Thanh Tien Truong

Portland State University

Interface OV7670 Camera with FPGA

Table of Contents

REFERENCE:

http://www.electronicaestudio.com/docs/sht001.pdf (OV7670 datasheet)

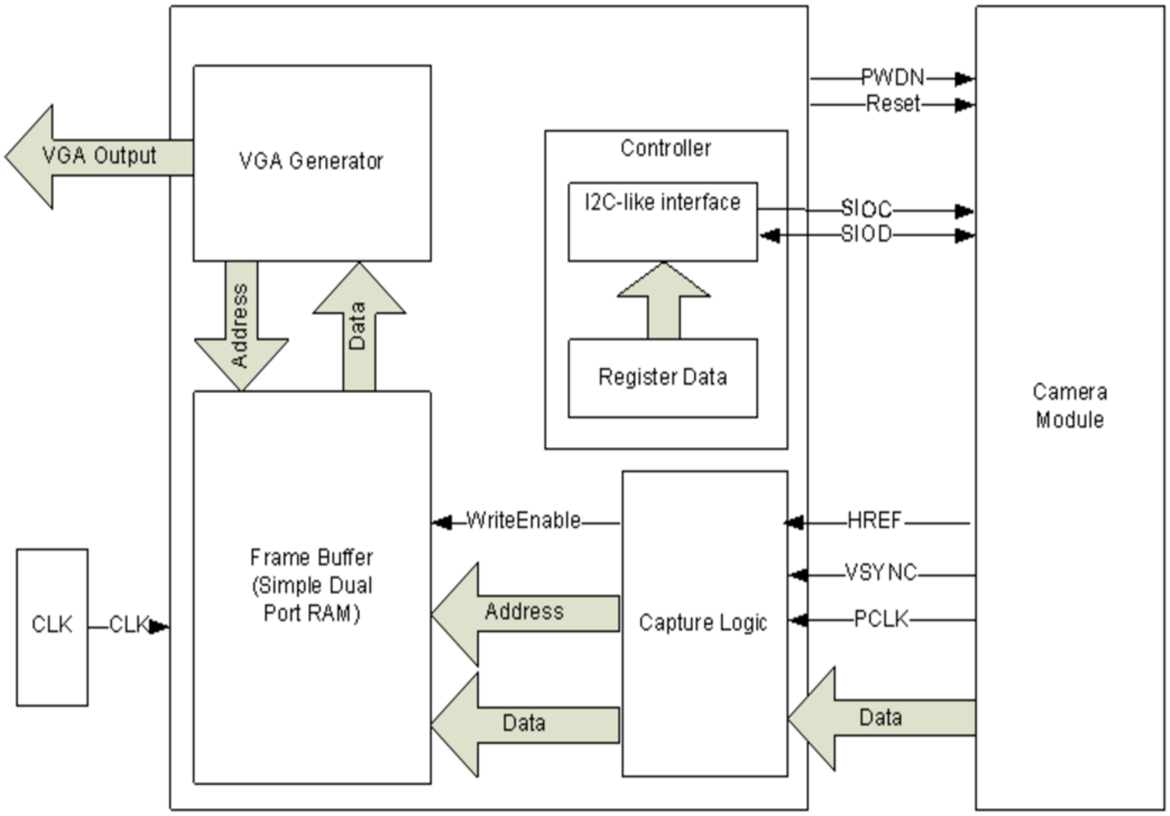
http://web.mit.edu/6.111/www/f2015/tools/OV7670app.pdf (OV7670 implementation guide)

http://www4.cs.umanitoba.ca/~jacky/Robotics/DataSheets/ov-sccb.pdf (SCCB interface)

I. Introduction

The OV7670 camera is a low voltage CMOS image sensor that provides the full functionality of a single-chip VGA camera and image processor in a small footprint package. The OV7670 camera is controlled through the Serial Camera Control Bus (SCCB) interface, supports Digital Video Port (DVP) to improve the speed of transferring data and supports multiple RGB, YUV and image scaling configurations. In this project, a Nexys4 FPGA board will be used to control the OV7670. The image data will be received and display on a VGA monitor.

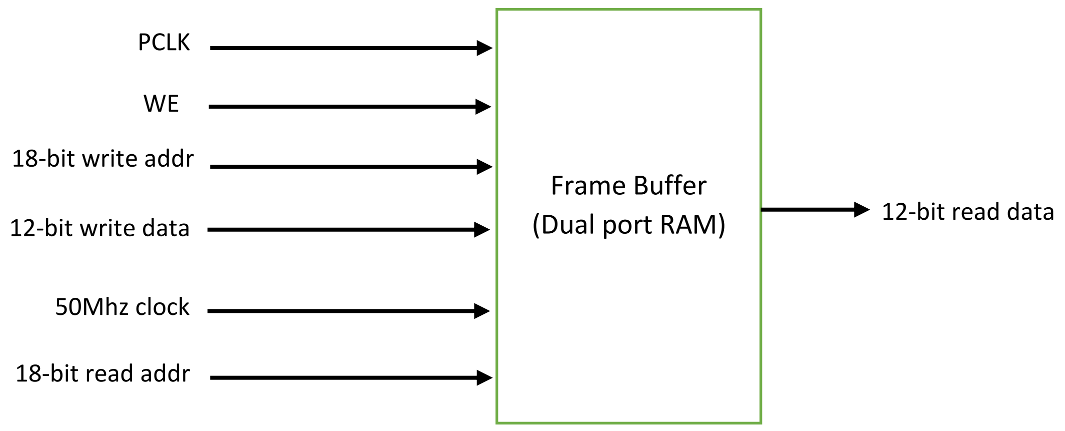
II. Block diagram



III. Implementation

3.1. Frame Buffer

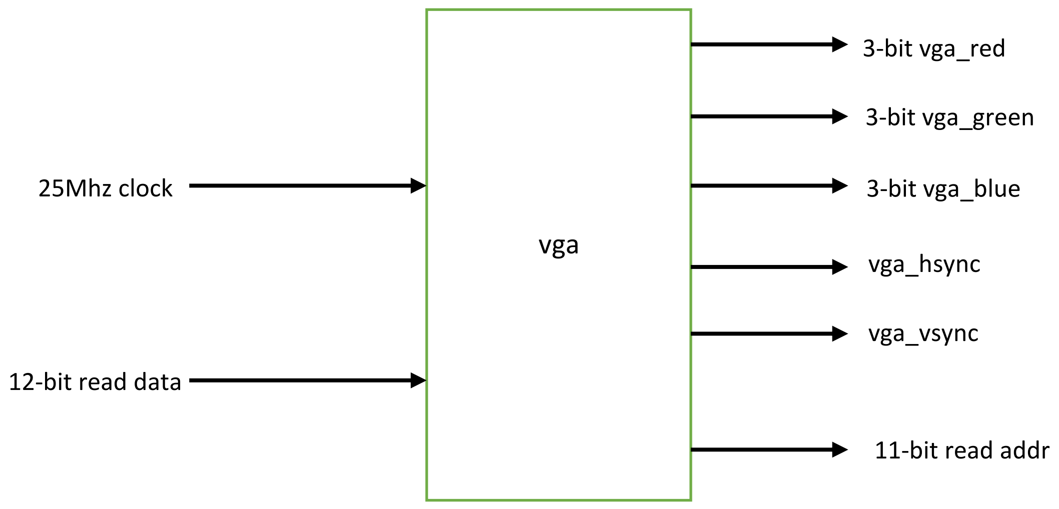
A frame buffer is simply a dual port RAM used to write and read data. The write and read transactions are controlled by 2 different clock signals. The reason that the frame buffer is required is to make everything synchronized. In fact, the data generated by the OV7670 is based on the PCLK frequency and other modules might require different frequency to read and analyze the data.



3.2. VGA Generator

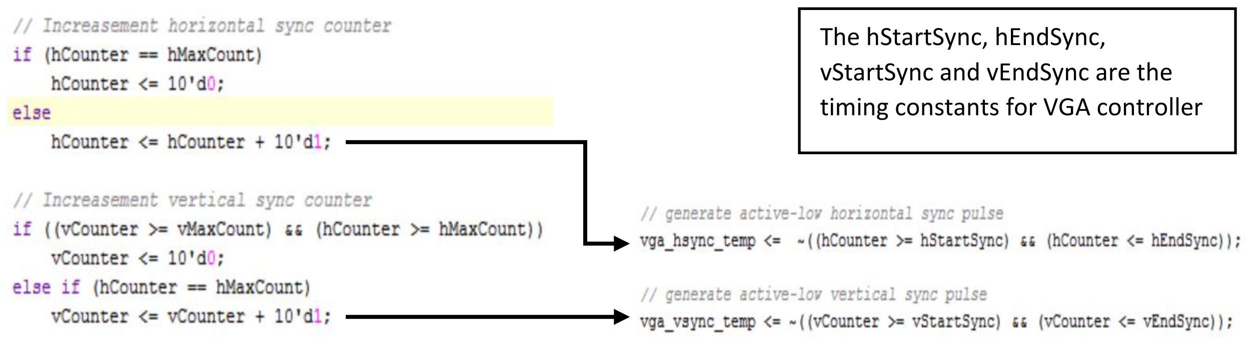
3.2.1. Description

The VGA generator is responsible for generating the horizontal sync pulse, vertical sync pulse for the VGA monitor and deciding the area on VGA monitor that the images will be displayed by providing the RGB values of each pixel. The RGB value of a pixel will be retrieved from the frame buffer or black.



3.2.2. Code explanation:

In order to generate the horizontal sync pulse and vertical sync pulse, there must be counters for keeping track of the current row pixel and the current column pixel to see if the current pixels is in the visible area

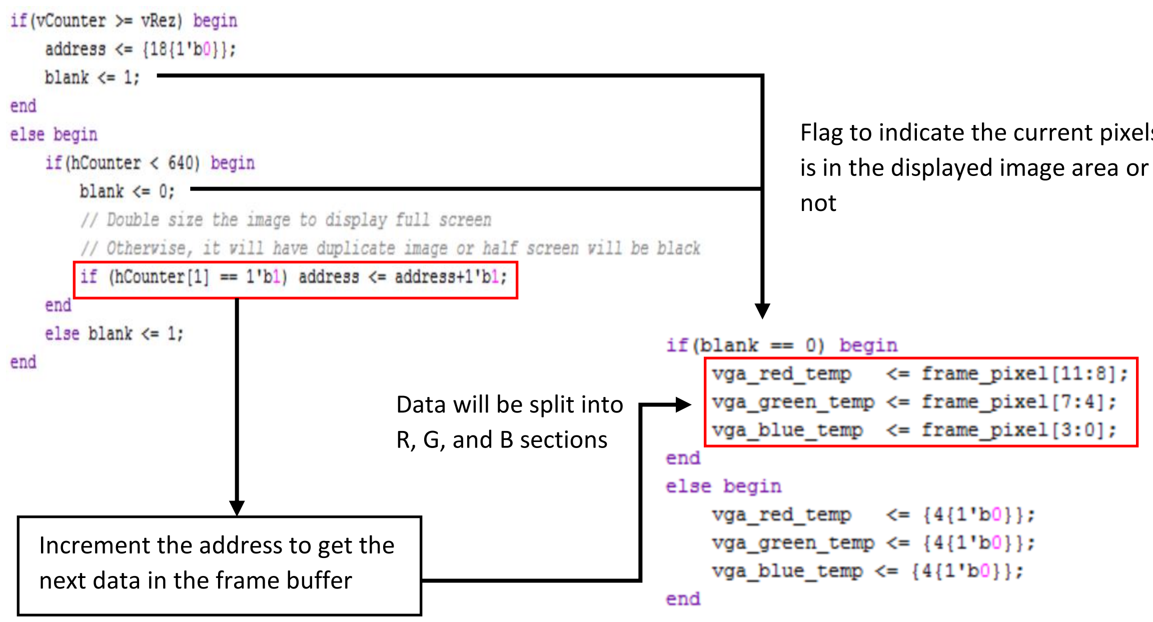


```
// Increasement horizontal sync counter
if (hCounter == hMaxCount)
    hCounter <= 10'd0;
else
    hCounter <= hCounter + 10'd1;

// Increasement vertical sync counter
if ((vCounter >= vMaxCount) && (hCounter >= hMaxCount))
    vCounter <= 10'd0;
else if (hCounter == hMaxCount)
    vCounter <= vCounter + 10'd1;
```

```
// generate active-low horizontal sync pulse
vga_hsync_temp <=  ~((hCounter >= hStartSync) && (hCounter <= hEndSync));

// generate active-low vertical sync pulse
vga_vsync_temp <= ~((vCounter >= vStartSync) && (vCounter <= vEndSync));
```

The hStartSync, hEndSync, vStartSync and vEndSync are the timing constants for VGA controller

In order to generating the RGB value, there must be a mechanism to keep track of the current row pixel and the current column pixel to see if the current pixels is in the area that we want the image to be displayed



```
if(vCounter >= vRez) begin
    address <= {18{1'b0}};
    blank <= 1;
end
else begin
    if(hCounter < 640) begin
        blank <= 0;
        // Double size the image to display full screen
        // Otherwise, it will have duplicate image or half screen will be black
        if (hCounter[1] == 1'b1) address <= address+1'b1;
    end
    else blank <= 1;
end
```

```
if(blank == 0) begin
    vga_red_temp   <= frame_pixel[11:8];
    vga_green_temp <= frame_pixel[7:4];
    vga_blue_temp  <= frame_pixel[3:0];
end
else begin
    vga_red_temp   <= {4{1'b0}};
    vga_green_temp <= {4{1'b0}};
    vga_blue_temp <= {4{1'b0}};
end
```

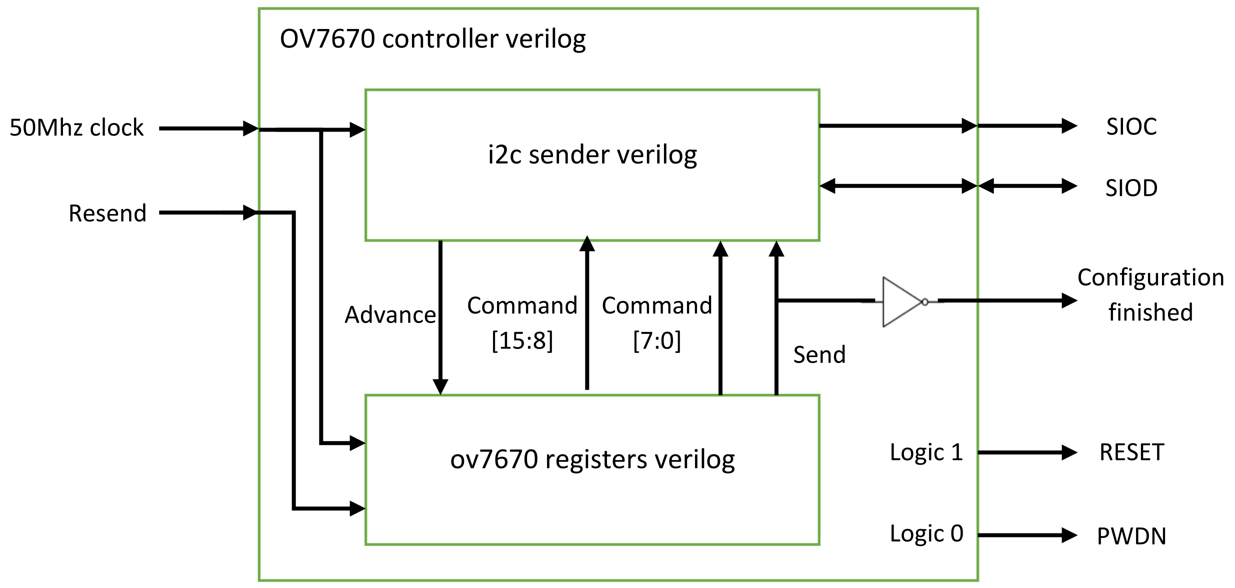
Flag to indicate the current pixels is in the displayed image area or not

Data will be split into R, G, and B sections

Increment the address to get the next data in the frame buffer

3.3. Controller

The module is responsible for reading data from a LUT, get the register address and value, then write the value into the register address of OV7670 through SCCB interface. This section is based on sections **3.3, 3.5. 3.6, 3.7** of **ov-sccb.pdf**



Signal name	Description
Resend (re-configuration)	Re-sending all the register address and data to OV7670
Advance	Move to the next register address and data
Command [15:8]	Register address
Command [7:0]	Data to write into the register address
Send	Flag indicating new data to send to OV7670
Configuration Finished	Flag indicating all data is sent to OV7670
SIOC	Clock signal for SCCB interface
SIOD	Data signal for SCCB interface
RESET	Constantly driven by logic 1 so that OV7670 is in normal mode
PWDN	Constantly driven by logic 0 so that OV7670 is in normal mode

3.3.1. SCCB protocol

The SCCB protocol has two types of transmission:

Three-Wire Data Transmission

Two-Wire Data Transmission

The difference of two type are the START and STOP conditions and the Three-Wire Data Transmission requires an additional signal named SCCB_E (SCCB enable) while Two-Wire Data Transmission requires only SIOC and SIOD.

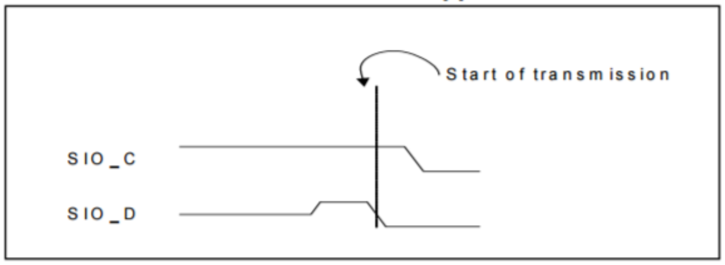
The OV7670 uses **Two-Wire Data Transmission** to communicate with the master device.

3.3.1.1. Two-Wire Data Transmission (START and STOP conditions)

The Two-Wire Data Transmission is quite similar to I2C protocol. It requires only 2 signals SIOC and SIOD. There are START condition to start transferring data and STOP condition to stop transferring data. The START and STOP conditions are defined by the combination behaviors of SIOC and SIOD.

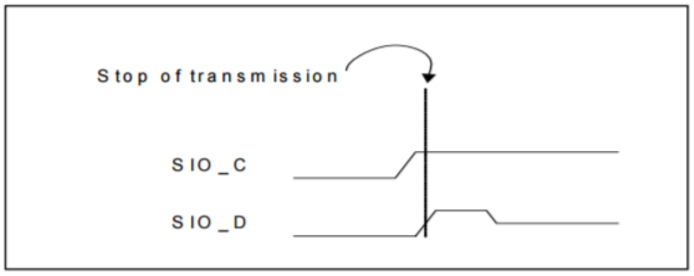
A START condition is indicated by a transition from tri-state ("Z") to high ("1") of SIOD, followed by a transition from high ("1") to low ("0") of SIOD. In addition, during those 2 transitions, SIOC must be high ("1").

Two-Wire Start of Transmission – Tri-State Supported



A STOP condition is indicated by a transmission from low ("0") to high ("1") of SIOD while SIOC is high

Two-Wire Stop of Transmission – Tri-State Supported



3.3.1.2. Two-Wire Data Transmission (Transmission cycles)

Between the START and STOP conditions, the data is transmitted through the SIOD wire from Master to Slave for writing or from Slave to Master for reading. There are three different transmission types:

3-phase write transmission cycle

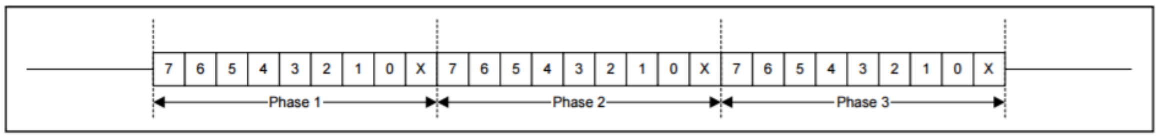
2-phase write transmission cycle

2-phase read transmission cycle

The implementation in this project will use 3-phase write transmission cycle to transmit the data from the Master (FPGA) to Slave (OV7670)

3.3.1.3. What is a phase of transmission and how 3-phase data is created?

A phase contains a total of 9 bits. The 9 bits consist of an 8-bit sequential data transmission follow by a 9th bit. The 9th bit is a "don't care" bit.



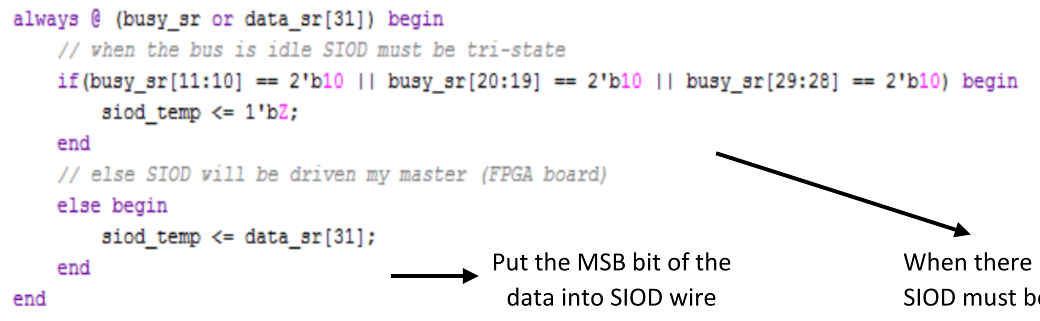
Phase 1: ID Address
Phase 2: Sub-address / Read data
Phase 3: Write Data

Based on the description above, the data that the SIOD will carry must be

Name	START condition	8-bit id	Don't care bit	8-bit register address	Don't care bit	8-bit value	Don't care bit	STOP condition
data	3'b100	8'h42	1'b0	xxxx_xxxx	1'b0	xxxx_xxxx	1'b0	2'b01

3.3.1.4. Code explanation

Controlling SIOD wire – The SIOD wire either is driven by the data that the Master wants to send to the Slave or tri-state.

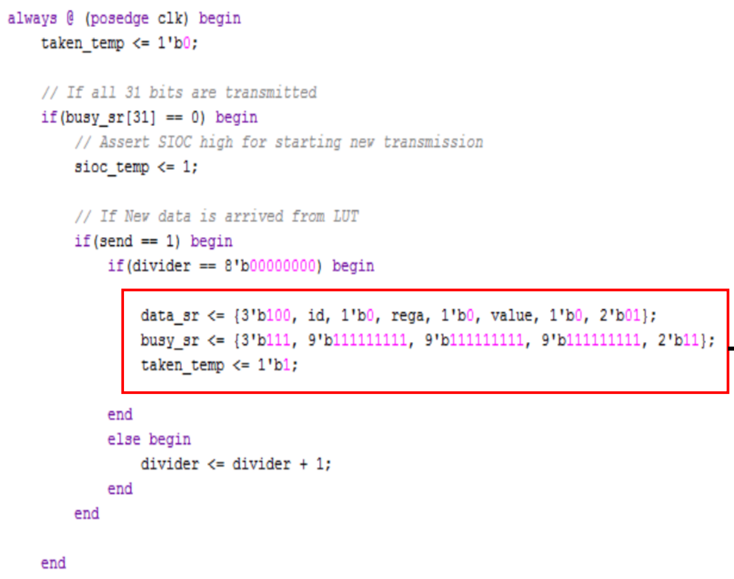


```
always @ (busy_sr or data_sr[31]) begin
    // when the bus is idle SIOD must be tri-state
    if(busy_sr[11:10] == 2'b10 || busy_sr[20:19] == 2'b10 || busy_sr[29:28] == 2'b10) begin
        siod_temp <= 1'bZ;
    end
    // else SIOD will be driven my master (FPGA board)
    else begin
        siod_temp <= data_sr[31];
    end
end
```

$\longrightarrow$ Put the MSB bit of the data into SIOD wire

When there is no data SIOD must be tri-state

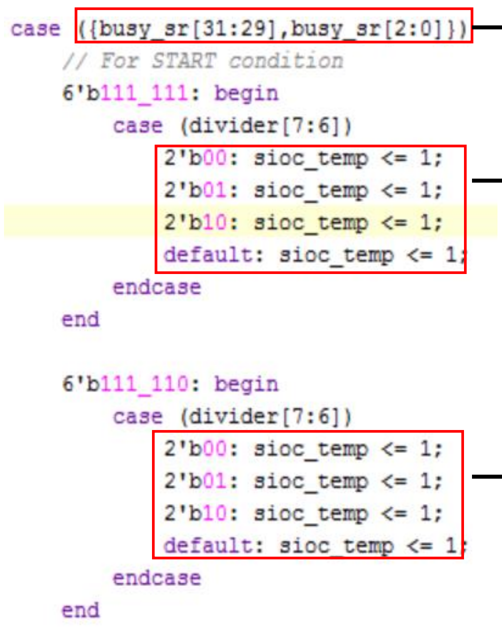
Generating data for 3-phase write transmission and the flag for the LUT to taking new register address and new data



```
always @ (posedge clk) begin
    taken_temp <= 1'b0;

    // If all 31 bits are transmitted
    if(busy_sr[31] == 0) begin
        // Assert SIOC high for starting new transmission
        sioc_temp <= 1;

        // If New data is arrived from LUT
        if(send == 1) begin
            if(divider == 8'b00000000) begin

                data_sr <= {3'b100, id, 1'b0, rega, 1'b0, value, 1'b0, 2'b01};
                busy_sr <= {3'b111, 9'b111111111, 9'b111111111, 9'b111111111, 2'b11};
                taken_temp <= 1'b1;

            end
            else begin
                divider <= divider + 1;
            end
        end

    end
```

- Creating data based on 3-phase write transmission
- The busy_sr is used to tracked the current bit position of the data
- Taken is the flag for the LUT to take new register address and data

Controlling SIOC wire - The SIOC wire either is driven by the data that the Master wants to send to the Slave or tri-state.

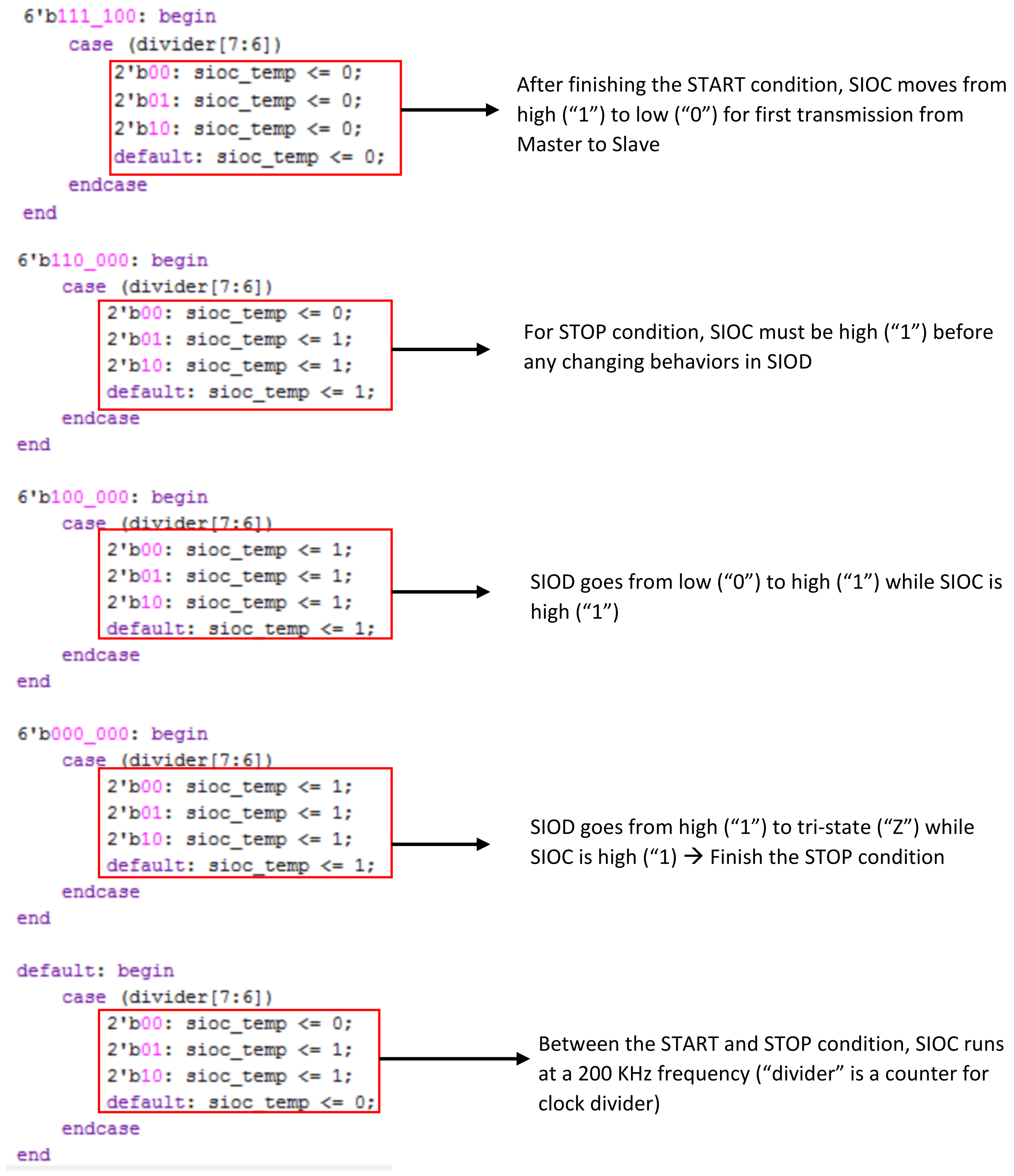


```
case ({busy_sr[31:29],busy_sr[2:0]})
    // For START condition
    6'b111_111: begin
        case (divider[7:6])
            2'b00: sioc_temp <= 1;
            2'b01: sioc_temp <= 1;
            2'b10: sioc_temp <= 1;
            default: sioc_temp <= 1;
        endcase
    end

    6'b111_110: begin
        case (divider[7:6])
            2'b00: sioc_temp <= 1;
            2'b01: sioc_temp <= 1;
            2'b10: sioc_temp <= 1;
            default: sioc_temp <= 1;
        endcase
    end
```

When SIOD has data for START and STOP conditions

For START condition, SIOC must be high while SIOD move from tri-state to high ("1")

SIOC must be high while SIOD moves from high ("1") to low ("0") → Finish the START condition



```
6'b111_100: begin
    case (divider[7:6])
        2'b00: sioc_temp <= 0;
        2'b01: sioc_temp <= 0;
        2'b10: sioc_temp <= 0;
        default: sioc_temp <= 0;
    endcase
end
```

After finishing the START condition, SIOC moves from high ("1") to low ("0") for first transmission from Master to Slave

```
6'b110_000: begin
    case (divider[7:6])
        2'b00: sioc_temp <= 0;
        2'b01: sioc_temp <= 1;
        2'b10: sioc_temp <= 1;
        default: sioc_temp <= 1;
    endcase
end
```

For STOP condition, SIOC must be high ("1") before any changing behaviors in SIOD

```
6'b100_000: begin
    case (divider[7:6])
        2'b00: sioc_temp <= 1;
        2'b01: sioc_temp <= 1;
        2'b10: sioc_temp <= 1;
        default: sioc_temp <= 1;
    endcase
end
```

SIOD goes from low ("0") to high ("1") while SIOC is high ("1")

```
6'b000_000: begin
    case (divider[7:6])
        2'b00: sioc_temp <= 1;
        2'b01: sioc_temp <= 1;
        2'b10: sioc_temp <= 1;
        default: sioc_temp <= 1;
    endcase
end
```

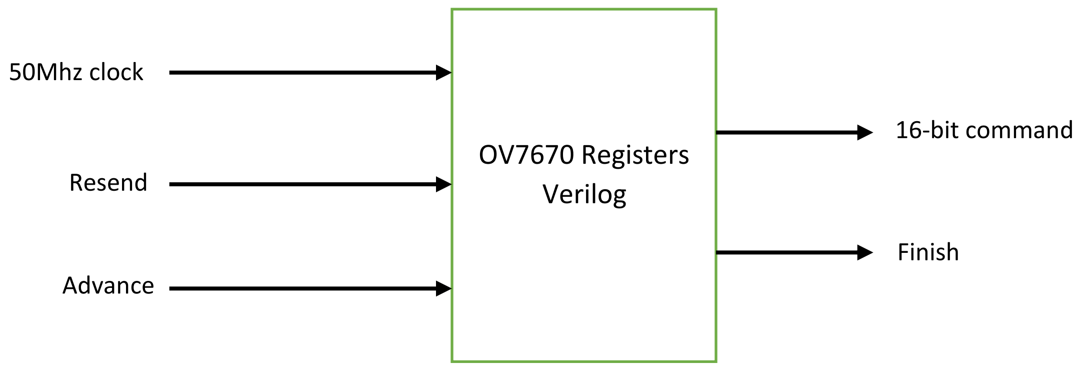
SIOD goes from high ("1") to tri-state ("Z") while SIOC is high ("1) → Finish the STOP condition

```
default: begin
    case (divider[7:6])
        2'b00: sioc_temp <= 0;
        2'b01: sioc_temp <= 1;
        2'b10: sioc_temp <= 1;
        default: sioc_temp <= 0;
    endcase
end
```

Between the START and STOP condition, SIOC runs at a 200 KHz frequency ("divider" is a counter for clock divider)

3.3.2. Data for configuration

The module is similar to a LUT. The module will provide the data based on the receiving address. The data is a 16-bit combination of register address and value; the 8 MSB represents the register address and the 8 LSB represents the value to write into the register address. The register address and the value is based on the **Resister Set** section in the **OV7670 datasheet.**



Code explanation

Generating the flag to indicating all the data for configuration is sent

```
always @ (sreg) begin
    if(sreg == 16'hFFFF) begin
        finished_temp <= 1;
    end
    else begin
        finished_temp <= 0;
    end
end
```

The "FFFF" value indicating it's the end of the LUT which means all the data for configuration is sent. Therefore, the finish signal is asserted

Re-send all the data or move to the next data

```
// Get value out of the LUT
always @ (posedge clk) begin
    if(resend == 1) begin
        address <= {8{1'b0}};
    end
    else if(advance == 1) begin
        address <= address+1;
    end

    case (address)
        0 : sreg <= 16'h12_80;
        1 : sreg <= 16'h12_80;
        2 : sreg <= 16'h12_04;
        3 : sreg <= 16'h11_00;
        4 : sreg <= 16'h0C_00;
        5 : sreg <= 16'h3E_00;
       …
       54 : sreg <= 16'hB3_82;
       55 : sreg <= 16'hB8_0A;
       default : sreg <= 16'hFF_FF;
   endcase
```

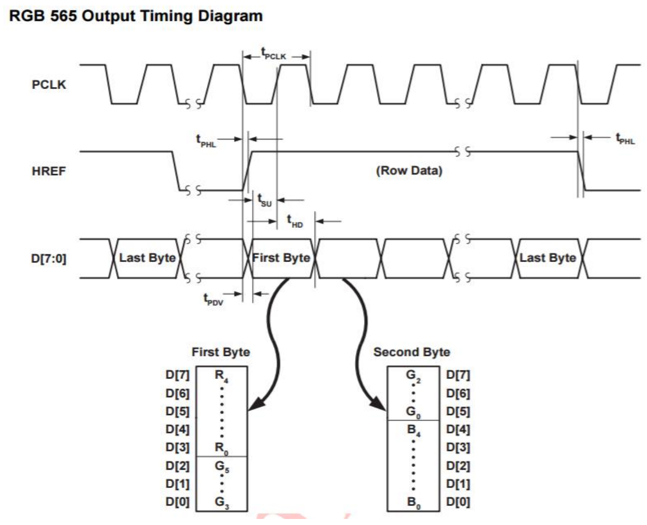
Re-send all the data by resetting the index value

Move to next data by incrementing the index value

The register address and the value to write into the register address at each index

3.4. Capture Logic

OV7670 has multiple models for sending a color of a pixel. In this implementation, the **RGB 565 model** is used. Based on the **timing specification** in the **OV7670 datasheet**, the OV7670 requires 2 PCLK cycles after HREF signal is asserted to produce a color of a pixel.



Because the full RGB of a pixel is generated after 2 PCK cycles, it requires at least 3 PCLK cycles to write the full RGB color into a memory.

PCLK	Signal name	Data	Description of the cycle
0	Href	1	Href is asserted, the first byte sent from the OV7670 is latched. Write-enable signal is not asserted
	D latch (16-bit)	xxxx_xxxx_$R_oR_oR_oR_o$_$R_oG_oG_oG_o$	
	D capture (16-bit)	xxxx_xxxx_xxxx_xxxx	
	WE	0	
1	Href	0	Href is de-asserted, the second byte sent from the OV7670 is latched. Write-enable signal is not asserted
	D latch (16-bit)	$R_oR_oR_oR_o$_$R_oG_oG_oG_o$_$G_oG_oG_oB_o$_$B_oB_oB_oB_o$	
	D capture (16-bit)	xxxx_xxxx_$R_oR_oR_oR_o$_$R_oG_oG_oG_o$	
	WE	0	
2	Href	1	Href is asserted, a new first byte sent from the OV7670 is latched. Write-enable signal is asserted to write the old 16-bit data into the memory
	D latch (16-bit)	$R_oR_oR_oR_o$_$R_oG_oG_oG_o$_$R_NR_NR_NR_N$_$R_NG_NG_NG_N$	
	D capture (16-bit)	$R_oR_oR_oR_o$_$R_oG_oG_oG_o$_$G_oG_oG_oB_o$_$B_oB_oB_oB_o$	
	WE	1	

Subscript of “O” means old value of RGB and Subscript of “N” means new value of RGB