# 一、 软件平台与硬件平台

软件平台：

1、操作系统：Windows-8.1

2、开发套件：ISE14.7

3、仿真工具：ModelSim-10.4-SE

4、Matlab 版本：Matlab2014b/Matlab2016a

硬件平台：

1、 FPGA 型号：Xilinx 公司的 XC6SLX45-2CSG324

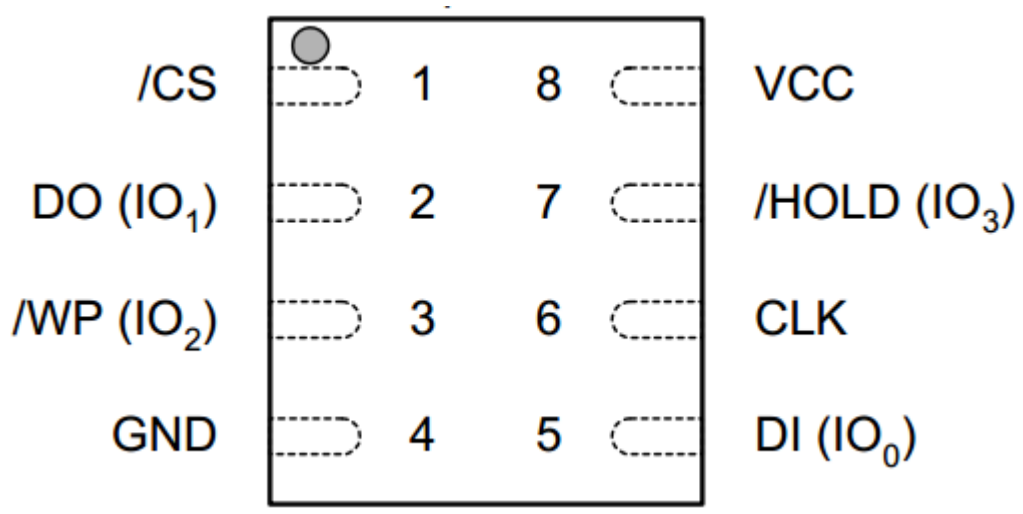
2、 Flash 型号：WinBond 公司的 W25Q128BV Quad SPI Flash 存储器

**提示：如果图片不清晰，请把图片在浏览器的新建标签页打开或保存到本地打开。**

# 二、 原理介绍

上一篇博客《SPI 总线的原理与 FPGA 实现》中已经有关于标准 SPI 协议的原理与时序的介绍，这里不再赘述。本节主要是讨论 QSPI(Quad SPI，四线 SPI 总线)的相关内容。我的开发板上有一片型号是 W25Q128BV 的 Quad SPI Flash 存储器，本文将以它为例子来说明 QSPI 操作的一些内容。

W25Q128BV 的 Quad SPI Flash 存储器的 Top View 如下图所示




这块芯片一共有 8 个有用的管脚，其每个管脚的功能定义如下图所示

| PAD NO. | PAD NAME | I/O | FUNCTION |
|---|---|---|---|
| 1 | /CS | I | Chip Select Input |
| 2 | DO (IO1) | I/O | Data Output (Data Input Output 1)*1 |
| 3 | /WP (IO2) | I/O | Write Protect Input ( Data Input Output 2)*2 |
| 4 | GND | | Ground |
| 5 | DI (IO0) | I/O | Data Input (Data Input Output 0)*1 |
| 6 | CLK | I | Serial Clock Input |
| 7 | /HOLD (IO3) | I/O | Hold Input (Data Input Output 3)*2 |
| 8 | VCC | | Power Supply |

*1: IO0 and IO1 are used for Standard and Dual SPI instructions

*2: IO0 – IO3 are used for Quad SPI instructions

由上图可知 2 号管脚 DO(IO1)，3 号管脚 /WP(IO2)，5 号管脚 DI(IO0)以及 7 号管脚/HOLD(IO3)均为双向 IO 口，所以在编写 Verilog 代码的时候要把它们定义为 inout 类型，inout 类型的信号既可以做输出也可以作为输入，具体在代码里面如何处理后文会有介绍。

QSPI Flash 每个引脚的详细描述如下：

1、Chip Select(/CS)

片选信号 Chip Select(/CS)的作用是使能或者不使能设备的操作，当 CS 为高时，表示设备未被选中，串行数据输出线(DO 或 IO0，IO1，IO2，IO3)均处于高阻态，当 CS 为低时，表示设备被选中，FPGA 可以给 QSPI Flash 发送数据或从 QSPI Flash 接收数据。

2、串行数据输入信号 DI 以及串行输出信号 DO

W25Q128BV支持标准SPI协议 ,双线SPI(Dual SPI)协议与四线SPI(Quad SPI)协议。标准的SPI协议在串行时钟信号(SCLK)的上升沿把串行输入信号DI上的数据存入QSPI Flash 中，在串行时钟信号(SCLK)的下降沿把 QSPI Flash 中的数据串行化通过单向的 DO 引脚输出。而在 Dual SPI 与 Quad SPI 中，DI 与 DO 均为双向信号(既可以作为输入，也可以作为输出)。

3、Write Project(/WP)

写保护信号的作用是防止 QSPI Flash 的状态寄存器被写入错误的数据，WP 信号低电平有效，但是当状态寄存器 2 的 QE 位被置 1 时，WP 信号失去写保护功能，它变成 Quad SPI 的一个双向数据传输信号。

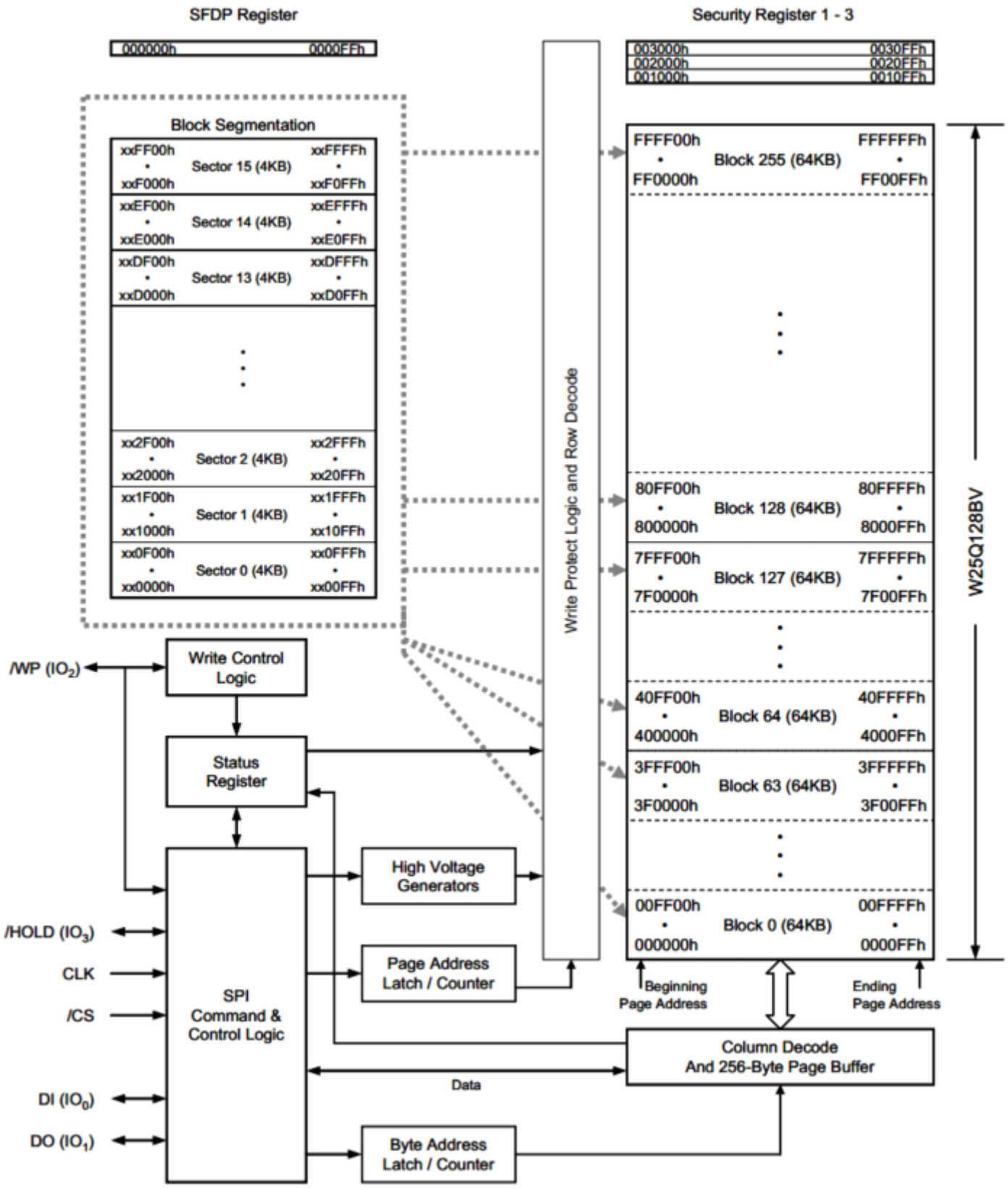
4、HOLD(/HOLD)

HOLD 信号的作用是暂停 QSPI Flash 的操作。当 HOLD 信号为低 ,并且 CS 也为低时 ,串行输出信号 DO 将处于高阻态 ,串行输入信号 DI 与串行时钟信号 SCLK 将被 QSPI Flash 忽略。当 HOLD 拉高以后，QSPI Flash 的读写操作能继续进行。当多个 SPI 设备共享同一组 SPI 总线相同的信号的时候，可以通过 HOLD 来切换信号的流向。和 WP 信号一样，当当状态寄存器 2 的 QE 位被置 1 时，HOLD 信号失去保持功能，它也变成 Quad SPI 的一个双向数据传输信号。

5、串行时钟线

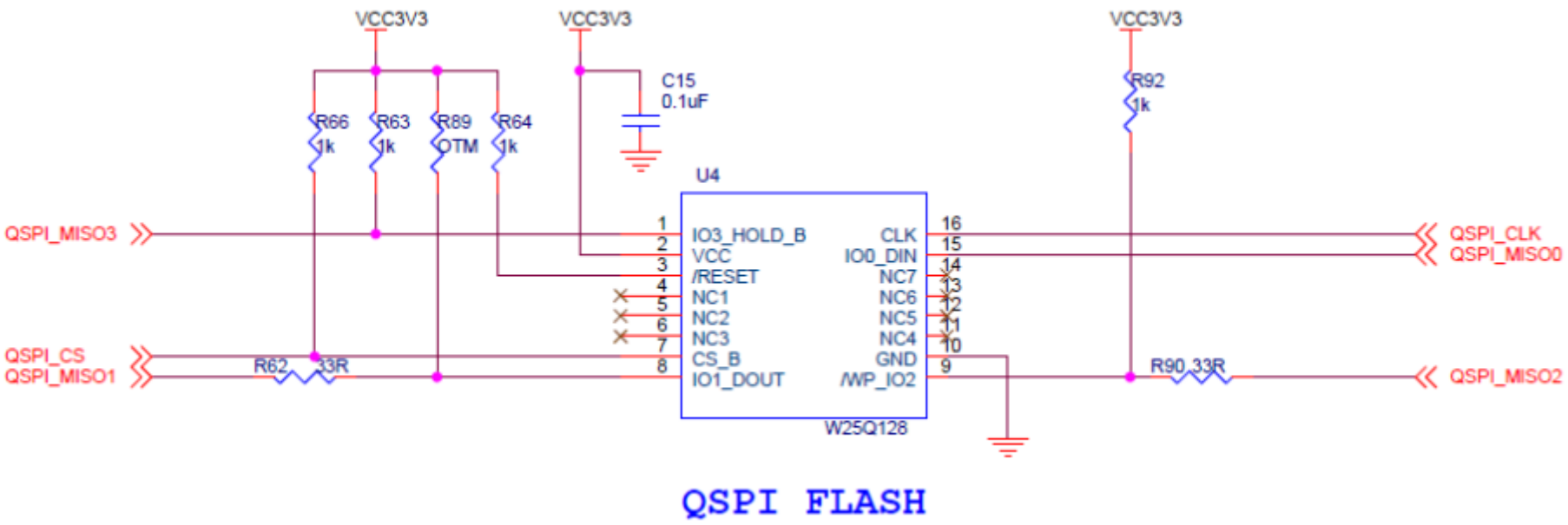
串行时钟线用来提供串行输入输出操作的时钟。

W25Q128BV 的内部结构框图如下图所示：




更多详细的内容请阅读 W25Q128BV 的芯片手册。由于本文要进行 4 线 SPI 的操作，但 QSPI Flash 默认的操作模式是标准单线 SPI 模式，所以在每次进行 4 线 SPI 操作的时候一定要先把状态寄存器 2 的 QE 位(倒数第 2 位)置 1，然后才能进行 QSPI 操作。

最后介绍一下我的开发板上 QSPI Flash 硬件原理图如下图所示：



# 三、 目标任务

1、编写标准 SPI 协议 Verilog 代码来操作 QSPI Flash，并用 ChipScope 抓出各个指令的时序与芯片手册提供的时序进行对比

2、在标准 SPI 协议的基础上增加 Quad SPI 的功能，并用 ChipScope 抓出 Quad SPI 的读写数据的时序

3、对比标准 SPI 与 Quad SPI 读写 W25Q128BV 的 ChipScope 时序，感受二者的效率差距

# 四、 设计思路与 Verilog 代码编写

## 4.1、 命令类型的定义

W25Q128BV 一共有 35 条命令，这里不可能把所有命令的逻辑都写出来，所以截取了一部分常用的命令作为示例来说明 QSPI Flash 的操作方法。由于命令数目很多，所以在这个部分先对各个命令类型做一个初步定义，下文的代码就是按照这个定义来编写的。

| 命令编号 | 命令类型(自定义) | 命令码(芯片手册定义) | 命令功能 |
|---|---|---|---|
| 1 | 5’b0XXXX | 8’h00 | 无 |
| 2 | 5’b10000 | 8’h90 | 读设备 ID |
| 3 | 5’b10001 | 8’h06 | 写使能 |
| 4 | 5’b10010 | 8’h20 | 扇区擦除 |
| 5 | 5’b10011 | 8’h05/8’h35 | 读状态寄存器 1/2 |
| 6 | 5’b10100 | 8’h04 | 关闭写使能 |
| 7 | 5’b10101 | 8’h02 | 写数据操作(单线模式) |
| 8 | 5’b10110 | 8’h01 | 写状态寄存器 |
| 9 | 5’b10111 | 8’h03 | 读数据操作(单线模式) |
| 10 | 5’b11000 | 8’h32 | 写数据操作(四线模式) |
| 11 | 5’b11001 | 8’h6b | 读数据操作(四线模式) |

说明：

1、命令类型是我自己随便定义的，可以随便修改。命令码是芯片手册上定义好，不能修改，更详细的内容请参考 W25Q128 芯片手册。

2、命令类型的最高位是使能位，只有当最高位为 1 时，命令才有效(在代码里面写的就是只有当最高位为 1 时才能进入 SPI 操作的状态机)。

3、进行四线读写操作之前，一定要把四线读写功能的使能位打开，方法是通过写状态寄存器命令把状态寄存器 2 的 QE 位(倒数第二位)置 1。

## 4.2、 如何用 Matlab 产生存放在 ROM 中的.coe 文件格式的数据

上一节设计了一个把存放在 ROM 中的数据用 SPI 总线发出来的例子 ,ROM 里面只存放了 10 个数据 ,所以可以直接把这 10 个数据填写到.coe 文件就可以了 ,由于 QSPI Flash 的页编程(写数据)指令最大支持 256 字节的写操作，所以下面的例子的功能是把 ROM 中存放的 256 个字节(8-bit)数据先写入 QSPI Flash 中，然后在读出来。由于数据太多(256 个),所以一个一个填写肯定不现实，所以可以利用 Matlab 来直接产生.coe 文件，Matlab 的完整代码如下：

```
width=8;    %rom 中数据的宽度
depth=256; %rom 的深度
y=0:255;
y=fliplr(y); %产生要发送的数据，255,254,253，.....  ，2,1,0
fid = fopen('test_data.coe', 'w'); %  打开一个.coe 文件%  存放在 ROM 中的.coe 文件第一行必须是这个字符串，16 表示 16 进制，可以改成其他进制
fprintf(fid,'memory_initialization_radix=16;\n');
%  存放在 ROM 中的.coe 文件第二行必须是这个字符串
fprintf(fid,'memory_initialization_vector=\n');
%  把前 255 个数据写入.coe 文件中，并用逗号隔开，为了方便知道数据的个数，每行只写一个数据
fprintf(fid,'%x,\n',y(1:end-1));
 %  把最后一个数据写入.coe 文件中，并用分号结尾
fprintf(fid,'%x;\n',y(end));
fclose(fid);   %  关闭文件指针
```

用 Matlab2014b 运行上面的代码以后会在与这个.m 文件相同的目录下产生一个.coe 文件，这个.coe 文件可以导入到 ROM 中。

## 4.3、 标准 SPI 总线操作 QSPI Flash 思路与代码编写

上一篇博客《SPI 总线的原理与 FPGA 实现》已经介绍过用 spi_module 这个模块去读取 QSPI Flash 的 Manufacturer/Device  ID，事实上除了上篇博客提供的那种方法以外，还可以直接在时钟信号的下降沿发送数据，时钟信号的上升沿来接受数据来完成读 ID 的操作，当 FPGA 在时钟的下降沿发送数据的时候，那么时钟的上升沿刚好在数据的正中间，QSPI Flash 刚好可以在这个上升沿把数据读进来，读操作则正好相反。但是有很多有经验的人告诉我在设计中如非必要最好不要使用时钟下降沿触发的设计方法，可能是因为大多数 FPGA 里面的 Flip Flops 资源都是上升沿触发的，如果在 Verilog 代码采用下降沿触发的话 ，综合的时候会在 CLK 输入信号前面综合出一个反相器，这个反相器可能会对时钟信号的质量有影响，具体的原因等我再 Google 上继续搜索一段时间在说。这个例子由于状态机相较前几篇博客来说相对复杂，所以接下来写代码我还是采用下降沿发送数据，上升沿接收数据的方式来描述这个状态机。

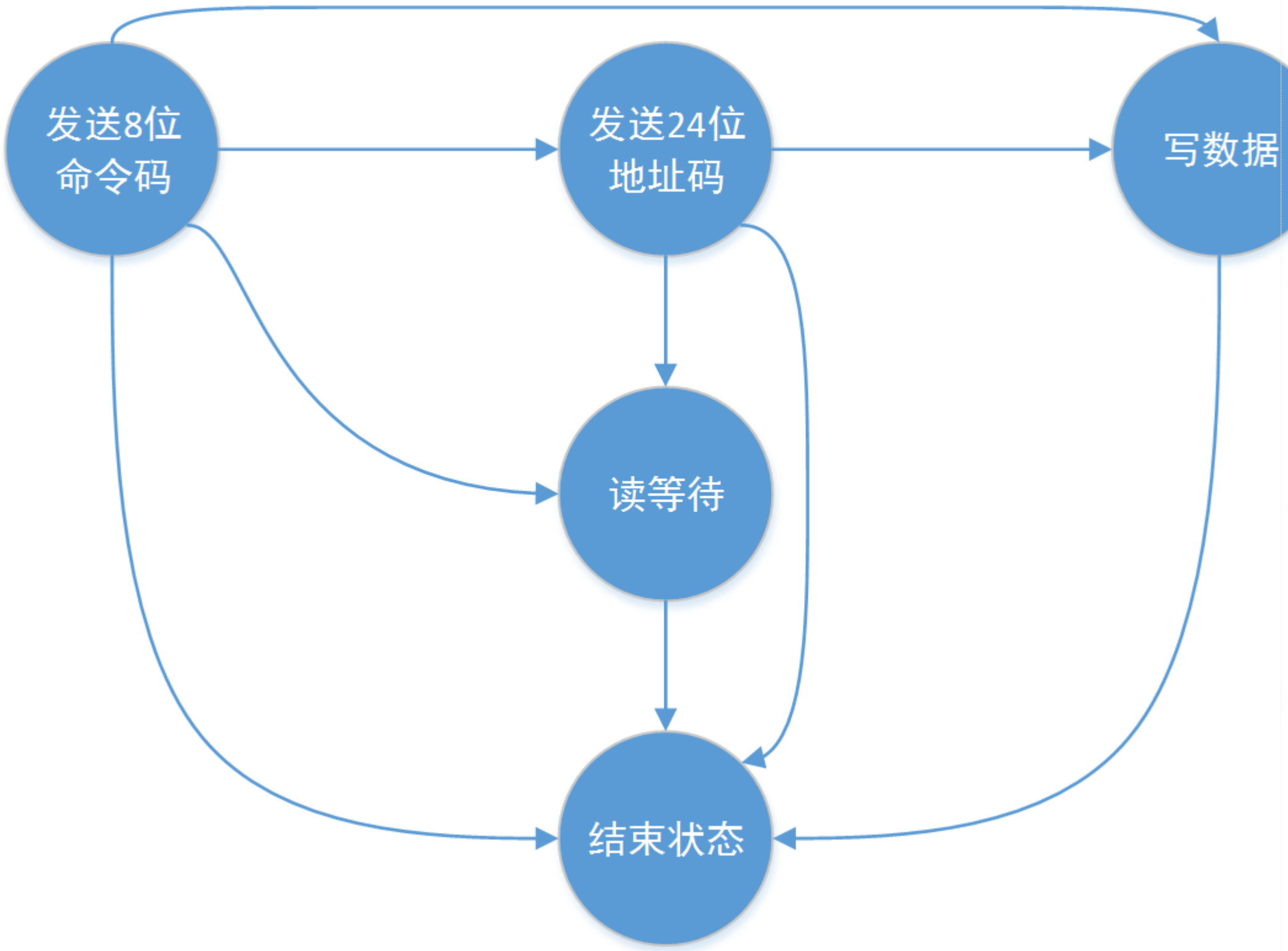
接下来的任务就是抽象出一个状态机。上一篇博客仅仅读一个 ID 就用了 6 个状态，所以采用上一篇博客的设计思路显然不太现实，但对于初学者而言，上一篇博客仍然有一个基本的指引作用。通过阅读 QSPI Flash 的芯片手册，可以发现，所有的命令其实至多由以下三个部分组成：

1、发送 8-bit 的命令码

2、发送 24-bit 的地址码

3、发送数据或接收数据

所有命令的状态跳变图可由下图描述




所以按照这个思路来思考的话抽象出来的状态机的状态并不多。单线模式的状态为以下几个：

1、空闲状态：用来初始化各个寄存器的值

2、发送命令状态：用来发送 8-bit 的命令码

3、发送地址状态：用来发送 24-bit 的地址码

4、读等待状态：当读数据操作正在进行的时候进入此状态等待读数据完毕

5、写数据状态(单线模式)：在这个状态 FPGA 往 QSPI Flash 里面写数据

6、结束状态：一条指令操作结束，并给出一个结束标志

完整的代码如下：

```
`timescale 1ns / 1ps

module qspi_driver
```

```
(output                 O_qspi_clk          , // SPI总线串行时钟线output reg          O_qspi_cs           , // SPI总线片
选信号 output reg           O_qspi_mosi         , // SPI 总线输出信号线，也是 QSPI Flash 的输入信号线 input
I_qspi_miso         , // SPI 总线输入信号线，也是 QSPI Flash 的输出信号线
                                    input               I_rst_n             , //  复位信号
input               I_clk_25M           , // 25MHz 时钟信号 input        [4:0]       I_cmd_type          , //  命令类型
input       [7:0]       I_cmd_code          , //  命令码 input        [23:0]      I_qspi_addr         , // QSPI Flash 地址
output reg          O_done_sig          , //  指令执行结束标志 output reg   [7:0]       O_read_data         , //  从 QSPI
Flash 读出的数据 output reg          O_read_byte_valid   , //  读一个字节完成的标志 output reg   [3:0]       O_qspi_state
//  状态机，用于在顶层调试用
);
```

```
parameter   C_IDLE          =   4'b0000  ; //  空闲状态 parameter   C_SEND_CMD      =   4'b0001  ; //  发送命令码
parameter   C_SEND_ADDR     =   4'b0010  ; //  发送地址码 parameter   C_READ_WAIT     =   4'b0011  ; //  读等待
parameter   C_WRITE_DATA    =   4'b0101  ; //  写数据 parameter   C_FINISH_DONE    =   4'b0110  ; //  一条指令执行结
束
reg         [7:0]   R_read_data_reg     ; //  从 Flash 中读出的数据用这个变量进行缓存，等读完了在把这个变量的值给输出 reg
R_qspi_clk_en       ; //  串行时钟使能信号 reg                R_data_come_single  ; //  单线操作读数据使能信号，当这个信号为高
时
            reg         [7:0]   R_cmd_reg           ; //  命令码寄存器 reg        [23:0]  R_address_reg       ; //  地址码寄存
器  reg         [7:0]   R_write_bits_cnt    ; //  写 bit 计数器，写数据之前把它初始化为 7，发送一个 bit 就减 1reg        [8:0]
R_write_bytes_cnt   ; //  写字节计数器，发送一个字节数据就把它加 1reg        [7:0]   R_read_bits_cnt     ; //  写 bit 计数器，接收
一个 bit 就加 1reg          [8:0]   R_read_bytes_cnt    ; //  读字节计数器，接收一个字节数据就把它加 1reg          [8:0]
R_read_bytes_num    ; //  要接收的数据总数 reg                R_read_finish       ; //  读数据结束标志位
wire        [7:0]   W_rom_addr          ;   wire        [7:0]   W_rom_out           ;
```

```
assign O_qspi_clk = R_qspi_clk_en ? I_clk_25M : 0   ; // 产生串行时钟信号assign W_rom_addr = R_write_bytes_cnt                ;
//////////////////////////////////////////////////////////////////////////////////////////////////////////////// 功能：用时钟的下降沿发送数据
////////////////////////////////////////////////////////////////////////////////////////////////////////////////always @(negedge I_clk_25M)begin
    if(!I_rst_n)
        begin
            O_qspi_cs           <=  1'b1    ;
            O_qspi_state        <=  C_IDLE ;
            R_cmd_reg           <=  0       ;
            R_address_reg       <=  0       ;
            R_qspi_clk_en       <=  1'b0    ;   //SPI clock 输出不使能
            R_write_bits_cnt    <=  0       ;
```

```
            R_write_bytes_cnt    <=   0        ;
            R_read_bytes_num     <=   0        ;
            R_address_reg        <=   0        ;
            O_done_sig           <=   1'b0     ;
            R_data_come_single   <=   1'b0     ;
        end
    else
        begin
            case(O_qspi_state)
                C_IDLE:   // 初始化各个寄存器，当检测到命令类型有效(命令类型的最高位位 1)以后,进入发送命令码状态
                    begin
                    R_qspi_clk_en   <=   1'b0          ;
                    O_qspi_cs       <=   1'b1          ;
                    O_qspi_mosi     <=   1'b0          ;
                    R_cmd_reg       <=   I_cmd_code    ;
                    R_address_reg   <=   I_qspi_addr   ;
                    O_done_sig      <=   1'b0          ;
                    if(I_cmd_type[4] == 1'b1)
                        begin                       //如果 flash 操作命令请求
                            O_qspi_state          <=   C_SEND_CMD   ;
                            R_write_bits_cnt      <=   7            ;
                            R_write_bytes_cnt     <=   0            ;
                            R_read_bytes_num      <=   0            ;
                        end
                    end
                C_SEND_CMD: // 发送 8-bit 命令码状态
                    begin
                        R_qspi_clk_en          <=   1'b1     ; // 打开 SPI 串行时钟 SCLK 的使能开关
                        O_qspi_cs              <=   1'b0     ; // 拉低片选信号 CS
                        if(R_write_bits_cnt > 0)
                            begin                           //如果 R_cmd_reg 还没有发送完
                                O_qspi_mosi          <=   R_cmd_reg[R_write_bits_cnt] ;          //发送 bit7~bit1 位
                                R_write_bits_cnt     <=   R_write_bits_cnt-1'b1        ;
                            end
                        else
```

```
begin                                   //发送 bit0
    O_qspi_mosi <=   R_cmd_reg[0]     ;
    if ((I_cmd_type[3:0] == 4'b0001) | (I_cmd_type[3:0] == 4'b0100))
        begin     //如果是写使能指令(Write Enable)或者写不使能指令(Write Disable)
            O_qspi_state     <=   C_FINISH_DONE    ;
        end
    else if (I_cmd_type[3:0] == 4'b0011)
        begin     //如果是读状态寄存器指令(Read Register)
            O_qspi_state          <=   C_READ_WAIT ;
            R_write_bits_cnt      <=   7              ;
            R_read_bytes_num      <=   1              ;//读状态寄存器指令需要接收一个数据
                                    end
                                else if( (I_cmd_type[3:0] == 4'b0010) || (I_cmd_type[3:0] == 4'b0101) || (I_cmd_type[3:0] ==
4'b0111) || (I_cmd_type[3:0] == 4'b0000) )
                                    begin // 如果是扇区擦除(Sector Erase),页编程指令(Page Program),读数据指令(Read Data),读设
备 ID 指令(Read Device ID)
                                        O_qspi_state          <=   C_SEND_ADDR ;
                                        R_write_bits_cnt      <=   23             ; // 这几条指令后面都需要跟一个 24-bit 的地址码
                                    end
                            end
                    end
            C_SEND_ADDR: // 发送地址状态
                begin
                    if(R_write_bits_cnt > 0)   //如果 R_cmd_reg 还没有发送完
                    begin
                        O_qspi_mosi          <=   R_address_reg[R_write_bits_cnt] ; //发送 bit23~bit1 位
                        R_write_bits_cnt     <=   R_write_bits_cnt      -    1          ;
                    end
                else
                    begin
                        O_qspi_mosi <=   R_address_reg[0]     ;    //发送 bit0
                        if(I_cmd_type[3:0] == 4'b0010) // 扇区擦除(Sector Erase)指令
                            begin   //扇区擦除(Sector Erase)指令发完 24-bit 地址码就执行结束了，所以直接跳到结束状态
                                O_qspi_state <= C_FINISH_DONE    ;
                            end
```

```
else if (I_cmd_type[3:0] == 4'b0101) // 页编程(Page Program)指令
    begin
        O_qspi_state          <=  C_WRITE_DATA     ;
        R_write_bits_cnt      <=  7                ;
    end
else if (I_cmd_type[3:0] == 4'b0000) // 读 Device ID 指令
    begin
        O_qspi_state          <=  C_READ_WAIT      ;
        R_read_bytes_num      <=  2                ; //接收 2 个数据的 Device ID
    end
else if (I_cmd_type[3:0] == 4'b0111) // 读数据(Read Data)指令
                    begin
                        O_qspi_state          <=  C_READ_WAIT      ;
                        R_read_bytes_num      <=  256              ;   //接收 256 个数据
                    end
                end
        end
    C_READ_WAIT: // 读等待状态
        begin
            if(R_read_finish)
                begin
                    O_qspi_state          <=  C_FINISH_DONE    ;
                    R_data_come_single    <=  1'b0             ;
                end
            else
                begin
                    R_data_come_single    <=  1'b1             ; // 单线模式下读数据标志信号 ,此信号为高标志正在接
收数据
                end
        end
                C_WRITE_DATA: // 写数据状态
        begin
            if(R_write_bytes_cnt < 256) // 往 QSPI Flash 中写入 256 个数据
                begin
                    if(R_write_bits_cnt > 0) //如果数据还没有发送完
```

```
                begin
                    O_qspi_mosi             <=   W_rom_out[R_write_bits_cnt] ; //发送 bit7~bit1 位
                    R_write_bits_cnt        <=   R_write_bits_cnt   - 1'b1      ;
                end
            else
                begin
                    O_qspi_mosi             <=   W_rom_out[0]                   ; //发送 bit0
                    R_write_bits_cnt        <=   7                              ;
                    R_write_bytes_cnt       <=   R_write_bytes_cnt + 1'b1       ;
                end
        end
            else
                begin
                    O_qspi_state        <=   C_FINISH_DONE    ;
                    R_qspi_clk_en       <=   1'b0             ;
                end
        end
    C_FINISH_DONE:
        begin
            O_qspi_cs               <=   1'b1     ;
            O_qspi_mosi             <=   1'b0     ;
            R_qspi_clk_en           <=   1'b0     ;
            O_done_sig              <=   1'b1     ;
            R_data_come_single      <=   1'b0     ;
                        O_qspi_state            <=   C_IDLE   ;
                    end
                default:O_qspi_state        <=   C_IDLE         ;
            endcase
        endend
////////////////////////////////////////////////////////////////////////////////// 功能：接收 QSPI Flash 发送过来的数据
////////////////////////////////////////////////////////////////////////////////always @(posedge I_clk_25M)begin
    if(!I_rst_n)
        begin
            R_read_bytes_cnt        <=   0          ;
            R_read_bits_cnt         <=   0          ;
```

```
            R_read_finish          <=   1'b0      ;
            O_read_byte_valid      <=   1'b0      ;
            R_read_data_reg        <=   0         ;
            O_read_data            <=   0         ;
        end
    else if(R_data_come_single)     //  此信号为高表示接收数据从 QSPI Flash 发过来的数据
        begin
            if(R_read_bytes_cnt < R_read_bytes_num)
                begin
                    if(R_read_bits_cnt < 7)   //接收一个 Byte 的 bit0~bit6
                        begin
                            O_read_byte_valid     <=   1'b0                                  ;
                            R_read_data_reg       <=   {R_read_data_reg[6:0],I_qspi_miso} ;
                            R_read_bits_cnt       <=   R_read_bits_cnt +     1'b1             ;
                        end
                    else
                        begin
                            O_read_byte_valid     <=   1'b1                                  ;   //一个 byte 数据有效
                            O_read_data           <=   {R_read_data_reg[6:0],I_qspi_miso} ;   //接收 bit7
                            R_read_bits_cnt       <=   0                                     ;
                            R_read_bytes_cnt      <=   R_read_bytes_cnt      +     1'b1        ;
                        end
                end
            else
                begin
                    R_read_bytes_cnt      <=   0         ;
                    R_read_finish         <=   1'b1      ;
                    O_read_byte_valid     <=   1'b0      ;
                end
        end
    else
        begin
            R_read_bytes_cnt      <=   0         ;
            R_read_bits_cnt       <=   0         ;
            R_read_finish         <=   1'b0      ;
```

```
                O_read_byte_valid     <=   1'b0      ;
                R_read_data_reg       <=   0         ;
            endend


rom_data rom_data_inst (
    .clka(I_clk_25M), // input clka
    .addra(W_rom_addr), // input [7 : 0] addra
    .douta(W_rom_out) // output [7 : 0] douta
);
endmodule
```

接下来就是写一个测试代码对这个单线模式 SPI 驱动，为了保证把上面的所有指令都测试一遍，测试代码如下：

```
module qspi_top
(
    input           I_clk           ,
    input           I_rst_n         ,

    output          O_qspi_clk      , // SPI 总线串行时钟线
    output          O_qspi_cs       , // SPI 总线片选信号
    output          O_qspi_mosi     , // SPI 总线输出信号线，也是 QSPI Flash 的输入信号线
    input           I_qspi_miso       // SPI 总线输入信号线，也是 QSPI Flash 的输出信号线
);
      reg [3:0]    R_state                ;reg [7:0]    R_flash_cmd            ;reg [23:0]   R_flash_addr           ;reg
R_clk_25M              ;reg [4:0]    R_cmd_type             ;
                                wire          W_done_sig             ;wire [7:0]   W_read_data            ;wire
W_read_byte_valid    ;wire [2:0]   R_qspi_state           ;

//////////////////////////////////////////////////////////////////////////////////          //功能：二分频逻辑
//////////////////////////////////////////////////////////////////////////////////          always @(posedge I_clk or negedge I_rst_n)begin
    if(!I_rst_n)
        R_clk_25M    <=   1'b0          ;
```

```
        else
            R_clk_25M     <=    ~R_clk_25M   ;end//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////功能：测试状态机
////////////////////////////////////////////////////////////////////////////////always @(posedge R_clk_25M or negedge I_rst_n)begin
        if(!I_rst_n)
            begin
                R_state             <=   4'd0           ;
                R_flash_addr        <=   24'd0          ;
                R_flash_cmd         <=   8'h00          ;
                R_cmd_type          <=   5'b0_0000     ;
            end
        else
            begin
                case(R_state)
                    4'd0://读 Device ID 指令
                        begin
                            if(W_done_sig)
                                begin
                                    R_flash_cmd <= 8'h00                ;
                                    R_state        <= R_state + 1'b1    ;
                                    R_cmd_type    <= 5'b0_0000           ;
                                end
                            else
                                begin
                                    R_flash_cmd    <= 8'h90             ;
                                    R_flash_addr <= 24'd0               ;
                                    R_cmd_type     <= 5'b1_0000         ;
                                end
                        end
                4'd1://写 Write disable instruction
                    begin
                        if(W_done_sig)
                            begin
                                R_flash_cmd <= 8'h00                ;
                                R_state        <= R_state + 1'b1    ;
```

```
                R_cmd_type    <= 5'b0_0000          ;
            end
        else
            begin
                R_flash_cmd <= 8'h04              ;
                R_cmd_type    <= 5'b1_0100          ;
            end
    end
4'd2://写使能(Write Enable)指令
    begin
        if(W_done_sig)
            begin
                R_flash_cmd <= 8'h00              ;
                R_state       <= R_state + 1'b1    ;
                R_cmd_type    <= 5'b0_0000          ;
            end
        else
            begin
                R_flash_cmd <= 8'h06              ;
                R_cmd_type    <= 5'b1_0001          ;
            end
    end
4'd3:// 扇区擦除(Sector Erase)指令
    begin
        if(W_done_sig)
            begin
                R_flash_cmd <= 8'h00              ;
                R_state       <= R_state + 1'b1    ;
                R_cmd_type    <= 5'b0_0000          ;
            end
        else
            begin
                R_flash_cmd <= 8'h20              ;
                R_flash_addr<= 24'd0              ;
                R_cmd_type    <= 5'b1_0010          ;
```

```
                end
        end

    4'd4://读状态寄存器 1，当 Busy 位(状态寄存器 1 的最低位)为 0 时表示擦除操作完成
        begin
            if(W_done_sig)
                begin
                    if(W_read_data[0]==1'b0)
                        begin
                            R_flash_cmd <= 8'h00                ;
                            R_state     <= R_state + 1'b1       ;
                            R_cmd_type  <= 5'b0_0000            ;
                        end
                    else
                        begin
                            R_flash_cmd <= 8'h05            ;
                            R_cmd_type  <= 5'b1_0011        ;
                        end
                end
            else
                begin
                    R_flash_cmd <= 8'h05            ;
                    R_cmd_type  <= 5'b1_0011        ;
                end
        end
    4'd5://写使能(Write Enable)指令
        begin
            if(W_done_sig)
                begin
                    R_flash_cmd <= 8'h00                ;
                    R_state     <= R_state + 1'b1       ;
                    R_cmd_type  <= 5'b0_0000            ;
                end
            else
                begin
```

```
                R_flash_cmd <= 8'h06                ;
                R_cmd_type   <= 5'b1_0001            ;
            end
        end
    4'd6: //页编程操作(Page Program): 把存放在 ROM 中的数据写入 QSPI Flash 中
        begin
            if(W_done_sig)
                begin
                R_flash_cmd <= 8'h00                ;
                R_state       <= R_state + 1'b1     ;
                R_cmd_type   <= 5'b0_0000            ;
                end
            else
                begin
                R_flash_cmd <= 8'h02                ;
                R_flash_addr<= 24'd0                ;
                R_cmd_type   <= 5'b1_0101            ;
                end
        end
    4'd7://读状态寄存器 1，当 Busy 位(状态寄存器 1 的最低位)为 0 时表示写操作完成
        begin
            if(W_done_sig)
                begin
                if(W_read_data[0]==1'b0)
                    begin
                    R_flash_cmd <= 8'h00            ;
                    R_state       <= R_state + 1'b1     ;
                    R_cmd_type   <= 5'b0_0000            ;
                    end
                else
                    begin
                    R_flash_cmd <= 8'h05            ;
                    R_cmd_type   <= 5'b1_0011      ;
                    end
            end
```

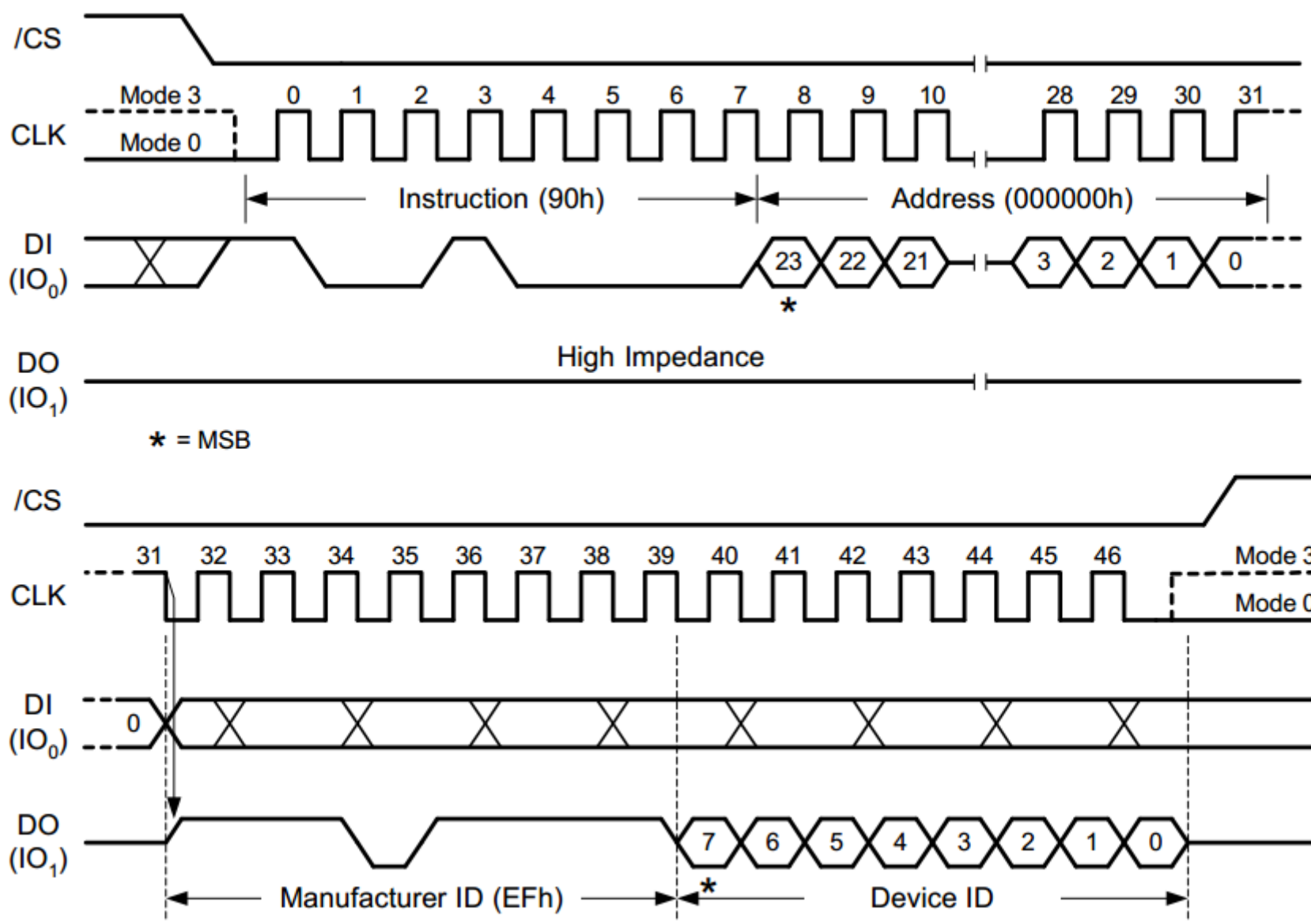
```
                else
                    begin
                        R_flash_cmd <= 8'h05            ;
                        R_cmd_type    <= 5'b1_0011      ;
                    end
            end
        4'd8://读 256 Bytes
            begin
                if(W_done_sig)
                    begin
                        R_flash_cmd <= 8'h00                ;
                        R_state       <= R_state + 1'b1     ;
                        R_cmd_type    <= 5'b0_0000          ;
                    end
                else
                    begin
                        R_flash_cmd <= 8'h03                ;
                        R_flash_addr<= 24'd0                ;
                        R_cmd_type    <= 5'b1_0111          ;
                    end
            end

        4'd9:// 空闲状态
            begin
                            R_flash_cmd <= 8'h00                ;
                            R_state       <= 4'd9               ;
                            R_cmd_type    <= 5'b0_0000          ;
                        end
                    default :     R_state       <= 4'd0             ;
                endcase
            end                 end
qspi_driver U_qspi_driver
(
.O_qspi_clk                 (O_qspi_clk             ), // SPI 总线串行时钟线
.O_qspi_cs                  (O_qspi_cs              ), // SPI 总线片选信号
```

```
.O_qspi_mosi            (O_qspi_mosi        ), // SPI 总线输出信号线，也是 QSPI Flash 的输入信号线
.I_qspi_miso            (I_qspi_miso        ), // SPI 总线输入信号线，也是 QSPI Flash 的输出信号线

.I_rst_n                (I_rst_n            ), //  复位信号

.I_clk_25M              (R_clk_25M          ), // 25MHz 时钟信号
.I_cmd_type             (R_cmd_type         ), //  命令类型
.I_cmd_code             (R_flash_cmd        ), //  命令码
.I_qspi_addr            (R_flash_addr       ), // QSPI Flash 地址

.O_done_sig             (W_done_sig         ), //  指令执行结束标志
.O_read_data            (W_read_data        ), //  从 QSPI Flash 读出的数据
.O_read_byte_valid      (W_read_byte_valid ), //  读一个字节完成的标志
.O_qspi_state           (R_qspi_state       )   //  状态机，用于在顶层调试用
);
      wire [35:0]       CONTROL0      ;wire [69:0]       TRIG0          ;
icon icon_inst (
    .CONTROL0(CONTROL0) // INOUT BUS [35:0]
);

ila ila_inst (
    .CONTROL(CONTROL0)    , // INOUT BUS [35:0]
    .CLK(I_clk)                 ,         // IN
    .TRIG0(TRIG0)         // IN BUS [255:0]
);
assign  TRIG0[7:0]      =   W_read_data         ;                                   assign  TRIG0[8]        =
W_read_byte_valid   ;   assign  TRIG0[12:9]     =   R_state             ;       assign  TRIG0[16:13]    =
R_qspi_state        ;   assign  TRIG0[17]       =   W_done_sig          ;   assign  TRIG0[18]       =
I_qspi_miso         ;   assign  TRIG0[19]       =   O_qspi_mosi         ;   assign  TRIG0[20]       =
O_qspi_cs           ;   assign  TRIG0[21]       =   O_qspi_clk          ; assign  TRIG0[26:22]    =   R_cmd_type          ;
assign  TRIG0[34:27]    =   R_flash_cmd         ; assign  TRIG0[58:35]    =   R_flash_addr        ; assign  TRIG0[59]
=   I_rst_n             ;

endmodule
```

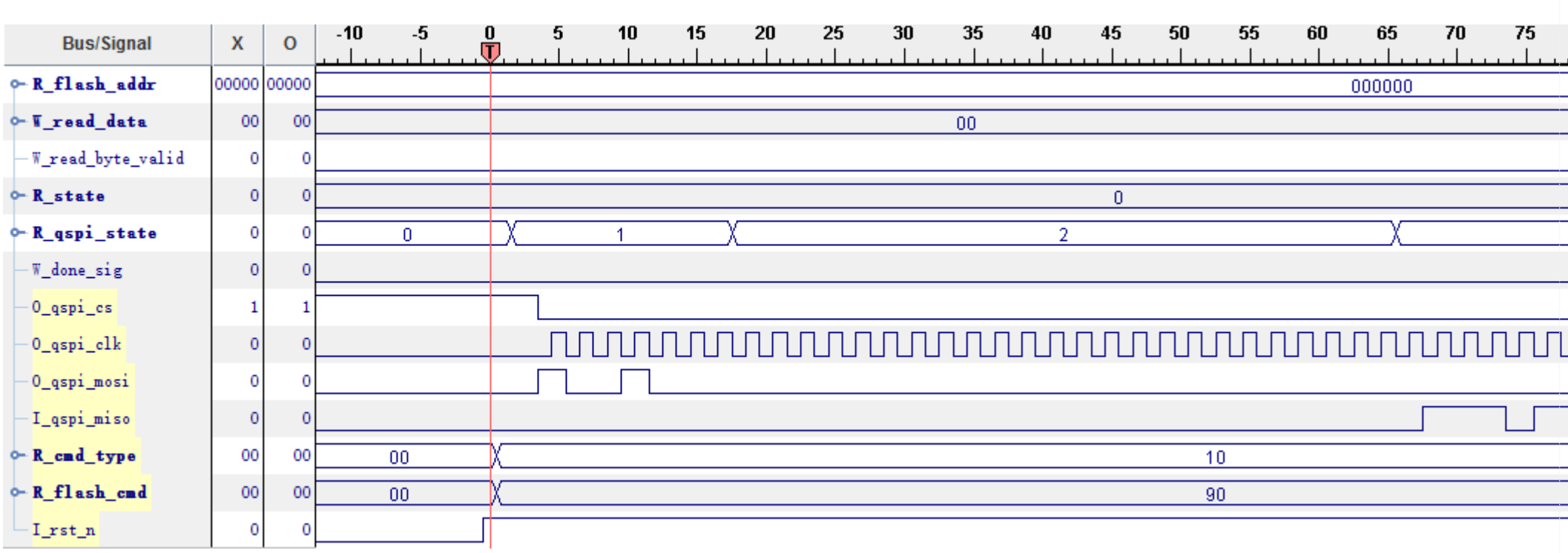
接下来主要看看用 ChipScope 抓出来的时序图和芯片手册规定的时序图是否完全一致。

1、读 ID 指令

芯片手册指令的读 ID 指令时序图：

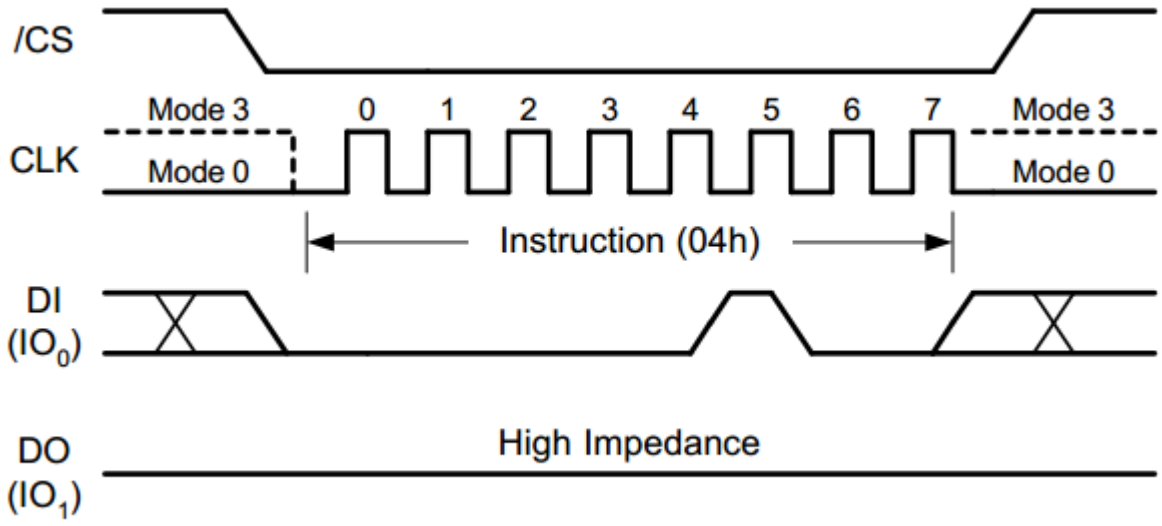



ChipScope 抓出来的时序图

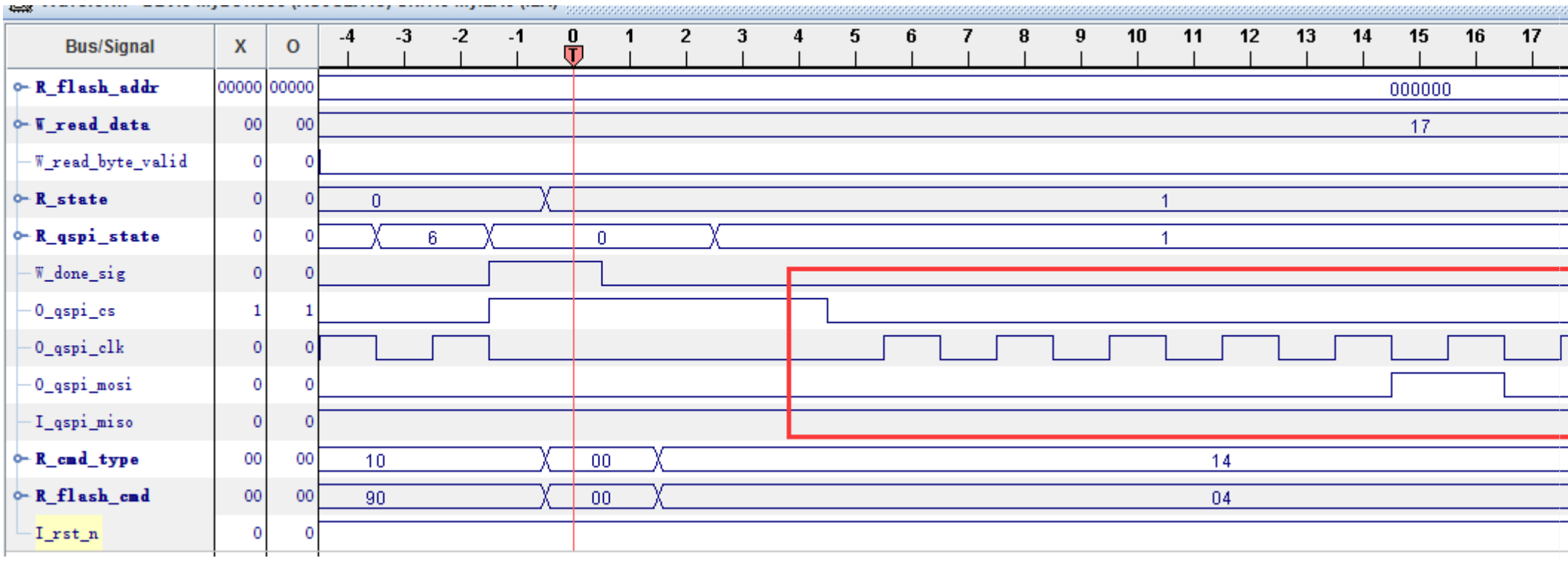



2、写不使能(Write Disable)指令

芯片手册指令的写不使能时序图：

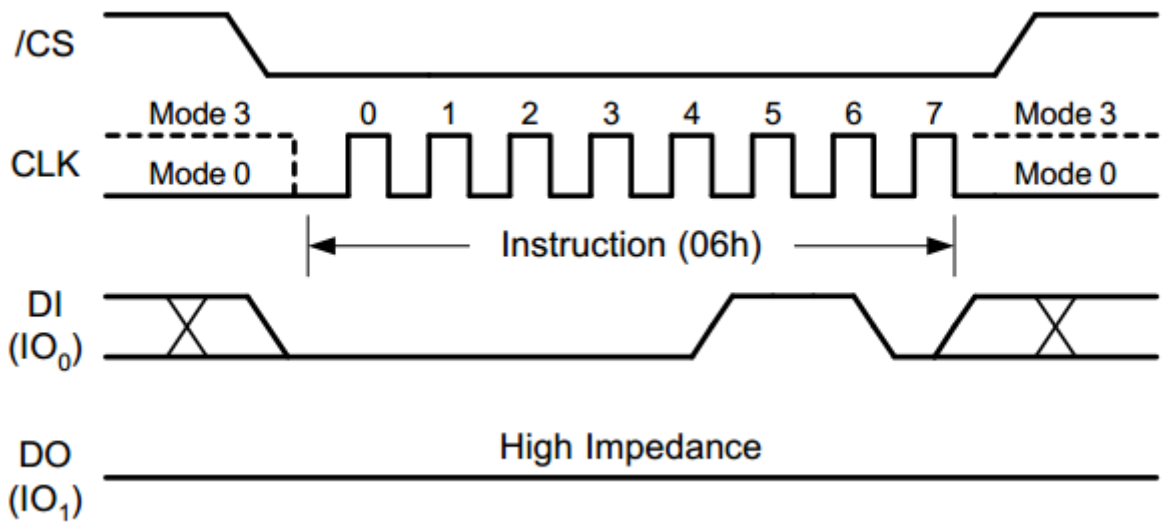



ChipScope 抓出来的写不使能时序图：

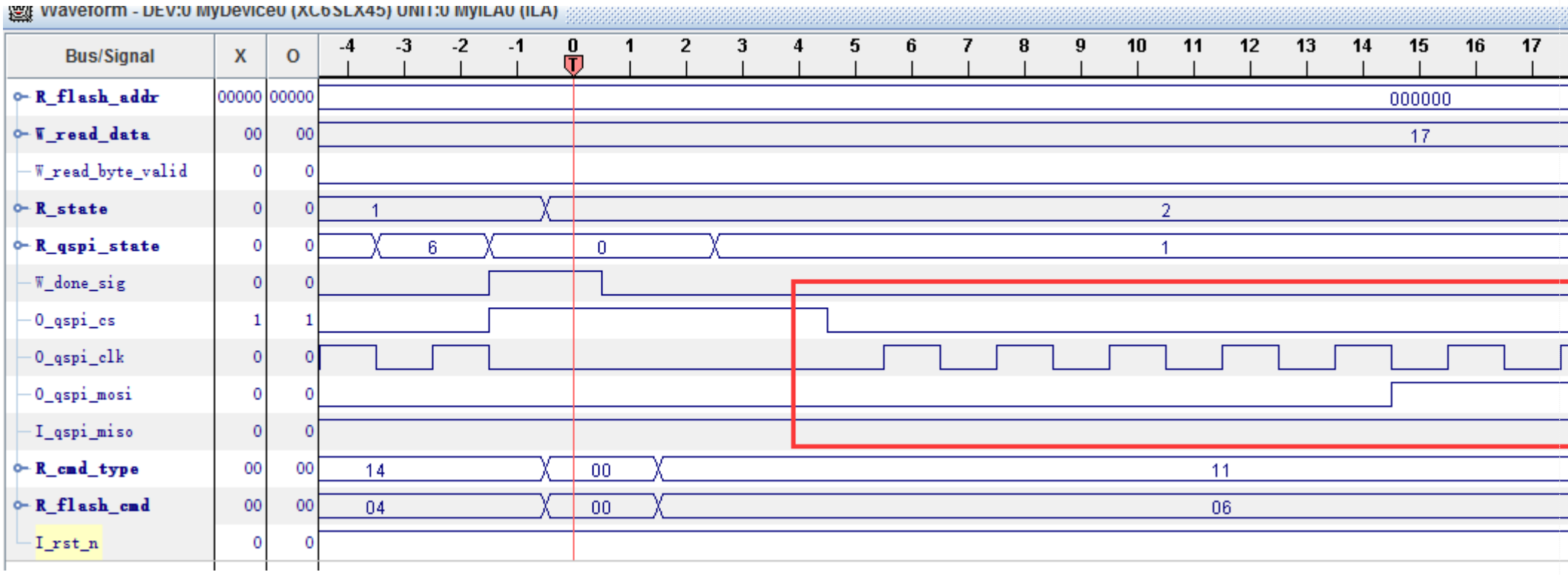



3、写使能(Write Enable)指令

芯片手册指令的写使能时序图：

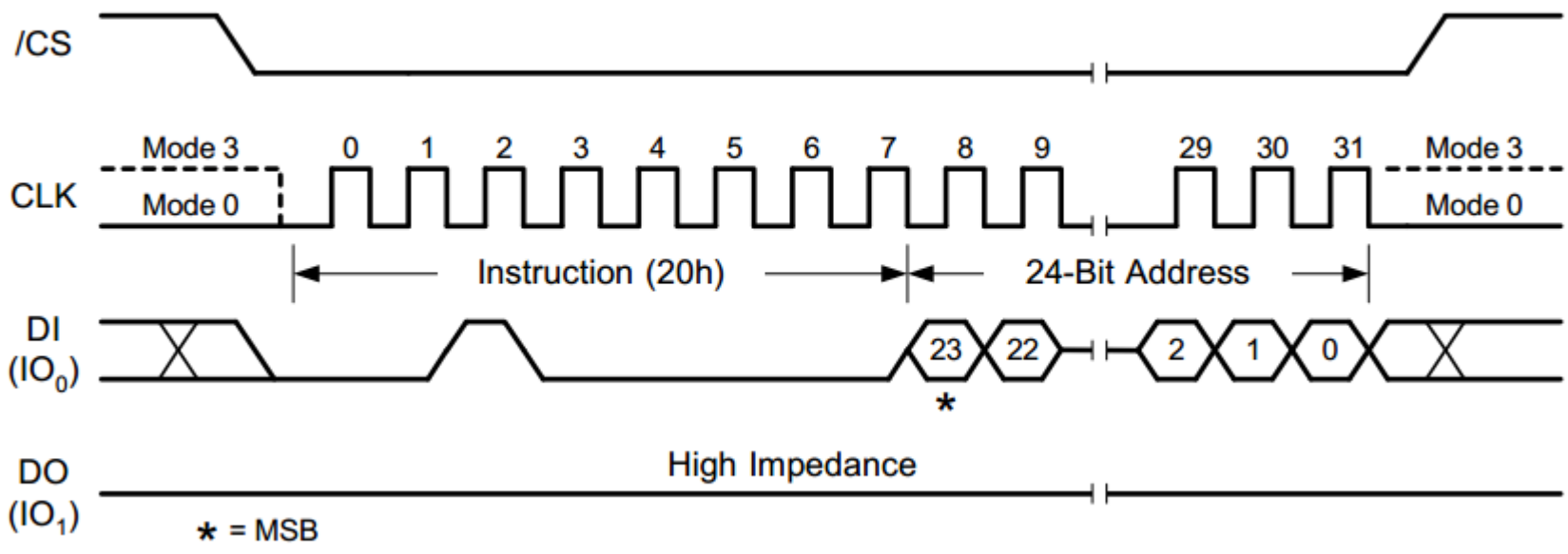



ChipScope 抓出来的写使能时序图

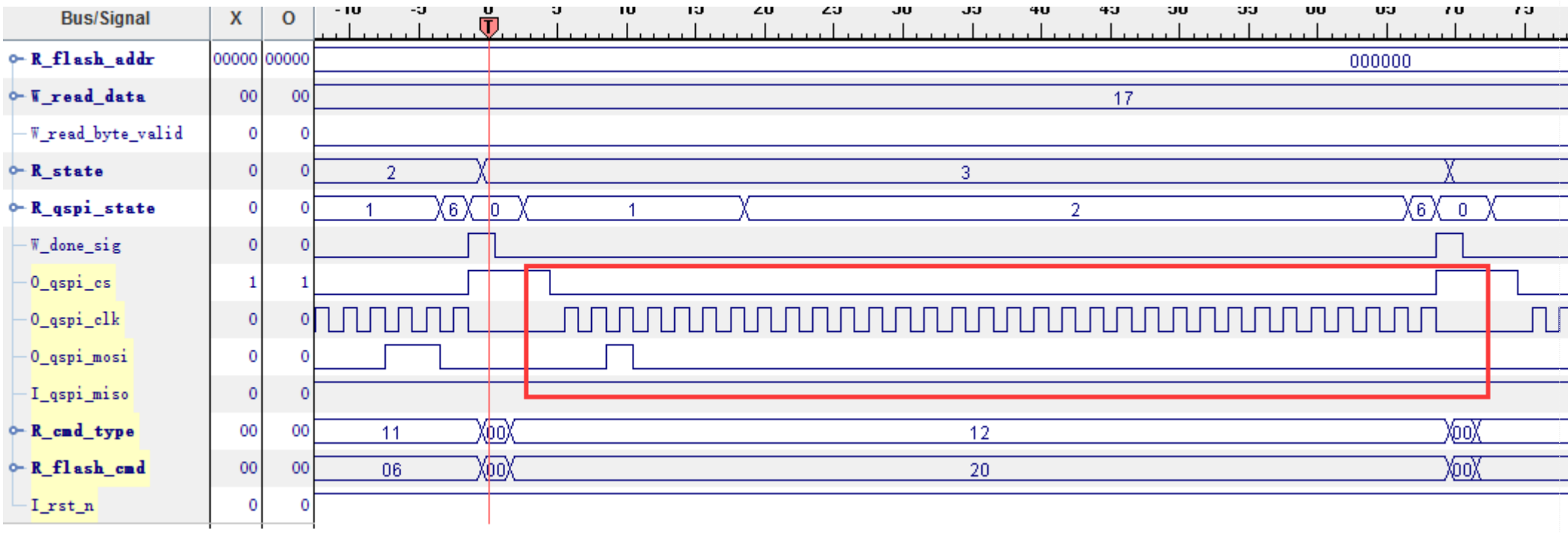



3、扇区擦除(Sector Erase)指令

芯片手册的扇区擦除指令时序图：

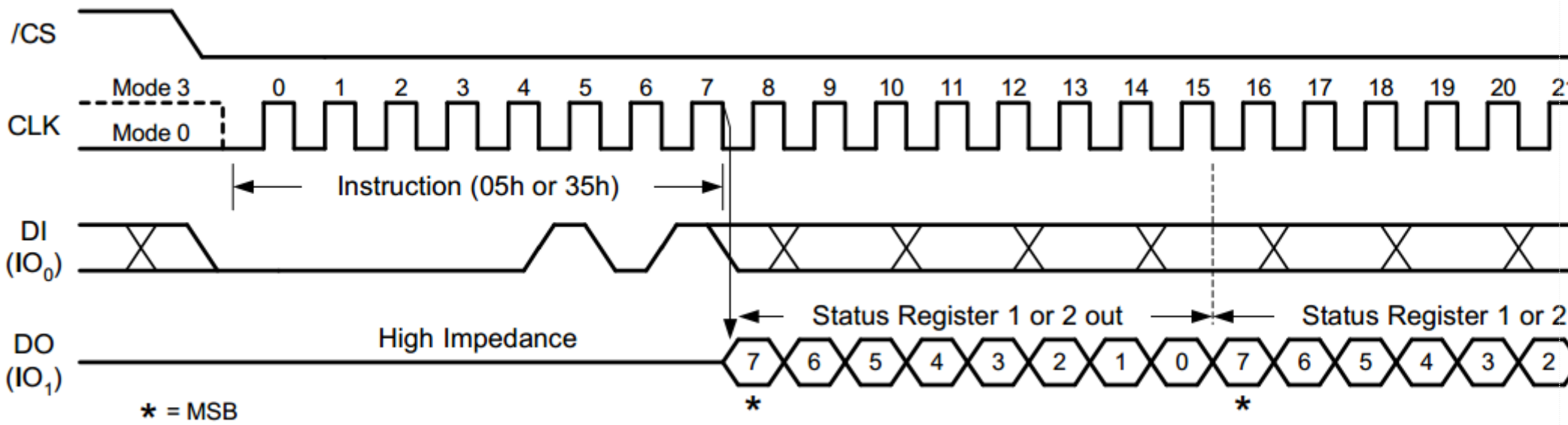



芯片手册的扇区擦除指令时序图：

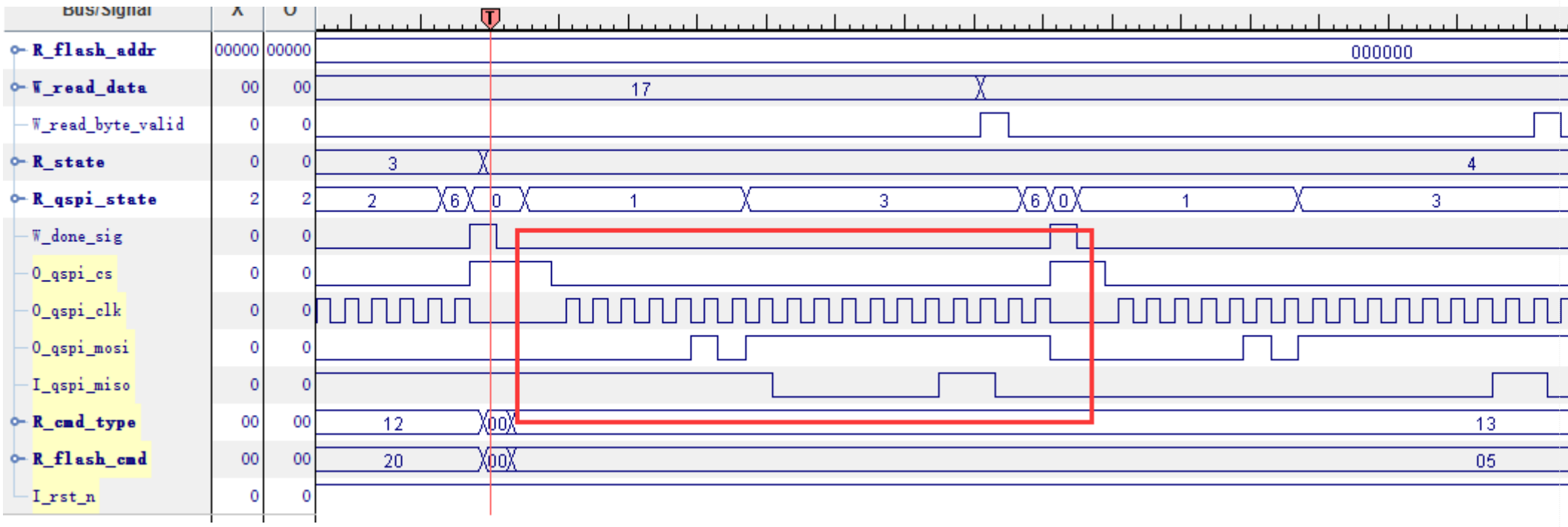



4、读状态寄存器(Read Status Register)指令

芯片手册的读状态寄存器指令时序图：




ChipScope 抓回来的读状态寄存器指令时序图：




由于在擦除操作和写操作(Page Program)操作指令发送完毕以后，芯片内部会自己执行相关的操作并把状态寄存器 1 中的最低位 Busy 位拉高，状态寄存器 1 的各位如下：