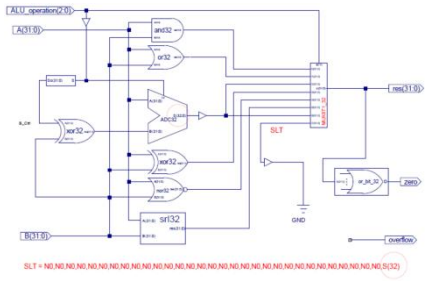

图 2.1.1 ALU 原理图

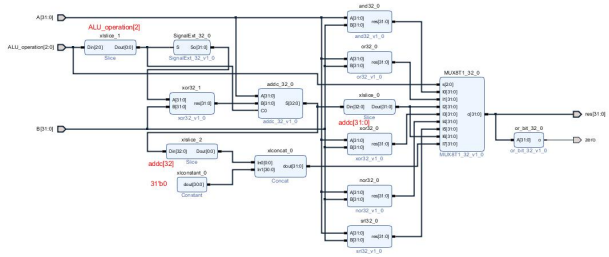

图 2.1.2 ALU 顶层逻辑连接

2. 实验二：Register files 设计

要求实现 32×32bit 寄存器组，优化逻辑实验 Regs，并完成行为描述和仿真

端口要求：

二个读端口：Rs1_addr,Rs1_data；Rs2_addr,Rs2_data。

一个写端口，带写信号:Wt_addr, Wt_data, RegWrite。

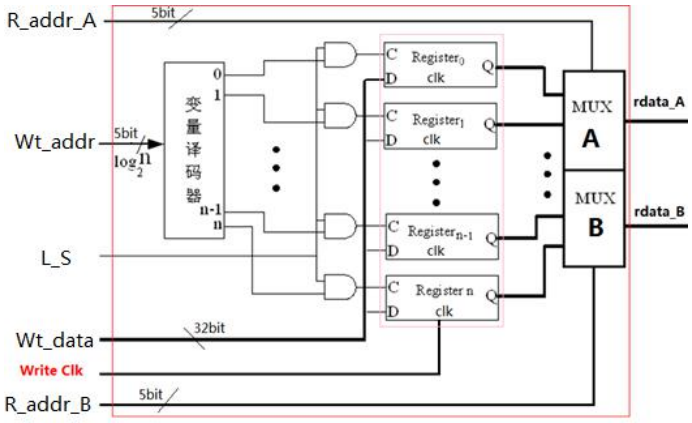

图 2.2.1 Register Files 原理图

3. 实验三：有限状态机设计

3.1 状态机定义

有限状态机（Finite State Machine，简称 FSM）在有限个状态之间按一定规律转换的时序电路。

有限状态机通常是由寄存器组和组合逻辑组成时序电路，根据当前状态和输入信号可以控制下一个状态的跳转，有限状态机在电路中通常是作为控制模块，作为整个电路模块的核心而存在。

3.2 状态机分类

它主要包括两大类：Mealy 型状态机和 Moore 型状态机。

Mealy 型状态机:其组合逻辑的输出不仅与当前状态有关，还与输入有关。

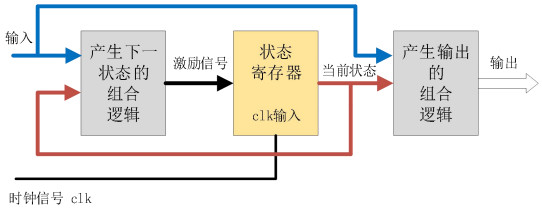

图 2.3.2.1 Mealy 状态机示意图

Moore 型状态机：其组合逻辑的输出只与当前的状态有关。

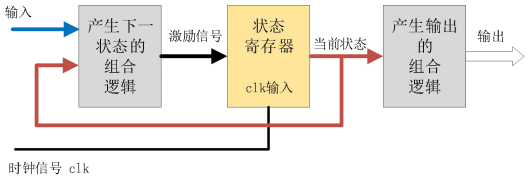

图 2.3.2.2 Moore 状态机示意图

状态寄存器由一组触发器组成，用来记忆状态机当前所处的状态，状态的改

变只发生在时钟的跳变沿。状态是否改变、如何改变，取决于组合逻辑 F 的输出，F 是当前状态和输入信号的函数。状态机的输出是由输出组合逻辑 G 提供的，G 也是当前状态和输入信号的函数。

3.3 状态机设计方法

一段式描述：即状态跳转与输出信号都在同一个 always 块里面进行描述；

二段式描述：即将输出信号,与状态跳转分开描述，便于设计代码管理；

三段式描述：即将输出信号,与状态跳转分开描述，并且状态跳转用组合逻辑来控制；

3.4 状态机设计步骤

系统架构和接口定义：

表 2.3.4.1 接口定义

接口	接口定义
clk	系统时钟
rst_n	系统复位
X	序列输入
Y	检测输出

状态定义和编码：

状态机的编码方式主要包括二进制码（Binary），格雷码（gray），独热码（one hot）。格雷码相对于二进制码而言，在状态跳转的时候，只有单比特翻转，它的功耗相对比较低。独热码相对于格雷码或者二进制码而言，它增加了两个寄存器来表示状态，但是它会更节省组合逻辑电路，因为它在比较状态的时候，只需要比较一个比特位，那么其电路的速度和可靠性就会增加。

绘制状态转换图：

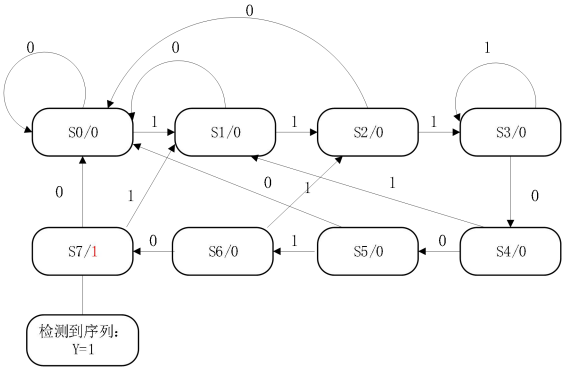

图 2.3.4.1 状态转换图示例

RTL 实现：通过 HDL 语言将状态转换图进行描述实现设计。

三、实验过程和数据记录

实验一：ALU 设计

1. 工程文件建立

新建工程文件，命名为“Exp01_ALU”。

2. Top 模块设计

新建 Verilog 文件“ALU. v”，设为顶层模块，输入以下代码。

```
module ALU(
    input [31:0] A,
    input [2:0] ALU_operation,
    input [31:0] B,
    output [31:0] res,
    output zero
    );
     wire [31:0]MUX8T1_32_0_o;
     wire [31:0]SignalExt_32_0_So;
     wire [32:0]ADC32_0_S;
     wire [31:0]and32_0_res;
     wire [31:0]nor32_0_res;
     wire [31:0]or32_0_res;
     wire [31:0]srl32_0_res;
     wire [31:0]xor32_0_res;
     wire [31:0]xor32_1_res;

    assign res = MUX8T1_32_0_o;
    MUX8T1_32 MUX8T1_32_0(
    .I0(and32_0_res),
    .I1(or32_0_res),
    .I2(ADC32_0_S[31:0]),
    .I3(xor32_0_res),
    .I4(nor32_0_res),
    .I5(srl32_0_res),
    .I6(ADC32_0_S[31:0]),
    .I7({{31'b0,ADC32_0_S[32]}),
    .s(ALU_operation),
    .O(MUX8T1_32_0_o));

    and32 and32_0(
    .A(A),
    .B(B),
    .res(and32_0_res));

    or32 or32_0(
    .A(A),
    .B(B),
    .res(or32_0_res));

    nor32 nor32_0(
    .A(A),
    .B(B),
    .res(nor32_0_res));

    srl32 srl32_0(
    .A(A),
    .B(B),
    .res(srl32_0_res));

    xor32 xor32_0(
    .A(A),
    .B(B),
    .res(xor32_0_res));

    or_bit_32 or_bit_32_0(
    .A(MUX8T1_32_0_o),
    .o(zero));

    xor32 xor32_1(
    .A(SignalExt_32_0_So),
    .B(B),
    .res(xor32_1_res));

    ADC32 ADC32_0(
    .A(A),
    .B(xor32_1_res),
    .CO(ALU_operation[2]),
    .S(ADC32_0_S));

    SignalExt_32 SignalExt_32_1(
    .S(ALU_operation[2]),
    .So(SignalExt_32_0_So));
endmodule
```

图 3.1.1 ALU 结构化描述

3. 模块调用及仿真

向工程中加入实验 0 完成的 and32、or32、ADC32、xor32、nor32、srl32、SignalExt_32、mux8to1_32、or_bit_32 模块，并对 ALU 进行仿真，输入仿真代码如下。

```
module ALU_tb;
        reg [31:0] A, B;
         reg[2:0] ALU_operation;
         wire[31:0]  res;
         wire zero;
    ALU ALU_u(
        .A(A),
        .B(B),
        .ALU_operation(ALU_operation),
        .res(res),
        .zero(zero)
    );
    initial begin
        A=32'hA5A5A5A5;
        B=32'h5A5A5A5A;
        ALU_operation =3'b111;
        #100;
        ALU_operation =3'b110;
        #100;
        ALU_operation =3'b101;
        #100;
        ALU_operation =3'b100;
        #100;
        ALU_operation =3'b011;
        #100;
        ALU_operation =3'b010;
        #100;
        ALU_operation =3'b001;
        #100;
        ALU_operation =3'b000;
        #100;
        A=32'h01234567;
        B=32'h76543210;
        ALU_operation =3'b111;
    end
endmodule
```

图 3.1.2 ALU 仿真代码

得到仿真结果图如下：

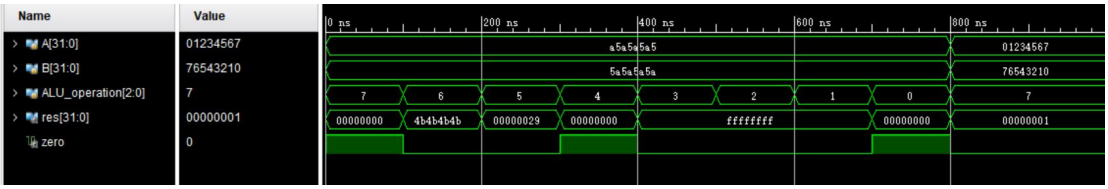

图 3.1.3 ALU 仿真结果图

4. 补充：功能性描绘实现

在结构化描述之外，用功能性描述重新设计 ALU，修改 ALU top 模块，输入代码如下，得到的仿真结果一致。

```
module ALU(
    input [31:0] A,
    input [2:0] ALU_operation,
    input [31:0] B,
    output reg [31:0] res,
    output zero
);
parameter one = 32'h00000001, zero_0 = 32'h00000000;
always @ (*)
    case (ALU_operation)
    3'b000: res = A&B;
    3'b001: res = A|B;
    3'b010: res = A+B;
    3'b011: res = A^B;
    3'b100: res = ~(A|B);
    3'b101: res = A>>B[4:0];
    3'b110: res = A-B;
    3'b111: res = (A<B)?one:zero_0;
    default: res = 32'hx;
    endcase

    assign zero = (res==0)? 1: 0;
endmodule
```

图 3.1.4 ALU 功能性描述

实验二：Register File 设计

1.工程文件建立

新建工程文件“Exp01_Reg”。

2. Register 模块

新建 Verilog 文件“regs.v”，输入如下代码。

```
23  module regs(
24      input clk, rst, RegWrite,
25      input [4:0] Rs1_addr, Rs2_addr, Wt_addr,
26      input [31:0] Wt_data,
27      output [31:0] Rs1_data, Rs2_data
28  );
29  reg [31:0] register [1:31]; // r1 - r31
30  integer i;
31  assign Rs1_data = (Rs1_addr== 0) ? 0 : register[Rs1_addr]; // read
32  assign Rs2_data = (Rs2_addr== 0) ? 0 : register[Rs2_addr]; // read
33  always @(posedge clk or posedge rst)
34  begin if (rst==1) for (i=1; i<32; i=i+1) register[i] <= 0; // reset
35  else if ((Wt_addr != 0) && (RegWrite == 1))
36  register[Wt_addr] <= Wt_data; // write
37  end
38  endmodule
```

图 3.2.1 Register 代码

对该顶层文件进行仿真，输入如下仿真代码。

```
module regs_tb(
    );
    reg clk, rst, RegWrite;
    reg [4:0] Rs1_addr, Rs2_addr, Wt_addr;
    reg [31:0] Wt_data;
    wire [31:0] Rs1_data, Rs2_data;

regs regs1(
    .clk(clk),
    .rst(rst),
    .RegWrite(RegWrite),
    .Rs1_addr(Rs1_addr),
    .Rs2_addr(Rs2_addr),
    .Wt_addr(Wt_addr),
    .Wt_data(Wt_data),
    .Rs1_data(Rs1_data),
    .Rs2_data(Rs2_data)
    );
    always begin
    #10;
    clk = ~clk;
    end
initial begin
    clk = 0;
    rst = 1;
    RegWrite = 0;
    Wt_data = 0;
    Wt_addr = 0;
    Rs1_addr = 0;
    Rs2_addr = 0;
    #100;
    rst = 0;
    RegWrite=1;
    Wt_addr[4:0]=5'h05;
    Wt_data[31:0]=32'ha5a5a5a5;
    #50;
    Wt_addr[4:0]=5'h0a;
    Wt_data[31:0]=32'h5a5a5a5a;
    #50;
    RegWrite=0;
    Rs1_addr[4:0]=5'h05;
    Rs2_addr[4:0]=5'h0a;
end
endmodule
```

图 3.2.2 Register 仿真代码

得到仿真结果如下。

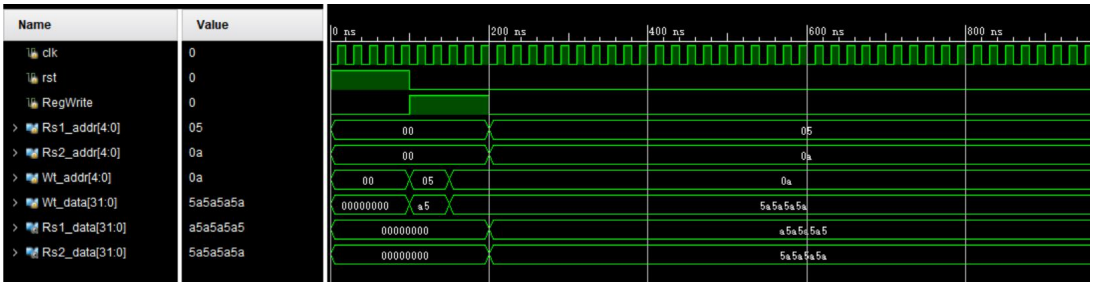

图 3.2.3 Register 仿真结果图

实验三：有限状态机设计

1. 工程文件建立

新建工程文件，命名“Exp01_FSM”。

2. 状态机设计

新建 Verilog 文件“seq.v”，输入以下代码。

```
module seq(
    clk,
    reset,
    in,
    out
    );
    input clk;
    input reset;
    input in;
    output out;

    parameter [2:0] S0 = 3'b000,
                    S1 = 3'b001,
                    S2 = 3'b010,
                    S3 = 3'b011,
                    S4 = 3'b100,
                    S5 = 3'b101,
                    S6 = 3'b110,
                    S7 = 3'b111;
    reg[2:0] cur_state;
    reg[2:0] next_state;
    wire out;

    always@(posedge clk or negedge reset) begin
        if(!reset)
            cur_state <= S0;
        else
            cur_state <= next_state;
    end

    always@(cur_state or in) begin
        case(cur_state)
            S0:begin
                if(in==0) next_state = S0;
                else      next_state = S1;
                end
            S1:begin
                if(in==0) next_state = S0;
                else      next_state = S2;
                end
            S2:begin
                if(in==0) next_state = S0;
                else      next_state = S3;
                end
            S3:begin
                if(in==0) next_state = S4;
                else      next_state = S3;
                end
            S4:begin
                if(in==0) next_state = S5;
                else      next_state = S1;
                end
            S5:begin
                if(in==0) next_state = S0;
                else      next_state = S6;
                end
            S6:begin
                if(in==0) next_state = S7;
                else      next_state = S2;
                end
            S7:begin
                if(in==0) next_state = S0;
                else      next_state = S1;
                end
            default: next_state = S0;
            endcase
    end

        assign out = (cur_state == S7)?1:0;
endmodule
```

图 3.3.1 有限状态机代码

对该原理图进行仿真，输入如下仿真代码。

```
module tb_seq();
    reg clk;
    reg reset;
    reg in;
    wire out;

seq seq_u1(
       .clk(clk),
       .reset(reset),
       .in(in),
       .out(out)
      );
always #20 clk = ~clk;

initial
    begin
        clk = 0;
        reset = 0;
        #20 reset = 1;
   end

//011100101
initial
    begin
        in = 0;
        #30
        in = 1;
        #40
        in = 1;
        #40
        in = 1;
        #40
        in = 0;
        #40
        in = 0;
        #40
        in = 1;
        #40
        in = 0;
        #40
        in = 1;
        #40
        $finish;
        end
```

图 3.3.2 有限状态机仿真代码

得到仿真结果。

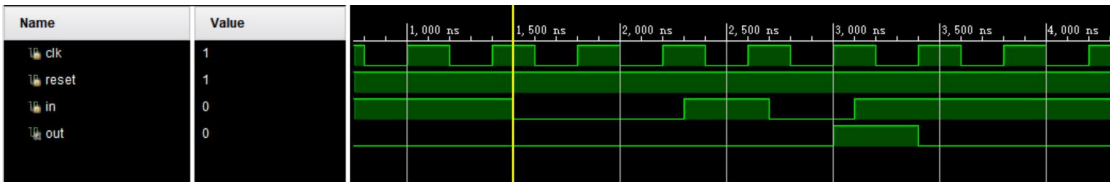

图 3.3.3 有限状态机仿真结果图

四、实验结果分析

实验一：ALU 设计

1. ALU 代码分析

对于结构化描述代码，此处不做展开，其原理即根据 ALU 顶层逻辑图，描述图中给出的电路，用 Verilog 语言代替电路连接。

对于功能性描述代码，可以看到其整体用一个 always 语句，对于 operation 的改变做出响应。由于前期实验 0 中实现的元件的 Verilog 代码大多十分简洁，使用功能性描述，可以很大提高编程效率，但是整体上架构上，使用元件能够使得结构更加清晰，有助于后续其他复杂模块设计

2.ALU 仿真结果分析

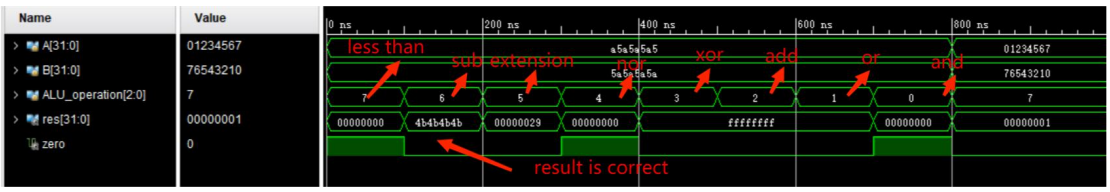

图 4.1.1 ALU 仿真结果分析

在仿真时，ALU_operation 表示的操作可以对照 ALU 设计要求，上图也进行了标注，可以看到算数操作结果达到预想要求。

实验二：Register Files

1. 代码分析

```
23  module regs(
24      input clk, rst, RegWrite,
25      input [4:0] Rs1_addr, Rs2_addr, Wt_addr,
26      input [31:0] Wt_data,
27      output [31:0] Rs1_data, Rs2_data
28  );
29  reg [31:0] register [1:31]; // r1 - r31
30  integer i;
31  assign Rs1_data = (Rs1_addr== 0) ? 0 : register[Rs1_addr]; // read
32  assign Rs2_data = (Rs2_addr== 0) ? 0 : register[Rs2_addr]; // read
33  always @(posedge clk or posedge rst)
34  begin if (rst==1) for (i=1; i<32; i=i+1) register[i] <= 0; // reset
35  else if ((Wt_addr != 0) && (RegWrite == 1))
36  register[Wt_addr] <= Wt_data; // write
37  end
38  endmodule
```

get data from regs

if rst is 1, set all regs as 0

write when write signal set

图 4.2.1 Register 代码分析

2. 仿真结果分析

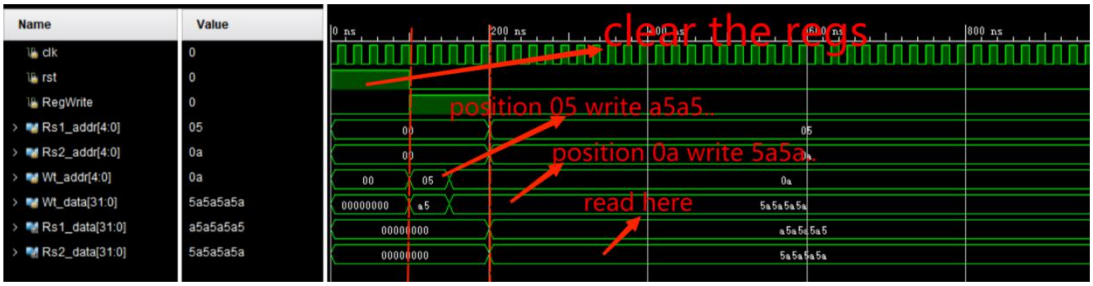

图 4.2.2 Register 仿真分析

首先，在最开始，rst 置 1，表示清零；此后 write 信号置 1，此时，我们可以看到对于地址 0x05，我们写入 a5a5a5a5；对于地址 0x0a，我们写入了 5a5a5a5a。此时 Rs_1，Rs_2 的读取地址为 0，则不读内容。当 write 信号置 0 后，我们对两个寄存器的读取地址做更改，则读到的内容符合我们写入的内容，符合实验预期。

实验三：有限状态机设计

1. 状态机代码分析

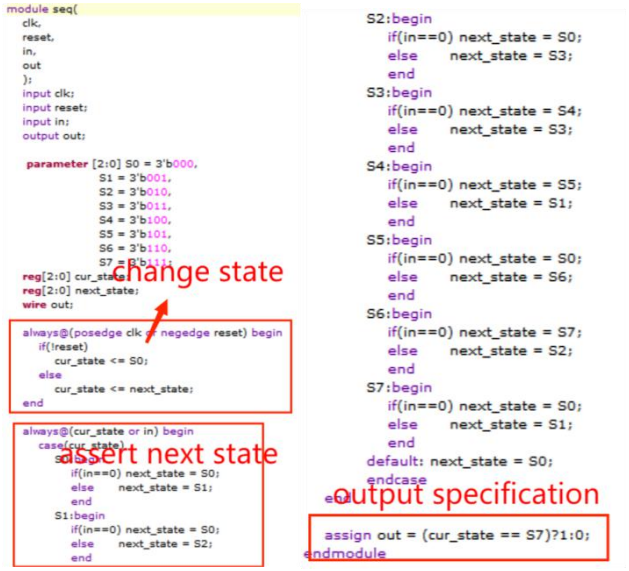

图 4.3.1 FSM 代码分析

2. 状态机仿真分析

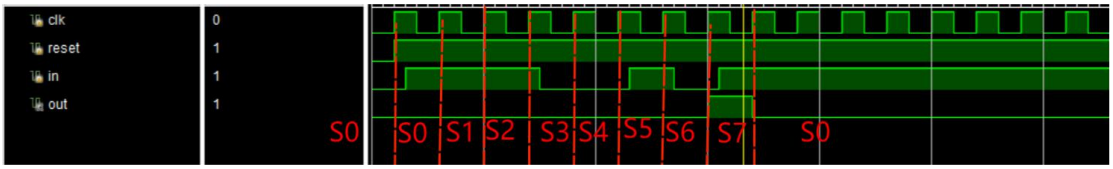

图 4.3.2 FSM 仿真结果分析

对于该段仿真，我们可以通过代码中第二段，即 next state 确定这段代码为突破点，分析各时钟周期状态机状态值，具体分析见上图。

五、讨论与心得

1. 通过本次实验，我复习了数字逻辑设计实验的相关内容，也进一步熟悉了 Vivado 实验平台的操作方法。同时，注意到本次实验对于 Register Files 的设计，不同于之前数字逻辑实验，本次实验采用的是 Write 的数据/地址与 Register 本身分离，且一个地址对应于一个数值，这样的设计比起之前原理图的寄存器设计更简洁，且能够方便的操作。

2. 本次实验也是对于实验 0 设计的几个元件的检查，发现其中依然存在一些错误，也顺利通过本次实验，纠正了一些 Verilog 代码的问题，能够方便后续进一步调用。

3. 本次实验整体相对顺利，为后续 CPU 设计做好铺垫。