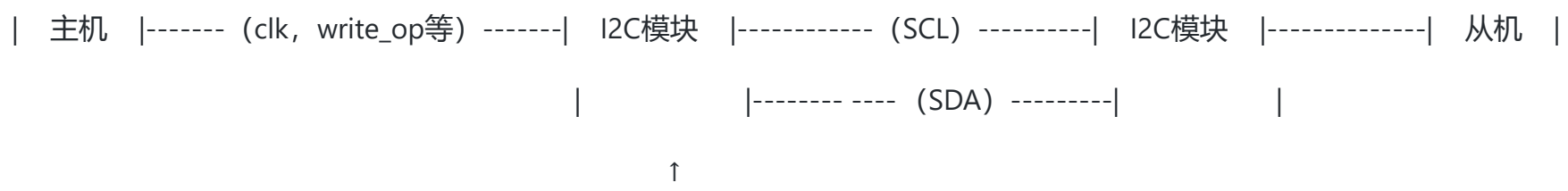
I2C

I2C协议用于主从机之间的数据传输，主机和从机设备靠SCL和SDA两条线连接，按照I2C协议的规定进行数据传输。只有两条线连接在一起，主机和从机不知道何时接收和发送数据，不知道怎样给出相应信号，因此还需要模块实现与解释I2C协议，主机或从机只需要给模块命令和数据，让模块决定怎样在SCL和SDA线上收发数据及信号。本次实验完成的是主机的I2C模块。



我们要实现的是这个

该模块的接口如下：

输入：

- clk：时钟信号
- rstn：复位，低电平有效
- write_op：写命令，低电平有效
- [7:0]write_data：写入的数据
- read_op：读命令，低电平有效
- [7:0]addr：需要读写数据的地址

输出

- op_done：操作结束
- scl：用于同步
- [7:0]read_data：读到的数据

输入输出：inout sda

主机要执行写操作，只要给I2C模块发出write_op=0，write_data和addr的数据就可以了，I2C模块将会根据I2C协议向SDA发数据，确认向从机写入了数据后，输出op_done=1，主机就知道操作已经完成了。读操作是类似的。scl和sda用于和其他设备收发数据，其他接口都是和主机进行交流使用的。

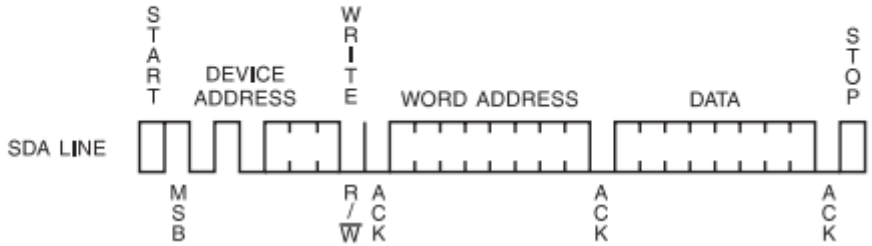
随机读与写操作

默认大家对I2C协议有大致的了解，接下来只看随机读和写操作时，在sda线上到底发生了什么。注意，接下来提到的主机发数据，从机发响应等，其实都是主机和从机的I2C模块做的，不是主从机做的，主从机只负责与I2C模块连接，因为这个模块也看作主从机设备的一部分，所以才这么说的。

从机是存储器EEPROM。

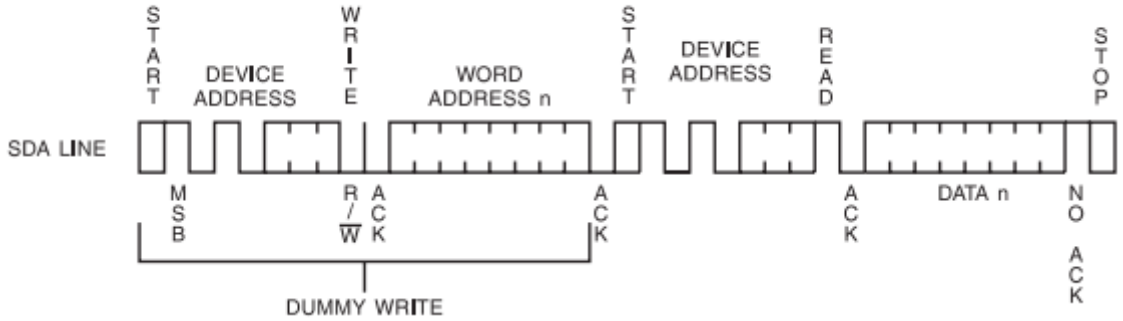
进行写入：

Byte Write



进行读

Random Read



可以看到读和写操作前面有一部分是相同的（DUMMYWRITE），即start信号发出后，先由主机写入器件地址，然后写入数据地址，每次写入后从机都会回复一个ACK信号表示收到。其中器件地址是从机EEPROM的地址**1010000X**，X为0表示写，1表示读，一开始写入器件地址时，X都是0。如果真的要进行写操作，接下来只需要由主机发出数据就可以了，从机发出ACK响应回复收到。如果要进行读操作，再重新写一遍器件地址，这次X就是1了，表示接下来真的要读数据了，从机读出数据，主机收到数据以后回复NOACK给从机表示收到数据。

以上的数据传输都发生在SDA线上，同一时刻，主机和从机只有一个可以向SDA发数据。还有许多其他细节在具体的实现代码中，在代码中进行分析。

代码分析

端口

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

状态

Byte Write : START + DEVICE +ACK + ADDR + ACK + DATA + ACK + STOP　　Random Read : START + DEVICE + ACK +ADDR + ACK+START + DEVICE + DATA + NO ACK + STOP

每个状态用8位16进制数表示，对照上面图中SDA上数据传输情况，每一位数据传输都有一个对应的状态，多出了一些等待状态。

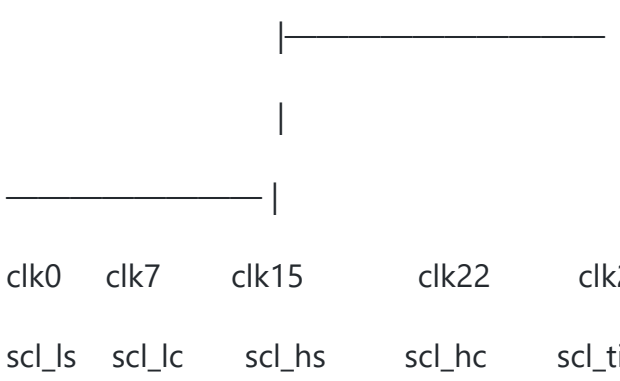
```
parameter IDLE =8'h00,
          WAIT_WTICK0=8'h01,
          WAIT_WTICK1=8'h02,
          W_START=8'h03,
          W_DEVICE7=8'h04,
          W_DEVICE6=8'h05,
          W_DEVICE5=8'h06,
          W_DEVICE4=8'h07,
          W_DEVICE3=8'h08,
          W_DEVICE2=8'h09,
          W_DEVICE1=8'h0a,
          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,
            R_DATA5=8'h2c,
            R_DATA4=8'h2d,
            R_DATA3=8'h2e,
            R_DATA2=8'h2f,
            R_DATA1=8'h30,
            R_DATA0=8'h31,
            R_NOACK=8'h32,
            S_STOP=8'h33,
            S_STOP0=8'h34,
            S_STOP1=8'h35,
            W_OPOVER=8'h36;
reg [7:0]i2c,next_i;                    //当前状态，下一状态
```

SCL同步

时钟的频率很高，读写数据不能以时钟周期为周期进行，设备进行响应和读写需要的时间远大于时钟周期，因此使用SCL同步方式来同步时序。SCL周期是通过时钟周期实现的。一个scl周期是30个时钟周期。使用div_cnt来记录时钟周期数，30个一次循环。以下只是时间单位声明，SCL的具体实现在下方↓。

SCL周期：




```
reg [7:0]div_cnt;                           //时钟计数器
wire scl_tick;
//计数，一个时钟周期div_cnt+1
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时间
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
```

状态更新

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

读写命令的判断

读写命令是通过端口输入write_op和read_op确定的，这两个信号是低电平有效，用了wr_op和rd_op两个寄存器把输入的write_op和read_op取反，这样如果有读或写命令，wr_op或rd_op为1，复位和操作结束(wr_opover)时都要清零，接下来就用wr_op和rd_op判断是否读写了，这里仅仅是把原来低电平有效的信号替换为两个高电平有效信号。

```
reg wr_op,rd_op;                    //读写操作

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;                    //等待
```

下一状态的判断

Byte Write : START + DEVICE +ACK + ADDR + ACK + DATA + ACK + STOP　　Random Read : START + DEVICE + ACK +ADDR + ACK+START + DEVICE + DATA + NO ACK + STOP

下一状态的更新是在scl_tick（每30个clk），上面下划线划出的部分是相同的，无论读写，一开始的状态更新都是按照上面下划线的顺序依次更新状态，这时候的器件地址都是10100000（write），到了ADDR的ACK，即W_AACK时会根据wr_op和rd_op决定接下来进入怎样的状态，wr_op会开始到状态W_DATA读数据；rd_op会到WAIT_WTICK3，然后继续START。操作结束后需要等待一个信号d5ms_over才能回到空闲，控制器件工作的频率不要太高。

```
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
```

```
//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

//ACK
W_DACK:begin next_i=W_DACK; if(scl_tick) next_i=S_STOP;end

//Current Address Read
//START: SCL=1,SDA=1->0(scl_lc)

WAIT_WTICK3:begin next_i=WAIT_WTICK3; if(scl_tick) next_i=R_START;end
R_START:begin next_i=R_START; if(scl_tick)next_i=R_DEVICE7;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
endcase
```

SCL同步实现

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，high start，clk15
```

SDA实现

代码后半都是SDA实现的部分，因为太长所以一段一段分析。

下面的一段是实现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，告诉SDA该怎么做，i2c_reg用来暂存scl上的数据，i2c_rlf是在读写数据时用的，使用的部分在下一段代码。

```
//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，主机发响应
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)|
                (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));
```

接下来就是根据控制信号进行操作，把对应操作的数据放到i2c_reg里，准备向SDA线上传输，i2c_rlf为1时i2creg会左移一位，因为sda是单位宽的，每次把i2creg的最高位送到sda上，因此左移就是一位一位把数据送到sda上。

```
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，sda输出
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
```

接下来是sda输出的控制，sda使能在主机写数据地址，数据时都是1，使能为1时器件靠sda输出i2creg的最高位。NOACK以外的其他响应信号，以及读的数据，都是从机发到sda线上的，这时主机的sda使能为0，不可以发出数据。

```
assign sda_o = i2c_reg[7];                          //sda
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
```

以下是读数据的操作，用sda_wr控制，在scl高电平的中心（数据稳定时）往read_data里读数据，左移补sda，就一位一位的用sda发的数据更新了read_data，这时sda是从机在发数据。

```
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
```

操作结束，等待

最后就是操作结束后的等待时间，d5ms_cnt用来记时钟周期，记满了就会把d5ms_over置1，在上面状态转换的部分，最后进入STOP状态等待的就是这个信号，等到了这个信号，I2C就回到空闲状态了。

```
//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);
```

最后，这个模块里没有ACK处理的部分，只是在应该收到ACK的时候从sda读信号出来，没有进行判断，？或许是交给主机来做的。