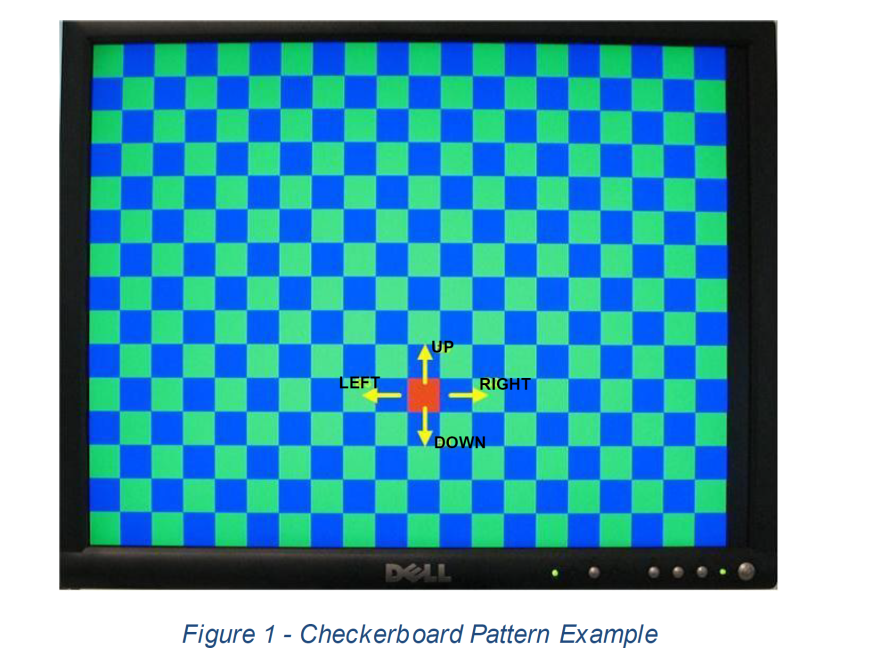
**Theory of VGA Checkerboard Display**

**Introduction**

The VGA checkerboard displays a red square that moves along the board in response to either pressing push-buttons or tilting the development board.

The design generates a checkerboard pattern as shown below in Figure 1, which will have alternating blue/green squares and one red square.



The design also uses a 7-segment display controller displaying the X-coordinates on digits 3 and 2 of the dev board display and the Y-coordinates

**Summary of Inputs, Outputs, and Signals:**

**Inputs:**

 Switch 0: Active high reset

 Button Up (BTNU): Move the red square up one position per button press

 Button Down (BTND): Move the red square down one position per button press

 Button Left (BTNL): Move the red square left one position per button press

 Button Right (BTNR): Move the red square right one position per button press

 Clock: System Clock, 100MHz

**Outputs:**

 VGA Ports: To drive VGA display

o VGA\_R, VGA\_G, VGA\_B: Red, Green, Blue color signals, each 4-bits wide

o VGA\_HS, VGA\_VS: Horizontal and Vertical Syncs, each 1-bit wide

 7-segment display: To display X-Y coordinates (SEG7\_CATH, AN)

o SEG7\_CATH: Cathode signals, 8-bits wide

o AN: Anode signals, 8-bits wide

**Theory and Implementation**

The checkerboard pattern is 15 rows by 20 columns occupying a total visible area of 480x640 pixels. Therefore, each square is 32x32 pixels to occupy the total visible area. The red square moves around the screen by pressing the up, down, left, and right push buttons. The red square wraps around the display and move appropriately at the edges.

Figure 2 depicts a block diagram of the VGA controller where each block signifies a section of VHDL code.

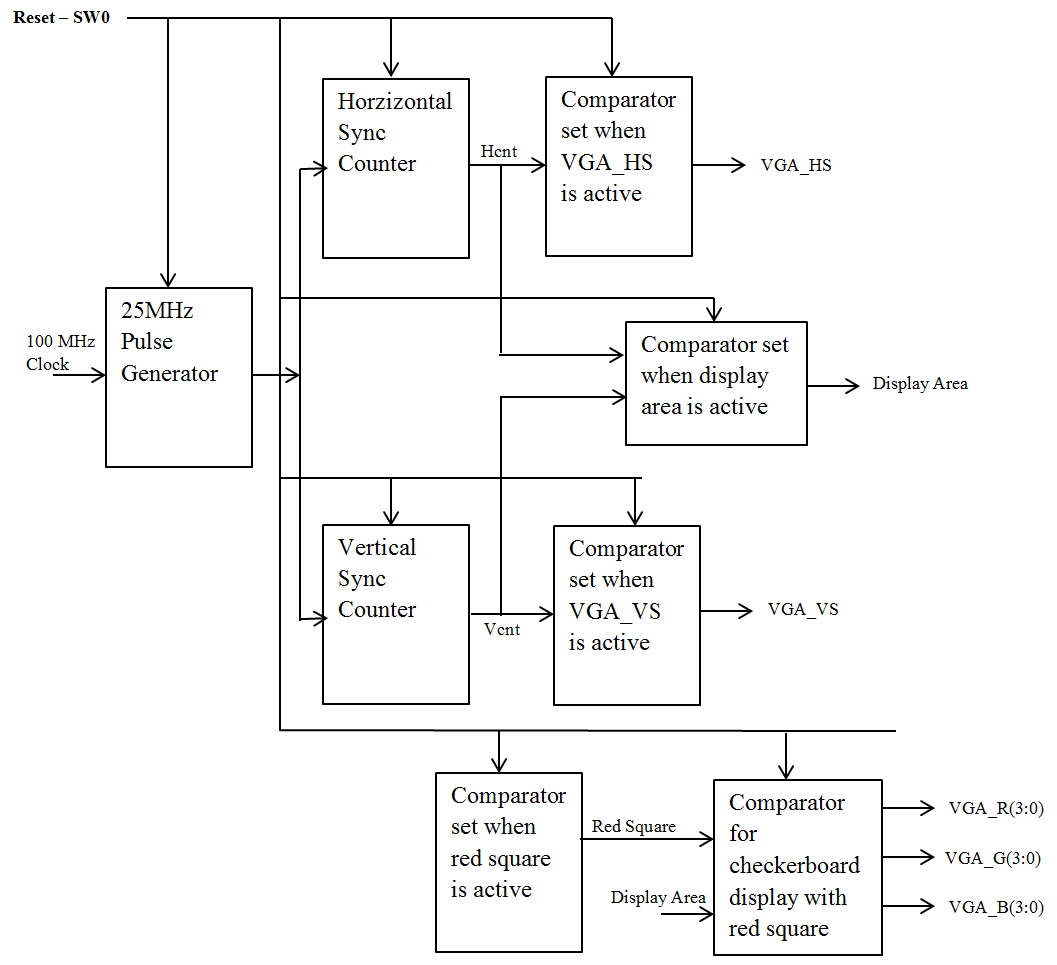


Figure 2: VGA Block Diagram

**VGA System Timing**

The following VGA system timing information is provided as an example of how a VGA monitor might be driven in 640 by 480 mode.

A standard monitor operates at 60 Hz. That is, the screen is refreshed 60 times per second, which is equal to once every 1/60th of a second. At this rate, the pixels appear bright and continuously illuminated without any noticeable "flicker" due to persistence of vision. Therefore, given a target resolution supported by the monitor, timing for each pixel, horizontal line, and vertical retrace can be derived.

**25 MHz pulse generator**

Figure 2 depicts a 25 MHz pulse generator that is used to drive the controller and as we will later discover, it is sufficient to accomplish the target resolution. We will refer to this as the pixel clock. The pixel clock defines the time available to display one pixel of information and thus influences the refresh rate.

A VGA controller circuit must generate the HS and VS timings signals and coordinate the delivery of video data based on the pixel clock.

Generate a 25 MHz enable pulse using a 100 MHz clock. Assume the pulse is high for one period of the 100 MHz clock and low otherwise (will go high for ONE 10ns interval (100MHz width) every 40 ns or a period of 25MHz). Name the pulse en25. You should generate a 25MHz enable signal derived from the 100MHz system clock.

constant maxCount : unsigned(1 downto 0) := "11"; --Binary value for 3 to divide by 4

signal reset : std\_logic;

signal en25 : std\_logic;

signal pulseCnt : unsigned(1 downto 0);

-- 25MHz Pulse Generator

process(CLK100MHZ, reset)

begin

if(reset = '1') then

pulseCnt <= (others=>'0'); = 2 bits

elsif(rising\_edge(CLK100MHZ)) then

if (en25 = '1') then

pulseCnt <= (others=>'0');

else

pulseCnt <= pulseCnt + 1; -- upcounter

end if;

end if;

end process;

en25 <= '1' when (pulseCnt = maxCount) else '0';

**Horizontal and Vertical Sync Counters**

Next from figure 2, we examine the horizontal sync and vertical sync counters. Recall a VGA controller circuit must generate the HS and VS timings signals and coordinate the delivery of video data based on the pixel clock.

**Horizontal Counters**

The **horizontal sync** **signal**, *HS* consists of four regions of a VGA monitor’s screen, the sync pulse *(SP),* back porch *(BP),* horizontal video *(HV)* display*,* and front porch *(FP).* The sync pulse signals the beginning of a new line and is accomplished by bringing *HS* low. The signal is then brought high for the back porch where pixels are not yet written to the screen at the left. After the back porch, the *HS* signal remains high during the horizontal video period (display time) where pixels are written to the screen proceeding from left to right. Finally, the *HS* signal also remains high during the front porch where no pixels are written to the screen at the right. Fig.3 shows the horizontal sync regions and timing.

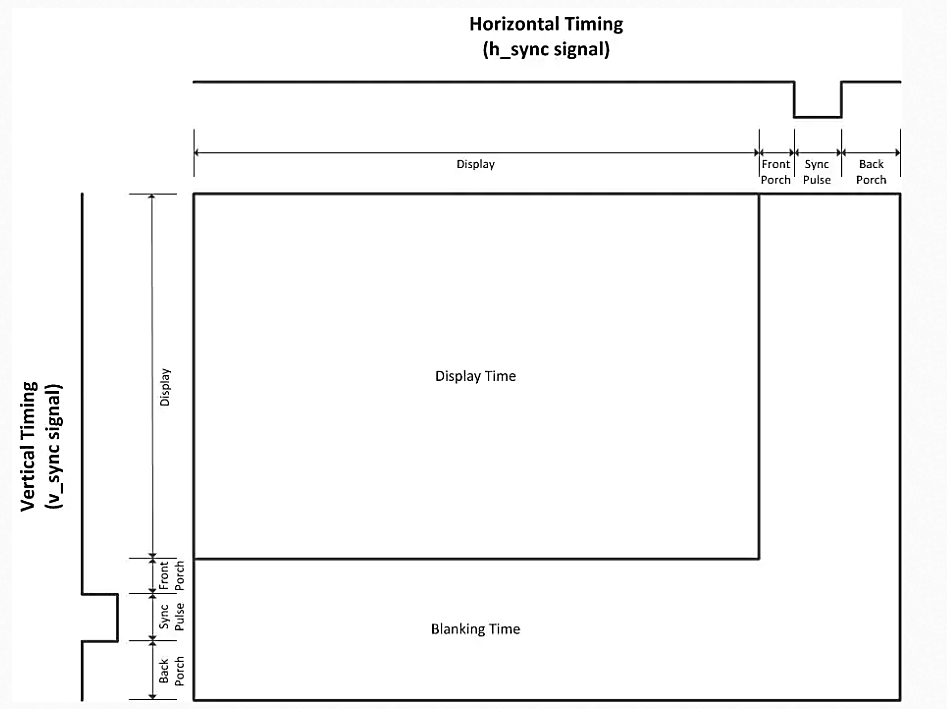


Figure 3: Regions of a VGA monitor’s screen

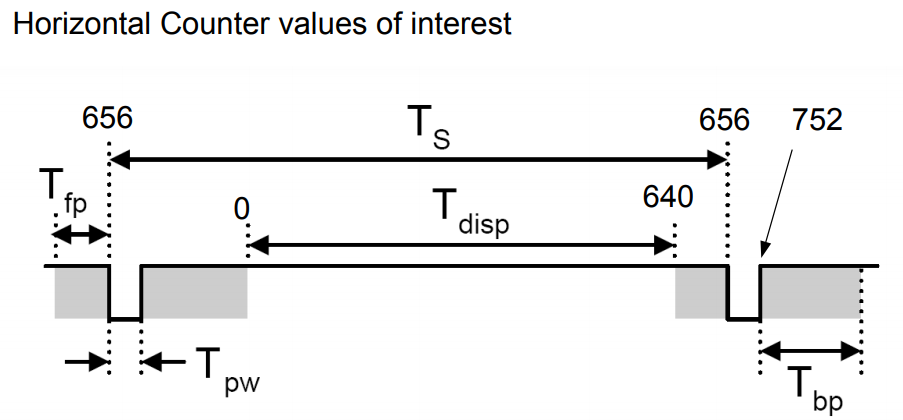


Figure 4: Horizontal Counter

Considering the length of a horizontal line by counting pixels, we have 96 invisible pixels for the sync pulse where *HS* is low. The horizontal video region, where pixels are visible, starts after both the sync pulse and invisible back porch at 48 + 96 = 144 and continues for 640 pixels stopping at 144 + 640 = 784. At this point, there is no visible video again at the front porch for 16 pixels until 784 + 16 = 800 where the sequence resets to zero, upon which it starts counting again from 0 to 799. Therefore in the code for the counter for the horizontal sync signal VGA\_HS, it is evident why the counter counts to a value of 800. These values are summarized in Figure 5 for the Horiz. sync Clks column below.

When the counter equals hpixels, it has reached 800 and must start over. At that time, it also must signal the vertical line counter to count a line by bringing vsenable high for one clock cycle (The vertical counter increments every time the horizontal counter resets. This is required in order to know when it reaches the bottom of the screen, where a vertical sync pulse causes it to start over again from the top left corner). Also, recall that the sync pulse signals the beginning of a new line and is accomplished by bringing VGA\_HSlow. This is depicted in Figure 4 and executed in the last line of code below. Otherwise, VGA\_HS is high. At this point, we are only setting the VGA\_HS signal. Later we will consider another signal, vidon, to differentiate the visible video ranges from the porches.

constant hpixels : std\_logic\_vector (9 downto 0) := "1100100000"

-- 1100100000 is decimal 800 which is the quantity of pixels in a horizontal line

signal hcs: std\_logic\_vector(9 downto 0); -- Horizontal counters

signal vsenable: std\_logic; --Enable for the Vertical counter

VGA\_HS: out std\_logic; -- Horizontal sync pulse 1-bit wide

--Counter for the horizontal sync signal VGA\_HS

process(CLK100MHZ, reset)

begin

if(reset = '1') then

horz\_cntr <= (others=>'0'); = 10 bits

elsif (rising\_edge(CLK100MHZ)) then

if horz\_cntr = hpixels - 1 then --The horizontal counter has reached the end of pixel count from 0 to 799

horz\_cntr <= (others => '0'); --reset the counter

vsenable <= '1'; --Enable the vertical counter = 1 bit

elsif (en25 = '1') then

horz\_cntr <= horz\_cntr + 1; --Increment the horizontal counter

vsenable <= '0'; --Leave the vsenable off

end if;

end if;

end process;

VGA\_HS <= '0' when (horz\_cntr >=656 and horz\_cntr < 752) else '1';

--Horizontal Sync Pulse is '0' between [656,752) otherwise '1'.

**Vertical Sync Counters**

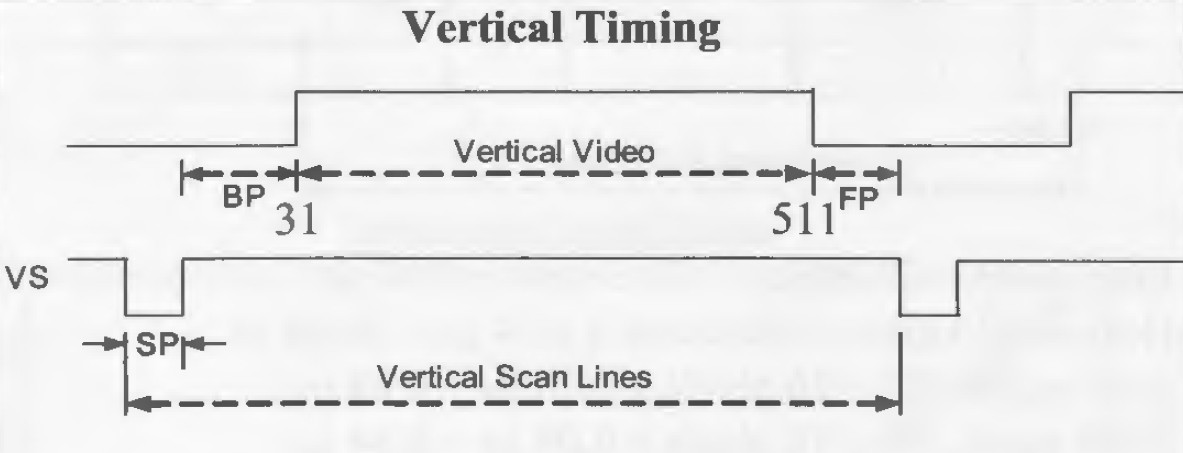


Figure 5: Vertical Sync

Considering the length of a vertical line by counting pixels, we have 2 invisible lines for the vertical sync pulse SP where the signal is low for 2 lines and the invisible back porch for 29 lines for a total count of 31. Then, vertical scan line is brought high for the next 480 lines in the vertical video visible region for a total of 511 lines. Finally we have the front porch for 10 lines and are invisible where the vertical scan drops low again for a total of 521 lines. Figure 5 summarizes these values.

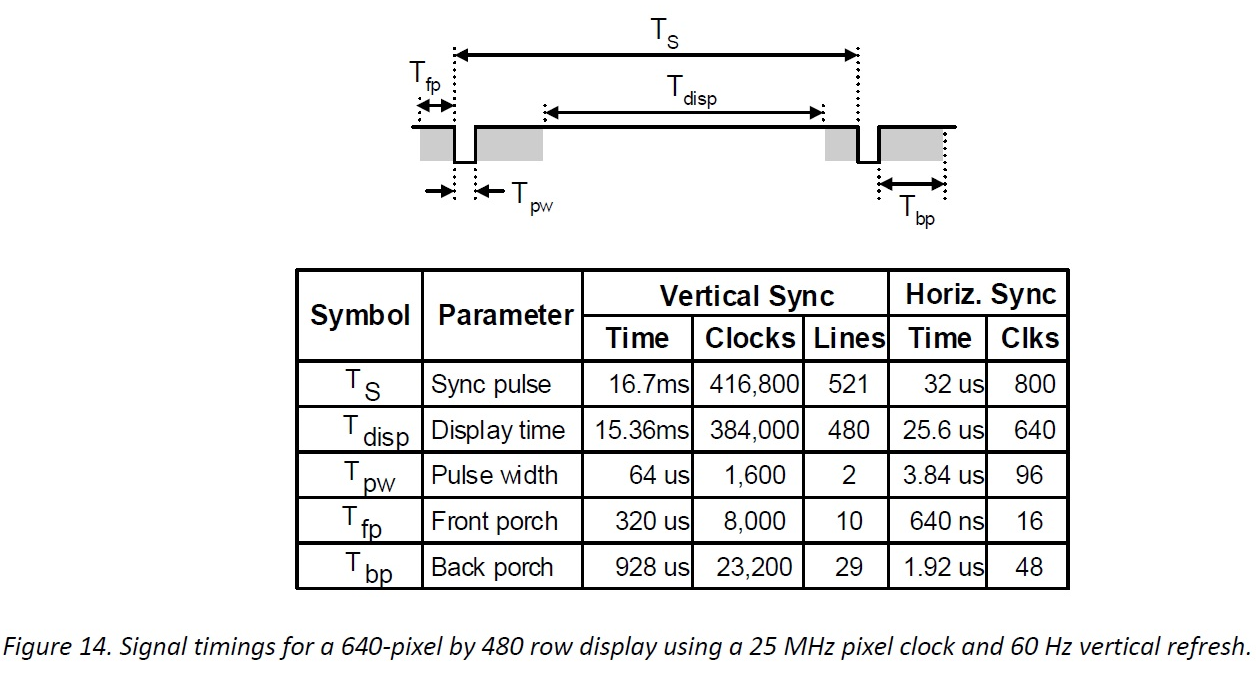


Figure 6: Vertical Timing

When the **counter** reaches 521 vertical scan lines, the sequence starts over again and draws a new frame.

Now to turn our attention to why a 25 MHz for a pixel clock rate was chosen. The horizontal sync timing required 800 clock ticks. The period of 25 MHz is 0.04 μs. 800 clock ticks \* 0.04 μs = 32 μs is the time to scan a cross one line. We then multiple this by 521 vertical lines or 32 μs \* 521 = 16.7 ms which is equal to a refresh rate of once every 1/60th of a second. That is, the screen is refreshed 60 times per second or 60 Hz which is where standard monitor operates at.

Recall when the horizontal counter equals hpixels, it has reached 800 and must start over. At that time, it also must signal the vertical line counter to count a line by bringing vsenable high for one clock cycle (The vertical counter increments every time the horizontal counter resets. This is required in order to know when it reaches the bottom of the screen, where a vertical sync pulse causes it to start over again from the top left corner).

constant vlines : unsigned (9 downto 0) := to\_unsigned (521,10);

--"1000001001"; --decimal 521 has 10 binary digits (10 bits wide word)

signal vert\_cntr : unsigned(9 downto 0); -- Vertical counter

signal vsenable: std\_logic; --Enable for the Vertical counter

VGA\_VS: out std\_logic; -- Vertical sync pulse 1-bit wide

--Counter for the vertical sync siglal VGA\_VSV

process(CLK100MHZ, reset)

begin

if(reset = '1') then

vert\_cntr <= (others=>'0'); = 10 bits

elsif (rising\_edge(CLK100MHZ) and vsenable = '1') then -- Increment when enabled

if vert\_cntr = vlines - 1 then --The vertical counter has reached the end of pixel count from 0 to 520

vert\_cntr <= (others => '0'); --reset the counter

elsif (en25 = '1') then

vert\_cntr <= vert\_cntr + 1 ; -- Increment vertical upcounter

end if;

end if;

end process;

VGA\_VS <= '0' when (vert\_cntr >= 490 and vert\_cntr < 492) else '1'; --Vertical Sync Pulse is '0' between

We set the *vidon* signal to high when the horizontal counter is between the horizontal back porch and front porch and the vertical counter is between the vertical back porch and front porch. Otherwise, *vidon* is low where there is no visible video.

--Enable video out when within the porches

display\_sig <= '1' when (((horz\_cntr >= 0) and (horz\_cntr < 640))

and (vert\_cntr >= 0) and (vert\_cntr < 480))

else '0';

--display <= display\_sig;

**Controlling Red Square Movement**

The default position for the red square upon initialization is the upper left hand corner square.

The red square moves around the screen in response by pressing the up, down, left, and right push buttons. The red square wraps around the display and move appropriately at the edges. Switch 0 is used as an active high reset to the design. Resetting the design moves the red square back to its starting location.

All squares including the red square are 32x32 pixels (i.e. 32 x15 rows = 480 pixels and 32 x 20 columns = 640 pixels). Thus

constant widthRedSq : integer := 32; -- Width of the red square

constant heightRedSq : integer := 32; -- Height of the red square

-- Signals Col and Row keep track of the current column and row of the relative to the upper LH position of the red square.

signal Col, Row : std\_logic\_vector (10 downto 0);

signal display\_RedSq, R, G, B: std\_logic;

-- Location red square based on

col <= columnPbCntr & "00000"; -- affects horizontal location of square

row <= rowPbCntr & "00000"; -- affects vertical location of square

The columnPbCntr value is determined by the counters for the column BTNR and BTNL pushbuttons that move the red square left or right.

The rowPbCntr value is determined by the counters for the row BTND and BTNU pushbutton that move the red square down or up

-- Enable red square within these 32 pixel boundaries

display\_RedSq <= '1' when (((horz\_cntr >= 0 + col) and (horz\_cntr < 0 + col + widthRedSq))

and (vert\_cntr >= 0 + row) and (vert\_cntr < 0 + row + heightRedSq))

else '0';

signal display\_sig : std\_logic;

signal red\_sig, green\_sig, blue\_sig : std\_logic\_vector(3 downto 0); -- 4-bit color

-- Draw red square

process(display\_sig, vert\_cntr, horz\_cntr)

begin

red\_sig <= (others=>'0'); -- Intialize values to "0000";

green\_sig <= (others=>'0'); -- Intialize values to "0000";

blue\_sig <= (others=>'0'); -- Intialize values to "0000";

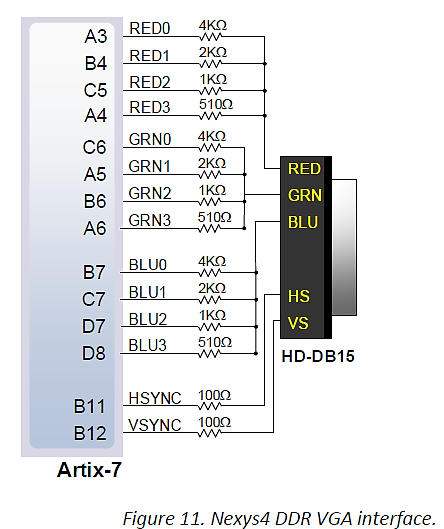
if(display\_sig = '1' and display\_RedSq = '1') then

red\_sig <= (others=>'1'); -- Draw red square

**RGB Color Signals**

A VGA controller is a component that controls five signals basic to video display. These signals are the horizontal sync HS, the vertical sync VS, and the three color signals R, G, and B which output the level of red, green, and blue, respectively. A VGA monitor works on the basis of emitting energy in the red, green, and blue spectrum proportional to the voltage on the corresponding R, G, or B signal input to the screen. Each colored dot on the screen is called a pixel (for picture element). How the monitor accomplishes this emission of color depends on the technology such as a cathode ray tube monitor or a liquid crystal display. In either case, the screen will display pixels starting from the top left corner of the screen moving towards the right and transitions, line-by-line in a direction moving towards the bottom of the screen. A horizontal sync pulse synchronizes each new line. Once it reaches the bottom of the screen, a vertical sync pulse causes it to start over again from the top left corner. The video controller must continuously output R, G, and B levels while synchronizing the horizontal lines and timing the vertical retrace to refresh the screen.

The R, G, and B inputs to a monitor are analog. However, the output signals from the FPGA are digital outputs that need to be converted to an analog signal using some type of D/A converter. The Artix7 use a simple 4-resistor circuit of the type shown in Fig. 11 to convert a 4-bit red signal to R(3:0) to an 16-level analog signal VR.



It uses a similar circuit to convert four bits of R green and four bits of blue to corresponding analog signals. Thus, these Digilent FPGA boards support 2^4 =16-bit VGA color - four bits of red, green, and blue. This will produce 16^3 = 4096 different colors.

Thus circuit produces video color signals that proceed in equal increments between 0V (fully off) and 0.7V (fully on). Using this circuit, 4096 different colors can be displayed, one for each unique 12-bit pattern. A video controller circuit must be created in the FPGA to drive the sync and color signals with the correct timing in order to produce a working display system.

Recall the checkerboard pattern is 15 rows by 20 columns occupying a total visible area of 480x640 pixels. Therefore, each square is 32x32 pixels (i.e. 32 x15 rows = 480 pixels and 32 x 20 columns = 640 pixels).

Therefore, we set the 4-bit **blue\_sig** with the argument to 5 (2^5 = 32 pixels.) to alternate with the **green\_sig.**

Use Power of 2 to simplify the design of checkerboard pattern In this case, the squares are 32x32, which is luckily a power of 2. With the state of bit 5 of the vertical (vc(5) and horizontal counters ((hc(5)), we can tell if the current pixel should be blue or green based on that pattern.

green <= vc(5) & vc(5) & vc(5) & vc(5) & hc(5) & hc(5) & hc(5) & hc(5)

blue <= not (vc(5) & vc(5) & vc(5) & vc(5) & hc(5) & hc(5) & hc(5) & hc(5))

Recall there are 15 rows and each row consist of 32 pixels. So for each row, we need to address 32 pixels at a time. For example, by keeping bits 5,6,7,8 active, others keep to 0, we get

row <= "0000000000"; = 1st row with all bits low

row <= "0000100000"; = 2nd row with 5th bit high only

row <= "0001000000"; = 3rd row with 6th bit high only

row <= "0001100000"; = 4th row with bits 5 and 6 high

row <= "0010000000"; = 5th row with bit 7 high only

row <= "0111000000"; = 15th row with bits 5,6,7,8 high 00111000000 = 448 and 448/32 = 14 rows from 0.

Y-coordinates range is from 0x00 to 0x0E (0 to14 in decimal for 15 squares) 14 Dec = 1110 binary

Similarly, there are 20 columns, each 32 pixels wide.

col <= "0000000000"; = 1st col with all bits low (coulmn 0)

col <= "0000100000"; = 2nd col with 5th bit high only (coulmn 1)

col <= "1001100000"; = 20th col 1001100000 = 608 and 608/32 = 19 columns from 0. (coulmn 19)

The X-coordinates range is from 0x00 to 0x13 (0 to 19 in decimal for 20 squares) 19 Dec = 10011 binary

Putting all this together, the code for the checkerboard display is:

signal display\_sig : std\_logic;

signal red\_sig, green\_sig, blue\_sig : std\_logic\_vector(3 downto 0); -- 4-bit color

process(display\_sig, vert\_cntr, horz\_cntr)

begin

red\_sig <= (others=>'0'); -- Intialize values to "0000";

green\_sig <= (others=>'0'); -- Intialize values to "0000";

blue\_sig <= (others=>'0'); -- Intialize values to "0000";

if(display\_sig = '1' and display\_RedSq = '1') then

red\_sig <= (others=>'1'); -- Draw red square

--Create checkerboard display wherever no red square is present

elsif(display\_sig = '1') then

blue\_sig <= (horz\_cntr(5) & horz\_cntr(5) & horz\_cntr(5) & horz\_cntr(5)) xor (vert\_cntr(5) & vert\_cntr(5) & vert\_cntr(5) & vert\_cntr(5));

green\_sig <= not ((horz\_cntr(5) & horz\_cntr(5) & horz\_cntr(5) & horz\_cntr(5)) xor (vert\_cntr(5) & vert\_cntr(5) & vert\_cntr(5) & vert\_cntr(5)));

end if;

end process;

VGA\_R <= red\_sig;

VGA\_G <= green\_sig;

VGA\_B <= blue\_sig;

**Seven-segment display controller**

The design also uses a 7-segment display controller displaying the X-coordinates on digits 3 and 2 of the dev board display and the Y-coordinates on digits 1 and 0 of the display. The X-coordinates range is from 0x00 to 0x13 and the Y-coordinates range is from 0x00 to 0x0E. For example, (x,y) of (0x00, 0x00) be displays as ‘0000’ and (0x11, 0x0C) would be shown as ‘110C.’ The origin is (0x00, 0x00) and is located in the upper left square of the grid.

The X-coordinates range is from 0x00 to 0x13 (0 to 19 in decimal for 20 squares) 19 Dec = 10011 binary

Y-coordinates range is from 0x00 to 0x0E (0 to14 in decimal for 15 squares) 14 Dec = 1110 binary