Design Document

Group Infomation

Term: 2024 Spring
Project name: Web camera

Group Members	**UNI**
Qiuhong Chen	qc2335
Shifei Zheng	sz3196
Yizhan Zhang	yz4703

Overview

This project implements a web camera that pushes the video captured by the camera to the internet stream. This system can be divided into 2 parts: hardware and software. In the hardware part, we adopt Dec-Soc1, inherited from the course's lab as the development board which is built around Altera FPGA. We use OV7670 as the external camera module to capture video pixels. The video is captured by the FPGA and outputs to the VGA interface, which is used for debugging. Meanwhile, the pixels are also transmitted by the Avalon bus to the software world, where more complex processing happens.
The software world is enabled by the Altera Cyclone SoC, where a Ubuntu Linux system resides. In the bottom layer, the camera driver reads the pixels from the I/O, which forms steady video signals. In the upper layer, the powerful FFmpeg stream encoder and Nginx web server are responsible for pushing the camera signals through RTMPto the internet. On the remote site, the clients can receive the RTMP video stream on their local media player.

Work division

Development

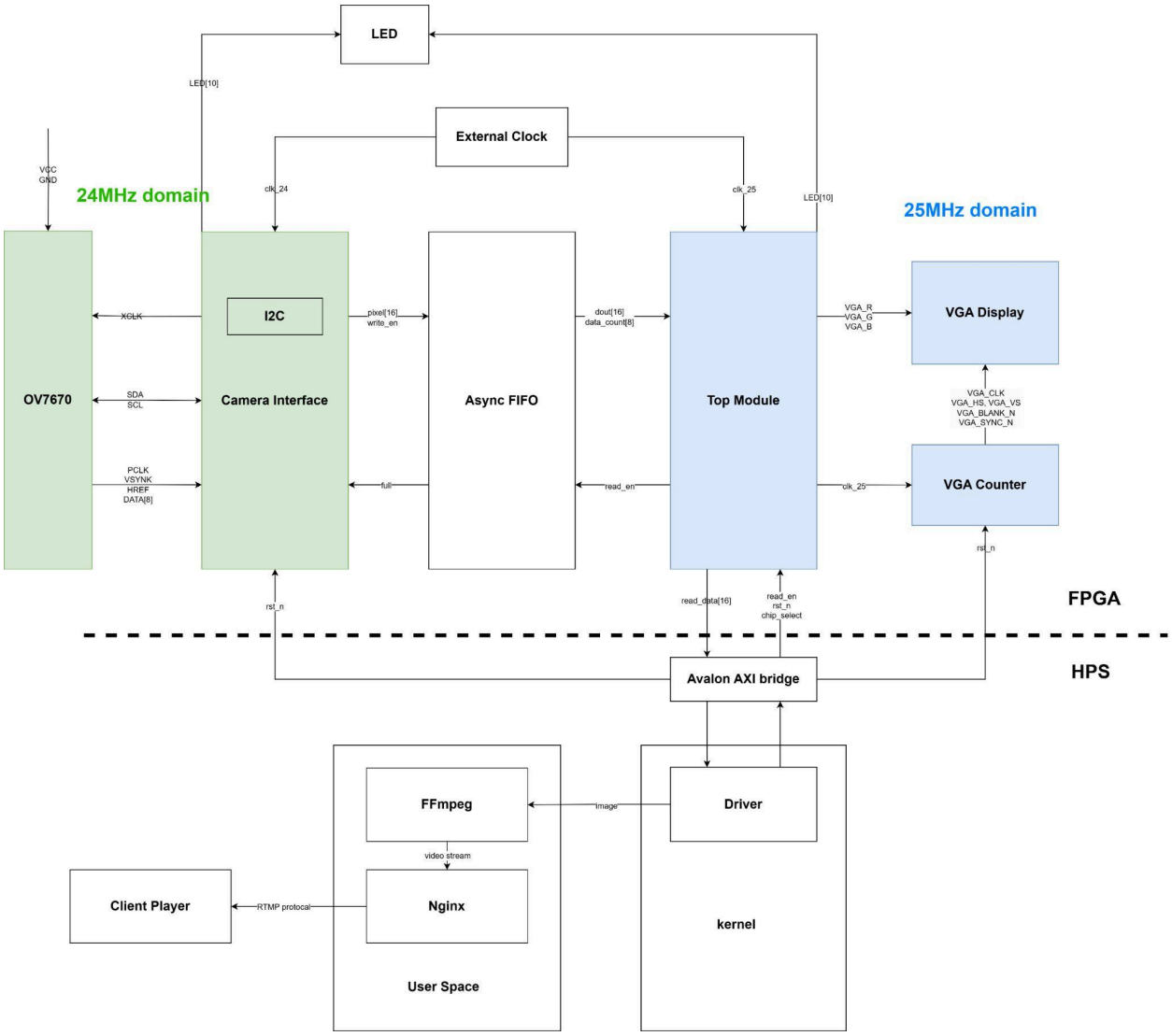
Qiuhong is responsible for 1) hardware design and debugging, and 2) building the video streaming solutions (video server & client).
Shifei is responsible for hardware synthesis and Qsys configuration.
Yizhan is responsible for the development of the camera driver and the sender program.

Documentation

Qiuhong writes the proposal and design documents. In the final report, he is responsible for the block diagram, hardware design, and web development section.
Shifei is responsible for calculating RAM usage in the design document.
Yizhan writes the camera driver and sender section in the design and report.

Block diagram

The block diagram describes the system components and hardware/software interfaces.

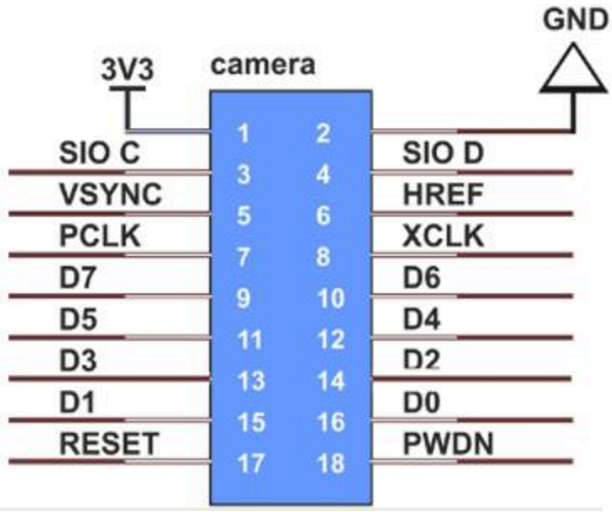


The system is generally divided into 2 parts: FPGA and HPS, which correspond to the hardware and software parts.

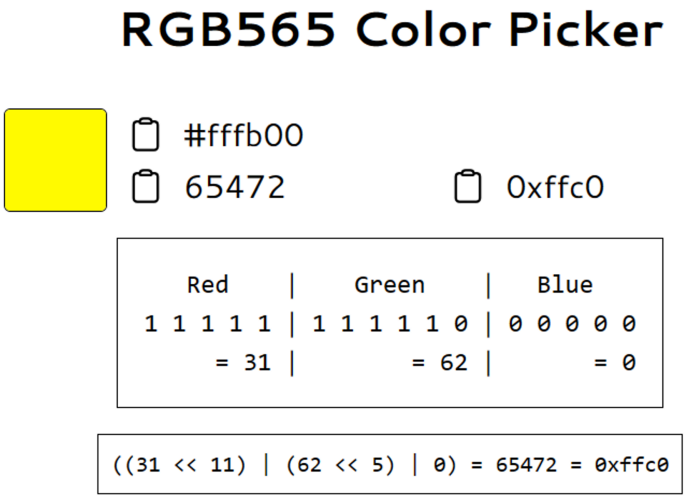
Hardware Design

Interfacing OV7670 Camera

OV7670 Camera communicates with the SCCB protocol, which is a subset of the I2C protocol, so we can use I2C to talk to it. The camera itself contains pixel encoding functionalities. To be specific, we can choose among Raw RGB (GRB 4:2:2, RGB565/555/444), YUV (4:2:2), and YCbCr (4:2:2) formats.

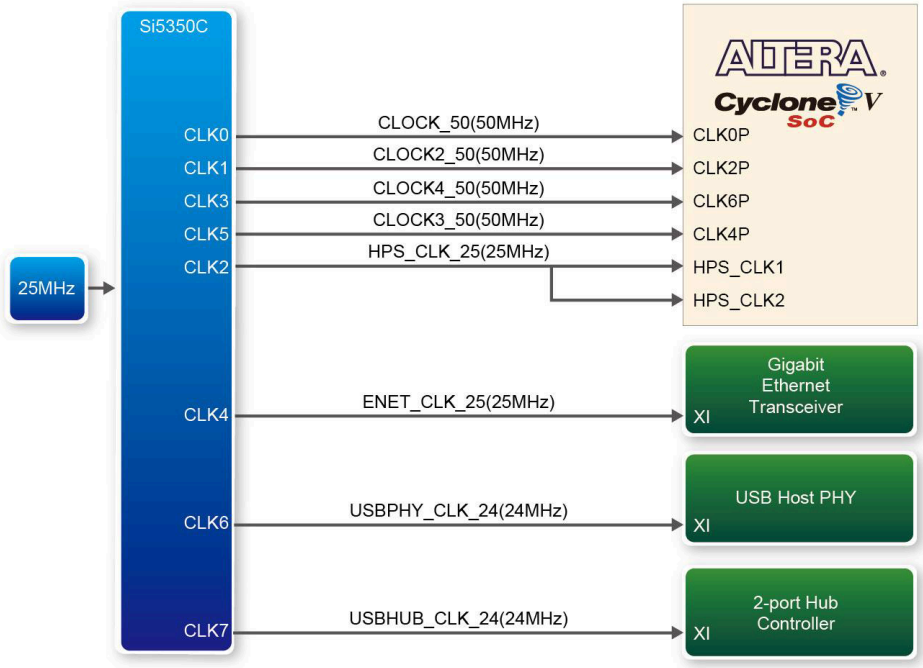


As the VGA display is in RGB format, for convenience, we choose RGB565 color format as camera output. RGB565 uses 5 bits for red, 6 bits for green, and 5 bits for blue. For instance, the color “yellow” can be expressed as the 16-bit ((31 << 11) | (62 << 5) | 0) = 65472 = 0xffc0. The pixel data signal pins out from D0-D7 (8 bits), which is synchronized to PCLK (pixel clock) at the falling edge. As there are only 8 data pins, every single 16-bit pixel should be transferred in two rounds. We will need to combine 2 cycles of data outputs to get a full pixel. The 24MHz external XCLK is provided by the FPGA. As it is slightly lower than the 25MHz VGA clock, we will need an asynchronous FIFO when it connects with the VGA module.



Unlike the “plug-and-play” VGA display and joypad we encountered in the lab, we have to initialize the OV7670 camera (e.g. output the specified color format) by sending certain sequences(address and data) to the SDA/SIO_D pin according to the SCCB protocol.

We found an implementation of the OV7670 camera interface in GitHub. The interface module configures the camera to output the RGB565 pixel and combines two consecutive pixels to output a 16-bit pixel channel. However, the implementation is on Xilinx FPGA and it uses some specific instantiations. The major obstacle is when we try to adapt it to our Cyclone FPGA by replacing clock sources. The module requires a 165MHz clock for general register updates. However, Cyclone V SoC FPGA can only generate at most 50MHz clock. This could be the root cause that we cannot receive a VSYNC control signal from the camera after initialization. We were unable to make the module function normally, partly because we are unfamiliar with signal-analyzing tools to debug the camera module. We can only rely on LED lights to reveal SCCB state signals, which is inefficient and ineffective.



Connecting 24MHz and 25MHz domains

The FIFO buffer is 8-bit wide and 12-bit addressable, meaning that the total RAM size is 8*(2^12)=2^15 bits, or 32kb. The write pointer is clocked by the 24MHz clock while the read pointer is clocked by the 25MHz clock. This concerns clock domain crossing, so we need asynchronous FIFO to pass data safely. We borrowed an asynchronous FIFO module implementation and figured out the basic idea.

The FIFO, either synchronous or asynchronous, software or hardware, includes a write pointer and a read pointer to indicate the address in the queue so that we can write to and read from the FIFO. To make the FIFO work, we also need two flags, empty and full, to indicate the status of the queue. We should stop reading when the FIFO is empty and stop writing when the FIFO is full. In the case of synchronous FIFO, the read and write pointers are generated by the same clock, so the empty and full flags can be expressed by simple combinational logic:

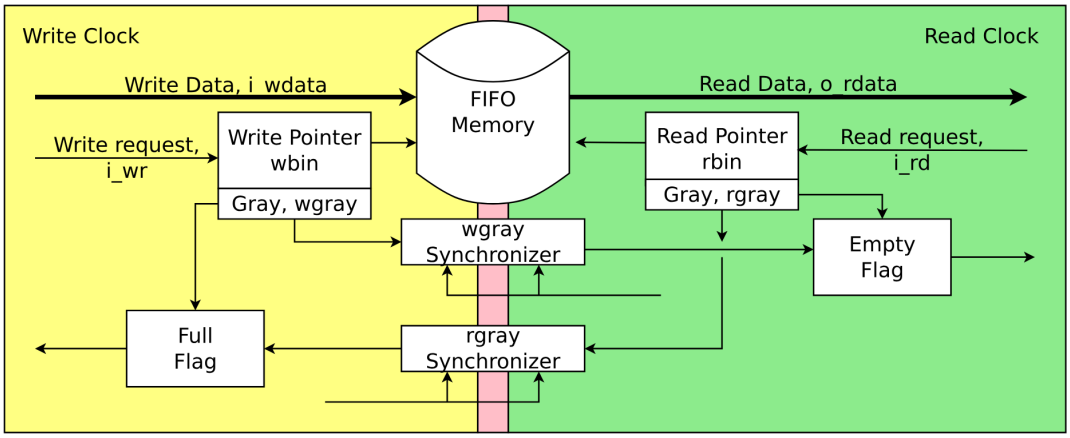
```
Unset

assign empty = (w_ptr == r_ptr);
assign full  = (w_ptr[ADDR_WIDTH] != r_ptr[ADDR_WIDTH])&&(w_ptr[ADDR_WIDTH-1:0]
== r_ptr[ADDR_WIDTH-1:0])
```

In the case of asynchronous FIFO, this flag may not be stable when it is read by either read or write clock, because the two pointers are not synced. This leads to the problem of metastability. To reduce the probability of metastability, the author introduces a temporary signal in the middle by using 2 D-Flipflops to cross the domain.

```
//////////////////////CLOCK DOMAIN CROSSING/////////////////////////////
reg [FIFO_DEPTH_WIDTH:0] r_grey_sync_temp;
reg [FIFO_DEPTH_WIDTH:0] w_grey_sync_temp;
always @(posedge clk_write) begin //2 D-Flipflops for reduced metastability in clock domain crossing from READ
DOMAIN to WRITE DOMAIN
    r_grey_sync_temp <= r_grey;
    r_grey_sync <= r_grey_sync_temp;
end
always @(posedge clk_read) begin //2 D-Flipflops for reduced metastability in clock domain crossing from WRITE
DOMAIN to READ DOMAIN
    w_grey_sync_temp <= w_grey;
    w_grey_sync <= w_grey_sync_temp;
end
```

Obviously, the probability of metastability of the flags is linked to the number of bits changing in the pointers each time. To further reduce metastability, gray code is used to make the pointers change only 1-bit at a time. We need two additional gray counters/pointers besides the two binary counters/pointers. This requires a gray-to-binary translator and new combinational logic for flag signals, which I won't cover here. At the end, the binary counter can access the RAM as normal, and an asynchronous FIFO is completed.



Output to VGA Display

We utilize the vga_counter module in Lab3 to produce the VGA control signals. The module takes in a 50MHz clock for synchronization. The module maintains two counters: the HCOUNT signal counts the pixels in a horizontal line and the VCOUNT signal counts the number of lines to determine when one frame ends and the next begins. Based on these two coordinates, the counter generates a series of control signals for the VGA port, including VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_N, and VGA_SYNC_N.
The 16-bit pixel data from the camera interface is in RGB565 format, so we need a RGB565 to RGB888 converter. We adopt the implementation that concatenates the RGB565 signals with their lower digits.

```
Unset
r_8 = {r_5, r_5[2:0]};
g_8 = {g_6, g_6[1:0]};
b_8 = {b_5, b_5[2:0]};
```

With the VGA display ready and camera images displaying on the screen(we finally display the bouncing ball as the camera module cannot work), the next step is to pump the pixels to the software, which assembles them into the video stream.

Software

Interconnect with AXI Bridge

The integration of the design's hardware and software components was achieved based on the skeleton project provided for Lab 3. This skeleton code not only offered a foundation to build upon but also included a pre-existing hardware module for VGA display interaction(which is

used for debugging in the project). The interfacing involved modifying the Lab 3 driver to enable the reception and transmission of the data needed to send our frame. All we have to do is to modify the clock frequency, create the OV7670 conduit according to the OV7670 datasheet and connect the corresponding signal correctly in the Platform Designer.

The signal we tried to transmit is under the listed format below:

```
output logic [63:0] readdata,  // {12'd0, column_num, row_num, endOfField, 15'd0, dout}
input logic read,
input logic [63:0] writedata,  // {32'd0, x, y}
input logic write,
```

, where the dout signal denoted the RGB565 value that we are trying to transmit. column_num and row_num indicates the location of the corresponding pixel.

Camera driver

The driver, named "top_module", is designed to control a VGA display's ball movement and read data from a camera, facilitating real-time interaction and data manipulation directly from user-space applications.
There are three modes, one is vga ball write, vga ball read and camera read. For the camera reading, it utilize `ioread32` to fetch data from the camera, offering basic data acquisition functionality directly linked to hardware operations.

```C/C++
// read data
static unsigned int camera_read(void)
{
```

```
        return ioread32((dev.virtbase));
}
```

Sender

This userspace program is designed to interface with custom device drivers, specifically targeting the control and visualization of a moving "ball" (possibly a graphical element) on a VGA display and the capture of frame data from a camera. The program combines functionalities of device communication via `ioctl` system calls, network communication using TCP sockets, and basic animation logic.

1. Device Communication

- **Device Files**: The program interacts with device drivers through special files (`/dev/top_module`). These drivers presumably control the VGA display and camera.
- **`ioctl` Calls**: The program uses `ioctl` to send commands to and receive data from these device drivers, specifically for reading the current position of the ball on the display and updating its position.

2. TCP Socket Communication

- **Socket Setup**: Establishes a TCP socket connection to a specified server (`SERVER_IP` and `PORT`). This is used to send frame data captured from the camera to a remote server.
- **Data Transmission**: Captured frame data is sent over this socket, demonstrating a typical client-server architecture where the client (this program) captures and forwards data to a server for further processing or display.

```C/C++
void send_frame(const char* buffer) {
    int bytes_sent = 0;
    int total_bytes = BUFFER_SIZE;

    while (bytes_sent < total_bytes) {
        int n = send(sockfd, buffer + bytes_sent, total_bytes - bytes_sent, 0);
        if (n < 0) {
            perror("send failed");
            exit(EXIT_FAILURE);
        }
        bytes_sent += n;
    }
    printf("Frame sent. Bytes sent: %d\n", bytes_sent);
```

```
}
```

3. Frame Processing

- **Frame Reading**: Utilizes a while loop to continuously read pixel data using `ioctl` calls to the camera driver, illustrating how raw image data is processed at a low level.
- **Frame Sending**: After capturing a frame, it immediately sends this data to a server using the established TCP connection, showcasing a real-time data streaming application.

C/C++

```
void read_frame(char* buf)
{
    unsigned int info;
    unsigned int hcount, vcount;
    unsigned int address;
    unsigned char pixel;

    while (1) {
        if (ioctl(cam_fd, CAMERA_READ, &info)) {
            perror("ioctl(CAMERA_READ) failed");
            return;
        }

        // Extract fields from the info
        hcount = (info >> 29) & 0x3FF;      // Extract bits 29-38 (10 bits for hcount)
        vcount = (info >> 19) & 0x3FF;      // Extract bits 19-28 (10 bits for vcount)
        pixel = info & 0xFF;                // Extract bits 0-7 for the pixel value (dout)

        // Calculate the address
        address = vcount * 640 + hcount;

        if (address < READ_PIXELS) {
            buf[address] = pixel;
        }

        // Check for end of field
        if (info & (1 << 18)) {
            break; // Exit the loop when end of field is encountered
        }
    }
}
```

Web development

The server reads video frames from the camera driver and streams the video to the internet. There are corresponding clients which receives the frames and displays on the screen. We prepare 2 alternatives for server development.

Solution 1: Nginx Streaming Server + FFmpeg encoder

The first solution that comes into our minds is the existing video streaming framework. On the server side, we set up an Nginx server on the HPS's Ubuntu system, which hosts an RTMP video stream. FFmpeg is used to encode the stream and send it to the server. On the receiver side, we only need to type in the streaming address on the internet browser, and the OS will call the local player for video playback.
The configuration is simple and we first use an existing video file as the streaming source. The command line is as follows:

```
Unset
VIDEO_FILE="big_buck_bunny_240p_30mb.mp4"
ffmpeg -re -i $VIDEO_FILE -c copy -f flv rtmp://localhost/live/test
```

The next step is to replace the video source with camera input. Luckily, FFmpeg as a powerful video streaming tool also offers support.

```
Unset
ffmpeg -re -f v4l2 -i /dev/video0 -f flv rtmp://localhost/live/test
```

However, this approach requires a fully functional camera driver. We worry that we may spend too much time figuring out the relative standards. Thus, we choose to implement one with simple functionality. We write our own server & client which offers a simple interface that only acquires raw images from the driver.

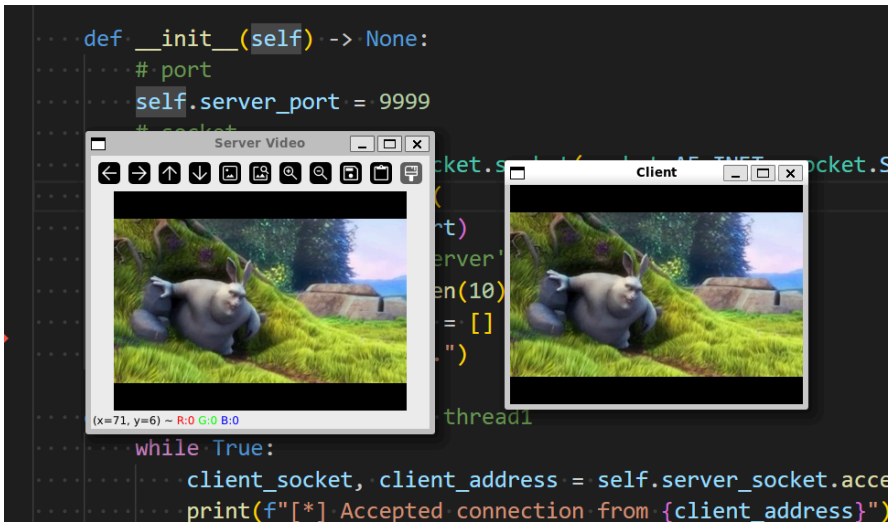
Solution 2: Using Python's CV library

The second solution implements a Python server program. The server keeps 2 threads:

- Thread 1 accepts client TCP connections, and maintains the client list.
- Thread 2 encodes each video frame in JPEG format, serializes to bytes, and broadcasts the image to all clients.

Although this solution has poor performance, it is easy to control because we can call the driver on our own. The CV library can also read images from both video files and camera input. We set the frame rate to 30 fps. Note that this is not an exact rate. In practice, the server waits 1/30 second and then starts to send the next image.

The client is written in C++. It establishes the connection with the server, receives the image, decodes it into JPEG format, and displays it using the CV library.[1]
We also first tested this program on a 240p video file. The bitrate is low and the display occasionally has a delay, but this is a truly playable stream.



By design, the internet server should directly call the driver in the kernel module to get the video frames. But as mentioned in the “driver” section, in reality, we implement a sender, as a user program. The sender interface with the driver and sends the video frames to the server via socket. This can be redundant design but we don’t have time to change it.

Lessons learned

Although we learned a lot from this project, it was too challenging for our 3 FPGA beginners. In the end, we failed to present a workable prototype, for the following reasons.

1. We fail to embed the 3rd-party camera interface module. We use the bouncing ball to replace the camera output to continue the remaining part.
    a. Due to the closure of the lab during the final, the debugging phase of our project is postponed. Qiuhong and Shifei returned to China and started their internship by the end of May. Considering the working schedule and time zone, we can only arrange a time to debug on weekends.
    b. We are not familiar with signal-analyzing tools to debug the camera module. In practice, we have to rely on LED lights to reveal suspicious signals, which is inefficient and ineffective.
2. For the software part, it is challenging to figure out how data is sent from hardware and through the driver and how data is displayed in software code, especially for the video form, where data is in a more complex form. And we also faced some problems installing opencv on the board and also setup the network between devices.

[1] Part of this code is reused in Qiuhong Chen’s another course project in 24Spring.

File Listing

Our files include:

```
Unset
|
├─Hardware
|      asyn_fifo.v
|      camera_interface.v
|      i2c_top.v
|      top_module.sv
|      vga_counter.sv
|
└─Software
        client.cpp
        client.h
        sender.c
        CMakeLists.txt
        main.cpp
        server.py
```

asyn_fifo.v

```
Unset
`timescale 1ns / 1ps

module asyn_fifo #(
    parameter DATA_WIDTH = 8,
    FIFO_DEPTH_WIDTH = 11  //total depth will then be 2**FIFO_DEPTH_WIDTH
) (
    input wire rst_n,
    input wire clk_write,
    clk_read,  //clock input from both domains
    input wire write,
    read,
    input wire [DATA_WIDTH-1:0] data_write,  //input FROM write clock domain
    output wire [DATA_WIDTH-1:0] data_read,  //output TO read clock domain
    output reg full,
    empty,  //full=sync to write domain clk , empty=sync to read domain clk
    output reg [FIFO_DEPTH_WIDTH-1:0] data_count_w,
    data_count_r //counts number of data left in fifo memory(sync to either
write or read clk)
```

```
);


    localparam FIFO_DEPTH = 2 ** FIFO_DEPTH_WIDTH;

    initial begin
        full  = 0;
        empty = 1;
    end


    ////////////////////WRITE CLOCK DOMAIN////////////////////////////////
    reg [FIFO_DEPTH_WIDTH:0] w_ptr_q = 0;  //binary counter for write pointer
    reg[FIFO_DEPTH_WIDTH:0] r_ptr_sync; //binary pointer for read pointer sync
to write clk
    wire[FIFO_DEPTH_WIDTH:0] w_grey,w_grey_nxt; //grey counter for write
pointer
    reg[FIFO_DEPTH_WIDTH:0] r_grey_sync; //grey counter for the read pointer
synchronized to write clock

    wire we;
    reg [3:0] i;  //log_2(FIFO_DEPTH_WIDTH)

    assign w_grey=w_ptr_q^(w_ptr_q>>1); //binary to grey code conversion for
current write pointer
    assign w_grey_nxt=(w_ptr_q+1'b1)^((w_ptr_q+1'b1)>>1);  //next grey code
    assign we = write && !full;

    //register operation
    always @(posedge clk_write, negedge rst_n) begin
        if (!rst_n) begin
            w_ptr_q <= 0;
            full <= 0;
        end else begin
            if (write && !full) begin  //write condition
                w_ptr_q <= w_ptr_q + 1'b1;
                full <= w_grey_nxt ==
{~r_grey_sync[FIFO_DEPTH_WIDTH:FIFO_DEPTH_WIDTH-1],r_grey_sync[FIFO_DEPTH_WIDTH
-2:0]}; //algorithm for full logic which can be observed on the grey code table
            end else
                full <= w_grey ==
{~r_grey_sync[FIFO_DEPTH_WIDTH:FIFO_DEPTH_WIDTH-1],r_grey_sync[FIFO_DEPTH_WIDTH
-2:0]};
```

```
            for (i = 0; i <= FIFO_DEPTH_WIDTH; i = i + 1)
                r_ptr_sync[i]=^(r_grey_sync>>i); //grey code to binary
converter
            data_count_w <= (w_ptr_q>=r_ptr_sync)?
(w_ptr_q-r_ptr_sync):(FIFO_DEPTH-r_ptr_sync+w_ptr_q); //compares write pointer
and sync read pointer to generate data_count
        end
    end

    ////////////////////////////////////////////////////////////////


    ////////////////////READ CLOCK DOMAIN//////////////////////////////
    reg [FIFO_DEPTH_WIDTH:0] r_ptr_q = 0;  //binary counter for read pointer
    wire [FIFO_DEPTH_WIDTH:0] r_ptr_d;
    reg[FIFO_DEPTH_WIDTH:0] w_ptr_sync; //binary counter for write pointer sync
to read clk
    reg[FIFO_DEPTH_WIDTH:0] w_grey_sync; //grey counter for the write pointer
synchronized to read clock
    wire[FIFO_DEPTH_WIDTH:0] r_grey,r_grey_nxt; //grey counter for read pointer


    assign r_grey = r_ptr_q ^ (r_ptr_q >> 1);  //binary to grey code conversion
    assign r_grey_nxt= (r_ptr_q+1'b1)^((r_ptr_q+1'b1)>>1); //next grey code
    assign r_ptr_d = (read && !empty) ? r_ptr_q + 1'b1 : r_ptr_q;

    //register operation
    always @(posedge clk_read, negedge rst_n) begin
        if (!rst_n) begin
            r_ptr_q <= 0;
            empty   <= 1;
        end else begin
            r_ptr_q <= r_ptr_d;
            if (read && !empty)
                empty <= r_grey_nxt ==  ;  //empty condition
            else empty <= r_grey == w_grey_sync;

            for (i = 0; i <= FIFO_DEPTH_WIDTH; i = i + 1)
                w_ptr_sync[i]=^(w_grey_sync>>i); //grey code to binary
converter
            data_count_r = (w_ptr_q>=r_ptr_sync)?
(w_ptr_q-r_ptr_sync):(FIFO_DEPTH-r_ptr_sync+w_ptr_q); //compares read pointer
to sync write pointer to generate data_count
        end
```

```
    end
    ////////////////////////////////////////////////////////////////////////


    ////////////////////CLOCK DOMAIN CROSSING/////////////////////////////
    reg [FIFO_DEPTH_WIDTH:0] r_grey_sync_temp;
    reg [FIFO_DEPTH_WIDTH:0] w_grey_sync_temp;
    always @(posedge clk_write) begin //2 D-Flipflops for reduced metastability
in clock domain crossing from READ DOMAIN to WRITE DOMAIN
        r_grey_sync_temp <= r_grey;
        r_grey_sync <= r_grey_sync_temp;
    end
    always @(posedge clk_read) begin //2 D-Flipflops for reduced metastability
in clock domain crossing from WRITE DOMAIN to READ DOMAIN
        w_grey_sync_temp <= w_grey;
        w_grey_sync <= w_grey_sync_temp;
    end

    ////////////////////////////////////////////////////////////////////////



    //instantiation of dual port block ram
    dual_port_sync #(
        .ADDR_WIDTH(FIFO_DEPTH_WIDTH),
        .DATA_WIDTH(DATA_WIDTH)
    ) m0 (
        .clk_r(clk_read),
        .clk_w(clk_write),
        .we(we),
        .din(data_write),
        .addr_a(w_ptr_q[FIFO_DEPTH_WIDTH-1:0]),  //write address
        .addr_b(r_ptr_d[FIFO_DEPTH_WIDTH-1:0] ), //read address ,addr_b is
already buffered inside this module so we will use the "_d" ptr to advance the
data(not "_q")
        .dout(data_read)
    );

endmodule



//inference template for dual port block ram
module dual_port_sync #(
```

```
    parameter ADDR_WIDTH=11, //2k by 8 dual port synchronous ram(16k block ram)
    DATA_WIDTH = 8
) (
    input clk_r,
    input clk_w,
    input we,
    input [DATA_WIDTH-1:0] din,
    input [ADDR_WIDTH-1:0] addr_a,
    addr_b,  //addr_a for write, addr_b for read
    output [DATA_WIDTH-1:0] dout
);

    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
    reg [ADDR_WIDTH-1:0] addr_b_q;

    always @(posedge clk_w) begin
        if (we) ram[addr_a] <= din;
    end
    always @(posedge clk_r) begin
        addr_b_q <= addr_b;
    end
    assign dout = ram[addr_b_q];

endmodule
```

Camera_interface.v

```
Unset
`timescale 1ns / 1ps

module camera_interface (
    input wire clk_50,
    clk_24,
    rst_n,
    //asyn_fifo IO
    input wire rd_en,
    output wire [9:0] data_count_r,
    output wire [15:0] dout,
    //camera pinouts
    input wire cmos_pclk,
```

```
    cmos_href,
    cmos_vsync,
    input wire [7:0] cmos_db,
    inout cmos_sda,
    cmos_scl,  //i2c comm wires
    output wire cmos_rst_n,
    cmos_pwdn,
    cmos_xclk,
    //Debugging
    output wire [9:0] led
);
    //FSM state declarations
    localparam idle=0,
                                start_sccb=1,
                                write_address=2,
                                write_data=3,
                                digest_loop=4,
                                delay=5,
                                vsync_fedge=6,
                                byte1=7,
                                byte2=8,
                                fifo_write=9,
                                stopping=10;

    localparam wait_init=0,
                                sccb_idle=1,
                                sccb_address=2,
                                sccb_data=3,
                                sccb_stop=4;

    localparam MSG_INDEX=77; //number of the last index to be digested by SCCB



    reg [3:0] state_q = 0, state_d;
    reg [2:0] sccb_state_q = 0, sccb_state_d;
    reg [7:0] addr_q, addr_d;
    reg [7:0] data_q, data_d;
    reg start, stop;
    reg [7:0] wr_data;
    wire rd_tick;
    wire [1:0] ack;
    wire [7:0] rd_data;
    wire [3:0] state;
```

```
    reg [3:0] led_q = 0, led_d;
    reg [30:0] delay_q = 0, delay_d;
    reg start_delay_q = 0, start_delay_d;
    reg delay_finish;
    reg [15:0] message[250:0];
    reg [7:0] message_index_q = 0, message_index_d;
    reg [15:0] pixel_q, pixel_d;
    reg  wr_en;
    wire full;
    wire key0_tick, key1_tick, key2_tick, key3_tick;

    //buffer for all inputs coming from the camera
    reg pclk_1, pclk_2, href_1, href_2, vsync_1, vsync_2;


    initial begin //collection of all adddresses and values to be written in
the camera
        //{address,data}
        message[0] = 16'h12_80;  //reset all register to default values
        message[1] = 16'h12_04;  //set output format to RGB
        message[2] = 16'h15_20;  //pclk will not toggle during horizontal blank
        message[3] = 16'h40_d0;  //RGB565

        // These are values scalped from
https://github.com/jonlwowski012/OV7670_NEXYS4_Verilog/blob/master/ov7670_regis
ters_verilog.v
        message[4] = 16'h12_04;  // COM7,     set RGB color output
        message[5] = 16'h11_80;  // CLKRC     internal PLL matches input clock
        message[6] = 16'h0C_00;  // COM3,     default settings
        message[7] = 16'h3E_00;  // COM14,    no scaling, normal pclock
        message[8] = 16'h04_00;  // COM1,     disable CCIR656
        message[9] = 16'h40_d0;  //COM15,     RGB565, full output range
        message[10]= 16'h3a_04; //TSLB       set correct output data sequence
(magic)
        message[11] = 16'h14_18;  //COM9       MAX AGC value x4 0001_1000
        message[12]= 16'h4F_B3; //MTX1       all of these are magical matrix
coefficients
        message[13] = 16'h50_B3;  //MTX2
        message[14] = 16'h51_00;  //MTX3
        message[15] = 16'h52_3d;  //MTX4
        message[16] = 16'h53_A7;  //MTX5
        message[17] = 16'h54_E4;  //MTX6
        message[18] = 16'h58_9E;  //MTXS
```

```
        message[19]= 16'h3D_C0; //COM13      sets gamma enable, does not
preserve reserved bits, may be wrong?
        message[20] = 16'h17_14;  //HSTART     start high 8 bits
        message[21]= 16'h18_02; //HSTOP      stop high 8 bits //these kill the
odd colored line
        message[22] = 16'h32_80;  //HREF       edge offset
        message[23] = 16'h19_03;  //VSTART     start high 8 bits
        message[24] = 16'h1A_7B;  //VSTOP      stop high 8 bits
        message[25] = 16'h03_0A;  //VREF       vsync edge offset
        message[26] = 16'h0F_41;  //COM6       reset timings
        message[27]= 16'h1E_00; //MVFP       disable mirror / flip //might have
magic value of 03
        message[28] = 16'h33_0B;  //CHLF       //magic value from the internet
        message[29] = 16'h3C_78;  //COM12      no HREF when VSYNC low
        message[30] = 16'h69_00;  //GFIX       fix gain control
        message[31] = 16'h74_00;  //REG74      Digital gain control
        message[32]= 16'hB0_84; //RSVD       magic value from the internet
*required* for good color
        message[33] = 16'hB1_0c;  //ABLC1
        message[34] = 16'hB2_0e;  //RSVD       more magic internet values
        message[35] = 16'hB3_80;  //THL_ST
        //begin mystery scaling numbers
        message[36] = 16'h70_3a;
        message[37] = 16'h71_35;
        message[38] = 16'h72_11;
        message[39] = 16'h73_f0;
        message[40] = 16'ha2_02;
        //gamma curve values
        message[41] = 16'h7a_20;
        message[42] = 16'h7b_10;
        message[43] = 16'h7c_1e;
        message[44] = 16'h7d_35;
        message[45] = 16'h7e_5a;
        message[46] = 16'h7f_69;
        message[47] = 16'h80_76;
        message[48] = 16'h81_80;
        message[49] = 16'h82_88;
        message[50] = 16'h83_8f;
        message[51] = 16'h84_96;
        message[52] = 16'h85_a3;
        message[53] = 16'h86_af;
        message[54] = 16'h87_c4;
        message[55] = 16'h88_d7;
        message[56] = 16'h89_e8;
```

```
    //AGC and AEC
    message[57] = 16'h13_e0;  //COM8, disable AGC / AEC
    message[58] = 16'h00_00;  //set gain reg to 0 for AGC
    message[59] = 16'h10_00;  //set ARCJ reg to 0
    message[60] = 16'h0d_40;  //magic reserved bit for COM4
    message[61] = 16'h14_18;  //COM9, 4x gain + magic bit
    message[62] = 16'ha5_05;  // BD50MAX
    message[63] = 16'hab_07;  //DB60MAX
    message[64] = 16'h24_95;  //AGC upper limit
    message[65] = 16'h25_33;  //AGC lower limit
    message[66] = 16'h26_e3;  //AGC/AEC fast mode op region
    message[67] = 16'h9f_78;  //HAECC1
    message[68] = 16'ha0_68;  //HAECC2
    message[69] = 16'ha1_03;  //magic
    message[70] = 16'ha6_d8;  //HAECC3
    message[71] = 16'ha7_d8;  //HAECC4
    message[72] = 16'ha8_f0;  //HAECC5
    message[73] = 16'ha9_90;  //HAECC6
    message[74] = 16'haa_94;  //HAECC7
    message[75] = 16'h13_e5;  //COM8, enable AGC / AEC
    message[76] = 16'h1E_23;  //Mirror Image
    message[77] = 16'h69_06;  //gain of RGB(manually adjusted)
end

//register operations
always @(posedge clk_50, negedge rst_n) begin
    if (!rst_n) begin
        state_q <= 0;
        led_q <= 0;
        delay_q <= 0;
        start_delay_q <= 0;
        message_index_q <= 0;
        pixel_q <= 0;

        sccb_state_q <= 0;
        addr_q <= 0;
        data_q <= 0;
    end else begin
        state_q <= state_d;
        led_q <= led_d;
        delay_q <= delay_d;
        start_delay_q <= start_delay_d;
        message_index_q <= message_index_d;
        pclk_1 <= cmos_pclk;
```

```
            pclk_2 <= pclk_1;
            href_1 <= cmos_href;
            href_2 <= href_1;
            vsync_1 <= cmos_vsync;
            vsync_2 <= vsync_1;
            pixel_q <= pixel_d;

            sccb_state_q <= sccb_state_d;
            addr_q <= addr_d;
            data_q <= data_d;
        end
    end


    //FSM next-state logics
    always @* begin
        state_d = state_q;
        led_d = led_q;
        start = 0;
        stop = 0;
        wr_data = 0;
        start_delay_d = start_delay_q;
        delay_d = delay_q;
        delay_finish = 0;
        message_index_d = message_index_q;
        pixel_d = pixel_q;
        wr_en = 0;

        sccb_state_d = sccb_state_q;
        addr_d = addr_q;
        data_d = data_q;

        //delay logic
        if (start_delay_q) delay_d = delay_q + 1'b1;
        if(delay_q[17] && message_index_q!=(MSG_INDEX+1) &&
(state_q!=start_sccb))  begin  //delay between SCCB transmissions (0.66ms)
            delay_finish = 1;
            start_delay_d = 0;
            delay_d = 0;
        end
             else if((delay_q[27] && message_index_q==(MSG_INDEX+1)) ||
(delay_q[27] && state_q==start_sccb)) begin //delay BEFORE SCCB transmission,
AFTER SCCB transmission, and BEFORE retrieving pixel data from camera (0.67s)
            delay_finish = 1;
```

```
            start_delay_d = 0;
            delay_d = 0;
        end

        case (state_q)

            ////////Begin: Setting register values of the camera via
SCCB//////////

            idle:
            if (delay_finish) begin  //idle for 0.6s to start-up the camera
                state_d = start_sccb;
                start_delay_d = 0;
            end else start_delay_d = 1;

            start_sccb: begin  //start of SCCB transmission
                start   = 1;
                wr_data = 8'h42;  //slave address of OV7670 for write
                state_d = write_address;
            end
            write_address:
            if (ack == 2'b11) begin
                wr_data = message[message_index_q][15:8];  //write address
                state_d = write_data;
            end
            write_data:
            if (ack == 2'b11) begin
                wr_data = message[message_index_q][7:0];  //write data
                state_d = digest_loop;
            end
            digest_loop:
            if (ack == 2'b11) begin  //stop sccb transmission
                stop = 1;
                start_delay_d = 1;
                message_index_d = message_index_q + 1'b1;
                state_d = delay;
            end
            delay: begin
                if (message_index_q == (MSG_INDEX + 1) && delay_finish) begin
                    state_d=vsync_fedge; //if all messages are already
digested, proceed to retrieving camera pixel data
                    led_d = 4'b0110;
                end else if (state == 0 && delay_finish)
```

```
                    state_d=start_sccb; //small delay before next SCCB
transmission(if all messages are not yet digested)
            end



            ///////////////Begin: Retrieving Pixel Data from Camera to be
Stored to SDRAM/////////////////

            vsync_fedge:
            if (vsync_1 == 0 && vsync_2 == 1) begin
                state_d=byte1; //vsync falling edge means new frame is incoming
                led_d = 4'b1000;
            end else if (cmos_vsync == 1) begin
                led_d = 4'b0111;
            end
            byte1:
            if(pclk_1==1 && pclk_2==0 && href_1==1 && href_2==1) begin //rising
edge of pclk means new pixel data(first byte of 16-bit pixel RGB565) is
available at output
                pixel_d[15:8] = cmos_db;
                state_d = byte2;
                led_d = 4'b1001;
            end else if (vsync_1 == 1 && vsync_2 == 1) begin
                state_d = vsync_fedge;
            end
            byte2:
            if(pclk_1==1 && pclk_2==0 && href_1==1 && href_2==1) begin //rising
edge of pclk means new pixel data(second byte of 16-bit pixel RGB565) is
available at output
                pixel_d[7:0] = cmos_db;
                state_d = fifo_write;
                led_d = 4'b1010;
            end else if (vsync_1 == 1 && vsync_2 == 1) begin
                state_d = vsync_fedge;
            end
            fifo_write: begin //write the 16-bit data to asynchronous fifo to
be retrieved later by SDRAM
                wr_en   = 1;
                state_d = byte1;
                // if (full) led_d = 4'b1001;  //debugging led
            end
            default: state_d = idle;
        endcase
```

```
    end


    assign cmos_pwdn  = 0;
    assign cmos_rst_n = 1;
    assign led[3:0]   = led_q;
    assign led[7:4]   = state_q;
    //module instantiations
    i2c_top #(
        .freq(50_000)  // ----------------------- [change
    ) m0 (
        .clk(clk_50),
        .rst_n(rst_n),
        .start(start),
        .stop(stop),
        .wr_data(wr_data),
        .rd_tick(rd_tick), //ticks when read data from servant is ready,data
will be taken from rd_data
        .ack(ack), //ack[1] ticks at the ack bit[9th bit],ack[0] asserts when
ack bit is ACK,else NACK
        .rd_data(rd_data),
        .scl(cmos_scl),
        .sda(cmos_sda),
        .state(state)
    );

    assign cmos_xclk = clk_24;


    asyn_fifo #(
        .DATA_WIDTH(16),
        .FIFO_DEPTH_WIDTH(10)
    ) m2  //1024x16 FIFO mem
    (
        .rst_n(rst_n),
        .clk_write(clk_50),
        .clk_read(clk_50),  //clock input from both domains
        .write(wr_en),
        .read(rd_en),
        .data_write(pixel_q),  //input FROM write clock domain
        .data_read(dout),  //output TO read clock domain
        .full(full),
        .empty(), //full=sync to write domain clk , empty=sync to read domain
clk
```

```
        .data_count_r(data_count_r) //asserted if fifo is equal or more than
than half of its max capacity
    );

endmodule
```

I2c_top.v

```
Unset
`timescale 1ns / 1ps

module i2c_top  //works on both i2c and SCCB mode(no pullups resistors needed)
#(
    parameter freq = 50_000
) (
    input wire clk,
    rst_n,
    input wire start,
    stop,
    input wire [7:0] wr_data,
    output reg rd_tick, //ticks when read data from servant is ready,data will
be taken from rd_data
    output reg[1:0] ack, //ack[1] ticks at the ack bit[9th bit],ack[0] asserts
when ack bit is ACK,else NACK
    output wire [7:0] rd_data,
    inout scl,
    sda,
    output wire [3:0] state
);

    /*
        i2c_top #(.freq(100_000)) m0
       (
              .clk(clk),
              .rst_n(rst_n),
              .start(start),
              .stop(stop),
              .wr_data(wr_data),
```

```
            .rd_tick(rd_tick), //ticks when read data from servant is
ready,data will be taken from rd_data
            .ack(ack), //ack[1] ticks at the ack bit[9th bit],ack[0] asserts
when ack bit is ACK,else NACK
            .rd_data(rd_data),
            .scl(scl),
            .sda(sda)
    );
       */

    localparam full= (100_000_000)/(2*freq),
                               half= full/2,
                               counter_width=log2(
        full
    );

    function integer log2(
        input integer n
    );  //automatically determines the width needed by counter
        integer i;
        begin
            log2 = 1;
            for (i = 0; 2 ** i < n; i = i + 1) log2 = i + 1;
        end
    endfunction

    //FSM state declarations
    localparam[3:0] idle=0,
                                              starting=1,
                                              packet=2,
                                              ack_servant=3,
                                              renew_data=4,
                                              read=5,
                                              ack_master=6,
                                              stop_1=7,
                                              stop_2=8;
    reg [3:0] state_q = idle, state_d;
    reg start_q = 0, start_d;
    reg [3:0] idx_q = 0, idx_d;
    reg [8:0] wr_data_q = 0, wr_data_d;
    reg [7:0] rd_data_q, rd_data_d;
    reg scl_q = 0, scl_d;
    reg sda_q = 0, sda_d;
    reg [counter_width-1:0] counter_q = 0, counter_d;
```

```
wire scl_lo, scl_hi;

//register operations
always @(posedge clk, negedge rst_n) begin
    if (!rst_n) begin
        state_q <= idle;
        start_q <= 0;
        idx_q <= 0;
        wr_data_q <= 0;
        scl_q <= 0;
        sda_q <= 0;
        counter_q <= 0;
        rd_data_q <= 0;
    end else begin
        state_q <= state_d;
        start_q <= start_d;
        idx_q <= idx_d;
        wr_data_q <= wr_data_d;
        scl_q <= scl_d;
        sda_q <= sda_d;
        counter_q <= counter_d;
        rd_data_q <= rd_data_d;
    end
end


//free-running clk, freq depends on parameter "freq"
always @* begin
    counter_d = counter_q + 1;
    scl_d = scl_q;
    if (state_q == idle || state_q == starting) scl_d = 1'b1;
    else if (counter_q == full) begin
        counter_d = 0;
        scl_d = (scl_q == 0) ? 1'b1 : 1'b0;
    end
end


//FSM next-state logic
always @* begin
    state_d = state_q;
    start_d = start_q;
    idx_d = idx_q;
    wr_data_d = wr_data_q;
```

```
        rd_data_d = rd_data_q;
        sda_d = sda_q;
        ack = 0;
        rd_tick = 0;
        case (state_q)
            idle: begin  //wait for the "start" to assert
                sda_d = 1'b1;
                if (start == 1'b1) begin
                    wr_data_d = {
                        wr_data, 1'b1
                    };  //the last 1'b1 is for the ACK coming from the
servant("1" means high impedance or "reading")
                    start_d= (wr_data[0])? 1:0; // if last bit(R/W bit) is
one:read after writing servant address, else write again after writing servant
address
                    idx_d=4'd8; //index to be used on transmitting the wr_data
serially(MSB first)
                    state_d = starting;
                end
            end

            starting:
            if(scl_hi) begin //start command, change sda to low while scl is
high
                sda_d   = 0;
                state_d = packet;
            end

            packet:
            if (scl_lo) begin  //transmit wr_data serially(MSB first)
                sda_d = (wr_data_q[idx_q] == 0) ? 0 : 1'b1;
                idx_d = idx_q - 1;
                if (idx_q == 0) begin
                    state_d = ack_servant;
                    idx_d   = 0;
                end
            end

            ack_servant:
            if (scl_hi) begin  //wait for ACK bit response(9th bit) from
servant
                ack[1] = 1;
                ack[0] = !sda;
                start_d = start;
```

```
                wr_data_d = {wr_data, 1'b1};
                if (stop)
                    state_d=stop_1; //master can forcefully stops the
transaction(even if response is either NACK or ACK)
                else if(start_q==1 && wr_data_q[1]) begin //if repeated start
happened before, choose if its read or write based on wr_data[1] (R/W bit)
                    start_d = 0;
                    idx_d   = 7;
                    state_d = read;
                end else state_d = renew_data;
            end

            renew_data: begin //new byte is comin(packets of data after the
slave address)
                idx_d = 8;
                if (start_q) begin  //if master wants a repeated start
                    state_d = starting;
                end else
                    state_d = packet;  //if master wants to continue writing
            end

            read:
            if (scl_hi) begin  //read data from slave(MSB first)
                rd_data_d[idx_q] = sda;
                idx_d = idx_q - 1;
                if (idx_q == 0) begin
                    state_d = ack_master;
                    idx_d   = 0;
                end
            end

            ack_master:
            if(scl_lo) begin //master must ACK after receiving data from
servant
                //sda_d=!sda_q; //acknowledge:1
                sda_d = 1;  //dont acknowledge if using SCCB
                if (sda_q == 0 || sda_d == 1) begin
                    rd_tick = 1;
                    idx_d   = 7;
                    if (stop)
                        state_d=stop_1; //after receiving data, master can opt
to stop
                    else if(start) begin //after receiving data, master can opt
to repeat start
```

```
                        start_d = 1;
                        state_d = starting;
                    end else
                        state_d=read; ///after receiving data, master can also
just continue receving more data
                end
            end
            stop_1:
            if (scl_lo) begin
                sda_d   = 1'b0;
                state_d = stop_2;
            end
            stop_2:
            if (scl_hi) begin
                sda_d   = 1'b1;
                state_d = idle;
            end
            default: state_d = idle;
        endcase
    end

    /*
        //i2c output logic
        assign scl=scl_q? 1'bz:0; //bidiectional logic for pull-up scl
        assign sda=sda_q? 1'bz:0; //bidirectional logic for pull-up scl
        */

    //sccb output logic
    assign scl = scl_q ? 1'b1 : 0;  //sccb scl does not scl
    assign sda=(state_q==read || state_q==ack_servant)? 1'bz : sda_q?1'b1:0;
//sccb scl does not scl

    assign scl_hi= scl_q==1'b1 && counter_q==half && scl==1'b1; //scl is on the
middle of a high(1) bit
    assign scl_lo= scl_q==1'b0 && counter_q==half; //scl is on the middle of a
low(0) bit
    assign rd_data = rd_data_q;
    assign state = state_q;

endmodule
```

Top_module.sv

```
Unset
module top_module (
    // Driver I/O
    input logic clk,  // 50 MHz
    input logic clk_24,  // 24 MHz
    input logic reset,
    output logic [31:0] readdata,
    input logic read,
    input chipselect,
    // VGA
    output logic [7:0] VGA_R,
    VGA_G,
    VGA_B,
    output logic VGA_CLK,
    VGA_HS,
    VGA_VS,
    VGA_BLANK_N,
    output logic VGA_SYNC_N,
    // Debugging
    output [9:0] LED,
    // Camera
    output wire SCLK,  // GPIO_0[0]
    inout logic SDA,  // GPIO_0[1]
    input logic VS,  // GPIO_0[2]
    input logic HREF,  // GPIO_0[3]
    input logic PCLK,  // GPIO_0[4]
    output logic MCLK,  // GPIO_0[5]          25MHz
    input logic [7:0] DATA,  // GPIO_O[13:6]
    output logic CAM_RESET,  // GPIO_0[14]
    output wire PWDN

);

    ////////////////////////////////////////////////////////////
    wire read_en;

    always_ff @(posedge clk) begin
        if (data_count_r >= 128) begin // start reading when there are 128
pixels in the FIFO
            read_en <= 1;
        end else begin
            read_en <= 0;
        end
```

```
    end

    ////////////////////////////////////////////////////////////////
    // Asyn FIFO in Camera
    wire [ 9:0] data_count_r;
    wire [15:0] dout;  // RGB565 format
    wire [ 9:0] tmp_LED;

    // Camera module (camera_interface.sv), 24MHz
    camera_interface m0 (
        .clk_50(clk),  // -------------------------- [ clk frequency change to
50MHz
        .clk_24(clk_24),
        .rst_n(~reset),  // rst_n is active-low, reset is active-high
        //asyn_fifo IO
        .rd_en(read_en),
        .data_count_r(data_count_r),
        .dout(dout),
        //camera pinouts
        .cmos_pclk(PCLK),
        .cmos_href(HREF),
        .cmos_vsync(VS),
        .cmos_db(DATA),
        .cmos_sda(SDA),
        .cmos_scl(SCLK),
        .cmos_rst_n(CAM_RESET),  // CAM_RESET is active-low
        .cmos_pwdn(PWDN),
        .cmos_xclk(MCLK),
        //Debugging
        .led(tmp_LED)
    );


    ////////////////////////////////////////////////////////////////
    // VGA module
    logic [10:0]  hcount;
    logic [9:0]  vcount;
    logic    endOfField;
    logic clk_25;
    clk_div clk_div_0 (
        .clk_ref(clk),
        .clk_out(clk_25)
    );
```

```
vga_counter counter_0 (
    .clk50(clk),
    .reset(reset),
    .enable(1),  // ---------------------------- [Change]
    .hcount(hcount),
    .vcount(vcount),
    .VGA_CLK(VGA_CLK),  // 25 MHz
    .VGA_HS(VGA_HS),
    .VGA_VS(VGA_VS),
    .VGA_BLANK_N(VGA_BLANK_N),
    .VGA_SYNC_N(VGA_SYNC_N),
    .endOfField(endOfField)
);

////////////////////////////////////////////////////////////////////
logic [7:0] VGA_R_8, VGA_G_8, VGA_B_8;

RGB565_to_RGB888 rgb565_to_rgb888 (
    .r_5(dout[15:11]),
    .g_6(dout[10:5]),
    .b_5(dout[4:0]),
    .r_8(VGA_R_8),
    .g_8(VGA_G_8),
    .b_8(VGA_B_8)
);


always_ff @(posedge clk) begin
    // VGA display
    if (VGA_BLANK_N) begin  // RGB565
        VGA_R <= VGA_R_8;
        VGA_G <= VGA_G_8;
        VGA_B <= VGA_B_8;
        // Transfer to HPS
        if (chipselect && read) readdata <= {endOfField, 15'd0, dout};
    end
end

////////////////////////////////////////////////////////////////////
// assign LED[0] = read_en;
// assign LED[1] = 1'b0;
// assign LED[2] = 1'b1;
assign LED[7:0] = tmp_LED[7:0];
```

```
endmodule



module clk_div (
    input  logic clk_ref,
    output logic clk_out
);

    reg [27:0] counter = 28'd0;
    parameter DIV = 28'd2;

    always @(posedge clk_ref) begin
        counter <= counter + 28'd1;
        if (counter >= (DIV - 1)) counter <= 28'd0;
        clk_out <= (counter < DIV / 2) ? 1'b1 : 1'b0;
    end
endmodule


module RGB565_to_RGB888 (
  input logic [4:0] r_5, b_5;
  input logic [5:0] g_6
  output logic [7:0] r_8, g_8, b_8;
);

  always_comb begin
    r_8 = {r_5, r_5[2:0]};
    g_8 = {g_6, g_6[1:0]};
    b_8 = {b_5, b_5[2:0]};
  end

endmodule
```

Vga_counter.sv

```
Unset
module vga_counter (
    input  logic        clk50,
    reset,
    enable,  // ------------------------------ [Change]
    output logic [10:0] hcount,  // hcount[10:1] is pixel column
```

```
    output logic [ 9:0] vcount,  // vcount[9:0] is pixel row
    output logic        VGA_CLK,
    VGA_HS,
    VGA_VS,
    VGA_BLANK_N,
    VGA_SYNC_N,
    endOfField
);

    /*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *             _________________              ________
 * ___________|    Video      |_____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _______________________      _____________
 * |____|       VGA_HS          |____|
 */
    // Parameters for hcount
    parameter       HACTIVE      = 11'd 1280,
                            HFRONT_PORCH = 11'd 32,
                            HSYNC        = 11'd 192,
                            HBACK_PORCH  = 11'd 96,
                            HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                                   HBACK_PORCH; // 1600

    // Parameters for vcount
    parameter       VACTIVE      = 10'd 480,
                            VFRONT_PORCH = 10'd 10,
                            VSYNC        = 10'd 2,
                            VBACK_PORCH  = 10'd 33,
                            VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                                           VBACK_PORCH; // 525

    logic endOfLine;

    always_ff @(posedge clk50 or posedge reset)
        if (reset) hcount <= 0;
        else if (enable) begin  // enable counting
-------------------------------- [Change]
            if (endOfLine) hcount <= 0;
```

```
            else hcount <= hcount + 11'd1;
        end

    assign endOfLine = hcount == HTOTAL - 1;

    //logic endOfField;

    always_ff @(posedge clk50 or posedge reset)
        if (reset) vcount <= 0;
        else if (endOfLine)
            if (endOfField) vcount <= 0;
            else vcount <= vcount + 10'd1;

    assign endOfField = vcount == VTOTAL - 1;

    // Horizontal sync: from 0x520 to 0x5DF (0x57F)
    // 101 0010 0000 to 101 1101 1111
    assign VGA_HS = !((hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
    assign VGA_VS = !(vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

    assign VGA_SYN_N = 1'b0;  // For putting sync on the green signal; unused

    // Horizontal active: 0 to 1279     Vertical active: 0 to 479
    // 101 0000 0000   -  1280               01 1110 0000  -  480
    // 110 0011 1111   -  1599               10 0000 1100  -  524
    assign VGA_BLANK_N = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                         !( vcount[9] | (vcount[8:5] == 4'b1111) );

    /* VGA_CLK is 25 MHz
       *             __    __    __
       * clk50    __|  |__|  |__|
       *
       *             _____       __
       * hcount[0]__|     |_____|
       */
    assign VGA_CLK = hcount[0];  // 25 MHz clock: rising edge sensitive

endmodule
```

Sender.c

C/C++

```
/*
 * Userspace program that communicates with the vga_ball device driver
 * through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

#include <stdio.h>
#include <stdlib.h>
#include "top_module.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <string.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>

#define VGA_BALL_MAX_X 239
#define VGA_BALL_MAX_Y 239
#define VGA_BALL_MIN_X 15
#define VGA_BALL_MIN_Y 15

#define SERVER_IP "127.0.0.1"  // replace with server IP address
#define PORT 9999

#define BUFFER_SIZE 640*480 + 1
#define READ_PIXELS 640*480

int vga_ball_fd;
int cam_fd;
int sockfd;

/* Read and print the background color */
void read_vga_ball()
{
    VGA_BALL_ARG vla;

    if (ioctl(vga_ball_fd, VGA_BALL_READ, &vla))
    {
```

```
        perror("ioctl(VGA_BALL_READ) failed");
        return;
    }
    printf("Ball position: %d %d\n", vla.position.x, vla.position.y);
    }

/* Set the background color */
void set_vga_ball(const VGA_BALL_ARG *arg)
{
    VGA_BALL_ARG vla = *arg;


    if (ioctl(vga_ball_fd, VGA_BALL_WRITE, &vla))
    {
        perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
        return;
    }
}
void read_frame(char* buf)
{
    unsigned int info;
    unsigned int hcount, vcount;
    unsigned int address;
    unsigned char pixel;

    while (1) {
        if (ioctl(cam_fd, CAMERA_READ, &info)) {
            perror("ioctl(CAMERA_READ) failed");
            return;
        }

        // Extract fields from the info
        hcount = (info >> 29) & 0x3FF;      // Extract bits 29-38 (10 bits for
hcount)
        vcount = (info >> 19) & 0x3FF;      // Extract bits 19-28 (10 bits for
vcount)
        pixel = info & 0xFF;                // Extract bits 0-7 for the pixel
value (dout)

        // Calculate the address
        address = vcount * 640 + hcount;

        if (address < READ_PIXELS) {
            buf[address] = pixel;
```

```
        }

        // Check for end of field
        if (info & (1 << 18)) {
            break; // Exit the loop when end of field is encountered
        }
    }
}


void send_frame(const char* buffer) {
    int bytes_sent = 0;
    int total_bytes = BUFFER_SIZE;

    while (bytes_sent < total_bytes) {
        int n = send(sockfd, buffer + bytes_sent, total_bytes - bytes_sent, 0);
        if (n < 0) {
            perror("send failed");
            exit(EXIT_FAILURE);
        }
        bytes_sent += n;
    }
    printf("Frame sent. Bytes sent: %d\n", bytes_sent);
}

int main()
{
    VGA_BALL_ARG vla;
    int i, j;

    static const char filename[] = "/dev/top_module";
    struct sockaddr_in server_addr;

    char buffer[BUFFER_SIZE] = {0};

    // Initialize TCP socket
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));

    // Filling server information
```

```
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
        perror("Invalid address / Address not supported");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Connect to the server
    if (connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) <
0) {
        perror("Connection failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("VGA ball Userspace program started\n");

    if ((vga_ball_fd = open(filename, O_RDWR)) == -1)
    {
        printf("here 1");
        fprintf(stderr, "could not open %s\n", filename);
        close(sockfd);
        return -1;
    }

    printf("initial state: ");
    read_vga_ball();
    i = 0;
    vla.position.x = 100;
    vla.position.y = 20;
    int x_dir = 0;
    int y_dir = 1;
    set_vga_ball(&vla);
    read_vga_ball();

    while (1)
    {
        // adjust ball settings
        if (x_dir)
        {
            vla.position.x = (vla.position.x - 1 + VGA_BALL_MAX_X) %
VGA_BALL_MAX_X;
            if (vla.position.x == VGA_BALL_MIN_X)
```

```
                x_dir = 0;
        }
        else
        {
            vla.position.x = (vla.position.x + 1) % VGA_BALL_MAX_X;
            if (vla.position.x == VGA_BALL_MAX_X - 1)
                x_dir = 1;
        }
        if (y_dir)
        {
            vla.position.y = (vla.position.y - 1 + VGA_BALL_MAX_Y) %
VGA_BALL_MAX_Y;
            if (vla.position.y == VGA_BALL_MIN_Y)
                y_dir = 0;
        }
        else
        {
            vla.position.y = (vla.position.y + 1) % VGA_BALL_MAX_Y;
            if (vla.position.y == VGA_BALL_MAX_Y - 1)
                y_dir = 1;
        }

        // system call
        read_vga_ball();
        set_vga_ball(&vla);

        read_frame(buffer);
        send_frame(buffer);

        usleep(200000);

        // increment
        i++;
    }

    // Cleanup
    close(cam_fd);
    close(vga_ball_fd);
    close(sockfd);
    printf("VGA BALL Userspace program terminating\n");
    return 0;
}
```

Client.cpp

```
Unset
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <string>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#define PAYLOAD_SIZE 8
#define BUFFER_SIZE 1 << 30
#define MSG_LEN 230564
using namespace std;

// Server

int server_port = 9999;
string server_addr = "127.0.0.1";
int client_socket;
char *received_data;
char *buffer;

int open_connection()
{
    // Create a socket client
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(server_port);
    inet_pton(AF_INET, server_addr.c_str(), &(server_address.sin_addr));
    if (connect(client_socket, (struct sockaddr *)&server_address,
sizeof(server_address)) != -1)
    {
        printf("Connected to the server\n");
        received_data = new char[MSG_LEN];
        memset(received_data, 0, sizeof(received_data));
        buffer = new char[BUFFER_SIZE];
        memset(buffer, 0, sizeof(buffer));
        return 1;
    }
```

```
    else
    {
        printf("Failed to connect to the server\n");
        return 0;
    }
}

int receive_frame()
{
    int count;
    cv::namedWindow("Client", cv::WINDOW_AUTOSIZE);
    while ((count = recv(client_socket, buffer, BUFFER_SIZE, 0)) > 0)
    {
        printf("Received %d bytes\n", count);

        cv::Mat rawData(1, count, CV_8UC1, (void *)buffer);
        cv::Mat decoded_frame = cv::imdecode(rawData, cv::IMREAD_COLOR);
        cv::imshow("Client", decoded_frame);
        cv::waitKey(25);
    }
    return 1;
}
```

Client.h

```
Unset
#ifndef CLIENT
#define CLIENT

int open_connection();
int receive_frame();
void test();

#endif
```

CMakeLists.txt

```
Unset
cmake_minimum_required(VERSION 3.5)
project (host)

set(SOURCES
client.cpp
main.cpp
)

find_package (OpenCV REQUIRED)
add_executable(host ${SOURCES})
# include_directories ("/usr/include/opencv4/")
include_directories( ${OpenCV_INCLUDE_DIRS} )
target_link_libraries( host ${OpenCV_LIBS} )
```

main.cpp

```
Unset
#include <stdio.h>
#include <string.h>
#include "client.h"

extern char *received_data;

int main()
{
    if (open_connection())
    {
        receive_frame();
    }
    return 0;
}
```

server.py

Unset

```
import cv2
import socket
import threading
import queue

# Video file
frame_rate = 30  # fps
video_file = "big_buck_bunny_240p_30mb.mp4"

class Server:
    def __init__(self) -> None:
        # port
        self.server_port = 9999
        # socket
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.bind(("127.0.0.1", self.server_port))
        self.server_socket.listen(10)
        self.client_socket_list = []
        self.frame_queue = queue.Queue(maxsize=10)
        print("Server running...")

    def accept_client(self):
        while True:
            client_socket, client_address = self.server_socket.accept()
            print(f"[*] Accepted connection from {client_address}")
            self.client_socket_list.append((client_socket, client_address[0]))

    def stream_video(self, video_file, frame_rate):
        video_capture = cv2.VideoCapture(video_file)
        if not video_capture.isOpened():
            print("Error: Could not open video file.")
            return

        while True:
            ret, frame = video_capture.read()
            if not ret:
                print("Failed to read frame; breaking loop...")
                break

            try:
                # serialize the frame
                serialized_frame = cv2.imencode(".jpg", frame)[1].tobytes()
                self.frame_queue.put(frame)  # Put the frame in the queue
                for client_socket, client_address in self.client_socket_list:
```

```
                    try:
                        client_socket.sendall(serialized_frame)
                    except Exception as e:
                        print(f"Error sending frame to {client_address}:
{str(e)}")
                        self.client_socket_list.remove((client_socket,
client_address))
                        client_socket.close()
            except Exception as e:
                print(f"Failed to encode or send frame: {str(e)}")

        video_capture.release()
        self.server_socket.close()

    def __del__(self):
        self.server_socket.close()

def display_video(frame_rate, frame_queue):
    while True:
        if not frame_queue.empty():
            frame = frame_queue.get()
            if frame is not None:
                cv2.imshow("Server Video", frame)
                if cv2.waitKey(int(1000 / frame_rate)) & 0xFF == ord('q'):
                    break
            else:
                print("Received an empty frame.")
    cv2.destroyAllWindows()

my_server = Server()
# launch threads
thread1 = threading.Thread(target=my_server.accept_client)
thread2 = threading.Thread(target=my_server.stream_video, args=(video_file,
frame_rate))

thread1.start()
thread2.start()

# Run GUI in the main thread
display_video(frame_rate, my_server.frame_queue)
```