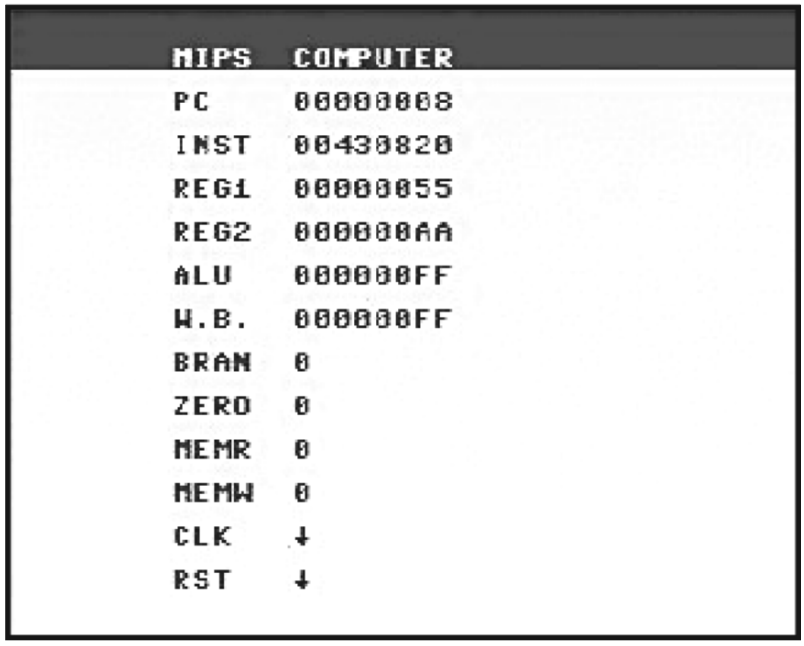
CHAPTER 10

VGA Video Display Generation using FPGAs



The video image above was produced by an FPGA board design.

10 VGA Video Display Generation using FPGAs

To understand how it is possible to generate a video image using an FPGA board, it is first necessary to understand the various components of a video signal. A VGA video signal contains 5 active signals. Two signals compatible with TTL logic levels, horizontal sync and vertical sync, are used for synchronization of the video. Three analog signals with 0.7 to 1.0-Volt peak-to-peak levels are used to control the color. The color signals are Red, Green, and Blue. They are often collectively referred to as the RGB signals. By changing the analog levels of the three RGB signals all other colors are produced.

10.1 Video Display Technology

The first technology used to display video images dictated the nature of the video signals. Even though LCD monitors are now in common use, the major component inside early VGA computer monitors was the color CRT or Cathode Ray Tube shown in Figure 10.1. The electron beam must be scanned over the viewing screen in a sequence of horizontal lines to generate an image. The deflection yoke uses magnetic or electrostatic fields to deflect the electron beam to the appropriate position on the face of the CRT. The RGB color information in the video signal is used to control the strength of the electron beam. Light is generated when the beam is turned on by a video signal and it strikes a color phosphor dot or line on the face of the CRT. The face of a color CRT contains a series of rows with three different phosphors. One type of phosphor is used for each of the primary colors of red, green, and blue.

In standard VGA format, as seen in Figure 10.2, the screen contains 640 by 480 picture elements or pixels. The video signal must redraw the entire screen 60 times per second to provide for motion in the image and to reduce flicker. This period is called the refresh rate. The human eye can detect flicker at refresh rates less than 30 to 60Hz.

To reduce flicker from interference from fluorescent lighting sources, refresh rates higher than 60 Hz at around 70Hz are sometimes used in PC monitors. The color of each pixel is determined by the value of the RGB signals when the signal scans across each pixel. In 640 by 480-pixel mode, with a 60Hz refresh rate, this is approximately 40 ns per pixel. A 25MHz clock has a period of 40 ns. A slightly higher clock rate will produce a higher refresh rate.

10.2 Video Refresh

The screen refresh process seen in Figure 10.2 begins in the top left corner and paints 1 pixel at a time from left to right. At the end of the first row, the row increments and the column address is reset to the first column. Each row is painted until all pixels have been displayed. Once the entire screen has been painted, the refresh process begins again.

The video signal paints or refreshes the image using the following process. The vertical sync signal, as shown in Figure 10.3 tells the monitor to start displaying a new image or frame, and the monitor starts in the upper left corner

with pixel 0,0. The horizontal sync signal, as shown in Figure 10.4, tells the monitor to refresh another row of 640 pixels.

After 480 rows of pixels are refreshed with 480 horizontal sync signals, a vertical sync signal resets the monitor to the upper left comer and the process continues. During the time when pixel data is not being displayed and the beam is returning to the left column to start another horizontal scan, the RGB signals should all be set to the color black (all zeros).

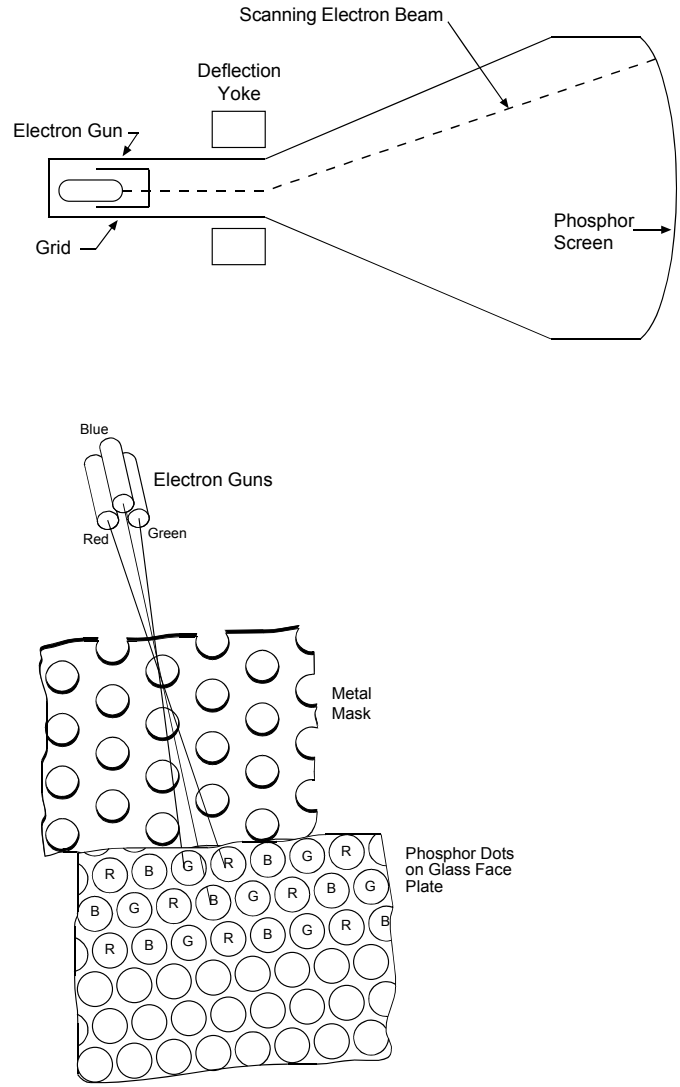


Figure 10.1 Color CRT and Phosphor Dots on Face of Display.

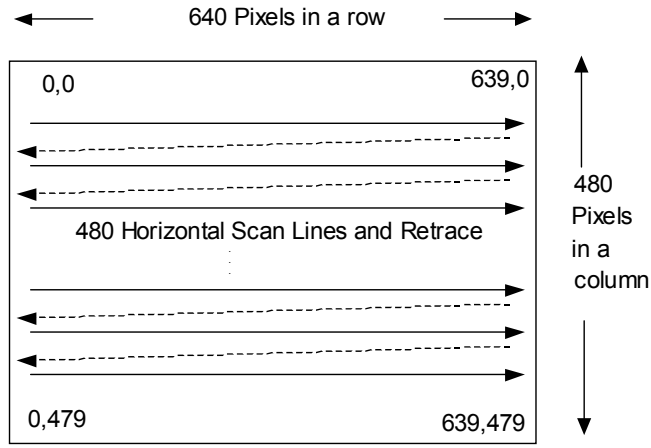


Figure 10.2 VGA Image - 640 by 480 Pixel Layout**.**

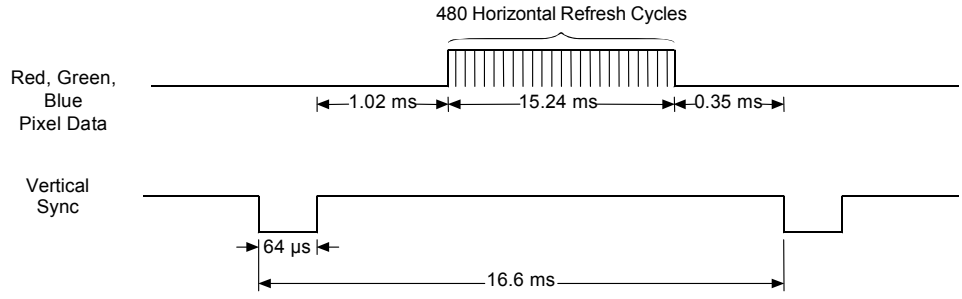


Figure 10.3 Vertical Sync Signal Timing for 640 by 480 at 60Hz.

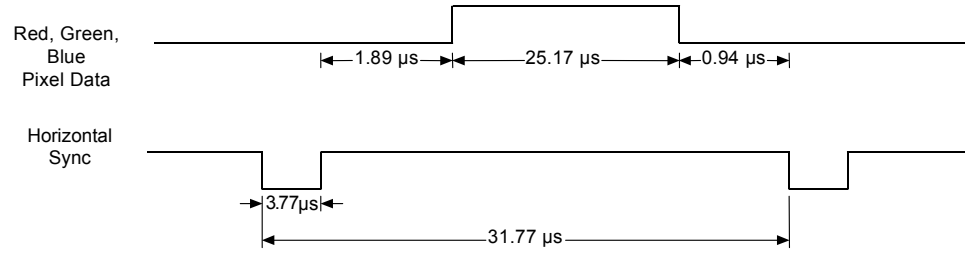


Figure 10.4 Horizontal Sync Signal Timing for 640 by 480 at 60Hz.

Many VGA monitors will shut down if the two sync signals are not the correct values. Most PC monitors have an LED that is green when it detects valid sync signals and yellow when it does not lock in with the sync signals. Modern monitors will sync up to an almost continuous range of refresh rates up to their design maximum. In a PC graphics card, a dedicated video memory location is used to store the color value of every pixel in the display. This memory is read out as the beam scans across the screen to produce the RGB signals. There is not enough memory inside current generation FPGA chips for this approach, so other techniques will be developed which require less memory.

10.3 Using an FPGA for VGA Video Signal Generation

To provide interesting output options in complex designs, video output can be developed using hardware inside the FPGA. Only five signals or pins are required, two sync signals and three RGB color signals. A simple resistor and diode circuit is used to convert TTL output pin signals from the FPGA to the low voltage analog RGB signals for the video signal. This supports two levels for each signal in the RGB data and thus produces a total of eight colors. This circuit and a VGA connector for a monitor are already installed on the Altera UP3 board. The FPGA's Phase Locked Loop (PLL) can be used to generate clocks for a wide variety of video resolutions and refresh rates.

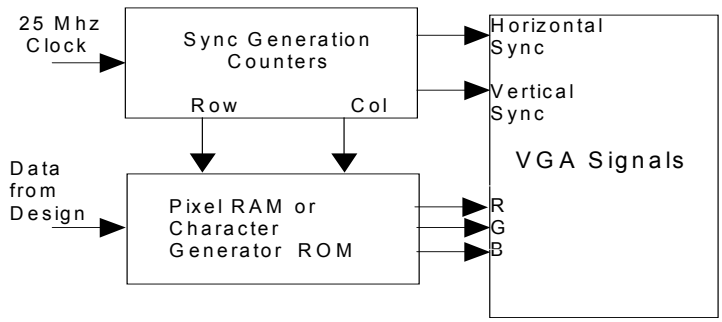
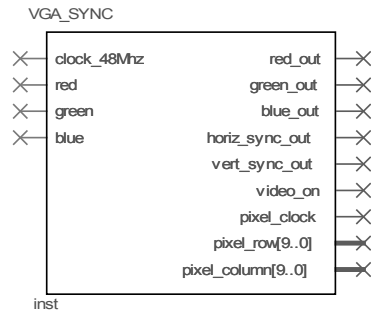


Figure 10.5 FPGA based generation of VGA Video Signals.

As seen in Figure 10.5, a 25.175 MHz clock, which is the 640 by 480 VGA pixel data rate of approximately 40ns is used to drive counters that generate the horizontal and vertical sync signals. Additional counters generate row and column addresses. In some designs, pixel resolution will be reduced from 640 by 480 to a lower resolution by using a clock divide operation on the row and column counters. The row and column addresses feed into a pixel RAM for graphics data or a character generator ROM when used to display text. The required RAM or ROM is also implemented inside the FPGA chip.

10.4 A VHDL Sync Generation Example: FPGAcore VGA_SYNC



The FPGAcore function, VGA_SYNC is used to generate the timing signals needed for a VGA video display. Although VGA_SYNC is written in VHDL, like the other FPGAcore functions it can be used as a symbol in a design created with any entry method.

The following VHDL code generates the horizontal and vertical sync signals, by using 10-bit counters, H_count for the horizontal count and V_count for the vertical count. H_count and V_count generate a pixel row and column address that is output and available for use by other processes. User logic uses these signals to determine the x and y coordinates of the present video location. The pixel address is used in generating the image's RGB color data. On all boards except the UP2 and UP1, the internal logic uses a 25 MHz clock generated by a PLL in the design file *Video_PLL.vhd*. Counters are used to produce video sync timing signals like those seen in figures 10.3 and 10.4. This process is used in all of the video examples that follow.

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY VGA_SYNC IS
    PORT(  clock_25MHz, red, green, blue : IN    STD_LOGIC;
           red_out, green_out, blue_out  : OUT  STD_LOGIC;
           horiz_sync_out, vert_sync_out : OUT  STD_LOGIC;
           pixel_row, pixel_column       : OUT  STD_LOGIC_VECTOR( 9 DOWNTO 0 ));
END VGA_SYNC;

ARCHITECTURE a OF VGA_SYNC IS
    SIGNAL horiz_sync, vert_sync                : STD_LOGIC;
    SIGNAL video_on, video_on_v, video_on_h : STD_LOGIC;
    SIGNAL h_count, v_count                     : STD_LOGIC_VECTOR( 9 DOWNTO 0 );

BEGIN

                                -- video_on is High only when RGB data is displayed
video_on <= video_on_H AND video_on_V;
```

```
PROCESS
    BEGIN
    WAIT UNTIL( clock_25MHz'EVENT ) AND ( clock_25MHz = '1' );

                        --Generate Horizontal and Vertical Timing Signals for Video Signal
                        -- H_count counts pixels (640 + extra time for sync signals)
                        --
                        -- Horiz_sync  ------------------------------------------_________--------
                        -- H_count      0             640                    659    755   799
                        --
    IF ( h_count = 799 ) THEN
        h_count <= "0000000000";
    ELSE
        h_count <= h_count + 1;
    END IF;

                        --Generate Horizontal Sync Signal using H_count
    IF ( h_count <= 755 ) AND  (h_count => 659 ) THEN
        horiz_sync <= '0';
    ELSE
        horiz_sync <= '1';
    END IF;
                        --V_count counts rows of pixels (480 + extra time for sync signals)
                        --
                        -- Vert_sync     ----------------------------------------_______------------
                        -- V_count       0               480             493-494       524
                        --
    IF ( v_count >= 524 ) AND ( h_count => 699 ) THEN
        v_count <= "0000000000";
    ELSIF ( h_count = 699 ) THEN
        v_count <= v_count + 1;
    END IF;
                        -- Generate Vertical Sync Signal using V_count
    IF ( v_count <= 494 ) AND ( v_count = >493 ) THEN
        vert_sync <= '0';
    ELSE
        vert_sync <= '1';
    END IF;
                        -- Generate Video on Screen Signals for Pixel Data
    IF ( h_count <= 639 ) THEN
        video_on_h   <= '1';
        pixel_column <= h_count;
    ELSE
        video_on_h <= '0';
    END IF;

    IF ( v_count <= 479 ) THEN
        video_on_v <= '1';
        pixel_row    <= v_count;
    ELSE
        video_on_v <= '0';
```

```
    END IF;

                                -- Put all video signals through DFFs to eliminate
                                -- any delays that can cause a blurry image
                                -- Turn off RGB outputs when outside video display area
    red_out          <= red AND video_on;
    green_out        <= green AND video_on;
    blue_out         <= blue AND video_on;
    horiz_sync_out   <= horiz_sync;
    vert_sync_out    <= vert_sync;

    END PROCESS;
END a;
```

To turn off RGB data when the pixels are not being displayed the video_on signals are generated. Video_on is gated with the RGB inputs to produce the RGB outputs. Video_on is low during the time that the beam is resetting to the start of a new line or screen. They are used in the logic for the final RGB outputs to force them to the zero state. VGA_SYNC also puts the all of video outputs through a final register to eliminate any timing differences in the video outputs. VGA_SYNC outputs the pixel row and column address. See the comments at the end of VGA_SYNC.VHD for information on setting up other screen resolutions and refresh rates.

10.5 Final Output Register for Video Signals

The final video output for the RGB and sync signals in any design should be directly from a flip-flop output. Even a small time delay of a few nanoseconds from the logic that generates the RGB color signals will cause a blurry video image. Since the RGB signals must be delayed a pixel clock period to eliminate any possible timing delays, the sync signals must also be delayed by clocking them through a D flip-flop. If the outputs all come directly from a flip-flop output, the video signals will all change at the same time and a sharper video image is produced. The last few lines of VHDL code in the FPGAcore VGA_SYNC design generate this final output register.

10.6 Required Pin Assignments for Video Output

The FPGA board requires the chip pins as seen in Table 10.1 to be defined in the project's *.qsf file, or elsewhere in your design in order to display the video signals. These pins are hard wired on the FPGA board to the VGA connector and cannot be changed.

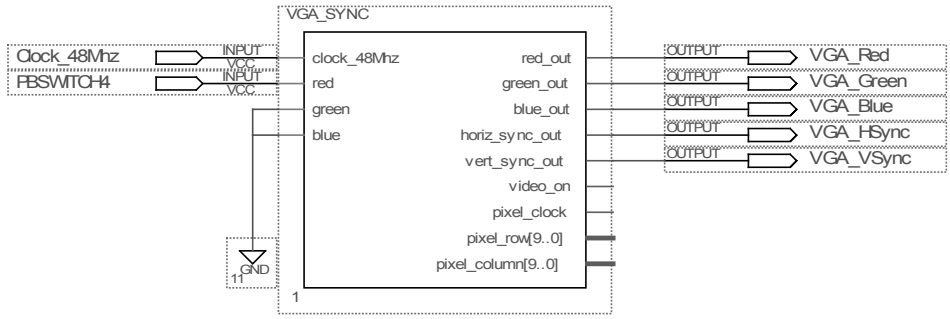
A pixel clock is also needed at the appropriate rate for the screen resolution and refresh rate. A PLL is used to generate this clock on the FPGA. The FPGA's external crystal controlled clock is used as input for the PLL on all boards except the UP2 and UP1 (no PLL on these boards). On the UP3, set jumper JP3 to short pins 3-4 for the 48Mhz clock. A table of the common screen resolutions and refresh rates with the required pixel clocks and sync counter values can be found at the end of the VGA_SYNC IP core code.

Table 10.1 The VGA Video Display Pin Assignments

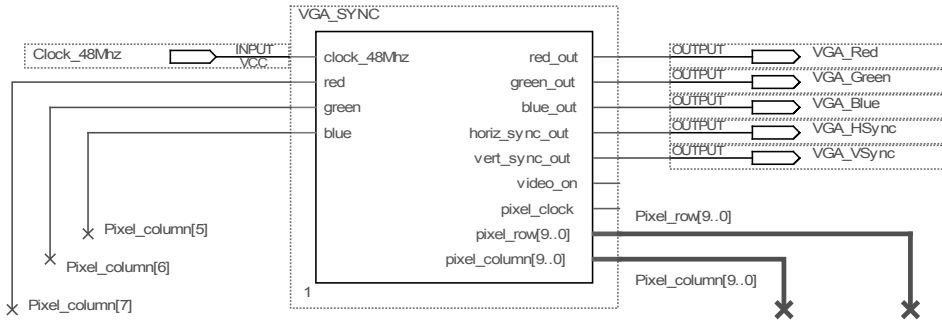
Pin Name	DE1	DE2	UP3	UP2, UP1	Pin Type	Function of Pin
CLOCK	L1 50Mhz	N2 50Mhz	153 48Mhz	91 25Mhz	Input	25-50MHz Clock
VGA_RED	B7	E10	228	236	Output	VGA Red Video Signal (highest bit on DE1/2)
VGA_GREEN	A8	D12	122	237	Output	VGA Green Video Signal (highest bit on DE1/2)
VGA_BLUE	B10	B12	170	238	Output	VGA Blue Video Signal (highest bit on DE1/2)
VGA_VSYNC	B11	D8	226	239	Output	VGA Connector Vertical Sync Signal
VGA_HSYNC	A11	A7	227	240	Output	VGA Connector Horizontal Sync Signal

10.7 Video Examples

For a simple video example with the VGA_SYNC function, the following schematic produces a video simulation of a red LED. When the PB1 pushbutton is hit, the color of the entire video screen will change from black to red. The VGA_LED project setup is seen below:



VGA_SYNC outputs the pixel row and column address. Pixel_row and Pixel_column are normally inputs to user logic that in turn generates the RGB color data. Here is a simple example that uses the pixel_column output to generate the RGB inputs. Bits 7, 6, and 5 of the pixel_column count are connected to the RGB data. Since bits 4 through 0 of pixel column are not connected, RGB color data will only change once every 32 pixels across the screen. This in turn generates a sequence of color bars in the video output. The color bars display the eight different colors that can be generated by the three digital RGB outputs in the VGA_BAR project.



10.8 A Character Based Video Design

One option is a video display that contains mainly textual data. For this approach, a pixel pattern or font is needed to display each different character. The character font can be stored in a ROM implemented inside the FPGA. A memory initialization file, *.mif, can be used to initialize the ROM contents during download. Given the memory limitations inside many FPGAs, one option that fits is a display of 40 characters by 30 lines.

Each letter, number, or symbol is a pixel image from the 8 by 8 character font. To make the characters larger, each dot in the font maps to a 2 by 2 pixel block so that a single character requires 16 by 16 pixels. This was done by dividing the row and column counters by 2. Recall that in binary, division by powers of two can be accomplished by truncating the lower bits, so no hardware is needed for this step. The row and column counters provide inputs to circuits that address the character font ROM and determine the color of each pixel. The clock used is the onboard 25.175MHz clock and other timing signals needed are obtained by dividing this clock down in hardware.

10.9 Character Selection and Fonts

Because the screen is constantly being refreshed and the video image is being generated on-the-fly as the beam moves across the video display, it is necessary to use other registers, ROM, or RAM inside the FPGA to hold and select the characters to be displayed on the screen. Each location in this character ROM or RAM contains only the starting address of the character font in font ROM. Using two levels of memory results in a design that is more compact and uses far less memory bits. This technique was used on early generation computers before the PC.

Here is an example implementation of a character font used in the FPGAcore function, char_ROM. To display an "A" the character ROM would contain only the starting address 000001 for the font table for "A". The 8 by 8 font in the character generation ROM would generate the letter "A" using the following eight memory words:

Address	Font Data
000001000 :	*00***11***000* ;
000001001 :	*00***111***00* ;
000001010 :	*0***1***00***11***0* ;
000001011 :	*0***1111***1*0 ;
000001100 :	*0***1***00***11***0* ;
000001101 :	*0***1***00***11***0* ;
000001110 :	*0***1***00***11***0* ;
000001111 :	*00000000* ;

Figure 10.6 Font Memory Data for the Character "A".

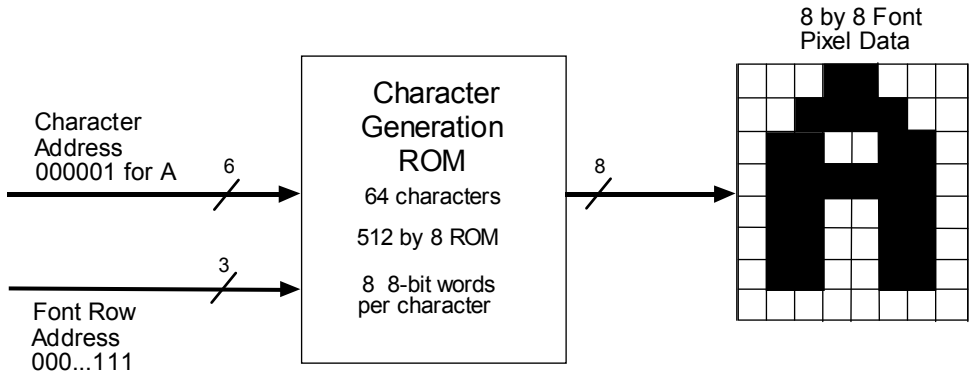
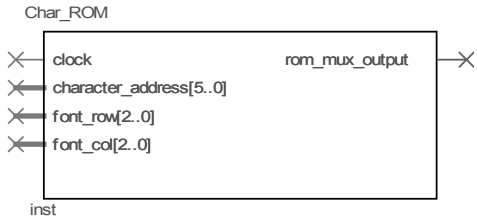


Figure 10.7 Accessing a Character Font Using a ROM.

The column counters are used to select each font bit from left to right in each word of font memory as the video signal moves across a row. This value is used to drive the logic for the RGB signals so that a "0" font bit has a different color from a "1". Using the low three character font row address bits, the row counter would select the next memory location from the character font ROM when the display moves to the next row.

A 3-bit font column address can be used with a multiplexer to select the appropriate bit from the ROM output word to drive the RGB pixel color data. Both the character font ROM and the multiplexer are contained in the FPGAcore char_ROM as shown below. The VHDL code declares the memory size using the LPM_ROM function and the tcgrom.mif file contains the initial values or font data for the ROM.




```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY Char_ROM IS
   PORT(    character_address    : IN     STD_LOGIC_VECTOR( 5 DOWNTO 0 );
            font_row, font_col   : IN     STD_LOGIC_VECTOR( 2 DOWNTO 0 );
            rom_mux_output       : OUT    STD_LOGIC);
END Char_ROM;

ARCHITECTURE a OF Char_ROM IS
   SIGNAL   rom_data             : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
   SIGNAL   rom_address          : STD_LOGIC_VECTOR( 8 DOWNTO 0 );
BEGIN
                        -- Small 8 by 8 Character Generator ROM for Video Display
                        -- Each character is 8 8-bit words of pixel data
 char_gen_rom: lpm_rom
    GENERIC MAP (
         lpm_widthad            => 9,
         lpm_numwords           => 512,
         lpm_outdata            => "UNREGISTERED",
         lpm_address_control    => "UNREGISTERED",
                                -- Reads in mif file for character generator font data
         lpm_file               => "tcgrom.mif",
         lpm_width              => 8)
    PORT MAP ( address => rom_address, q = > rom_data);
       rom_address <= character_address & font_row;
                        -- Mux to pick off correct rom data bit from 8-bit word
                        -- for on screen character generation
       rom_mux_output <= rom_data (
             (CONV_INTEGER( NOT font_col( 2 DOWNTO 0 ))) );
END a;
```

Table 10.2 Character Address Map for 8 by 8 Font ROM.

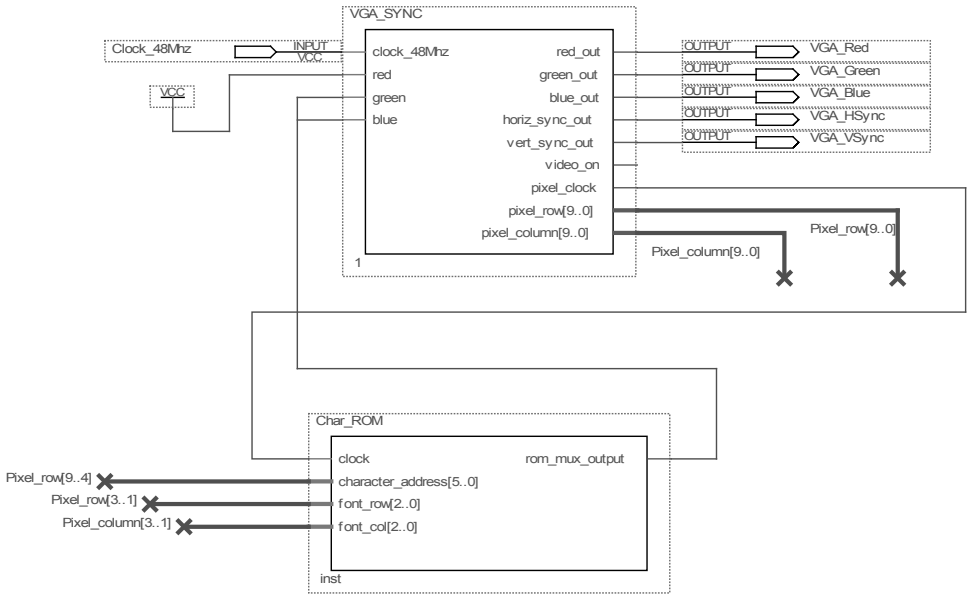
CHAR	ADDRESS	CHAR	ADDRESS	CHAR	ADDRESS	CHAR	ADDRESS
@	*00*	P	*20*	Space	*40*	0	*60*
A	*01*	Q	*21*	!	*41*	1	*61*
B	*02*	R	*22*	"	*42*	2	*62*
C	*03*	S	*23*	#	*43*	3	*63*
D	*04*	T	*24*	$	*44*	4	*64*
E	*05*	U	*25*	%	*45*	5	*65*
F	*06*	V	*26*	&	*46*	6	*66*
G	*07*	W	*27*	‘	*47*	7	*67*
H	*10*	X	*30*	(	*50*	8	*70*
I	*11*	Y	*31*	)	*51*	9	*71*
J	*12*	Z	*32*	*	*52*	A	*72*
K	*13*	[	*33*	+	*53*	B	*73*
L	*14*	Dn Arrow	*34*	,	*54*	C	*74*
M	*15*	]	*35*	-	*55*	D	*75*
N	*16*	Up Arrow	*36*	.	*56*	E	*76*
O	*17*	Lft Arrow	*37*	/	*57*	F	*77*

A 16 by 16 pixel area is used to display a single character with the character font. As the display moves to another character outside of the 16 by 16 pixel area, a different location is selected in the character RAM using the high bits of the row and column counters. This in turn selects another location in the character font ROM to display another character.

Due to limited ROM space, only the capital letters, numbers and some symbols are provided. Table 10.2 shows the alphanumeric characters followed by the high six bits of its octal character address in the font ROM. For example, a space is represented by octal code 40. The repeated letters A-F were used to simplify the conversion and display of hexadecimal values.

10.10 VHDL Character Display Design Examples

The FPGAcores VGA_SYNC and CHAR_ROM are designed to be used together to generate a text display. CHAR_ROM contains an 8 by 8 pixel character font. In the following schematic, a test pattern with 40 characters across with 30 lines down is displayed in the VGA_CHARACTER project. Examining the RGB inputs on the VGA_SYNC core you can see that characters will be white (111 = RGB) with a red (100 = RGB) background. Each character uses a 16 by 16 pixel area in the 640 by 480 display area. Since the low bit in the pixel row and column address is skipped in the font row and font column ROM inputs, each data bit from the font is a displayed in a 2 by 2 pixel area. Since pixel row bits 9 to 4 are used for the character address a new character will be displayed every 16^{th} pixel row or character line. Division by 16 occurs without any logic since the low four bits are not connected.



Normally, more complex user designed logic is used to generate the character address. The video example shown in Figure 10.8 is an implementation of the MIPS RISC processor core. The values of major busses are displayed in hexadecimal and it is possible to single step through instructions and watch the values on the video display. This example includes both constant and variable character display areas. The video setup is the same as the schematic, but additional logic is used to generate the character address.

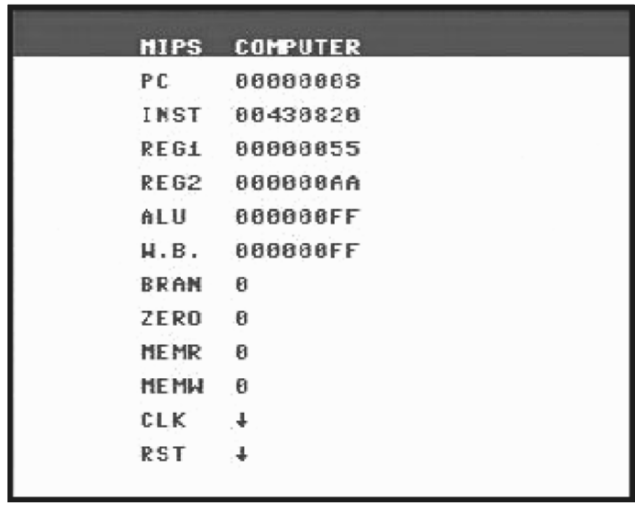


Figure 10.8 MIPS Computer Video Output.

Pixel row address and column address counters are used to determine the current character column and line position on the screen. They are generated as the image scans across the screen with the VGA_SYNC core by using the high six bits of the pixel row and pixel column outputs. Each character is a 16 by 16 block of pixels. The divide by 16 operation just requires truncation of the low four bits of the pixel row and column. The display area is 40 characters by 30 lines.

Constant character data for titles in the left column is stored in a small ROM called the character format ROM. This section of code sets up the format ROM that contains the character addresses for the constant character data in the left column of the video image for the display.

```
                        -- Character Format ROM for Video Display
                        -- Displays constant format character data
                        -- on left side of Display area
format_rom: lpm_rom
   GENERIC MAP (
     lpm_widthad          => 6,
     lpm_numwords         =>60,
     lpm_outdata          => "UNREGISTERED",
     lpm_address_control => "UNREGISTERED",
                        -- Reads in mif file for data display titles
     lpm_file             =>"format.mif",
     lpm_width            => 6)
```

Each pixel clock cycle, a process containing a series of nested CASE statements is used to select the character to display as the image scans across the screen. The CASE statements check the row and column counter outputs from the sync unit to determine the exact character column and character line that is currently being displayed. The CASE statements then output the character address for the desired character to the char_ROM FPGAcore.

Table 10.1 lists the address of each character in the font ROM. Alphabetic characters start at octal location 01 and numbers start at octal location 60. Octal location 40 contains a space that is used whenever no character is displayed. When the display is in the left column, data from the format_ROM is used. Any unused character display areas must select the space character that has blank or all zero font data.

Hexadecimal variables in the right column in Figure 10.8 are generated by using 4-bit data values from the design to index into the character font ROM. As an example, the value "11" & PC(7 DOWNTO 4), when used as the character address to the FPGAcore, char_ROM, will map into the character font for 0..9 and A..F. The actual hex character selected is based on the current value of the 4 bits in the VHDL signal, PC. As seen in the last column of Table 10.1, the letters, A..F, appear again after decimal numbers in the font ROM to simplify this hexadecimal mapping conversion.

10.11 A Graphics Memory Design Example

For another example, assume the display will be used to display only graphics data. The Cyclone EP1C6 FPGA contains 92K bits of memory. If only two colors are used in the RGB signals, one bit will be required for each pixel in the video RAM. If a 300 by 300 pixel video RAM was implemented in the Cyclone chip it would use all of the chip's 92K-bit memory. For full color RGB data of three bits per pixel, a 175 by 175 pixel RAM would use all of the 92K on-chip memory and no memory would be left for the remainder of the design.

Pixel memory must always be in read mode whenever RGB data is displayed. To avoid flicker and memory access conflicts on single port memory, designs should update pixel RAM and other signals that produce the RGB output, during the time the RGB data is not being displayed.

When the scan of each horizontal line is complete there are typically over 100 clock cycles before the next RGB value is needed, as seen in Figure 10.9. Additional clocks are available when a vertical sync signal resets the monitor to the first display line. The exact number of clocks available depends on the video resolution and refresh rate.

In most cases, calculations that change the video image should be performed during this off-screen period of time to avoid memory conflicts with the readout of video RAM or other registers which are used to produce the RGB video pixel color signals. Since on-chip pixel memory is limited, complex graphic designs with higher resolutions will require another approach.

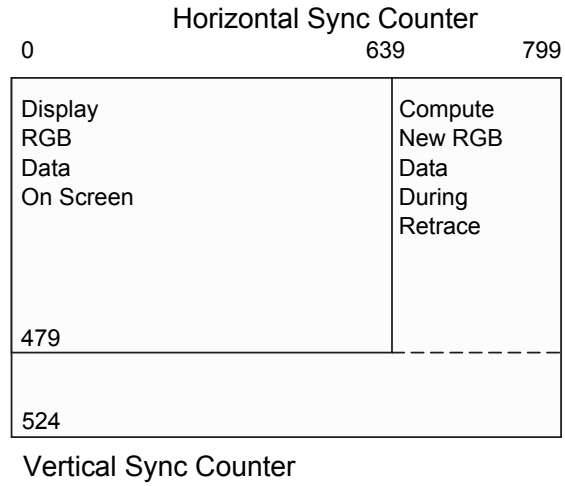


Figure 10.9 Display and Compute clock cycles available in a single 640 by 480 Video Frame.

10.12 Video Data Compression

Here are some ideas to save memory and produce more complex graphics. Compress the video pixel data in memory and uncompress it on-the-fly as the video signal is generated. One compression technique that works well is run length encoding (RLE). The RLE compression technique only requires a simple state machine and a counter for decoding.

In RLE, the pixels in the display are encoded into a sequence of length and color fields. The length field specifies the number of sequentially scanned pixels with the same color. In simple color images, substantial data compression can be achieved with RLE and it is used in PCs to encode color bitmaps. Matlab can be used to read bitmaps into a two-dimensional array and then write the output as an RLE encoded version directly to a *.mif file. An example program is available on the DVD. Bitmap file formats and some C utilities to help read bitmaps can be found on the web.

Many early video games, such as Pong, have a background color with a few moving images. In such cases, the background image can be the default color value and not stored in video RAM. Hardware comparators can check the row and column counts as the video signal is generated and detect when another image other than the background should be displayed. When the comparator signals that the row and column count matches the image location, the image's color data instead of the normal background data is switched into the RGB output using gates or a multiplexer.

The image can be made to move if its current row and column location is stored in registers and the output of these registers are used as the comparator input. Additional logic can be used to increment or decrement the image's location registers slowly over time and produce motion. Multiple hardware comparators can be used to support several fixed and moving images. These moving images are also called sprites. This approach was used in early-generation video games.

10.13 Video Color Mixing using Dithering

PC graphics cards use an analog to digital converter to drive the analog RGB color signals. Although the hardware directly supports only eight different pixel colors using digital color signals, there are some techniques that can be used to generate more colors. On analog CRTs, pixels can be overclocked at two to four times the normal rate to turn on and off the 1-bit color signal several times while scanning across a single pixel. The FPGA's PLL is handy to generate the higher frequency clocks need. Along the same lines, anding the final color signal output with the clock signal itself can further reduce the signal's on time to ½ a clock or less. Unfortunately, this technique does not work quite as well on LCD monitors due to the differences in the internal electronics.

The screen is refreshed at 60Hz, but flicker is almost undetected by the human eye at 30Hz. So, in odd refresh scans one pixel color is used and in even refresh scans another pixel color is used. This 30Hz color mixing or dithering technique works best if large areas have both colors arranged in a checkerboard pattern. Alternating scans use the inverse checkerboard colors. At 30Hz, the eye

can detect color intensity flicker in large regions unless the alternating checkerboard pattern is used.

10.14 VHDL Graphics Display Design Example

This simple graphics example will generate a ball that bounces up and down on the screen. As seen in Figure 10.10, in the VGA_BALL project the ball is red and the background is white. This example requires the VGA_SYNC design from Section 10.4 to generate the video sync and the pixel address signals. The pixel_row signal is used to determine the current row and the pixel_column signal determines the current column. Using the current row and column addresses, the process Display_Ball generates the red ball on the white background and produces the ball_on signal which displays the red ball using the logic in the red, green, and blue equations.

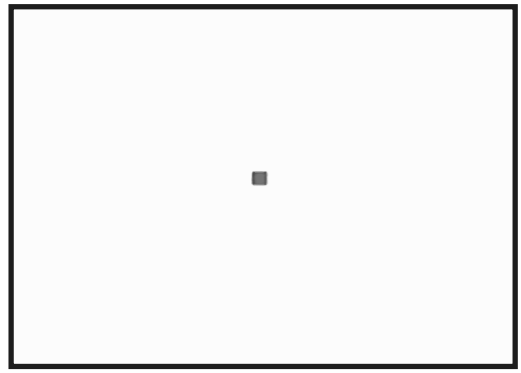


Figure 10.10 Bouncing Ball Video Output.

Ball_X_pos and Ball_y_pos are the current address of the center of the ball. Size is the size of the square ball.

The process Move_Ball moves the ball a few pixels every vertical sync and checks for bounces off of the walls. Ball_motion is the number of pixels to move the ball at each vertical sync clock. The VGA_SYNC core is also used to generate sync signals and pixel addresses, but is not shown in the code below.

```
ENTITY ball IS
    PORT(
        SIGNAL Red, Green, Blue          : OUT STD_LOGIC;
        SIGNAL vert_sync_out             : IN STD_LOGIC;
        SIGNAL pixel_row, pixel_column   : IN STD_LOGIC_VECTOR( 9 DOWNTO 0 ));
END ball;
ARCHITECTURE behavior OF ball IS
                                    -- Video Display Signals
        SIGNAL reset, Ball_on, Direction  : STD_LOGIC;
        SIGNAL Size                       : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
        SIGNAL Ball_Y_motion              : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
        SIGNAL Ball_Y_pos, Ball_X_pos     : STD_LOGIC_VECTOR( 9 DOWNTO 0 );
```

```
BEGIN                                   -- Size of Ball
    Size        <=  CONV_STD_LOGIC_VECTOR (8,10 );
                                        -- Ball center  X address
    Ball_X_pos <= CONV_STD_LOGIC_VECTOR( 320,10 );
                                        -- Colors for pixel data on video signal
    Red         <=  '1';                -- Turn off Green and Blue to make
                                        -- color Red when displaying ball
    Green       <= NOT Ball_on;
    Blue        <= NOT Ball_on;

Display_Ball:
    PROCESS ( Ball_X_pos, Ball_Y_pos, pixel_column, pixel_row, Size )
    BEGIN                               -- check row & column for ball area
                                        -- Set Ball_on = '1' to display ball
        IF  (  '0' & Ball_X_pos   <= pixel_column + Size ) AND
            ( Ball_X_pos + Size >= '0' & pixel_column    ) AND
            ( '0' & Ball_Y_pos    <= pixel_row + Size    ) AND
            ( Ball_Y_pos + Size >= '0' & pixel_row       ) THEN
                Ball_on <= '1';
        ELSE
                Ball_on <= '0';
        END IF;
    END PROCESS Display_Ball;

Move_Ball:
    PROCESS
    BEGIN
                        -- Move ball once every vertical sync
    WAIT UNTIL Vert_sync'EVENT AND Vert_sync = '1';
                        -- Bounce off top or bottom of screen
        IF ('0' & Ball_Y_pos) >= CONV_STD_LOGIC_VECTOR(480,10) - Size THEN
            Ball_Y_motion <=  CONV_STD_LOGIC_VECTOR(-2,10);
        ELSIF  Ball_Y_pos <= Size   THEN
            Ball_Y_motion <= CONV_STD_LOGIC_VECTOR(2,10);
        END IF;
                        -- Compute next ball Y position
        Ball_Y_pos <= Ball_Y_pos + Ball_Y_motion;
    END PROCESS Move_Ball;
END behavior;
```

10.15 Higher Video Resolution and Faster Refresh Rates

The Video Sync FPGAcore function is designed to support higher resolutions and refresh rates. The UP2 and UP1 boards can only support their Video Sync core's existing 640 by 480 60Hz video mode since it does not have an internal PLL to produce different pixel clocks. Table 10.3 shows several common screen resolutions and refresh rates. To change resolutions or refresh rates two changes are needed. First, change the PLL's video output pixel clock to the new frequency value by editing the Video_PLL.vhd file using the MegaWizard edit feature. Second, the six counter constant values used to generate the horizontal and vertical sync signals in the Video_Sync.vhd core need to be changed to the new six values for the desired resolution and refresh rate found in the large

table at the end of the Video_Sync.vhd file. Keep in mind that higher resolutions will require more pixel memory and smaller hardware delays that can support the faster clock rates needed.

Table 10.3 Pixel clock rates for some common video resolutions and refresh rates.

Mode	Refresh	Hor. Sync	Pixel clock
640x480	60Hz	31.5kHz	25.175MHz
640x480	63Hz	32.8kHz	28.322MHz
640x480	70Hz	36.5kHz	31.5MHz
640x480	72Hz	37.9kHz	31.5MHz
800x600	56Hz	35.1kHz	36.0MHz
800x600	60Hz	37.9kHz	40.0MHz
800x600	72Hz	48.0kHz	50.0MHz
1024x768	60Hz	48.4kHz	65.0MHz
1024x768	70Hz	56.5kHz	75.0MHz
1024x768	70Hz	56.25kHz	72.0MHz
1024x768	76Hz	62.5kHz	85.0MHz
1280x1024	61Hz	64.24kHz	110.0MHz
1280x1024	74Hz	78.85kHz	135.0MHz

10.16 Laboratory Exercises

1. Design a video output display that displays a large version of your initials. Hint: use the character generation ROM, the Video Sync FPGAcore, and some of the higher bits of the row and column pixel counters to generate larger characters.

2. Modify the bouncing ball example to bounce and move in both the X and Y directions. You will need to add code for motion in two directions and check additional walls for a bounce condition.

3. Modify the bouncing ball example to move up or down based on input from the two pushbuttons.

4. Modify the example to support different speeds. Read the speed of the ball from the FPGA switches.

5. Draw a more detailed ball in the bouncing ball example. Use a small ROM to hold a small detailed color image of a ball.

6. Make a Pong-type video game by using pushbutton input to move a paddle up and down that the ball will bounce off of.

7. Design your own video game with graphics. Some ideas include breakout, space invaders, Tetris, a slot machine, poker, craps, blackjack, pinball, and roulette. Keep the

graphics simple so that the design will fit on the FPGA chip. If the video game needs a random number generator, information on random number generation can be found in Appendix A.

8. Use the character font ROM and the ideas from the MIPS character output example to add video character output to another complex design.

9. Using Matlab or C, write a program to convert a color bitmap into a *.mif file with run-length encoding. Design a state machine to read out the memory and generate the RGB color signals to display the bitmap. Use a reduced resolution pixel size such as 160 by 120. Find a bitmap to display or create one with a paint program. It will work best if the bitmap is already 160 by 120 pixels or smaller. A school mascot or your favorite cartoon character might make an interesting choice. Limited internal memory is available in the FPGA, so a 12-bit RLE format with nine bits for length and three bits for color can be used with up to 7,600 locations on the UP3. Some boards have more slightly more memory as seen in Table 2.1. This means that the bitmap can only have several thousand color changes as the image is scanned across the display. Simple images such as cartoons have fewer color changes. A Matlab example is on the DVD.

10. Add color mixing or dithering with more than 8 colors to the previous problem. The 3-bit color code in the RLE encoded memory can be used to map into a color palette. The color palette contains the RGB patterns used for color mixing or interlacing. On boards that support more than 8 colors directly in hardware, the color palette value can be used directly to drive several of the higher RGB output bits. The color palette memory selects 8 different colors. The program translating the bitmap should select the 8 closest colors for the color palette.

11. Modify the VGA Sync core to support a higher screen resolution and demonstrate it using one of the earlier example video designs.