CE2003 Laboratory 3
A simple Video System

Introduction

In image processing systems, pixels for each frame of a video are streamed through the system serially. The pixels are ordered according to a raster scan format, which dictates that the leftmost pixel of the first row is followed by the one to its right, and so on until the end of the row. Immediately following the last pixel in the first row comes the first pixel in the second row, and then the one beside it, and so on.

If we index the top row as having *y* coordinate 0, and the leftost pixel having *x* coordinate 0, then, for an image of size *N x M* pixels, the order of pixels would be:

(0, 0), (1, 0), (2, 0), … (*N*-2, 0), (*N*-1, 0);
(0, 1), (1, 1), (2, 1), … (*N*-2, 1), (*N*-1, 1);
…
(0, *M*-2), (1, *M*-2), … (*N*-2, *M*-2), (*N*-1, *M*-2);
(0, *M*-1), (1, *M*-1), … (*N*-2, *M*-1), (*N*-1, *M*-1);

In a video system, each video frame (image) would be followed immediately by the next (with some standards dictating some empty cycles between).

In this lab, we will build a simple video system, consisting of:

1. A video frame buffer memory, containing an image to be output to the screen
2. A set of counters to generate pixel addresses
3. Some logic to write data into the frame buffer

Task 1

First, start a new project called *Lab3* in Xilinx ISE, targeting the ATLYS board, as usual. Add a directory called *src* into the *Lab3* directory. Download the lab3.zip file from NTUlearn and unzip it into the *src* directory. Now add all the files you unzipped using Project | Add Source (you will need to go to the *src* directory). You will see a number of modules, SysCon, VideoTiming, DVITransmitter, etc. These are modules provided by the board vendor to enable us to use the video ports. There is no need to delve into them. Their purpose is to manage control signals and clocks to ensure the interfaces function correctly. *atlys_vmodcam.ucf* contains all the required pin constraints for our design, including the physical pin connections for the HDMI port, and the timing constraints for the design. A diagram of the system is shown in Figure 1. You will be creating and instantiating the Frame Buffer, and writing the DataGen module.

Task 2

We need to instantiate a frame buffer memory for the design. This will hold the image we wish to display on screen. Existing logic will read the contents of this memory and send them to the HDMI controller.

The external HDMI video interface has a resolution of 1600 x 900. We cannot fit so many pixels in a buffer inside the FPGA. Hence, such systems are implemented in an entirely streaming manner, where the pixels pass through and are processed without being stored. Since we do want an image buffer inside the FPGA, we will use an image one quarter the size and upscale it when we output the signal. Hence, our image will be of size 400 x 225.

The number of bits used to represent colours depends on the video format. In this system, 16 bits are used. To make our memory size more manageable, we will use just 6 bits; 2 bits each for red, green, and blue. Hence, our buffer memory should be of size 400 x 225 x 6 bits, or 90000 x 6 bits.

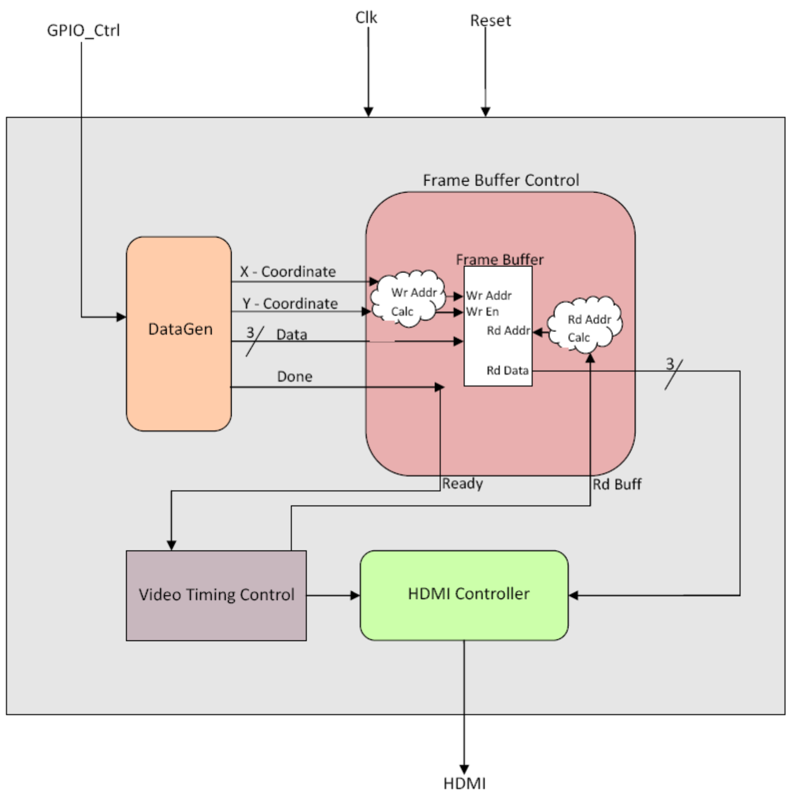

Figure 1. Block Diagram of the Video System

To implement the buffer perform the following steps

1. Go to Project | New Source..., and select IP (Core Generator...)
2. Name the module *frame_buffer* and click Next
3. Scroll down the list and expand the ***Memories & Storage Elements*** item
4. Expand ***RAMs and ROMs***, then select ***Block Memory Generator***, and click Next, then Finish
5. Wait for the generator interface to appear, then select Native for interface type, and click Next
6. Select ***Simple Dual Port RAM*** from the Memory Type dropdown, then click Next
7. Set the write width to 6, and the write depth to 90000, and click Next
8. Leave all other options as is, then click Generate

Now we need to instantiate this module in the main design and connect it:

1. Open the *FBCtl.v* file, and locate the line where you should instantiate the frame buffer
2. Select *frame_buffer* in the Hierarchy pane, then expand ***CORE Generator*** in the Processes pane
3. Double click View HDL Instantiation Template
4. Copy the non-commented lines and paste them back in the previously identified location in FBCtl.v
5. Edit the port list to make the following connections (The signals inside the brackets. Note all these signals are already instantiated at the top of the FBCtl module.
   - Both clock ports (*clka*, *clkb*) to *i_clk*
   - Write enable port (*wea*) to *frame_buffer_wr_en*
   - First address port (*addra*) to *wr_addr*
   - Second address port (*addrb*) to *rd_addr*
   - Data input (*dina*) to *wr_data*
   - Data output (*doutb*) to *buff_data*

Task 3

Now we will write the code to write pixels to the frame buffer, to be output onto screen. The first thing we need is to generate the x and y coordinates using two counters.

1. Open the *DataGen.v* file
2. Locate the first always block, where the *o_x*, *o_y*, and *o_done* signals are reset
3. Implement the else part, which should do the following:
   - *o_x* should count up in each cycle to 399, then wrap around to 0
   - *o_y* should count up only each time *o_x* wraps around to 0
   - *o_y* should wrap around to 0 each time it reaches 224
   - *o_done* should be asserted when *o_x* and *o_y* are at their maximum values
4. Implement a quick testbench to check your code. Add a new Verilog Test Fixture source, called DataGen_tb and click Next. Associate the DataGen source and click Next, then Finish. Set *i_rst* high for 15 timesteps then back to 0. Add a clock **always** block below the **initial** block toggling the value every #5. Simulate for 100us or more, and check that the count sequences are correct

The second always block in the code passes a colour value to the frame buffer for only those pixels where the x and y coordinate is equal, hence drawing a diagonal line. The *i_color* input is taken from the lowest 6 switches on the board, **so set the lower 6 switches to select a colour**. If the board switches are all off, you will see nothing.

Implement the design so far, and test it on the board. **Note that you will see a number of warnings which you can safely ignore.** To view the board's output after you have programmed it, press the button just to the left of the blue LED ON-light (on the lower edge of the monitor). When the menu appears, press it again to change the input. You can change back to the PC screen by repeating the process.

Next, comment out the equality **if** condition (in the 2nd **always** block of the *DataGen* module), and uncomment the one below it. Implement the design and test it again. You should see a square of 50 x 50 'pixels' (remember, each pixel in our memory is drawn as a 4 x 4 pixel on screen).

Task 4

Here we will try to add some variations to the existing code to modify the output. The 3rd **always** block contains partial code to move the square about the screen using the 4 push buttons. The existing code allows you to move the square drawn by the first always block to the left and right. This always block modifies the values of x_min and *x_max* when the buttons on the board are pressed.

(Note the button directions have been changed to match their current position on the lab bench). Also note that counter-based debouncers and pulse generators have been added to the 4 button inputs. These will generate a single pulse every so many cycles, allowing you to keep the button pressed for continuous movement. Using the previous implementation, verify that the square can be moved from left to right.

Now add similar code to the **always** block to allow the square to be moved in the vertical direction. You will need to modify the values of *y_min* and *y_max* based on the buttons *i_buffon_up* and *i_buffon_down*. Test your implementation

Task 5

1. Now, modify the way the colours are implemented. In *DataGen.v*:
2. Declare a 31-bit signal called *color_count*
3. Add a new synchronous always block that increments *color_count* whenever *i_color*[0] is high
4. Modify the assignment to *o_data* in the second **always** block so you assign *color_count*[30:25] instead of *i_color* if *i_color*[0] is true (high).
5. Test the design. The square should cycle through colours when the **least significant switch is high**

Inform your lab supervisor once you reach this point