

Lab 5. Capturing the image from an NTSC Camera and displaying it live on a SVGA monitor

Objective

In this lab, you will capture the video data from an NTSC camera, using the Video Decoder card (VDEC). The data obtained is in YCrCb format, you are required to convert it to RGB and display it on the SVGA monitor.

This lab is divided into three parts. In the first part, you will generate the SVGA signals, without actually connected the camera (or any external input). This part will be done using ISE. In the later parts, you will interface the camera along with the SVGA monitor. The hardware part will be done in the second week and software in the third week (XPS will be used for the last two parts).

Each of the three parts needs to be completed in a week and the deliverables submitted at the end of every week.

Overview

The block diagram for this system is shown in Figure 1. An overview of how this system works is given next, each of the components will be explained in further detail in the following sections.

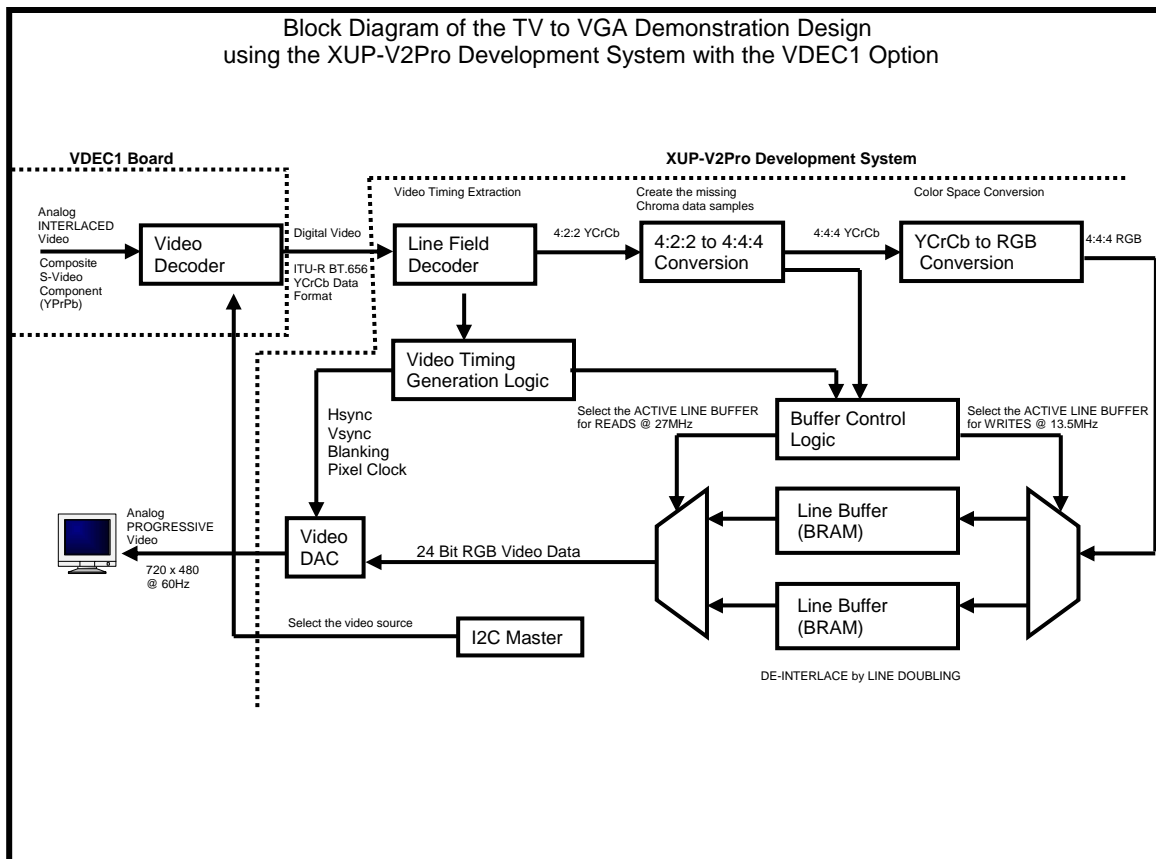


Figure 1. Block Diagram (from Xilinx)

The Video Decoder is configured using an FPGA IP Core via the I²C bus. The Video Decoder converts the analog video data into digital video in ITU-R BT.656 format. ITU-R BT.656 video consists of color and brightness information in YCrCb 422 format. You are required to extract the timing information from this video input, convert it to YCrCb 444, then convert it to RGB format. Send the resulting RGB signal to the Video DAC (Digital to Analog Converter, which converts the data back to analog and displays it on the SVGA monitor) along with proper timing signals.

In Part A of this experiment, you will be generating the timing signals for SVGA. We will explain ITU-R BT.656 standard and rest of the hardware of Figure 1 in more detail, before Part B.

Procedure

PART A: Generating SVGA Timing Signals:

We will do this part using ISE (you can also do this using XPS, by using the ‘Create and Import Custom IP’ wizard. But ISE is preferred for designs using only hardware, as it is faster). In this part, we will generate the timing signals for output format of 640 x 480 @ 60 Hz. A 640 x 480 format signal has the following components:

# of pixels →	Active	Front Porch	Sync	Back Porch	Total
Horizontal	640	16	96	48	800
Vertical	480	9	2	29	520

Table 1. Pixels count for 640 x 480 @ 60 Hz video format

The active period corresponds to the duration in which the video data is being transmitted. The remaining is the blanking portion. In the blanking interval, a sync pulse is generated which is used by the electron gun to align itself. The sync pulse is followed by a back porch, which was originally allocated to allow the slow electronics in early televisions time to respond to the sync pulse and prepare for the active line period. It now finds other uses, such as generating the color burst signal (which is used to decode the color information from a composite signal). The front porch is a brief period inserted between the end of each transmitted line of picture and the leading edge of the next line sync pulse. Its purpose is to allow voltage levels to stabilize preventing interference between picture lines.

The screen needs to be refreshed at a frequency of 60Hz and 800 * 520 pixels need to be scanned in this duration. Thus, the pixel clock is $60 \cdot 800 \cdot 520 \approx 25\text{MHz}$.

In your Verilog module, you will receive system clock and reset as inputs and you are required to generate the signals required by the SVGA VDAC. The signals required by the SVGA terminal are PIXEL_CLOCK, HORIZ_SYNC, VERTICAL_SYNC, BLANK, COMP_SYNC and 8-bit R, G, B signals.

1. Your Verilog module (say 'svga_timing') will have sys_clk and sys_rst as inputs and the SVGA outputs as the output ports.
2. Create the corresponding UCF called 'svga_timing.ucf' with the following data:

```

NET "sys_clk" TNM_NET = "sys_clk";
TIMESPEC "TS_sys_clk" = PERIOD "sys_clk" 10 ns HIGH 50 %;

Net "sys_clk" LOC=AJ15;
Net "sys_clk" IOSTANDARD = LVCMOS25;
Net "sys_rst" LOC=AH5;
Net "sys_rst" IOSTANDARD = LVTTTL;

#### Module VGA_FrameBuffer constraints
NET "B[0]" LOC = "D15" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "B[1]" LOC = "E15" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "B[2]" LOC = "H15" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "B[3]" LOC = "J15" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "B[4]" LOC = "C13" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "B[5]" LOC = "D13" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "B[6]" LOC = "D14" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "B[7]" LOC = "E14" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "BLANK_Z" LOC = "A8" | IOSTANDARD = LVTTTL | DRIVE = 12 ;
NET "COMP_SYNC" LOC = "G12" | IOSTANDARD = LVTTTL | DRIVE = 12 ;
NET "G[0]" LOC = "G10" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "G[1]" LOC = "E10" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "G[2]" LOC = "D10" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "G[3]" LOC = "D8" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "G[4]" LOC = "C8" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "G[5]" LOC = "H11" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "G[6]" LOC = "G11" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "G[7]" LOC = "E11" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "H_SYNC_Z" LOC = "B8" | IOSTANDARD = LVTTTL | DRIVE = 12 ;
NET "PIXEL_CLOCK" LOC = "H12" | IOSTANDARD = LVTTTL | DRIVE = 12 ;
;
NET "R[0]" LOC = "G8" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "R[1]" LOC = "H9" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "R[2]" LOC = "G9" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "R[3]" LOC = "F9" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "R[4]" LOC = "F10" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "R[5]" LOC = "D7" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "R[6]" LOC = "C7" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "R[7]" LOC = "H10" | IOSTANDARD = LVTTTL | DRIVE = 8 ;
NET "V_SYNC_Z" LOC = "D11" | IOSTANDARD = LVTTTL | DRIVE = 12 ;

```

As explained in earlier labs, UCF gives the location of the ports which are in the port list of your top-level module.

The following steps will guide you through writing the Verilog code for generating the timing signals.

3. Use 'define' to specify the values of horizontal and vertical front porch, back porch, sync, active and total pixels. Do not hardcode these values in your Verilog code. We shall be using a different output format (720 X 480 @ 60Hz with a 27MHz pixel clock) in the next 2 parts of this lab. You will be able to use the code from PART A for the next 2 parts by simply changing the 'define values.

Call these signals by following names:

H_ACTIVE, H_FRONT_PORCH, H_SYNCH, H_BACK_PORCH, H_TOTAL,
V_ACTIVE, V_FRONT_PORCH, V_SYNCH, V_BACK_PORCH and V_TOTAL.

4. Now, refer to Table 1 for generating the timing signals. Some hints:
 - The system clock you are using is a 100MHz clock. You can derive the 25MHz pixel clock by dividing it by four.
 - Create a 'reg' called 'pixel_count'. Use this to generate the HORIZ_SYNC and HORIZ_BLANK signals.
 - Similarly, maintain a 'reg' called 'line_count' which you can use for generating the VERTICAL_SYNC and VERTICAL_BLANK signals.
 - Use the reset signal in the sensitivity list of all your always blocks. Assign proper values to each of the signals on reset. All the signals should be made equal to zero, except line_count should be made equal to 'V_TOTAL - 33' and VERTICAL_BLANK should be made equal to '1' on reset (remember that system reset is active low as discussed in Lab 1 solution hint!).
 - The BLANK signal for the SVGA terminal is generated by taking an OR of the horizontal and vertical blank signals.
 - Set COMP_SYNC to zero. We shall not be using this signal.
5. In this part, we do not have the actual video data that needs to be displayed on the SVGA screen. Assign any arbitrary 8-bit value to each of the R, G and B signals.
6. Compile your Verilog code and generate the programming file using the steps learnt in Lab 1. Connect the extra monitor kept on your workstations to the SVGA port on the XUP board. Download the bitstream. If your timing signals have been correctly generated, based on the value of R, G and B signals, you should be able to see a constant color on the SVGA monitor.

DEBUGGING: In order to validate that your Verilog code for generating the timing signals is correct, you can take the help of Xilinx ISE Simulator.

- In the processes window, click on 'Create New Source'. Select 'Verilog Test Fixture'. It will generate a verilog test file for you, which instantiates your verilog module. Assign appropriate values to the inputs for your module (clock and reset).
- In the sources window. In the drop-down menu for 'Sources for:', change 'Synthesis/Implementation' to 'Behavioral Simulation'.
- In the Processes window, under 'Xilinx ISE Simulator', click on 'Check Syntax' followed by 'Simulate Behavioral Model'. This will show you the waveform for the various signals in your module. Check to see if a signal is not as expected, analyze and fix the same.

Deliverables – PART A

Demonstrate to the TA the constant color that you see on your SVGA monitor. Also, submit your commented Verilog code. [8 points]

What color do you see on the screen when you set R=0, G=0 and B=0. Similarly, when R=255, G=255 and B=255. [2 points]

PART B and C

As mentioned earlier, the Video Decoder converts the analog video data (from video camera) into digital video in ITU-R BT.656 format. ITU-R BT.656 video consists of color and brightness information in YCrCb 422 format. You are required to extract the timing information from this video input, convert it to YCrCb 444, then convert it to RGB format.

YUV color space

The input from VDEC is in YCrCb format, which is a scaled and offset version of YUV. The YUV color model is used in the PAL, NTSC, and SECAM composite color video standards. The camera provided to you generates data in NTSC standard.

YUV signals are created from an original RGB (red, green and blue) source. The weighted values of R, G, and B are added together to produce a single Y signal, representing the overall brightness, or luminance, of that spot and U and V signals containing the color information. The YUV format is useful for compatibility with old black and white televisions, as the black and white video can easily be derived by using only the Y component, and ignoring the color information (U and V).

Because the human eye is less sensitive to color than luminance, bandwidth can be optimized by storing more luminance detail than color detail. At normal viewing distances, there is no perceptible loss incurred by sampling the color detail at a lower rate. In video systems, this is achieved through the use of color difference components. The signal is divided into a luma (Y) component and two color difference components (chroma – Cb/U and Cr/V). In YUV 422, the two chroma components are sampled at half the sample rate of luma, so horizontal chroma resolution is cut in half. This reduces the bandwidth of a video signal by one-third with little to no visual difference.

We will first convert the 4:2:2 video data to 4:4:4 data. The process of 4:2:2 to 4:4:4 is simply creating the missing Cr and Cb components. This can be accomplished by merely duplicating the Cr and Cb information. You are then required to convert the YUV signal to RGB and send it to the SVGA port along with proper timing signals.

ITU-R BT.656 video format

The data obtained from the VDEC card is in ITU-R BT.656 video format. It consists of color information in YCrCb 422 format, along with timing information. As discussed above, human eye is less perceptible to color, than brightness. Thus, we store only half as much color information as brightness information. In order to transmit the data serially, it is reduced to pixel pairs, each of them 32bits wide with two luminance (Y) values, and one each of the chrominance values, red (Cr) and blue (Cb).

C_B Y C_R Y C_B Y C_R Y C_B Y C_R Y ...

ITU-R BT.656 conveys interlaced video signals, along with the blanking periods which old CRT TV sets used to align and move their electron guns. In CRT, with *progressive* scan, an image is captured, transmitted and displayed in a path similar to text on a page: line by line, from top to bottom. The *interlaced* scan pattern in a CRT display completes such a scan too, but only for every second line. This is carried out from the top left corner to the bottom right corner of a CRT display. This process is repeated again, only this time starting at the second row, in order to fill in those particular gaps left behind while performing the first progressive scan on alternate rows only. Such scan of every alternate line is called a field. There are thus two fields, which are often called 'even' field and 'odd' field.

The beam scans across the top of the monitor from left to right, is then blanked (horizontal blanking) and moved back to the left-hand side slightly below the previous trace (on the next scan line), scans across the second line and so on until the bottom right of the screen is reached. The beam is again blanked (vertical blanking), and moved back to the top left to start again.

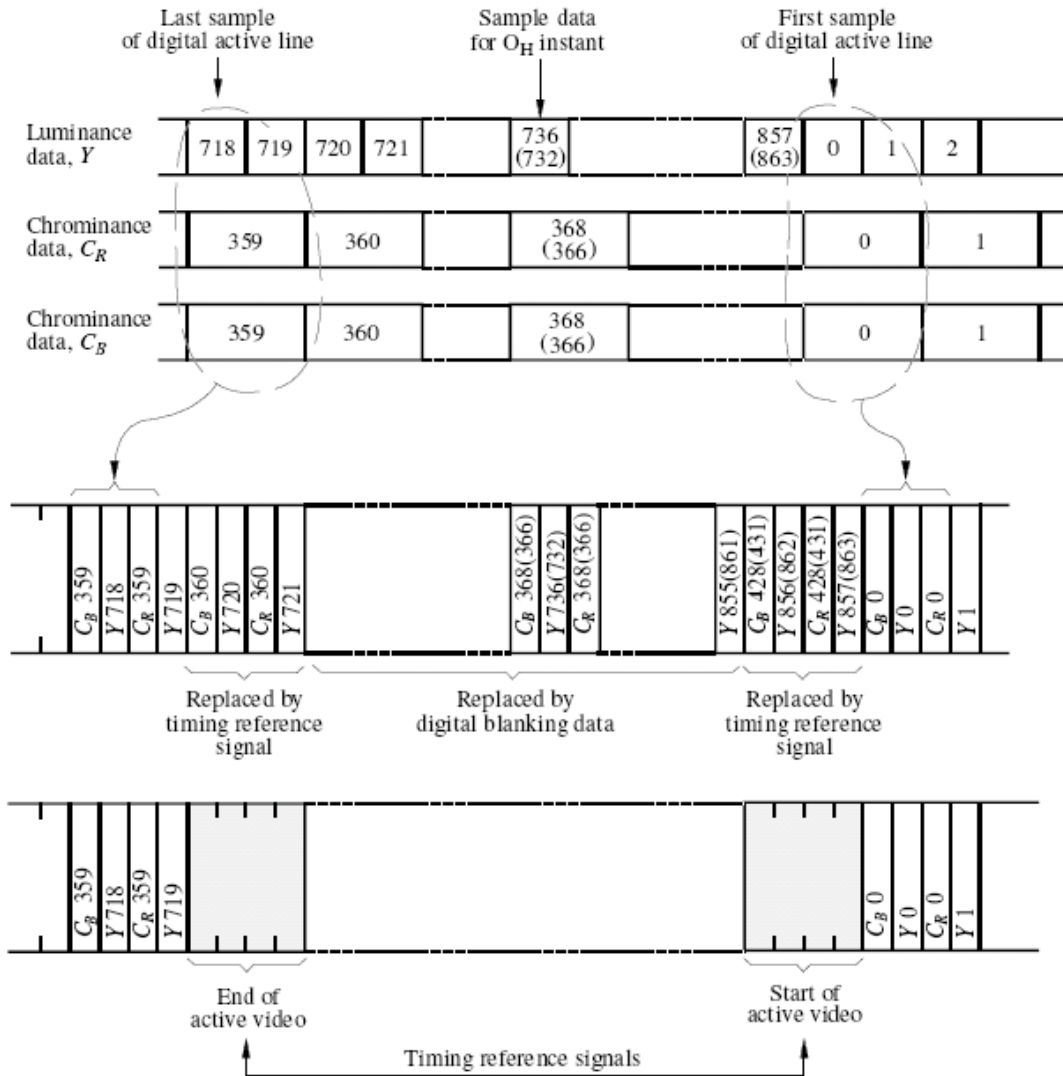


Figure 2. ITU-R BT.656 Format Detail (from berkely EECS 150 lab doc)

Figure 2 above shows two timing reference signals, one at the beginning of each video data block (start of active video, SAV) and one at the end of each video data block (end of active video, EAV). The portion between EAV and SAV corresponds to horizontal blanking. Figure 3 below shows the entire frame, including odd, even fields, horizontal and vertical blanking intervals. After horizontal blanking, the active portion of line will have the data for number of pixels in a row (= 720 for 720 X 480 format).

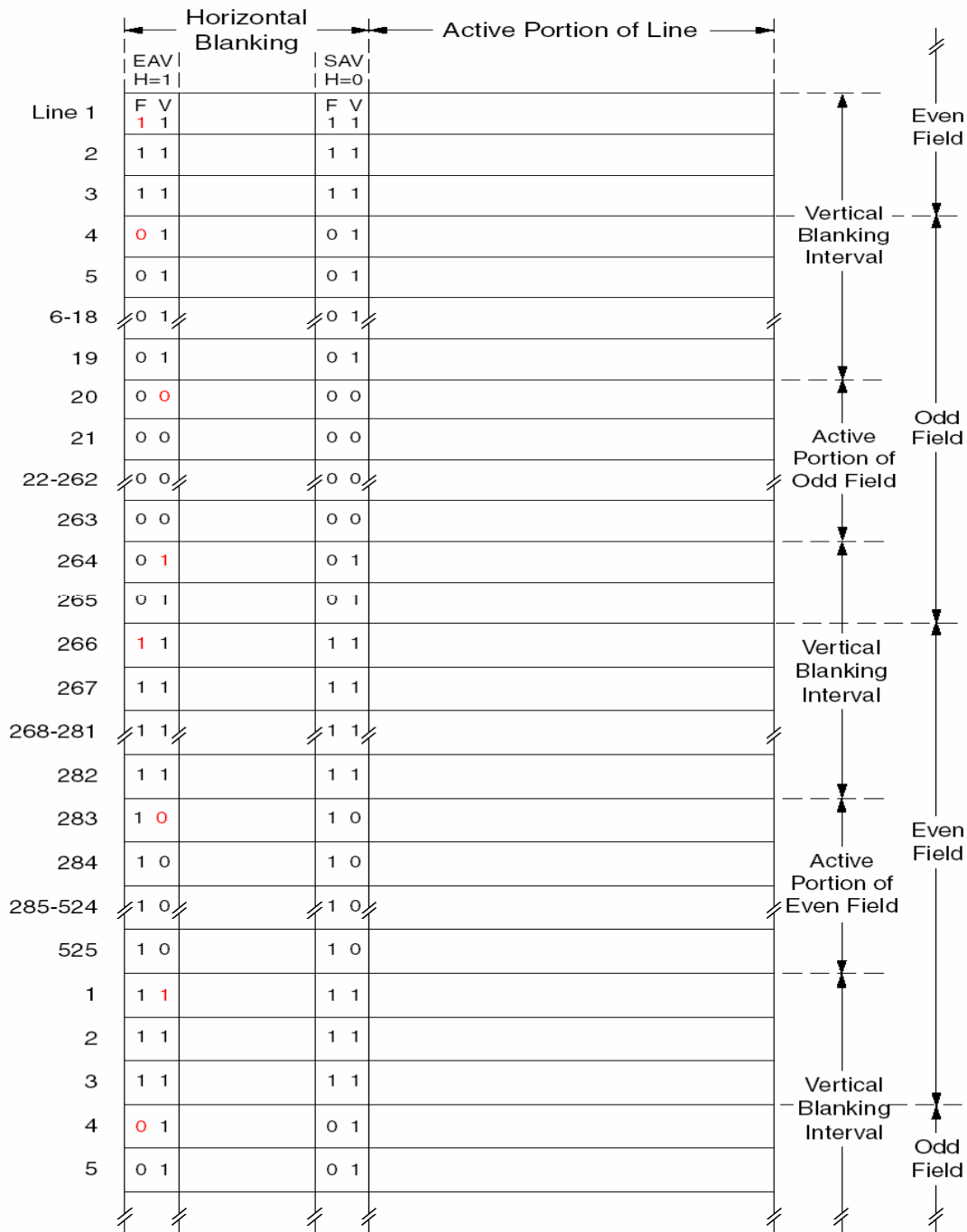


Figure 3. ITU-R BT.656 Frame Detail (from berkely EECS 150 lab doc)

Each timing reference signal consists of a four word sequence in the following format: FF 00 00 XY (values are expressed in hexadecimal notation). The first three words are a fixed preamble. The fourth word contains information regarding field identification, the state of field blanking, and the state of line blanking. The "F bit" or bit position 6, the "V bit" or bit position 5 and the "H bit" or bit position 4 of 'XY' are decoded as follows:

F = 0, denotes field 1, odd

F = 1, denotes field 2, even
V = 0, denotes no field blanking
V = 1, denotes field blanking
H = 0, in SAV
H = 1, in EAV

The remaining bits of XY are used for parity checking. We will not be decoding these bits in our experiment.

Let's again look at the Block Diagram in Figure 1 and try and understand how the entire system would work.

I²C and porting the design to XPS

The Video Decoder (VDEC card issued to you) uses the I²C bus for configuration information. The I²C protocol was invented by Philips and is used to attach low-speed peripherals to a motherboard, embedded system, or cellphone. The name stands for Inter-Integrated Circuit. The I²C bus offers several advantages over traditional interfaces, such as RS-232, including automatic adjustment of the baud-rate, support for multi-master operation, and simple plug-and-play network extensions. Robust serial inter-chip protocols such as this provide significant savings in cost, space and power which are all critical concerns for embedded systems designers.

For your experiment, the I²C part has already been written for you. For the software part – you will be provided a set of functions you can use to talk to the I²C. For the hardware part, you will be provided the signals obtained from VDEC (such as the YCrCb data, LLC (VDEC clock)), you can generate the RGB data from this and send it to the SVGA port along with the timing signals (generated in PART A).

Since we have to use both hardware and software, you are required to use XPS. For porting the hardware to XPS, you need to use the 'Import Custom IP' wizard. After importing the Custom IP, all the ports of your module need to be made External and connected to the appropriate pin names as defined in the UCF (similar to what we did for the LEDs and switches in Lab 2 and the comparator output in Lab 4).

This part has been done for you. You can download the 'lab5_partc.zip' file from the TA website. It contains an XPS setup, which includes I2C, as well as the imported hardware (video_capture.v). The video_capture.v file provides a framework to you to write your verilog code. It instantiates various modules and specifies the inputs and outputs for these modules. You are required to write the verilog code as per the required functionality of each of these instantiated modules. This will be explained in detail in the steps provided.

PART B: Completing the hardware part

7. Unzip lab5_partc.zip that you have downloaded to obtain the lab5_partc directory. Navigate to the lab5_partc/pcores/video_capture_v1_01_b/hdl/verilog directory. Open 'video_capture.v' file.

8. Look at the port list of the video_capture module. The signals YCrCb_in and LLC_CLOCK are the inputs to our module obtained from the VDEC card. PIXEL_CLOCK, HORIZ_SYNC, VERTICAL_SYNC, BLANK, COMP_SYNC, R, G, B signals are the signals to be generated for the SVGA terminal. RESET_VDEC1_Z, VDEC1_OE_Z and VDEC1_PWRDN_Z are signals for the VDEC card, which have already been assigned constant values in your module. system_dcm_locked is the reset from the top level module.
9. Now look at the various modules that are being instantiated.
 - IBUFG and BUFG are buffers for the clock signals.
 - 'extract_HVF' extract the H, V and F signals corresponding to horizontal and vertical blanking and field (odd/even) as explained for ITU-R BT.656 video format.
 - 'c422_444' converts the YCrCb 422 data to YCrCb 444 format.
 - YCrCb2RGB converts the video data to RGB format.
 - There are two LINE_BUFFERs which switch from read to write at the end of every horizontal blanking interval.
 - NEG_EDGE_DETECT generates a one clock wide pulse when the field signal (of the timing reference code) goes from one to zero. This pulse is used as a reset for the 'svga_timing' module.
 - 'svga_timing' generates the timing signals for SVGA.
 - 'pipe_line_delay' introduces a pipe line delay in the signals generated by the 'svga_timing' module and the signals sent to the SVGA terminal.
 - OFDDRSE is a DDR output flip flop which synchronizes the pixel clock with the LLC clock.

Following steps give hints for the modules that you need to write:

10. The LLC clock is a 27MHz clock. We will generate the video output using 720 X 480 @ 60Hz format with a 27MHz pixel clock. We are thus operating the SVGA with a pixel clock which has the same frequency as the LLC clock. We will however be adding buffers and a pipeline delay between the VDEC input and output to the SVGA, for the system to work smoothly.

Copy the code from PART A for timing signal generation in the 'svga_timing' module. Change the values of your `defines in accordance with the following table:

# of pixels →	Active	Front Porch	Sync	Back Porch	Total
Horizontal	720	7	62	69	858
Vertical	487	4	4	30	525

Table 2. Pixels count for 720 x 480 @ 60 Hz video format

11. In module, 'extract_HVF', extract the H, V and F signals from the timing reference code. That is, look for pattern FF 00 00 XY, extract HVF from the XY

portion. Also, introduce a pipeline such that YCrCb_out is a delayed version of YCrCb_in by 5 clock cycles. The 'H' signal is also delayed by 5 clock cycles.

12. The 422 data stream will appear serially as:

Cb₀₁ Y₀ Cr₀₁ Y₁ Cb₂₃ Y₂ Cr₂₃ Y₃.....

In the 'c422_444' module, extract out the Y, Cr and Cb data from the incoming serial YCrCb data. Look for start of active video (falling edge of 'H_in'), and extract the data knowing that it will appear in the order shown above.

Cb₀₁ and Cr₀₁ are the color information for first two pixels, Y₀ is the brightness information for the first pixel, and Y₁ for the second pixel, and so on. Thus, Y information is available for all the pixels, while Cb and Cr appear for every 2 pixels. In order to obtain the 444 data, we need to duplicate the Cb and Cr data for the 2 pixels they correspond to. This can be achieved as follows:

Update the value of Cr, Cb and Y signals in your Verilog module every time they are extracted from the serial input stream, otherwise hold their last value. Every time you receive the Y signal, Cb delayed by 4 clock cycles, Cr delayed by 2 and Y delayed by 3 forms the data for one pixel. This can be understood with the help of Figure 4 below

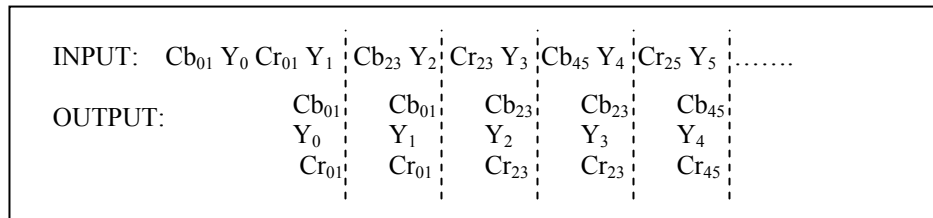


Figure 4. YCrCb 422 to 444

The output data stream thus appears at half the frequency of input stream. 'clk_out' is the clock for the 444 data. 'fo', 'ho' and 'vo' are the F, H and V signals for the 444 data stream; and are basically the F_in, H_in and V_in signals delayed by 4 each.

13. 'NEG_EDGE_DETECT' should generate the signal 'one_shot_out' when the field input to this module changes from one to zero. In the video_capture output, combine one_shot_out with system reset to generate the reset for the 'svga_timing' module.
14. Let's now look at the YCrCb2RGB conversion module. The transformation from YCbCr to RGB, which restricts the RGB signals between 0 and 255 is:

$$R = k1 * (Y - 64) + k2 * (Cr - 512)$$

$$G = k1 * (Y - 64) - k3 * (Cr - 512) - k4 * (Cb - 512)$$

$$B = k1 * (Y - 64) + k5 * (Cb - 512)$$

The constants k1 through k5 are defined in the YCrCb2RGB module.

It is important to note that it is normal for YCbCr values to go out of range due to image filtering, brightness controls, and other common video operations. Check that Y is in its legal range of 64-940, Cb and Cr are within 64-960. It is also normal for the computed R, G, or B values to be below 0 or above 255, even if YCbCr were in their legal range. Include the necessary code to keep all RGB result values within their legal range.

15. PIPE_LINE_DELAY, as the name suggested is a pipeline, which introduces one clock cycle delay between the input and output signals. The delayed signals are the outputs for the video_capture module.
16. Finally, we need to write the Buffer Control logic for the two line buffers (refer to the block diagram in Figure 1 again). You need to toggle between read and write to the two line buffers. That is, when data is being written into buffer 1, it is read from buffer 2. Similarly, the data is read from buffer 1, while it is being written into buffer 2.

The data written to the buffers is the output of the RGB output of the YCrCb2RGB module, and the data read from the buffers is assigned to the RGB signals in the video_decoder module (which are in turn connected to the VDAC shown in Figure 1).

- The data should be written to the line buffers with the 13.5MHz clock (27/2MHz clock output of the c422_444 module).
- Toggle the read_enable and write_enable signals for the line buffers at the start of horizontal blanking interval (HBI), which is H switching from 0 to 1.
- Increment the write_address of the line buffers with the 13MHz clock. Reset it to zero at the start of HBI.
- Read the data from line buffer 0 or 1 (depending on the read_enable signal) and assign it to the RGB outputs of the video_capture module.

PART C

I²C bus – some more details

I²C stands for inter integrated circuit bus. I²C bus is used to form a system in which one or more microprocessors control several devices. The interface is identical for master and slave devices. The only requirement is that the total load on the bus should be <400pF.

The architecture consists of two buses – SCLK (serial clock) and SDA (serial data). By default, both the lines are pulled to Vdd. Therefore, when the bus is idle, both SDA and SCLK are high. Start and Stop is indicated by transition on SDA when SCLK is high (Figure 4). During normal data transfer, SDA may change only when SCLK is low.

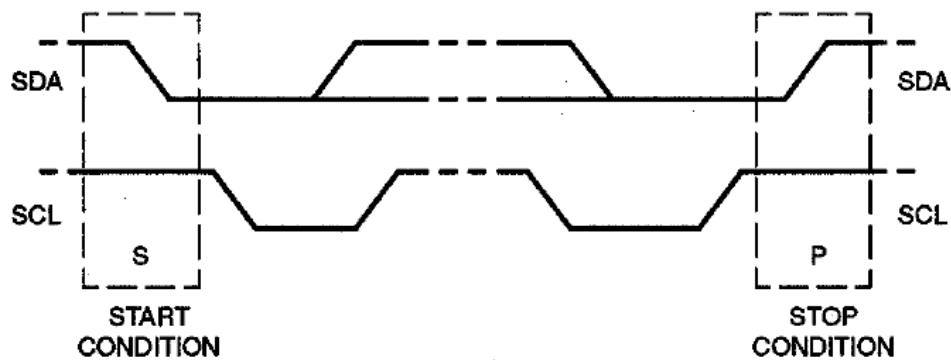


Figure 4. Start and Stop condition

There can be multiple masters and they can support many devices. To talk to a device, the master sends transfer start, followed by device address, data bytes and then transfer stop. All devices have a unique address. They look at the address sent by the master, to decide whether the data is meant for them or not. The device generates an acknowledge signal if it sees its address on the data bus. Master must abort transfer if it doesn't receive an acknowledge signal.

The video decoder we are using is Analog Devices ADV7183B. It has a large number of control registers, such as to select input format, output type, luminance/chroma control, etc which are programmed using the I²C interface. The manuals for I²C and the data sheet for ADV7183B can be found on the course website.

Page 88 of ADV7183B_0.pdf shows some I²C programming examples. For composite video, the register addresses and the config_values to be set are as follows:

Address	Value
0x00	0x04
0x15	0x00
0x17	0x41
0x27	0x58
0x3a	0x16
0x50	0x04
0x0e	0x80
0x50	0x20
0x52	0x18
0x58	0xed
0x77	0xc5
0x7c	0x93
0x7d	0x00
0xd0	0x48
0xd5	0xa0
0xd7	0xea

0xe4	0x3e
0xea	0x0f
0x0e	0x00

You can use the following piece of code to write to each of the registers:

```
XI2c_mWriteReg(XPAR_I2C_BASEADDR, GPO_REG_OFFSET, GPO_RESET_IIC);
XI2c_mWriteReg(XPAR_I2C_BASEADDR, GPO_REG_OFFSET, GPO_RESETS_OFF);
send_data[0] = address;
send_data[1] = config_value;

send_cnt = XI2c_Send(XPAR_I2C_BASEADDR, DECODER_ADDR, send_data, 2);
if( send_cnt != 2 )
{
    xil_printf("Error writing to address %02x\r\n", address);
    break;
}
```

Write your code in the 'configDecoder' function in the 'xup_config_decoder.c' file of the Application attached to your XPS project.

Deliverables

After Part C, your hardware and software are both in place. Compile XPS with the usual procedure. Download bitstream and demonstrate to the TA, that the live video can be seen on the SVGA monitor! Also, turn in your Verilog code for the same.