Embedded System
Final Project: Camera Control

Fatima Dantsoho, fd2508
Micheal Lanzano, mml2238

I. Introduction

The main objective of this project is to design a digital camera system using OV7670 camera module to illustrate the main concepts related to design with SystemVerilog and DE1 SoC Board, video format, CMOS camera, basic image processing such as edge detection in embedded programming of microcontrollers.

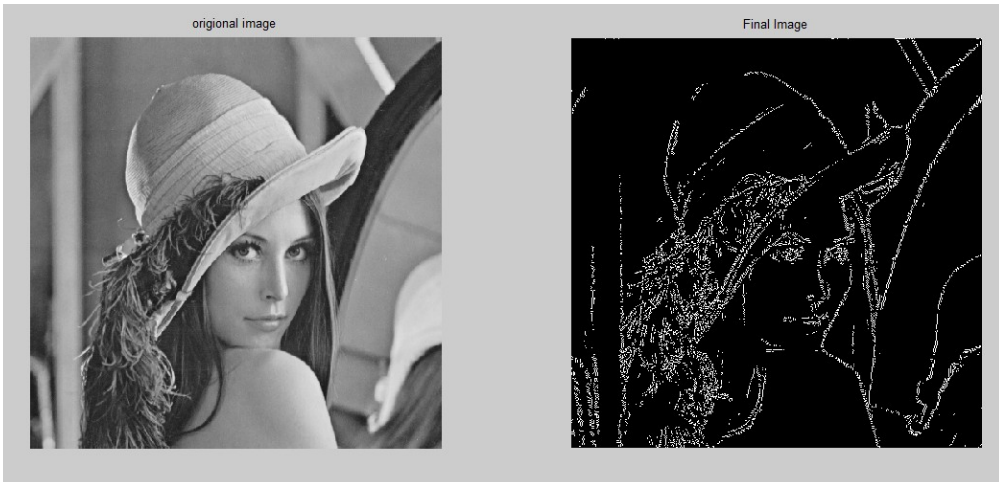


Figure 1 - grayscale and edge picture

A. System Architecture

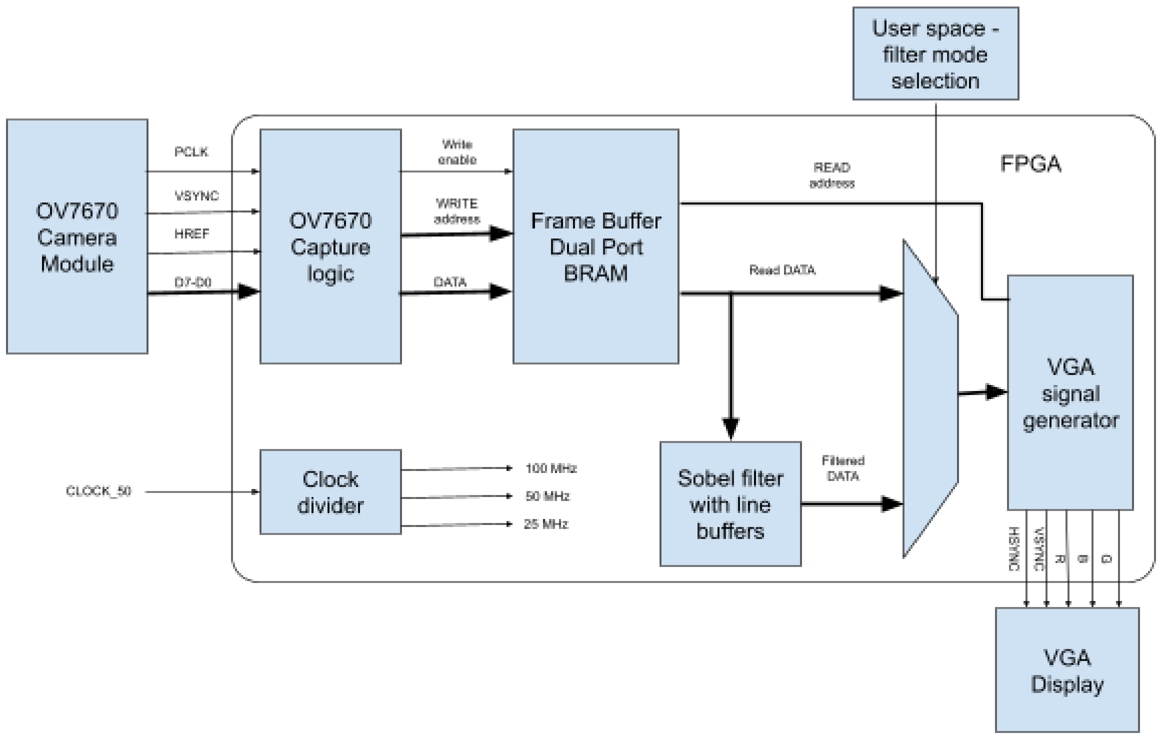


Figure 2 - System Block Diagram

In this application, real time video image processing on the FPGA, scene irradiance data collected by an OV7670 CMOS will be streamed to the FPGA target board where it will be processed by convolution filters and finally output as a signal to a VGA display. The software will generate the filtering mode - where it can alternate between normal mode and filtered mode. The user will interact with software with the keyboard. Avalon buses will be used for transferring video streams from the camera and to the VGA display.

Our project performs 3 main operations:

1. Interface with camera
2. Apply a filter to the video stream
3. Display the video on VGA output

We connect the OV7670 camera module to the FPGA, retrieve video frames from the camera module, store it in a DUAL port BRAM temporarily and feed the data to the sobel filter and input from the user space is used to choose the video output mode between the filtered and normal video.

II. Hardware

A. OV7670 camera module

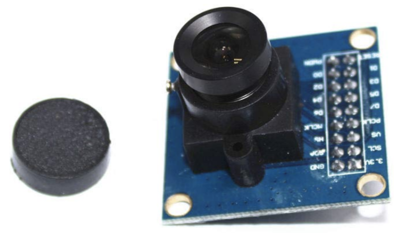


Figure 3 - OV7670 CMOS camera module

The OV7670 camera module is a low cost image sensor and a DSP which works at a maximum of 30 frames per second i.e, 640 by 480 resolution, which is the same as that of the VGA. This is equivalent to 0.3 MegaPixels. The OV7670 camera header comes with a 9 by 2 header. The I/O signals of the camera module are listed below:

Pin	**Type**	**Description**
VDD	Power Supply	Power Supply
GND	Power Supply	Ground level
SDIOC	Input	SCCB clock
SDIOD	Inout	SCCB data
VSYNC	Output	Vertical synchronization
HREF	Output	Horizontal synchronization
PCLK	Output	Pixel clock
XCLK	Input	System clock
D0-D7	Output	Video parallel output
RESET	Input	Reset (active low)
PWDN	Input	Power down (Active high)

Table 1 - camera I/O pins and description

B. YCbCr

The pixels from the OV7670 camera are encoded in YCbCr (or YUV) 4:2:2 format. The luminance component (Y) is the amount of white light of a color, while Cb/U and Cr/V are the chroma components, which encode the blue and red levels relative to the luminance component. Y channel encodes the gray scale levels of the image and it's the easiest way to get a monochrome image from our camera module. OV7670 uses the YUV 4:2:2 format which arrives in the following manner:

N	**Byte**
1st	U0
2nd	Y0
3rd	V0

4th	Y1
5th	U2
6th	Y2
7th	V2
8th	Y3
…	…

Table 2 - YUV 422 pixel arrival sequence

The pixels are arranged as:

Pixel 0	Y0	U0	V0
Pixel 1	Y1	U0	V0
Pixel 2	Y2	U2	V2
Pixel 3	Y3	U2	V2
Pixel 4	Y4	U4	V4
Pixel 5	Y5	U4	V4

Table 3 - pixel arrangement

C. Signal Communication

The D0 to D7 pixel data are sampled at the rising edge of the pclk clock signal. The D0-1 pixels are sampled only when HREF is high. The beginning of a line starts at the rising edge of the HREF signal and ends at the falling edge of HREF. Because the default pixel format is YUV 4:2:2 a pixel is represented by two bytes.

The display_480p module controls the VSYNC, HSYNC, line and frame signals. The HSYNC and VSYNC signals give us a point of reference to know when pixel data of a frame are being transmitted from the OV7670 camera module. Our project is designed to display a 640 by 480 VGA frame. When HSYNC is high, 640 pixels are captured which is a line 480 lines compose a frame which are captured when VSYNC is low.

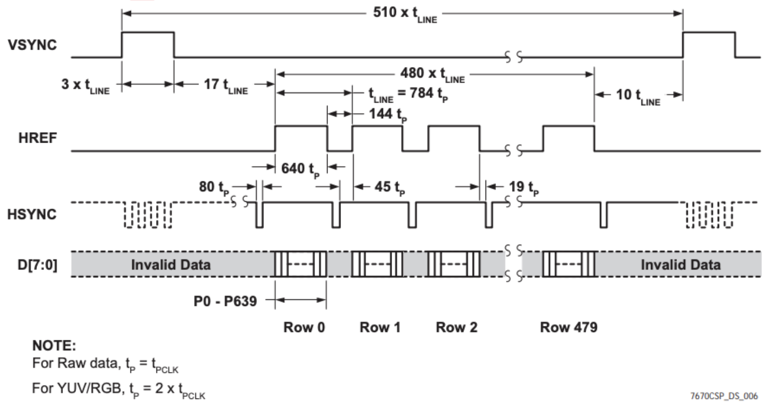


Figure 4 - VGA frame timing diagram

D. SCCB (Serial Camera Control Bus)

OV7670 has the ability to preprocess images before it leaves the camera module because of its internal DSP. The Serial Camera Control Bus allows us to access the DSP. If the SCCB is working properly, the OV7670 will answer with an ACK if it has been addressed properly
We used the De1-SoC board for this project. To write the System Verilog code and to synthesize and generate FPGA programming files, and program the FPGA, we use Quartus Software from Altera.

The De1-SoC board has an oscillator based clock signal generator that produces a 50MHz clock signal to the FPGA. We use the PLL divider available on the Cyclone V to produce five clocks using the Megawizard Plug-In Manager of Quartus II Software. These clocks have signals of 100MHz, 50MHz, 25MHz, 12.5 MHz and 130MHz respectively which were used in different aspects of the project. For the frame buffer, we used a 2-PORT BRAM with separate write and read clocks to accommodate the different clock signals that drive the camera module and VGA display.

E. Sobel Filter

A filter kernel is an n x n array of coefficients that are used to calculate the sum of products in the neighborhood of the target pixel. A very common and useful type of image filter is the gaussian. A 2D gaussian is an n x n matrix whose elements take the value of the gaussian probability distribution as a function of its distance from the center of the kernel. We perform our Sobel filtering by doing a 2D convolution of our video frames with two kernels of dimension 3x3 in both X and Y directions.

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

Figure 5 - Y and X convolution kernel

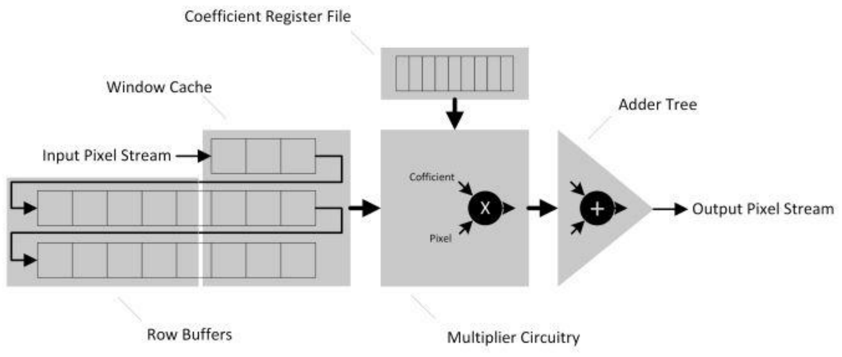
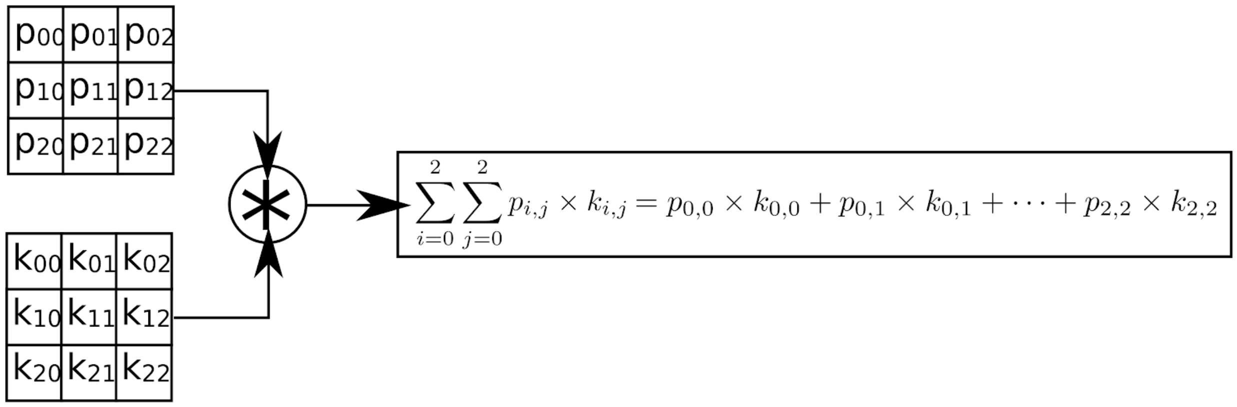


Figure 6 - Image filter block diagram



$$\sum_{i=0}^{2}\sum_{j=0}^{2} p_{i,j} \times k_{i,j} = p_{0,0} \times k_{0,0} + p_{0,1} \times k_{0,1} + \cdots + p_{2,2} \times k_{2,2}$$

Figure 7 - Convolution operation for a 3x3 kernel

As seen in Figure 5, we implemented 2 line buffers of size 800 which corresponds to the width of the VGA monitor. OV7670 camera module transfers images in 1D stream of bits so we use these buffers to stream the bits for the kernel convolution of 3x3 dimension.

III. Software

Since the sobel convolution module we implemented has a predetermined kernel, the only thing our software does is switch vga output from unfiltered video to video with edge detection applied. When a user presses the enter key, a get_filter() method in user space initiates an ioctl() call that reads a register in kernel space. The program prints the output mode corresponding to the register entry and calls the set_filter() method to make another call to ioctl() to cycle to the next output mode

IV. Discussion

With 4 different clock domains (FPGA,VGA, camera, and SCCB (I2C)) maintaining system stability and synchronizing the modules was a challenge. Further, as several modules took more than a single cycle to complete, these pipelines added further complexity to our project in the temporal domain. While we anticipated this conceptually, the reality of it was a lot to deal with. As a result we had to scale back our ambitions on the project. But by that same token it provided a tremendous learning opportunity that will be valuable going forward.

V. References

[1] *Web.Mit.Edu*, 2022, http://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf

[2] Guo, Yanhui, and Amira Ashour. *Neutrosophic Set In Medical Image Analysis*.

[3] Campos, Nelson. "2D Convolution In Hardware". *Sistenix.Com*, 2022, https://sistenix.com/sobel.html.

[4] "Digital Camera Project". *Dejazzer.Com*, 2022, http://www.dejazzer.com/eigenpi/digital_camera/digital_camera.html.

[5] “Gradient Filter Implementation on an FPGA - Part 1 Interfacing an FPGA with a Camera.” *Blog - FPGA - element14 Community*, https://community.element14.com/technologies/fpga-group/b/blog/posts/gradient-filter-implementation-on-an-fpga---part-1-interfacing-an-fpga-with-a-camera.

VI. Appendix

TOP MODULE:

```
module ov7670_vga_fb(CLOCK_50, HREF,VS, SDA, SCLK, clk_25, v_clk,
incoming_data, pclk,R,G,B,VGA_VS,VGA_HS, VGA_BLANK_N, LEDR,reset,
write, writedata, chipselect, address);
     input CLOCK_50;
     output [9:0] LEDR;
     assign LEDR[9:0] = 10'b0;

     input      reset;
     input [7:0]     writedata;
     input           write;
     input           chipselect;
     input [7:0]     address;

     output reg [7:0] R;
     output reg [7:0] G;
     output reg [7:0] B;                  //VGA
     output reg VGA_HS,VGA_VS;
     output wire VGA_BLANK_N;
     output wire v_clk;

     inout SDA;
     output SCLK;
     input VS,HREF;
     input pclk;                                  //OV7670
     input [7:0] incoming_data;
     output wire clk_25;
     reg [7:0] red, green , blue;
     wire [15:0] tx;
     wire [3:0] sccbState;

     assign v_clk = clk_25;


      // framebuffer (FB)
    localparam FB_WIDTH  = 640;
    localparam FB_HEIGHT = 480;
    localparam FB_PIXELS = FB_WIDTH * FB_HEIGHT;
    localparam FB_ADDRW  = $clog2(FB_PIXELS);
    localparam FB_DATAW  = 8;  // color bits
      localparam CORDW = 16;
```

```
 reg fb_we;
 reg [FB_ADDRW-1:0] fb_addr_write, fb_addr_read;
 reg [FB_DATAW-1:0] fb_colr_write;
 reg [FB_DATAW-1:0] fb_colr_read;


wire clk_100, clk_50, clk_12, clk_130, clk_150;

   clk_div clkdiv0 (
             .refclk(CLOCK_50),
             .rst(0),
             .outclk_0(clk_100),
             .outclk_1(clk_50),
             .outclk_2(clk_25),
             .outclk_3(clk_12),
             .outclk_4(clk_130),
             .outclk_5(clk_150)
        );

 bram_sdp #(
     .WIDTH(FB_DATAW),
     .DEPTH(FB_PIXELS)
 ) bram_inst (
     .clk_write(pclk),
     .clk_read(v_clk),
     .we(fb_we),
     .addr_write(fb_addr_write),
     .addr_read(fb_addr_read),
     .data_in(fb_colr_write),
     .data_out(fb_colr_read)
 );

 reg [1:0] state = 0;
   reg pixel_valid = 0;

 always @(posedge pclk) begin
     case (state)
        0:
             if (VS & conf_done)
                  state <= 1;
        1:
             if (~VS) begin
                  fb_we <= 1;
                  fb_addr_write <= 0;
```

```
                    fb_colr_write <= incoming_data;
                    state <= 2;
                end
            2:
                if (HREF) begin
                    pixel_valid <= ~pixel_valid;
                    if (pixel_valid) begin
                    fb_addr_write <= fb_addr_write + 1;
                    fb_colr_write <= incoming_data;
                end
                else if (VS)
                    state <= 1;
        endcase
    end

    wire paint;  // which area of the framebuffer should we paint?
    assign paint = (sy >= 0 && sy < (FB_HEIGHT) && sx >= 0 && sx <
(FB_WIDTH));

    always @(posedge v_clk) begin
        if (frame) begin  // reset address at start of frame
            fb_addr_read <= 7;
                    filter_rst <= 1;

        end else if (paint) begin  // increment address in painting
area
            fb_addr_read <= fb_addr_read + 1;
         filter_rst <=0;
        end
    end

    // reading from BRAM takes one cycle: delay display signals to
match
    reg paint_p1, hsync_p1, vsync_p1;
    always @(posedge v_clk) begin
        paint_p1 <= paint;
        hsync_p1 <= hsync;
        vsync_p1 <= vsync;
    end

      wire [7:0] filtered_pxl;
      reg filter_mode ;
      reg filter_rst;

     always@(posedge pclk)begin
```

```
          case(address)
                  8'h00: filter_mode <= writedata;
                  8'h01: filter_mode <= writedata;
         default: filter_mode <= 8'b00;
       endcase

  end

   sobel filter
   (    .clock(v_clk),
        .reset(filter_rst),
        .inputPixel(fb_colr_read),
        .outputPixel(filtered_pxl)
        );

   // VGA output

   wire signed [CORDW-1:0] sx, sy;
   wire vsync, hsync, de, line;
   //frame sets fb_addr_read to 0
 wire frame;
   reg vga_rst = 0;

   display_480p #(.CORDW(CORDW)) display_inst (
     .clk_pix(v_clk),
     .rst(vga_rst),
     .hsync(hsync),
     .vsync(vsync),
     .de(de),
     .frame(frame),
     .line(line),
  .sx(sx),
     .sy(sy)
 );


 always @(posedge v_clk) begin
     VGA_HS <= hsync_p1;
     VGA_VS <= vsync_p1;
          if(paint_p1)
               begin

                    R <= red;
                    G <= green;
```

```
                        B <= blue;

                   end
              end

        always @(filter_mode) begin
              case(filter_mode)
                   1'b0: begin
                        red <= filtered_pxl;
                        green <= filtered_pxl;
                        blue <= filtered_pxl;
                   end
                   1'b1: begin
                        red <= fb_colr_read;
                        green <= fb_colr_read;
                        blue <= fb_colr_read;
                   end
              endcase
        end

       assign VGA_BLANK_N = de;

       reg [2:0]  strt = 3'd0;
       wire conf_done;

       always @( posedge clk_25)
            if ( &strt )
                 strt  <= strt;
            else
                 strt  <= strt + 1'h1;

       wire [7:0] SCCB_addr;
       wire [7:0] SCCB_data;

       camera_configure #(.CLK_FREQ (25000000)) camera_configure_0
       (
      .clk             ( clk_25                  ),
      .start           (( strt == 3'h6      )),
      .sioc       ( SCLK                    ),
      .siod       ( SDA                     ),
      .done       ( conf_done               ),
        .SCCB_addr (SCCB_addr               ),
        .SCCB_data (SCCB_data               )
        );
Endmodule
```

FRAME BUFFER:

```
module bram_sdp #(
    parameter WIDTH=8,
    parameter DEPTH=256,
    parameter INIT_F=""
    )
      (
    input wire logic clk_write,               // write clock (port a)
    input wire logic clk_read,                // read clock (port b)
    input wire logic we,                      // write enable (port
a)
    input wire logic [ADDRW-1:0] addr_write,  // write address (port
a)
    input wire logic [ADDRW-1:0] addr_read,   // read address (port
b)
    input wire logic [WIDTH-1:0] data_in,     // data in (port a)
    output     logic [WIDTH-1:0] data_out     // data out (port b)
    );

    localparam ADDRW = $clog2(DEPTH);
    logic [WIDTH-1:0] memory [DEPTH];
    logic [WIDTH-1:0] edge_filter;
    initial begin
        if (INIT_F != 0) begin
            $display("Loading memory init file '%s' into bram_sdp.",
INIT_F);
            $readmemh(INIT_F, memory);
        end
    end


    // Port A: Sync Write
    always_ff @(posedge clk_write) begin
        if (we) memory[addr_write] <= data_in;
    end

    // Port B: Sync Read
    always_ff @(posedge clk_read) begin
        data_out <= memory[addr_read];
    end
Endmodule
```

SOBEL FILTER:

```
module sobel #(parameter WORD_SIZE= 8, ROW_SIZE = 800 , BUFFER_SIZE =
3)
             (input logic clock,
                       input logic reset,
              input logic [WORD_SIZE-1:0] inputPixel,
              output logic [WORD_SIZE-1:0] outputPixel);

  //localparam BUFFER_SIZE=3;

  logic [BUFFER_SIZE-1:0] [WORD_SIZE-1:0] sliding [BUFFER_SIZE-1:0];
  sliding_window #(WORD_SIZE,BUFFER_SIZE) my_window(.*);

  logic [WORD_SIZE+1:0] gx1, gx2, gy1, gy2;

   always_ff @(posedge clock)
     if (reset) begin
       gx1 <= 0;
       gx2 <= 0;
       gy1 <= 0;
       gy2 <= 0;
     end
     else begin

     gx1 <= sliding[0][0] + sliding[2][0] + (sliding[1][0]<<1);
     gx2 <= sliding[0][2] + sliding[2][2] + (sliding[1][2]<<1);
     gy1 <= sliding[0][0] + sliding[0][2] + (sliding[0][1]<<1);
     gy2 <= sliding[2][0] + sliding[2][2] + (sliding[2][1]<<1);

     end

  logic [WORD_SIZE+1:0] gx, gy;
   always_comb begin
     if (gx1 > gx2) gx <= gx1-gx2;
      else gx <= gx2 - gx1;
     if (gy1 > gy2) gy <= gy1-gy2;
      else gy <= gy2-gy1;
   end

  logic [WORD_SIZE+2:0] g;

   always_comb g <= gy+gx;
   always_ff @(posedge clock)
     if (reset)
        outputPixel <= 0;
     else
       if (g[WORD_SIZE+2]) outputPixel <= {WORD_SIZE{1'b1}};
         else outputPixel <= g[WORD_SIZE+1:2];

endmodule
```

SLIDING WINDOW with LINE BUFFERS:

```
module sliding_window #(parameter WORD_SIZE=8, BUFFER_SIZE=3,
ROW_SIZE =798)
               (input  logic clock,
                           input logic reset,
                input logic [WORD_SIZE-1:0] inputPixel,
                output logic
[BUFFER_SIZE-1:0][WORD_SIZE-1:0]sliding[BUFFER_SIZE-1:0]);

  logic [(BUFFER_SIZE-1)*WORD_SIZE-1:0] buffer[ROW_SIZE-1:0];
  logic [$clog2(ROW_SIZE)-1:0] ptr;

  always_ff @(posedge clock)
    if(reset) begin
      ptr <=0;
      sliding[0][0] <= inputPixel;
      sliding[0][1] <= 0;
      sliding[0][2] <= 0;
      sliding[1][0] <= 0;
      sliding[1][1] <= 0;
      sliding[1][2] <= 0;
      sliding[2][0] <= 0;
      sliding[2][1] <= 0;
      sliding[2][2] <= 0;
    end
    else begin
      sliding[0][0] <= inputPixel;
      sliding[1][0] <= sliding[0][0];
      sliding[1][1] <= sliding[0][1];
      sliding[1][2] <= sliding[0][2];
      sliding[2][0] <= sliding[1][0];
      sliding[2][1] <= sliding[1][1];
      sliding[2][2] <= sliding[1][2];

      buffer[ptr] <= sliding[BUFFER_SIZE-1][BUFFER_SIZE-2:0];
      sliding[0][BUFFER_SIZE-1:1] <= buffer[ptr];
      if(ptr < ROW_SIZE-BUFFER_SIZE) ptr <= ptr + 1;
       else ptr <= 0;
    end
endmodule
```

VGA SIGNAL GENERATOR:

```
module display_480p #(
    CORDW=16,    // signed coordinate width (bits)
    H_RES=640,   // horizontal resolution (pixels)
    V_RES=480,   // vertical resolution (lines)
    H_FP=16,     // horizontal front porch
    H_SYNC=96,   // horizontal sync
    H_BP=48,     // horizontal back porch
    V_FP=4,     // vertical front porch
    V_SYNC=2,    // vertical sync
    V_BP=35,     // vertical back porch
    H_POL=0,     // horizontal sync polarity (0:neg, 1:pos)
    V_POL=0      // vertical sync polarity (0:neg, 1:pos)
    ) (
    input  wire logic clk_pix,  // pixel clock
    input  wire logic rst,  // reset in pixel clock domain
    output      logic hsync,    // horizontal sync
    output      logic vsync,    // vertical sync
// output       logic blank_n,
    output      logic de,    // data enable (low in blanking
interval)
    output      logic frame,    // high at start of frame
    output      logic line,     // high at start of line
    output      logic signed [CORDW-1:0] sx,  // horizontal position
    output      logic signed [CORDW-1:0] sy   // vertical position
    );

    // horizontal timings
    localparam signed H_STA  = 0 - H_FP - H_SYNC - H_BP;    //
horizontal start
    localparam signed HS_STA = H_STA + H_FP;                // sync
start
    localparam signed HS_END = HS_STA + H_SYNC;             // sync
end
    localparam signed HA_STA = 0;                           // active
start
```

```
    localparam signed HA_END = H_RES - 1;                   // active
end

    // vertical timings
    localparam signed V_STA  = 0 - V_FP - V_SYNC - V_BP;    //
vertical start
     localparam signed VS_STA = V_STA + V_FP;        // sync start
    localparam signed VS_END = VS_STA + V_SYNC;      // sync end
    localparam signed VA_STA = 0;                    // active start
    localparam signed VA_END = V_RES - 1;            // active end

  logic signed [CORDW-1:0] x, y;  // screen position

    // generate horizontal and vertical sync with correct polarity
    always_ff @(posedge clk_pix) begin
        hsync <= H_POL ? (x > HS_STA && x <= HS_END)
                       : ~(x > HS_STA && x <= HS_END);
        vsync <= V_POL ? (y > VS_STA && y <= VS_END)
                       : ~(y > VS_STA && y <= VS_END);
    end

    // control signals
    always_ff @(posedge clk_pix) begin
        de    <= (y >= VA_STA && x >= HA_STA);
        frame <= (y == V_STA  && x == H_STA);
        line  <= (x == H_STA);
        if (rst) begin
            de <= 0;
            frame <= 0;
            line <= 0;
        end
    end

    // calculate horizontal and vertical screen position
    always_ff @(posedge clk_pix) begin
        if (x == HA_END) begin  // last pixel on line?
            x <= H_STA;
            y <= (y == VA_END) ? V_STA : y + 1;// last line on
screen?
        end else begin
            x <= x + 1;
        end
        if (rst) begin
            x <= H_STA;
            y <= V_STA;
        end
    end

    // delay screen position to match sync and control signals
    always_ff @ (posedge clk_pix) begin
        sx <= x;
        sy <= y;
        if (rst) begin
            sx <= H_STA;
            sy <= V_STA;
        end
    end
endmodule
```