COE3DQ5 Lab #3
Implementation and Utilization of a VGA Interface

Objective

To design and implement a digital system synchronized with an external device. To understand the video graphics array (VGA) interface, which is the main display interface used for personal computers (PCs).

Preparation

- Revise the first two labs and the programmable logic fabric, i.e., logic elements (LEs)
- Read this document and get familiarized with the source code and the in-lab experiments

Experiment 1

The aim of this experiment is to get you familiarized with the VGA interface.

The VGA interface is an analog interface that was designed for conventional cathode ray tube (CRT) displays. It is also compatible with modern liquid crystal displays (LCD). It carries ***synchronization (or control)*** and ***data (or color) signals***. The role of both synchronization and data signals can be explained as follows. CRTs have a phosphor-coated screen that "activates" a pixel (the smallest point on the screen that can generate light) each time an electron beam comes in contact with it. Color CRTs have three different electron beams and when in contact with the phosphor-coated screen they generate blue light (wavelength 450nm), green light (wavelength 510nm) and red light (wavelength 650nm) with different intensity. The intensity of each electron beam is controlled by the ***data signals*** (Red, Green and Blue or **RGB**) from the VGA interface in order to create a particular perception to the human eye. The human eye has photosensitive cells called rods and cones. Rod cells are 100 times more sensitive to light than the cone cells but because they have only one type of light sensitive pigment, they do not play any role in color vision. Cone cells respond to different wavelengths at higher light intensity, thus enabling us to see color images. The RGB data signals in the VGA interface are used to create an image on the phosphor-coated screen using the electron beam. The rate at which this image is displayed is controlled by the synchronization signals available in the VGA interface. If images (or frames) are updated approximately 60 times per second, an illusion of motion can be created to the human eye. Therefore, at this ***frame rate***, the electron beam traverses the screen approx 60 times per second as follows: it starts from the top-left corner of the screen and it activates the pixels from row 0 one by one with the intensity determined by the RGB signals. Then it comes back to the left side of the screen and it activates (one by one) all the pixels from row 1. After the final row is displayed, the electron beam returns to the top-left corner and it starts displaying the next frame. The two ***synchronization signals*** used to control the position and the speed of the electron beam are called horizontal synchronization (H_SYNC) and vertical synchronization (V_SYNC). Their waveforms are shown in Figure 1.

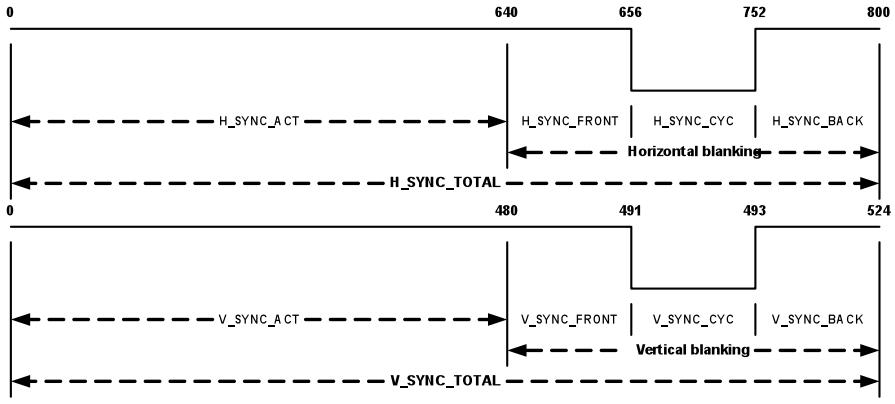

Figure 1 – H_SYNC and V_SYNC for the VGA mode: 640x480 @ 60 Hz using a 25 MHz pixel clock

The full period of the H_SYNC signal has four regions: ***active*** region (H_SYNC_ACT), ***front porch*** (H_SYNC_FRONT), ***synchronization pulse cycle*** (H_SYNC_CYC) and ***back porch*** (H_SYNC_BACK). The front porch together with the back porch and the synchronization pulse constitute the ***horizontal blanking*** period when no information is displayed on the screen. The synchronization pulse (active low) directs the electron beam to return to left side of the screen and start displaying a new row (the front and back porches isolate it from the active region, which is when useful data is displayed on the screen). The horizontal blanking period when added to the active region gives the total period (H_SYNC_TOTAL) for the horizontal synchronization signal. There is one period of the H_SYNC signal for each row (or line) of pixels that is displayed. Similarly there is one period of the V_SYNC signal for each frame. The RGB data signals carry useful information only during active periods (H_SYNC_ACT and V_SYNC_ACT). They vary between 0mV and approx 700mV and are driven by the output of a digital-to-analog converter (DAC). The DAC receives its input values from a digital system (e.g., a PC or the CycloneII device on the DE2 board). The precision of the DAC on the DE2 board is 10 bits per color, which translates into 1024 possible levels from 0mV to 700mV for each of the RGB signals in the VGA interface (and hence 1024 distinct intensities for each of the colors). There are two important points regarding synchronization and data signals:

- The period of the synchronization signals determines the resolution of the screen and the frame rate. For example if the reference clock is 25 MHz and the targeted resolution is 640x480 (i.e., 640 pixels per line and 480 lines per frame, also called the VGA mode) then the period of the H_SYNC signal is approx 32us. This is because the number of clock cycles for H_SYNC_TOTAL is 800. The number of clock cycles (for each of the regions of the H_SYNC signal) is shown in Figure 1. The period of the V_SYNC signal is 16.768ms (524 different lines and 32us are spent for each line). This translates into a frame rate of approx 60 frames per second. The number of lines (or H_SYNC periods) for each of the V_SYNC regions is also shown in Figure 1. Therefore, by changing the reference clock (also called the ***pixel_clock***) and the duration of the active and blanking periods, one can change the ***resolution*** of the video mode (obviously, so long as the targeted CRT or LCD can accept it). For example, as it can be found in the VGA parameters file from the ***experiment1*** directory, if the pixel clock is 50 MHz one can change the video mode to SVGA, i.e., 800x600 @ 72 Hz.

- The combination of the three data signals (RGB) can create other colors. For example, if R, G and B are all set to their highest intensity (700mV) then the resultant color is intense white. If R, G and B are all set to half their intensity (350mV) a gray color of medium intensity will be displayed on the screen. If red is combined with green, then yellow will be displayed. Green combined with blue will generate cyan. A mixture of red and blue will display magenta. Depending on the combination of RGB values we can obtain: black (000), blue (001), green (010), cyan (011), red (100), magenta (101), yellow (110) and white (111). If there are 1024 levels for each of the RGB signals, the total number of distinct colors that can be generated is 1024x1024x1024.

Having explained the fundamentals of the VGA interface, the next step is to understand the VGA controller provided in the ***experiment1*** directory. It generates a shifted version of the H_SYNC and V_SYNC signals as shown in Figure 2.

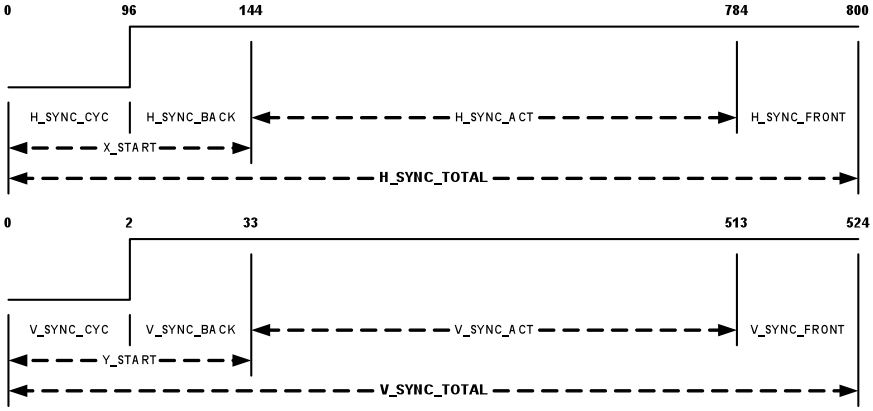

Figure 2 – H_SYNC and V_SYNC as generated by the VGA controller

The implementation of the VGA controller (shown in Figure 3) is simple and it comprises of a couple of modulo counters and steering logic. The two counters reset to zero after reaching H_SYNC_TOTAL-1 and V_SYNC_TOTAL-1 respectively. Two synchronization flip-flops drive the sync pulses to the VGA port. They are zero only when the two modulo counters are less than H_SYNC_CYC and V_SYNC_CYC respectively. The VGA controller provides the oCoord_X and oCoord_Y signals to the rest of the design. These signals carry the information on where exactly is the electron beam with respect to the H_SYNC and V_SYNC signals. Using the information from oCoord_X and oCoord_Y the user can drive the iRed, iGreen and iBlue signals. These three signals are inputs to the VGA controller and they are buffered in order to be synchronized with oVGA_H_SYNC and oVGA_V_SYNC before being sent to the VGA port. Note, the RGB signals are disabled during the blanking periods, as it is requested by the VGA interface. The clock signal comes from a phase locked loop (PLL) instantiated as a component. The input to the PLL is the 50 MHz on-board clock and the outputs are two internal clocks: 25 MHz and 50 MHz. For the 640x480@60Hz the 25 MHz should be used and for the 800x600@72Hz the 50 MHz should be used.

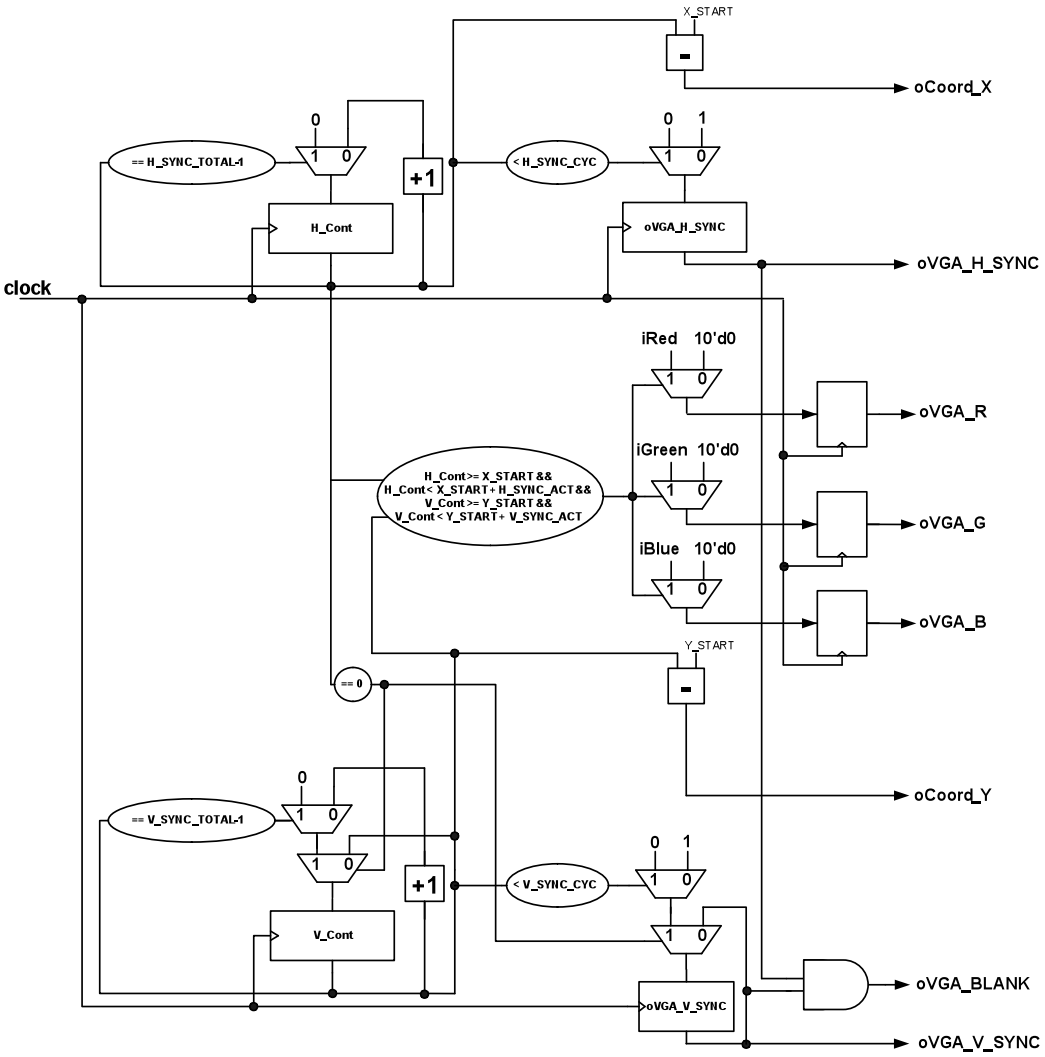

Figure 3 – The implementation of the VGA controller

In ***experiment1***, the iRed, iGreen and iBlue signals are driven by the complements of bits 8, 7 and 6 of the horizontal counter. This will create 10 different vertical color bars (for the VGA mode), each of them of width equal to 64. You have to perform the following tasks in the lab for this experiment:

- verify that the design works correctly and change the resolution from VGA to SVGA
- in the VGA mode, display 20 vertical color bars of width 32 each (8 colors are sufficient)

Experiment 2

This experiment shows how to put together user-defined logic blocks that are synchronized with the VGA controller in order to display objects on the screen. The oCoord_X and oCoordY signals from the VGA controller are mapped onto the pixel_X_pos and pixel_Y_pos signals in the top level design (as shown in Figure 4). Using these two signals we know the position of the electron beam on the screen (relative to H_SYNC and V_SYNC) and thus we can determine what color to display for each pixel. Therefore, if we drive a signal object_on to a "1" only when the electron beam is between columns 300 and 340 and rows 220 and 260 and then we use this signal to turn on VGA_blue, a blue square will be displayed.

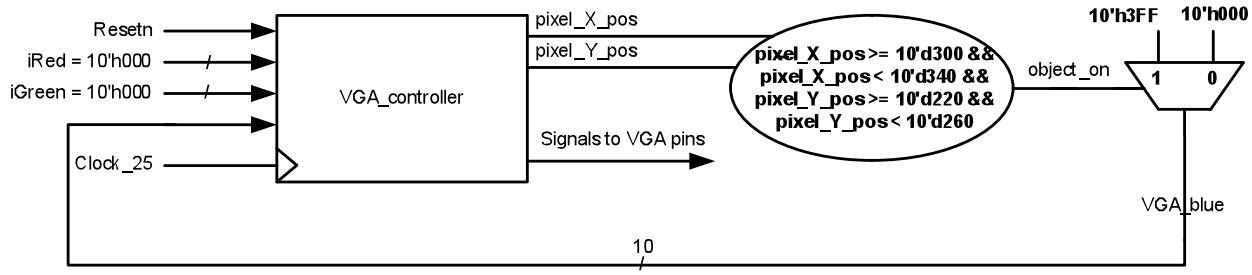

Figure 4 – Interfacing the VGA controller to user defined logic

The above-described principle can be used to display borders and objects. To understand how you can put them together, you have to perform the following tasks in the lab for this experiment:

- verify that the designs given in directories ***experiment2a*** and ***experiment2b*** work correctly
- display both the object and the borders on the VGA screen; understand the variation in LEs

Experiment 3

This experiment shows how to update the position of an object on the screen and it illustrates timing violations that occur in digital circuits that are "over-clocked".

In the previous example the coordinates of the object that was displayed were fixed. If one wishes to update the object's coordinates he can store them in registers however special care should be taken on when these registers are updated. Because changing their value at the same time when the electron beam is in the active area of the screen can cause flickering, it is recommended that this update occurs during the vertical blanking period. In order to employ a single clock in the design, we will use the negative edge of the vertical synchronization to generate a single pulse used as the enable signal for updating the coordinate registers. Other registers, such as the direction registers, should be updated in the same way. ***Experiment3a*** shows a white ball that bounces off the left and right borders of the screen.

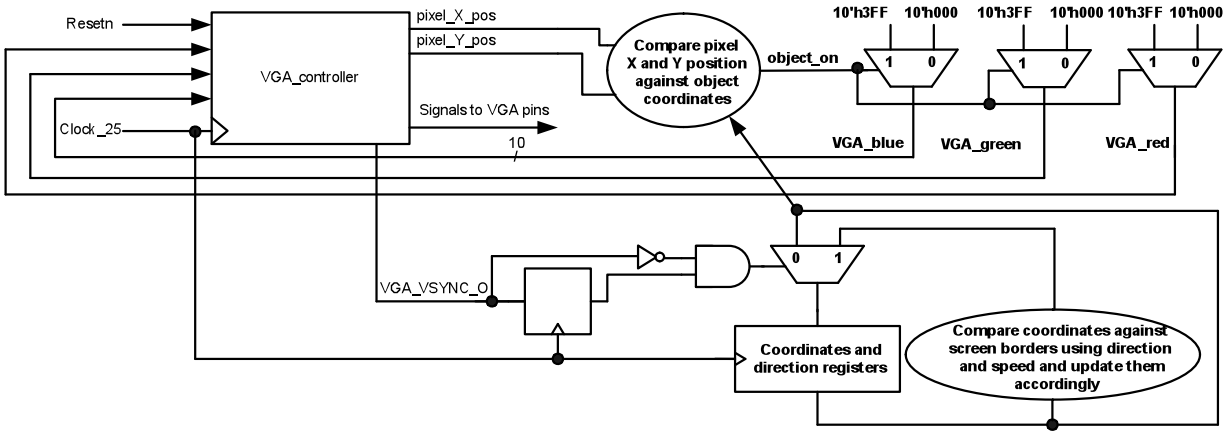

Figure 5 – Updating the position of objects during vertical blanking

A common problem in digital system design is ***timing violations***. The scope of ***experiment3b*** is not to discuss how these timing violations can be fixed, but rather to show you their effects. To achieve this, the object_on signal is put through a long combinational delay using the LCELL component (this primitive component maps onto an LE configured as a buffer that adds extra delay). Since object_on is used to drive all of the R, G and B signals, we create unequal propagation delays from the H_Cont register (in the VGA_controller) to the oVGA_R, oVGA_G and oVGA_B registers (also in the VGA controller). By construction, all these propagation delays are in excess of 40 ns, which is our clock period in the VGA mode (25 MHz). When the color registers (oVGA_R, oVGA_G and oVGA_B) update their values (i.e., when the color on the screen changes from black to white or vice-versa), the samples acquired on the edge of the clock will not be the correct ones (because the propagation delays through the circuit have not been settled). This explains the visual artifacts that can be observed on the screen. As this is an artificial example that creates unnecessarily long combinational delays in the circuit, fixing this timing problem is trivial (by removing the LCELL instances). For real cases, pipelining and retiming are used.

You have to perform the following tasks in the lab for this experiment:

- understand why the timing violations occur only in ***experiment3b*** and not in ***experiment3a***
- make the object (from ***experiment 3a***) move in diagonal (both X and Y directions with identical speed); note the object should now bounce off left, right, bottom and top display borders; explain your changes and understand the variation in LE and register count when compared to the original design

Experiment 4

In this experiment, a simple video game is implemented exclusively in hardware. It is important to note that this pong game is by no means robust. The purpose of this experiment is to better understand how to put a larger design together around the VGA controller and not how to “polish” a particular video game.

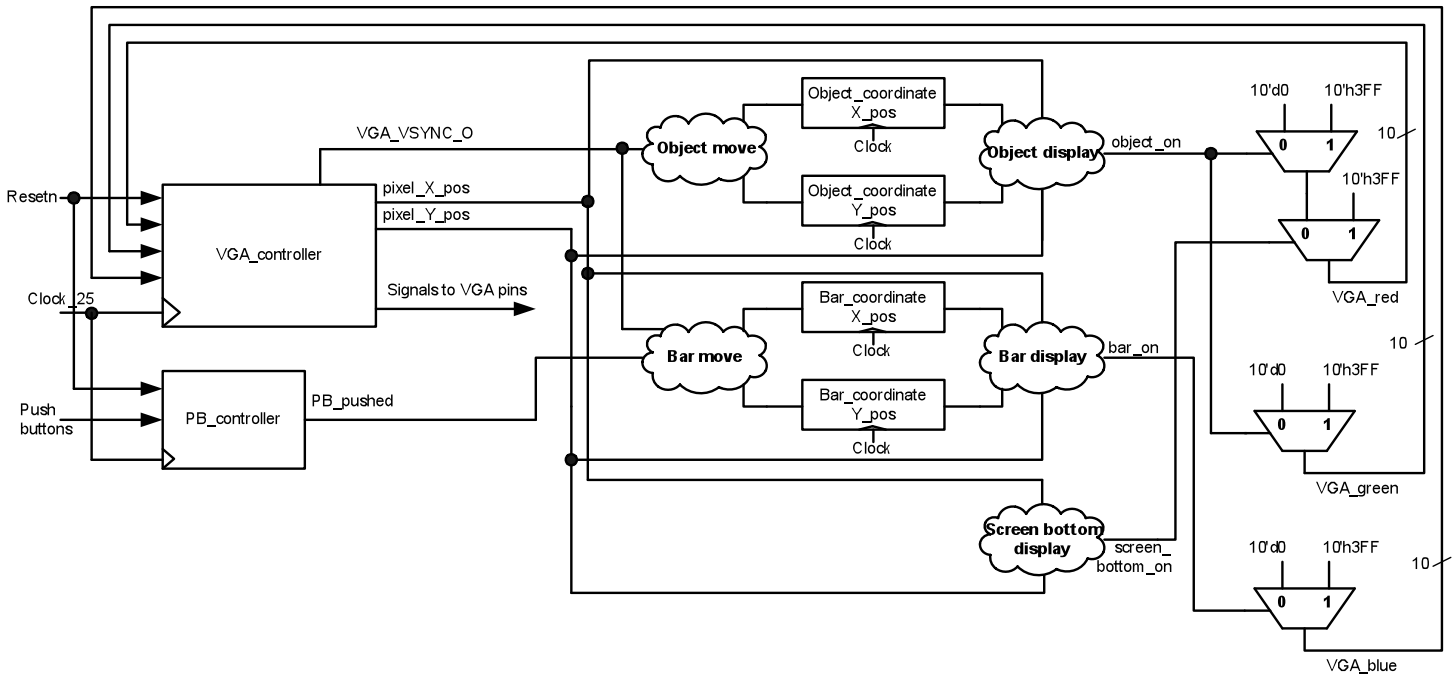

Figure 6 – The hardware architecture for the pong game

A pong game requires (in addition to a bouncing object or ball) a bar that can hit the ball and keep it in the game. Each time the ball is hit, the score is incremented and it is displayed on the 7-segment-displays (together with how many lives are left). If the ball reaches the screen bottom (as defined in the game) the number of lives is decreased. The ball speed is controlled by the three rightmost switches and the bar by push buttons 0 and 1 (note, push buttons 0 and 1 are monitored on the level and not on the edge).

The object_move and bar_move logic blocks update the coordinates of the bouncing object and the bar. All these coordinates are updated during the vertical blanking period, as explained in experiment 3. Because the ball is Yellow, the bar is Blue and the screen bottom is Red, the R, G and B equations are updated accordingly as shown on the right hand side of the figure. In the code given in the ***experiment4*** directory, the game resumes only if the asynchronous reset button (switch 17) is used.

You have to perform the following tasks in the lab for this experiment:

- understand the game behavior and the logic blocks used to update the position of the bar and the ball
- modify the design in such way that after the number of lives becomes zero the game re-starts itself

Experiment 5

The objective of this experiment is to show how to interface a ROM to the VGA controller.

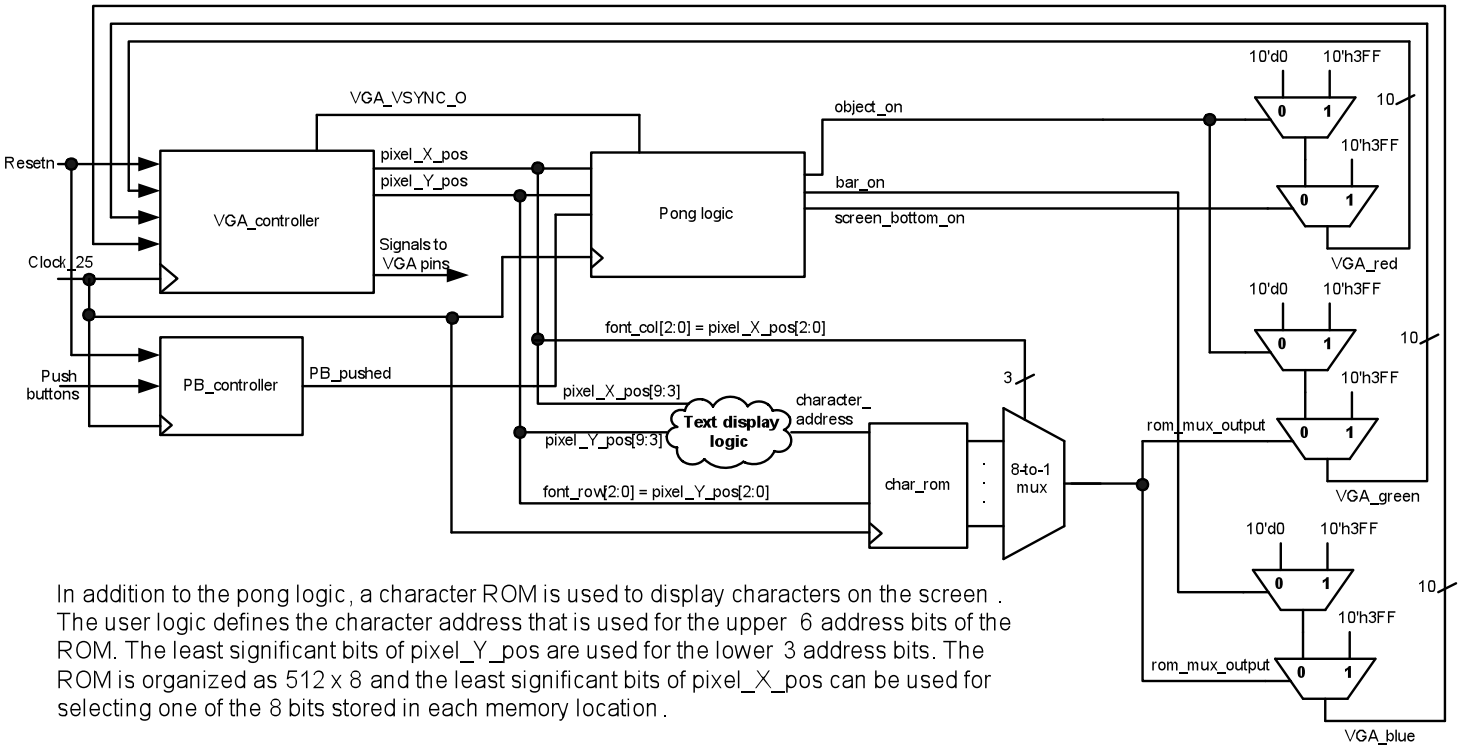

Figure 7 – Adding a character ROM to the pong game

A character ROM can be added to the design as shown and explained in Figure 7. The map of each character is provided in the "MIF" file from the ***experiment5*** folder. Although the design seems to work, there is a subtle problem. As it can be seen from the figure, the ROM data is available one clock cycle after the address has been provided and the least 3 significant bits of pixel_X_pos can be used to select one of the 8 bits stored in each ROM location. This implies that in order to properly synchronize the ROM output with the H_SYNC and V_SYNC signals, the ROM address (and hence pixel_X_pos[9:3]) must be provided one clock cycle earlier. This can be achieved by computing pixel_X_pos[9:0] + 1 and then use the 7 most significant bits of this sum as an input to the text display logic block shown in the figure. It should be noted that pixel_Y_pos[9:0] does not need to be updated because its value is constant for an entire line that is displayed.

You have to perform the following tasks in the lab for this experiment:

- understand how the character ROM is interfaced to the rest of the video game
- fix the ROM addressing to properly synchronize the VGA controller with the rest of the design

Write-up Template

COE3DQ5 – Lab #3 Report
Group Number
Student names and IDs
McMaster email addresses
Date

There are 2 take-home exercises that you have to complete within a week. When the source files are requested, submit the Verilog, "QSF" (and the "MIF" files, where appropriate). Label the top-level modules as exercise1 and exercise2. If, for any particular reason, you will add/remove/change the signals in the port list from the port names used in the design files from the in-lab experiments, make sure that these changes are properly documented in the source code.

Exercise 1 (2 marks) – Change the in-lab ***experiment1*** as follows. The screen is divided into four regions of 320x240 pixels each: top-left, top-right, bottom-left and bottom-right. It is assumed that immediately after you have downloaded the "sof" file to the FPGA, the asynchronous reset was applied using SWITCH_I[17]. Unless loaded (as explained below), the 3-bit RGB color of each region is incremented or decremented after 60 frames since its previous change. ~~After asynchronous reset~~ On load the color of the bottom-right region is defined by the position of SWITCH_I[2:0] (i.e., R = SWITCH_I[2], G = SWITCH_I[1], B = SWITCH_I[0]) and the counting direction is defined by the position of SWITCH_I[3] (i.e., count up if SWITCH_I[3] is high and count down if SWITCH_I[3] is low). Likewise, SWITCH_I[6:4] and SWITCH_I[7] define the color and direction for the bottom-left region, SWITCH_I[10:8] and SWITCH_I[11] define the color and direction for the top-right region and SWITCH_I[14:12] and SWITCH_I[15] define the color and direction for the top-left region. SWITCH_I[15:0] should be ~~read-only~~ loaded when SWITCH_I[16] toggles, i.e., it changes its position either from low to high or from high to low (what happens until the first toggle of SWITCH_I[16] is irrelevant). Note, to avoid any jittery images, the positions of the input switches, as well as the updates of the region colors, should occur only during the vertical blanking period. It is assumed that switches toggle at most once per frame. Submit your sources and in your report write at most a half-a-page paragraph describing your reasoning.

Exercise 2 (2 marks) – Extend your in-lab contribution to ***experiment5*** as follows. It is assumed that immediately after you have downloaded the "sof" file to the FPGA, the asynchronous reset was applied using SWITCH_I[17]. Thereafter, depending on the status of the system, one of the five messages (as specified below) will be displayed in white on black background using 8x8 fonts (you can freely choose the position of the message, so long as it is clearly readable). Note, while a message is displayed, all the objects and characters from the game should be deleted and only the corresponding message should be displayed. While the asynchronous reset is asserted, nothing is displayed. After asynchronous reset was deasserted, after one frame display "WELCOME" and wait for any of SWITCH_I[14:3] to be toggled before starting the first game (at which point in time, the message should be deleted and the new game screen, as the one used in the lab, is displayed). At the end of the first game, display "THE BAR IS SET" and keep the message for 300 frames (which amounts to approx 5 seconds) before starting a new game. Note, a game ends when the live counter reaches 0. From the second game onwards, if the result was lower than the worst result so far you should display "REALLY BAD" for 180 frames before starting a new game. If the result was greater than the best result so far you should display "BEAT THIS" for 350 frames before starting a new game (it is fair to assume that the largest possible score is the maximum value that can be represented on 4 bits). Otherwise, if the result was neither the best nor the worst, display "KEEP GOING" for 300 frames before starting a new game. Submit your sources and in your report write at most a half-a-page paragraph describing your reasoning.

VERY IMPORTANT NOTE:

This lab has a weight of 4% of your final grade. The report has no value without the source files, where requested. Your report must be in "pdf" format and together with the requested source files it should be included in a directory called "coe3dq5_group_xx_takehome3" (where xx is your group number). Archive this directory (in "zip" format) and upload it through Avenue to Learn before noon on the day you are scheduled for lab 4. Late submissions will be penalized.