1） 如何描述组合电路、时序电路、状态机？如何编写 TestBench？

组合电路是由多个逻辑门组成的电路，其输出只依赖于当前输入信号。这意味着，组合电路不包含存储元件，它的输出仅仅由输入决定，因此组合电路是一种无状态的电路。

时序电路是一种电子电路，它包含存储元件（例如触发器和计数器），可以在时钟脉冲和输入信号的变化下改变它们的状态。 时序电路的输出不仅取决于当前输入信号，还取决于输入信号的历史和时钟信号的变化。因此，时序电路是一种有状态的电路。

状态机是一种特殊的时序电路，它将输入信号映射到输出信号上，并根据预定义的状态转换规则转换其状态。状态机的状态可以通过外部输入信号的变化而发生改变。状态机通常用于控制系统、计算机等领域。在状态机中，有限状态自动机（Finite State Machine，FSM）是最为常见的一种形式。

TestBench 是用于验证设计的仿真环境，它可以模拟各种输入信号并观察输出结果。下面是编写 TestBench 的一般步骤：

定义输入信号：根据设计规格书，定义输入信号的类型、范围和时序。

1. 生成测试向量：根据输入信号的定义，生成一组或多组针对不同测试场景的测试向量。
2. 实例化被测设计：将被测设计实例化为一个模块，并与 TestBench 连接。
3. 应用测试向量：将测试向量应用到 TestBench 中的输入端口上。

4. 运行仿真：运行仿真工具以模拟被测设计和 TestBench 之间的交互，并产生仿真波形。
5. 分析仿真结果：分析仿真波形，检查设计是否满足规格书中的要求。
6. 优化测试向量：根据测试结果，优化测试向量，进一步提高测试覆盖率。
7. 重复以上步骤：反复进行测试和分析，直至确认被测设计的正确性。

需要注意的是，TestBench 的具体实现方式取决于所使用的仿真工具及其支持的硬件描述语言（例如 Verilog 或 VHDL）。在编写 TestBench 时，应该参照仿真工具的文档并遵循最佳实践。

2） ModelSim 工具的使用；

自定 FSM 说明：

设计如下状态，表示单日活动轨迹：

13 种状态描述，6 个判断选择

s0: 宿舍休息 7:00 闹钟（a=1）--》起床早饭

```
if(a)next_st=s1;
else next_st=s0;
```

s1: 起床吃早饭 12 节有课 （b=1）--》上课 否则 --》12 自习

```
if(b)next_st=s2;
else next_st=s3;
```

s2: 12 节课 34 节有课 （c=1）-》上课 否则 --》34 自习

if(c)next_st=s4;

else next_st=s5;

s3: 12 节自习 34 节有课 （c=1）-》上课 否则 --》34 自习

if(c)next_st=s4;

else next_st=s5;

s4: 34 节课 --》 午饭

next_st=s6;

s5: 34 节自习 --》午饭

next_st=s6;

s6: 午饭 56 节有课 （d=1）-》上课 否则 --》56 自习

if(d)next_st=s7;

else next_st=s8;

s7: 56 节课 78 节有课 （e=1）-》上课 否则 --》78 自习

if(e)next_st=s9;

else next_st=s10;

s8: 56 自习 78 节有课 （e=1）-》上课 否则 --》78 自习

if(e)next_st=s9;

else next_st=s10;

s9: 78 节课 --》晚饭

next_st=s11;

s10:78 自习 --》晚饭

```
      next_st=s11;
s11:晚饭 不下雨就夜跑（f=1）否则 --》宿舍休息
      if(f)next_st=s12;
      else next_st=s13;
s12:夜跑 --》宿舍休息
      next_st=s13;
s13:宿舍休息
      next_st=s0;
default: next_st=s0;
```

输出 pos 表示地点：

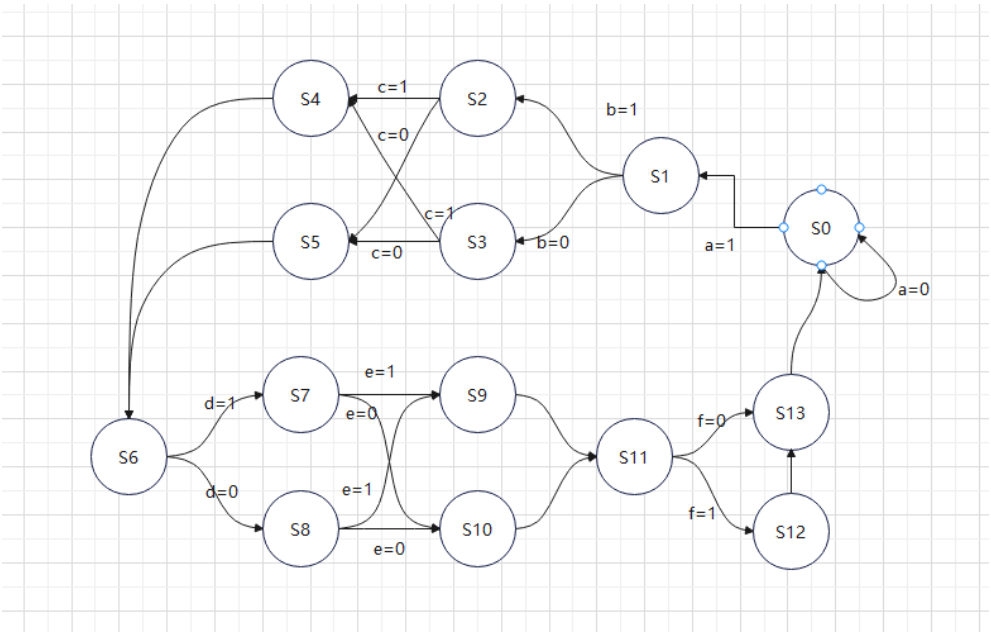
00：寝室

01：食堂

10：教室

11：荫马塘

状态机图：



设计代码说明：

输入端口为控制信号和时钟信号，输出端口为 pos，表示位置：

```
module fsm1(
    input clk,
    input a,
    input b,
    input c,
    input d,
    input e,
    input f,
    output reg [1:0] pos
);
```

使用 4 位 16 进制数表示所有状态：

```
parameter
    s0=4'h0,
    s1=4'h1,
    s2=4'h2,
    s3=4'h3,
    s4=4'h4,
    s5=4'h5,
    s6=4'h6,
    s7=4'h7,
    s8=4'h8,
    s9=4'h9,
    s10=4'ha,
    s11=4'hb,
    s12=4'hc,
    s13=4'hd;
    reg [3:0] state,next_st;
```

下一状态判断：

```
always @ (*)
case(state)
s0:
    if(a)next_st=s1;
    else next_st=s0;
s1:
    if(b)next_st=s2;
    else next_st=s3;
s2:
    if(c)next_st=s4;
    else next_st=s5;
s3:
    if(c)next_st=s4;
    else next_st=s5;
s4:
    next_st=s6;
s5:
    next_st=s6;
s6:
    if(d)next_st=s7;
    else next_st=s8;
s7:
    if(e)next_st=s9;
    else next_st=s10;
s8:
    if(e)next_st=s9;
    else next_st=s10;
s9:
    next_st=s11;
s10:
    next_st=s11;
s11:
    if(f)next_st=s12;
    else next_st=s13;
s12:
    next_st=s13;
s13:
    next_st=s0;
default: next_st=s0;
endcase
```

状态更新与输出：

```
always@(posedge clk) state<=next_st;

always@(*)
case(state)
s0:pos=2'b00;
s1:pos=2'b01;
s2:pos=2'b10;
s3:pos=2'b10;
s4:pos=2'b10;
s5:pos=2'b10;
s6:pos=2'b01;
s7:pos=2'b10;
s8:pos=2'b10;
s9:pos=2'b10;
s10:pos=2'b10;
s11:pos=2'b01;
s12:pos=2'b11;
s13:pos=2'b00;
default:pos=2'b00;
endcase
```

test_bench 代码如下：

第一次从 s0 开始，设定状态变化为：

s0-> s1->s2->s4->s6->s8->s9/s10(由随机产生的 e 决定)->s11->s12->s13->s0

第二次从 s0 开始

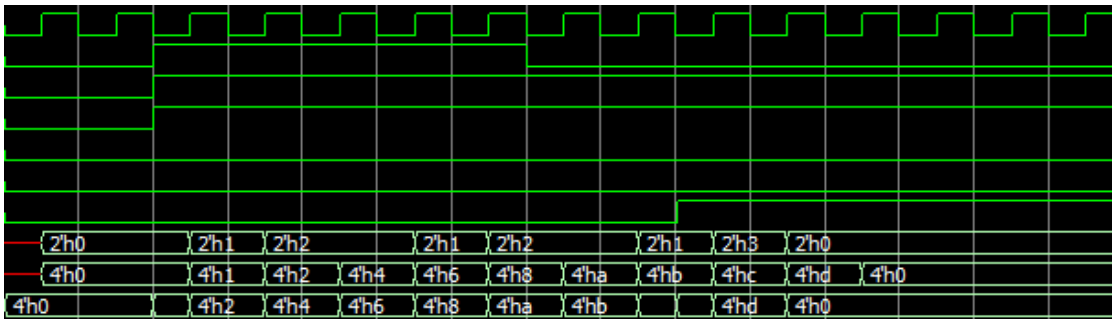
s0-> s1->s3->s5->s6->s8->s9->s11->s12->s13

此后设置 a=0，停留在 s0 状态

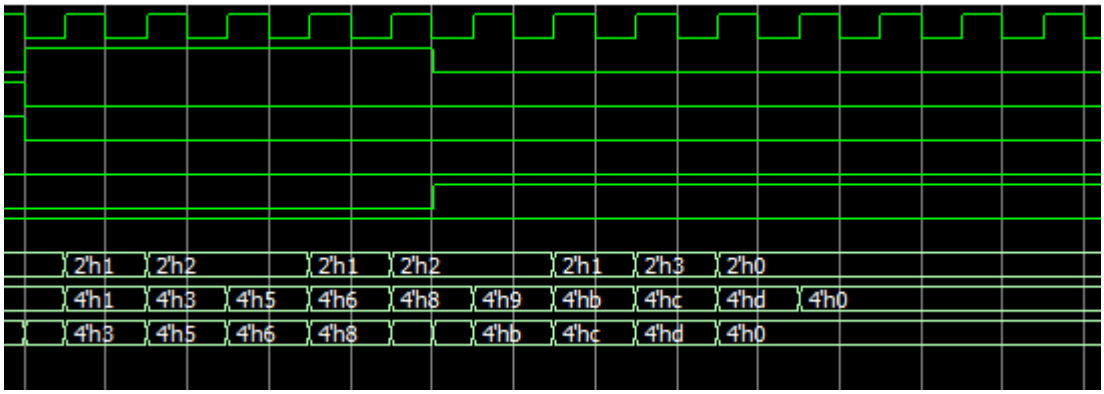
```
`timescale  1ns/100ps
module fsml_tb();
parameter
    s0=4'h0,
    s1=4'h1,
    s2=4'h2,
    s3=4'h3,
    s4=4'h4,
    s5=4'h5,
    s6=4'h6,
    s7=4'h7,
    s8=4'h8,
    s9=4'h9,
    s10=4'ha,
    s11=4'hb,
    s12=4'hc,
    s13=4'hd;
    reg clk,a,b,c,d,e,f;
    wire [1:0] pos;
    reg [3:0] state,next_st;
    fsml fsm(clk,a,b,c,d,e,f,pos);
    initial clk=0;
    always #50 clk=~clk;
    initial begin
        a=0;b=0;c=0;d=0;e=0;f=0;
        #1
        #200
        a=1;
        b=1;
        c=1;
        #500
        e=$random;
        a=0;
        #200
        f=1;
        #1000
        d=1;
        f=1;
        a=0;
        repeat(1024) @(posedge clk);
        $stop;
    end
endmodule
```

ModelSim 仿真波形

第一天：



第二天：



3） EEPROM 读写代码分析；

代码说明

输入：

clk，rstn 分别为时钟和复位信号

write_op：写命令，低电平有效

write_data：写数据

addr：地址

read_op：读命令，低电平有效

输出：

read_data：读到的数据

op_done：操作结束

I2C 协议信号：

scl：I2C 协议的 scl 信号

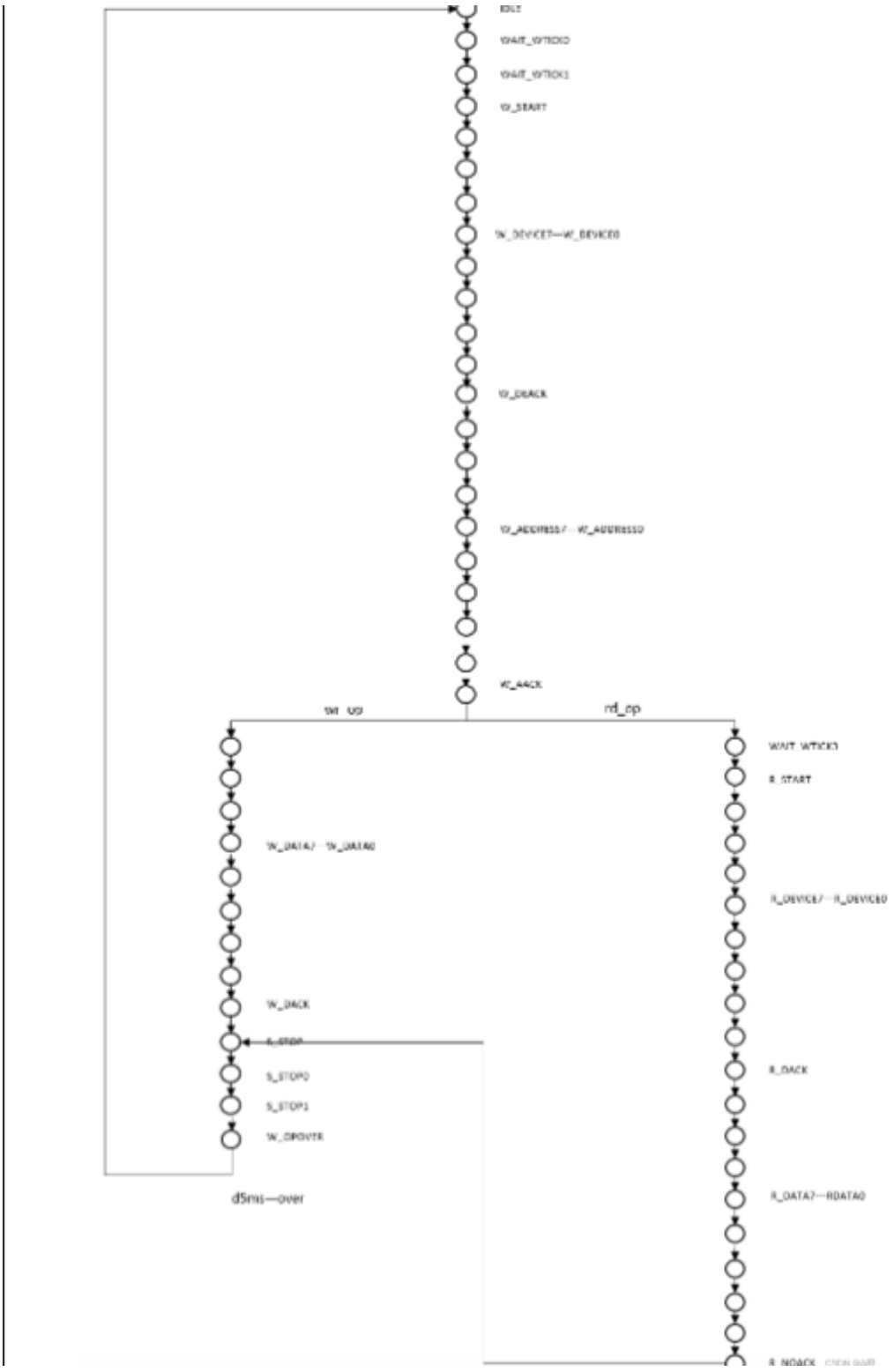
sda：I2C 协议的 sda 信号

```
`timescale 1ns / 1ps
module i2c(
    input clk,                  //时钟
    input rstn,                //复位
    input write_op,             //写操作
    input [7:0]write_data,          //写入的数据
    input read_op,              //读操作
    output reg [7:0]read_data,          //读出的数据
    input [7:0]addr,                //地址
    output op_done,             //操作结束
    output reg scl,             //scl
    inout sda                   //sda
);
```

使用 8 位 16 进制数表示所有状态，共 55 个：

```
parameter IDLE =8'h00,
          WAIT_WTICK0=8'h01,
          WAIT_WTICK1=8'h02,
          W_START=8'h03,
          W_DEVICE7=8'h04,
          W_DEVICE6=8'h05,
          W_DEVICE5=8'h06,
          W_DEVICE4=8'h07,
          W_DEVICE3=8'h08,
          W_DEVICE2=8'h09,
          W_DEVICE1=8'h0a,
          W_DEVICE0=8'h0b,
          W_DEVACK=8'h0c,
          W_ADDRES7=8'h0d,
          W_ADDRES6=8'h0e,
          W_ADDRES5=8'h0f,
          W_ADDRES4=8'h10,
          W_ADDRES3=8'h11,
          W_ADDRES2=8'h12,
          W_ADDRES1=8'h13,
          W_ADDRES0=8'h14,
          W_AACK=8'h15,
          W_DATA7=8'h16,
          W_DATA6=8'h17,
          W_DATA5=8'h18,
          W_DATA4=8'h19,
          W_DATA3=8'h1a,
          W_DATA2=8'h1b,
          W_DATA1=8'h1c,
          W_DATA0=8'h1d,
          W_DACK=8'h1e,
          WAIT_WTICK3=8'h1f,
          R_START=8'h20,
          R_DEVICE7=8'h21,
          R_DEVICE6=8'h22,
          R_DEVICE5=8'h23,
          R_DEVICE4=8'h24,
          R_DEVICE3=8'h25,
          R_DEVICE2=8'h26,
          R_DEVICE1=8'h27,
          R_DEVICE0=8'h28,
          R_DACK=8'h29,
          R_DATA7=8'h2a,
          R_DATA6=8'h2b,
```

状态转移情况如下：



scl 周期是使用计数器对时钟周期计数实现的，一个 scl 周期是 30 个时钟周期，30*200k=6Mhz，为所使用的 FPGA 板的时钟频率。

```
reg [7:0]i2c,next_i;                    //当前状态，下一状态
reg [7:0]div_cnt;                     //计数器
wire scl_tick;
//计数
always @(posedge clk or negedge rstn)
if(!rstn) div_cnt <=8'd0;
else if((i2c==IDLE)|scl_tick) div_cnt <=8'd0;
else div_cnt<=div_cnt+1'b1;
//scl同步
wire scl_ls =(div_cnt==8'd0);              //scl low
wire scl_lc = (div_cnt==8'd7);             //scl low center
wire scl_hs =(div_cnt==8'd15);           //scl high
wire scl_hc = (div_cnt==8'd22);          //scl high center
assign scl_tick = (div_cnt==8'd29);         //一个周期结束
```

下一状态的更新：

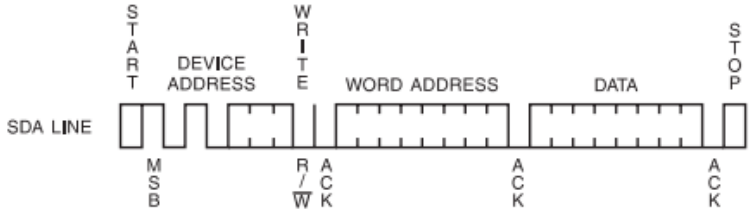
```
//状态
always @(posedge clk or negedge rstn)
if(!rstn) i2c <=0;
else i2c <= next_i;
```

使用 wr_op 和 rd_op 将输入信号 write_op，read_op 表示的读写命令用高电平表示：

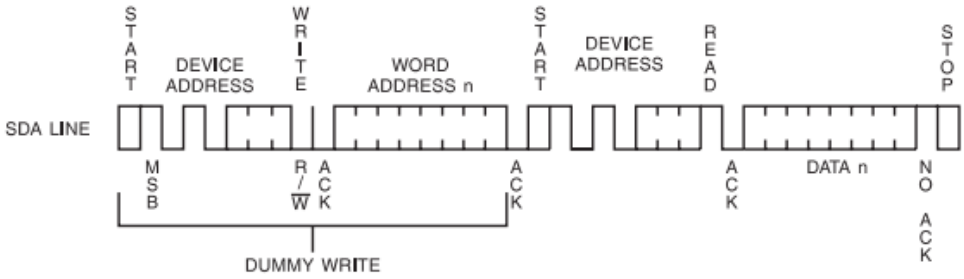
```
always @ (posedge clk or negedge rstn)
if(!rstn) wr_op <= 0;
else if (i2c==IDLE) wr_op <= ~write_op;
else if(i2c==W_OPOVER) wr_op <=1'b0;

always @(posedge clk or negedge rstn)
if(!rstn) rd_op <= 0;
else if (i2c==IDLE) rd_op <= ~read_op;
else if(i2c==W_OPOVER) rd_op <=1'b0;

wire  d5ms_over;
```

下一状态判断，与状态图一致，时间为 scl_tick，即 scl 周期结束。首先是在 scl=1 时，sda 由 1->0，开始数据传输，并先写入器件地址(10100000)和数据地址，然后根据 wr_op，rd_op 判断进行读还是写操作，写操作直接开始写入数据，读操作则需要重新写器件地址和数据地址，然后读取数据。

Byte Write



Random Read




```
always@(*)
case (i2c)
    IDLE: begin next_i = IDLE;if(wr_op|rd_op) next_i = WAIT_WTICK0;end      //有读写操作跳出空闲状态

    //wait tick
    WAIT_WTICK0:begin next_i = WAIT_WTICK0;if(scl_tick) next_i=WAIT_WTICK1;end
    WAIT_WTICK1:begin next_i = WAIT_WTICK1;if(scl_tick) next_i = W_START;end

    //START:SCL=1,SDA=1->0(scl_lc)
    W_START:begin next_i=W_START;if(scl_tick) next_i=W_DEVICE7;end

    //DEVICE ADDRESS (1010_000_0(WRITE))
    W_DEVICE7:begin next_i = W_DEVICE7;if(scl_tick) next_i=W_DEVICE6;end
    W_DEVICE6:begin next_i = W_DEVICE6;if(scl_tick) next_i=W_DEVICE5;end
    W_DEVICE5:begin next_i = W_DEVICE5;if(scl_tick) next_i=W_DEVICE4;end
    W_DEVICE4:begin next_i = W_DEVICE4;if(scl_tick) next_i=W_DEVICE3;end
    W_DEVICE3:begin next_i = W_DEVICE3;if(scl_tick) next_i=W_DEVICE2;end
    W_DEVICE2:begin next_i = W_DEVICE2;if(scl_tick) next_i=W_DEVICE1;end
    W_DEVICE1:begin next_i = W_DEVICE1;if(scl_tick) next_i=W_DEVICE0;end
    W_DEVICE0:begin next_i = W_DEVICE0;if(scl_tick) next_i=W_DEVACK;end

    //ACK
    W_DEVACK:begin next_i=W_DEVACK;if(scl_tick) next_i=W_ADDRES7;end

    //WORD ADDRESS
    W_ADDRES7 :begin next_i = W_ADDRES7;if(scl_tick) next_i=W_ADDRES6;end
    W_ADDRES6 :begin next_i = W_ADDRES6;if(scl_tick) next_i=W_ADDRES5;end
    W_ADDRES5 :begin next_i = W_ADDRES5;if(scl_tick) next_i=W_ADDRES4;end
    W_ADDRES4 :begin next_i = W_ADDRES4;if(scl_tick) next_i=W_ADDRES3;end
    W_ADDRES3 :begin next_i = W_ADDRES3;if(scl_tick) next_i=W_ADDRES2;end
    W_ADDRES2 :begin next_i = W_ADDRES2;if(scl_tick) next_i=W_ADDRES1;end
    W_ADDRES1 :begin next_i = W_ADDRES1;if(scl_tick) next_i=W_ADDRES0;end
    W_ADDRES0 :begin next_i = W_ADDRES0;if(scl_tick) next_i=W_AACK;end
```

```
//WORD ADDRESS
W_ADDRES7 :begin next_i = W_ADDRES7;if(scl_tick) next_i=W_ADDRES6;end
W_ADDRES6 :begin next_i = W_ADDRES6;if(scl_tick) next_i=W_ADDRES5;end
W_ADDRES5 :begin next_i = W_ADDRES5;if(scl_tick) next_i=W_ADDRES4;end
W_ADDRES4 :begin next_i = W_ADDRES4;if(scl_tick) next_i=W_ADDRES3;end
W_ADDRES3 :begin next_i = W_ADDRES3;if(scl_tick) next_i=W_ADDRES2;end
W_ADDRES2 :begin next_i = W_ADDRES2;if(scl_tick) next_i=W_ADDRES1;end
W_ADDRES1 :begin next_i = W_ADDRES1;if(scl_tick) next_i=W_ADDRES0;end
W_ADDRES0 :begin next_i = W_ADDRES0;if(scl_tick) next_i=W_AACK;end

//ACK
W_AACK:begin next_i = W_AACK;
            if(scl_tick&wr_op) next_i=W_DATA7;          //wr_op即写命令，开始写数据
            else if(scl_tick&rd_op) next_i=WAIT_WTICK3;         //rd_op读命令,则下一状态为WAIT_WTICK3
       end

//WRITE DATA[7:0]
W_DATA7:begin next_i=W_DATA7;if(scl_tick)next_i=W_DATA6;end
W_DATA6:begin next_i=W_DATA6;if(scl_tick)next_i=W_DATA5;end
W_DATA5:begin next_i=W_DATA5;if(scl_tick)next_i=W_DATA4;end
W_DATA4:begin next_i=W_DATA4;if(scl_tick)next_i=W_DATA3;end
W_DATA3:begin next_i=W_DATA3;if(scl_tick)next_i=W_DATA2;end
W_DATA2:begin next_i=W_DATA2;if(scl_tick)next_i=W_DATA1;end
W_DATA1:begin next_i=W_DATA1;if(scl_tick)next_i=W_DATA0;end
W_DATA0:begin next_i=W_DATA0;if(scl_tick)next_i=W_DACK;end
```

```
//DEVICE ADDRESS(1010_000_1(READ))
R_DEVICE7:begin next_i=R_DEVICE7; if(scl_tick) next_i=R_DEVICE6;end
R_DEVICE6:begin next_i=R_DEVICE6; if(scl_tick) next_i=R_DEVICE5;end
R_DEVICE5:begin next_i=R_DEVICE5; if(scl_tick) next_i=R_DEVICE4;end
R_DEVICE4:begin next_i=R_DEVICE4; if(scl_tick) next_i=R_DEVICE3;end
R_DEVICE3:begin next_i=R_DEVICE3; if(scl_tick) next_i=R_DEVICE2;end
R_DEVICE2:begin next_i=R_DEVICE2; if(scl_tick) next_i=R_DEVICE1;end
R_DEVICE1:begin next_i=R_DEVICE1; if(scl_tick) next_i=R_DEVICE0;end
R_DEVICE0:begin next_i=R_DEVICE0; if(scl_tick) next_i=R_DACK;end

//ACK
R_DACK:begin next_i=R_DACK;if(scl_tick) next_i=R_DATA7;end

//READ DATA[7:0], SDA:input
R_DATA7:begin next_i=R_DATA7;if(scl_tick) next_i=R_DATA6;end
R_DATA6:begin next_i=R_DATA6;if(scl_tick) next_i=R_DATA5;end
R_DATA5:begin next_i=R_DATA5;if(scl_tick) next_i=R_DATA4;end
R_DATA4:begin next_i=R_DATA4;if(scl_tick) next_i=R_DATA3;end
R_DATA3:begin next_i=R_DATA3;if(scl_tick) next_i=R_DATA2;end
R_DATA2:begin next_i=R_DATA2;if(scl_tick) next_i=R_DATA1;end
R_DATA1:begin next_i=R_DATA1;if(scl_tick) next_i=R_DATA0;end
R_DATA0:begin next_i=R_DATA0;if(scl_tick) next_i=R_NOACK;end

//NO ACK
R_NOACK:begin next_i=R_NOACK;if(scl_tick) next_i=S_STOP;end

//STOP
S_STOP:begin next_i=S_STOP;if(scl_tick) next_i=S_STOP0;end
S_STOP0:begin next_i=S_STOP0;if(scl_tick) next_i=S_STOP1;end
S_STOP1:begin next_i=S_STOP1;if(scl_tick) next_i=W_OPOVER;end

//WAIT write_op=0,read_op=0;
W_OPOVER:begin next_i = W_OPOVER;if(d5ms_over)next_i=IDLE;end       //操作结束回到空闲状态
default:begin next_i= IDLE;end
dcase
```

SCL 同步的实现：

空闲，等待，操作结束，start 开始等状态下 SCL 都是高电平，因此不需要 clr_scl 对 SCL 清零。另外 clr_scl 只在 scl_ls（scl 的低电平开始）处才置 1，把 scl 清 0，在 15 个 clk 周期的 scl_hs 处，再把 scl 拉高，就实现了 SCL 周期。

```
//SCL
assign clr_scl=scl_ls&(i2c!=IDLE)&(i2c!=WAIT_WTICK0)&                    //clr_scl，scl置0信号
               (i2c != WAIT_WTICK1)&(i2c!=W_START)&(i2c!=R_START)
               &(i2c!=S_STOP0)&(i2c!=S_STOP1)&(i2c!=W_OPOVER);

always @(posedge clk or negedge rstn)
if(!rstn) scl <= 1'b1;                                  //复位，scl为高电平
else if(clr_scl) scl <= 1'b0;                               //scl 1->0
else if(scl_hs) scl <=1'b1;                             //scl 0->1
```

SDA：

SDA 的控制信号声明，这些信号在对应的状态且 scl 在低电平的中心时置 1，根据这些控制信号，在 SDA 上进行数据读写。而 i2c_reg 用来暂存数据。

```
//SDA
reg [7:0]i2c_reg;
assign start_clr = scl_lc &((i2c==W_START)|(i2c==R_START));            //在scl low center开始读写操作
assign ld_wdevice = scl_lc&(i2c==W_DEVICE7);                        //加载器件地址
assign ld_waddres = scl_lc&(i2c==W_ADDRES7);                        //加载数据地址
assign ld_wdata= scl_lc&(i2c==W_DATA7);                     //加载数据
assign ld_rdevice = scl_lc&(i2c==R_DEVICE7);                        //读操作的器件地址
assign noack_set = scl_lc&(i2c==R_NOACK);                       //读操作完毕
assign stop_clr = scl_lc&(i2c==S_STOP);
assign stop_set = scl_lc&((i2c==S_STOP0)|(i2c==WAIT_WTICK3));

assign i2c_rlf =scl_lc&(                                //有读写则i2c_rlf
               (i2c == W_DEVICE6)|
```

使用信号 i2c_rlf 表示是否有读写操作，如果有，则 i2c_reg 将左移，一位一位处理数据。

```
assign i2c_rlf =scl_lc&(                                    //有读写则i2c_rlf
                (i2c == W_DEVICE6)|
                (i2c == W_DEVICE5)|
                (i2c == W_DEVICE4)|
                (i2c == W_DEVICE3)|
                (i2c == W_DEVICE2)|
                (i2c == W_DEVICE1)|
                (i2c == W_DEVICE0)|
                (i2c == W_ADDRES6)|
                (i2c == W_ADDRES5)|
                (i2c == W_ADDRES4)|
                (i2c == W_ADDRES3)|
                (i2c == W_ADDRES2)|
                (i2c == W_ADDRES1)|
                (i2c == W_ADDRES0)|
                (i2c == W_DATA6)|
                (i2c == W_DATA5)|
                (i2c == W_DATA4)|
                (i2c == W_DATA3)|
                (i2c == W_DATA2)|
                (i2c == W_DATA1)|
                (i2c == W_DATA0)|
                (i2c == R_DEVICE6)|
                (i2c == R_DEVICE5)|
                (i2c == R_DEVICE4)|
                (i2c == R_DEVICE3)|
                (i2c == R_DEVICE2)|
                (i2c == R_DEVICE1)|
                (i2c == R_DEVICE0));
```

根据上述控制信号，将输入的特定数据保存到 i2c_reg

```
always@(posedge clk or negedge rstn)
if(!rstn) i2c_reg <= 8'hff;                                 //复位，高电平
else if(start_clr) i2c_reg <= 8'h00;                        //开始读写，低电平
else if(ld_wdevice) i2c_reg <= {4'b1010,3'b000,1'b0};               //10100000 写
else if(ld_waddres) i2c_reg <= addr;                        //加载数据地址
else if(ld_wdata) i2c_reg <= write_data;                        //加载写入的数据
else if(ld_rdevice) i2c_reg <= {4'b1010,3'b000,1'b1};               //10100001 读
else if(noack_set) i2c_reg <= 8'hff;                        //NOACK
else if(stop_clr) i2c_reg <= 8'h00;
else if(stop_set) i2c_reg <= 8'hff;
else if(i2c_rlf) i2c_reg <= {i2c_reg[6:0],1'b0};                    //左移
```

sda 输出使用 sda 使能信号 sda_en 控制，写器件地址，数据地址，写数据时使能信号为 1，接收 ACK 响应时使能为 0。sda 输出 i2c_reg 的最高位，即一位一位完成读或写。

```
assign sda_o = i2c_reg[7];                                  //sda输出
assign clr_sdaen = (i2c==IDLE)|                             //sda使能置0信号
                   (scl_lc&(
                   (i2c==W_DEVACK)|
                   (i2c==W_AACK)|
                   (i2c==W_DACK)|
                   (i2c==R_DACK)|
                   (i2c==R_DATA7)));

assign set_sdaen = scl_lc&(                                 //sda使能置1信号
                   (i2c==WAIT_WTICK0)|
                   (i2c==W_ADDRES7)|
                   (i2c==W_DATA7)|
                   (i2c==WAIT_WTICK3)|
                   (i2c==S_STOP)|
                   (i2c==R_NOACK));

reg sda_en;
always @(posedge clk or negedge rstn)
if(!rstn) sda_en <= 0;
else if (clr_sdaen) sda_en <=0;
else if(set_sdaen) sda_en <= 1'b1;

assign sda= sda_en?sda_o: 1'bz;                             //sda使能为1时sda可工作
```

读取数据时将数据读到 read_data

```
assign sda= sda_en?sda_o: 1'bz;                             //sda使能为1时sda可工作

assign sda_wr = scl_hc &(                                   //读数据
                (i2c==R_DATA7)|
                (i2c==R_DATA6)|
                (i2c==R_DATA5)|
                (i2c==R_DATA4)|
                (i2c==R_DATA3)|
                (i2c==R_DATA2)|
                (i2c==R_DATA1)|
                (i2c==R_DATA0));

always@(posedge clk or negedge rstn)
if(!rstn) read_data <= 0;
else if(sda_wr) read_data <= {read_data[6:0],sda};                  //左移读入数据
```

最后使用 d5ms_count 计数时钟周期等待，使用时钟频率为 6Mhz，一个周期 166ns，等待约 1.36ms，然后重新开始完成新的读写命令。

```
//op_done
assign op_done = (i2c == W_OPOVER);                    //操作结束

//Write Cycle(5ms)
//6MHZ = 166ns,5ms/166ns = 31
reg [12:0] d5ms_cnt;
always @(posedge clk or negedge rstn)
if(!rstn)    d5ms_cnt <= 8'd0;
else if(i2c==IDLE) d5ms_cnt <= 8'd0;
else if(i2c==W_OPOVER) d5ms_cnt <= d5ms_cnt + 1'b1;

assign d5ms_over = (d5ms_cnt==13'h1FFF);

endmodule
```

TestBench 代码说明

模块声明与实例化：

```
`timescale 1ns / 1ps

// `include "../I2C_Tes/M24XXX_Memory.v"
// `include "./i2c.v"

module i2c_tb();

reg clk;
reg rstn;
reg write_op;
reg [7:0] write_data;
reg read_op;
wire [7:0] read_data;
reg [7:0] addr;

wire scl;
wire sda;

pullup(sda);
i2c i2c_dut(
 .clk (clk),
 .rstn(rstn),
 .write_op(write_op),
 .write_data(write_data),
 .read_op(read_op),
 .read_data(read_data),
 .addr(addr),
 .op_done(op_done),
 .scl(scl),
 .sda(sda)
);
//EEPROM
M24XXX M24XXX_dut(
    .Ei(3'b0),
    .SDA(sda),
    .SCL(scl),
    .WC(1'b0),
    .VCC(1'b1)
);
```

根据时钟频率 6Mhz 设置周期 166ns，并对信号初始化：

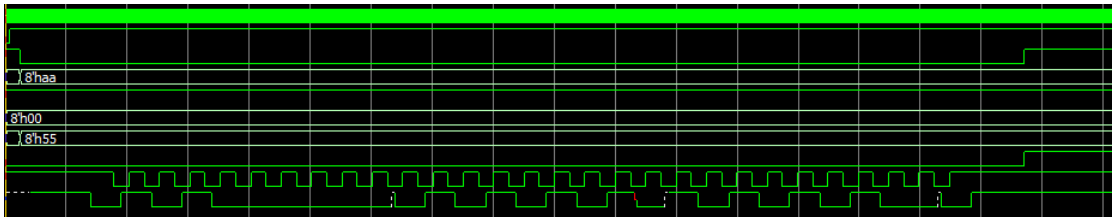
```
always #(166/2) clk = ~clk;                    //6Mhz

initial
begin
    clk = 0;
    rstn = 0;
    write_op=1'b1;
    write_data=8'h00;
    read_op=1'b1;
    addr=0;

    repeat(5) @(posedge clk);
    rstn = 1'b1;
end
```

首先输入写命令信号，地址为 8‘h55，写入的数据为 8’haa。等待操作完成后，将 write_op 设为 1（高电平无效），输入读命令信号，读出地址 8’h55 中的数据，读出的数据应该为刚刚写入的 8’haa。

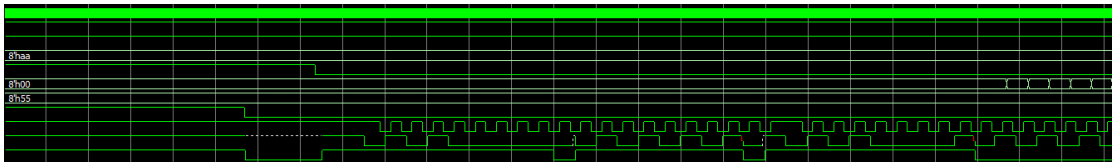
```
initial
begin
    wait(rstn);
    repeat(10) @(posedge clk);
    write_op=1'b0;
    addr = 8'h55;
    write_data= 8'haa;

    wait(op_done);
    write_op=1'b1;
    $display ($stime/1,"ns","Write:Addr(%h)=(%h)\n",addr,write_data);

    wait(!op_done);
    repeat(100)@(posedge clk);
    read_op=1'b0;
    addr = 8'h55;
    wait(op_done);
    read_op=1'b1;
    $display ($stime/1,"ns","Read:Addr(%h)=(%h)\n",addr,read_data);

    repeat(1000) @(posedge clk);
    $stop;
end

endmodule
```

ModelSim 仿真波形

首先是向地址为 8‘h55 处写入数据，sda 在 scl 为高电平时产生下降沿，表示开始工作，scl 开始翻转，依次写入器件地址，数据地址，以及数据 8’haa，并接收响应。最后 scl 为高电平，sda 产生上升沿，停止工作。



经过等待后开始读出 0x55 处的数据，先写入器件地址，数据地址（dummy write），然后再次写入器件地址，读出数据。



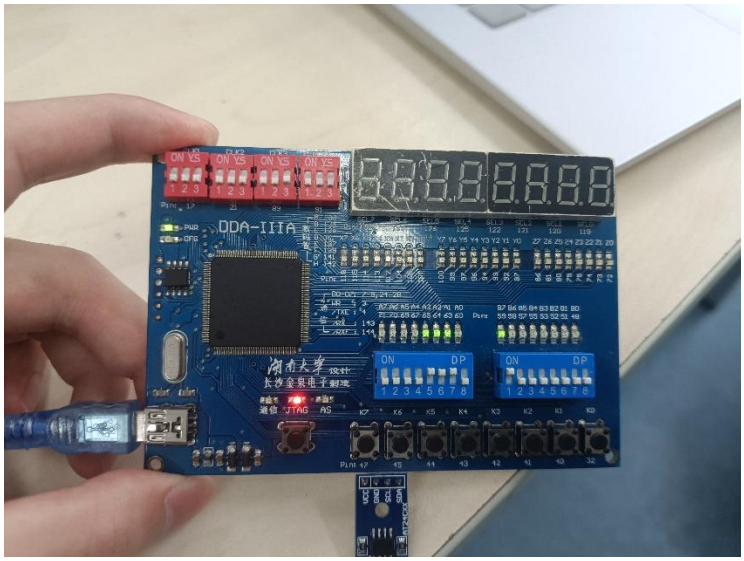
4） 实验总结；

通过实验，学习了 verilog 语言的基本语法以及使用 verilog 语言描述时序逻辑和组合逻辑的方式。学习了如何使用 verilog 编写有限状态机并练习编写了简单的有限状态机。了解了 test_bench 的编写以及熟悉了使用 ModelSim 进行波形仿真。理解了 I2C 接口协议以及 I2C 协议下 SCL，SDA 数据是如何传输的。对于给出的参考代码能够基本理解，也能够对应波形仿真结果解释 I2C 协议的数据传输，最后将代码下载至 FPGA 开发板，验证了 I2C 协议正常工作。对于实验中 I2C 协议的 verilog 实现的一些具体细节理解的还不够深刻，使用

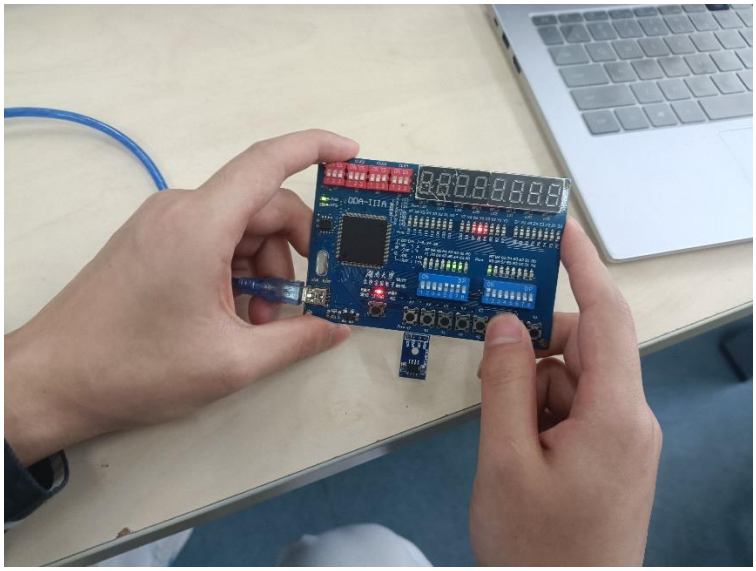
verilog 语言编写有限状态机的能力还比较基础，还需要后续的练习和进一步的学习。

EEPROM 读写代码下载及验证

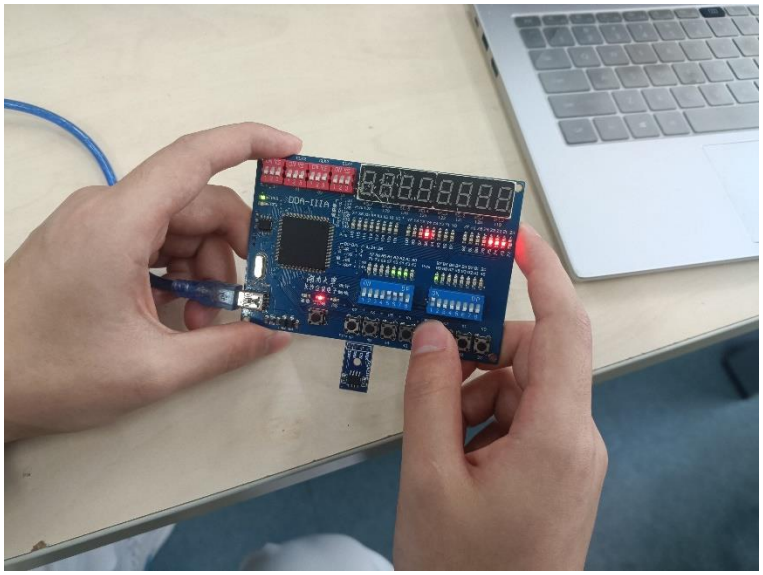
设定数据和地址



写入



读取



清空