CS/ECE 3710 – Computer Design Lab
VGA Tutorial/Exercise

This tutorial/exercise is meant to help you understand and design a simple VGA controller driving a VGA display. It has been said that "There are two kinds of engineers – those who think VGA is hard, and those who understand VGA." That is to say, it's a little intimidating when you first see it, but it's really a very simple protocol for sending output information to a video screen. The steps are:

1. Read and understand the VGA specification. I recommend breaking this down into a VGA control circuit that generates the timing signals, and a Bitgen circuit that defines what color each pixel should be as the information is being sent to the display.
2. Design and implement a VGA control/timing circuit and simulate that circuit using ISE.
3. Design and demonstrate a very simple Bitgen allow for the movement a square using the buttons and change its color using the switches while the rest of the screen changes color randomly.
4. Think about how the Bitgen needs to interface to the simple inputs (buttons/switches) and it can instead interface with one of the read ports of a dual port block RAM (frame buffer) while the processor writes VGA information into the block.

VGA Basics

The term VGA really means one of two things depending on how you use the acronym. It's either a standard 15-pin connector used to drive video devices (e.g. a VGA cable) or it's the protocol used to drive information out on that cable (e.g. a VGA interface spec.). The interface defines how information is sent across the wires from your board to the VGA device. The cable defines which pins you use on the standard connector for those signals.

There is a nice description of VGA (both the connector and the protocol) in the Nexys 3 board reference manual pages 15-17. The most basic thing to know about VGA is that it is a protocol designed to be used with analog CRT (cathode ray tube) output devices. On these devices the electron beam moves across the screen from left to right as you're looking at the screen at a fixed rate (the refresh rate defines how fast the beam moves), and also moves down the screen from top to bottom at a fixed rate. While it's moving across and down the screen, you can modify the Red, Green, and Blue values on the VGA interface to control what color is being painted to the screen at the current location. So, painting a certain color on the screen is as easy as keeping track of where the beam is, and making sure the R, G, and B signals are at the right values when the beam is over the point on the screen where you want that color.

If you don't do anything to stop it, the beam will move to the right and bottom of the screen and get stuck there. You can force the beam to move back to the left by asserting an active-low signal called hSync (horizontal sync). You can force the beam to move back to the top of the screen by asserting an active-low signal called vSync (vertical sync). Because the beam moves at a fixed rate (defined by the monitor's refresh rate), you can keep track of where the beam is on the screen by counting clock ticks after the hSync and vSync signals.

Original author: Prof. Brunvand
Nexys 3 Revisions: Paymon Saebi

So, the basics of the VGA control/timer circuit are just a pair of counters to count horizontal ticks and vertical ticks of the VGA clock. How many ticks are there? That depends on how fast your clock is, and how many pixels you want to paint during the time the beam moves across the screen. The basic (ancient) standard for “plain” VGA is 640 pixels on each line, and 480 lines down the screen. This is “640x480” mode. Figure 1 shows a 640x480 screen, and the horizontal sync (hSync) timing required to make it work. After the hSync pulse, you must wait for a certain number of ticks before painting pixels to the screen. This gives the beam time to get back to the left and start moving forward again. This time is called the “back porch” because it’s after the hSync timing pulse. Then you count 640 pixels as the beam moves. After the 640th pixel, you wait for some amount of time (this is the “front porch” because it’s before the hSync), then assert the hSync signal (asserted low) for a certain amount of time.

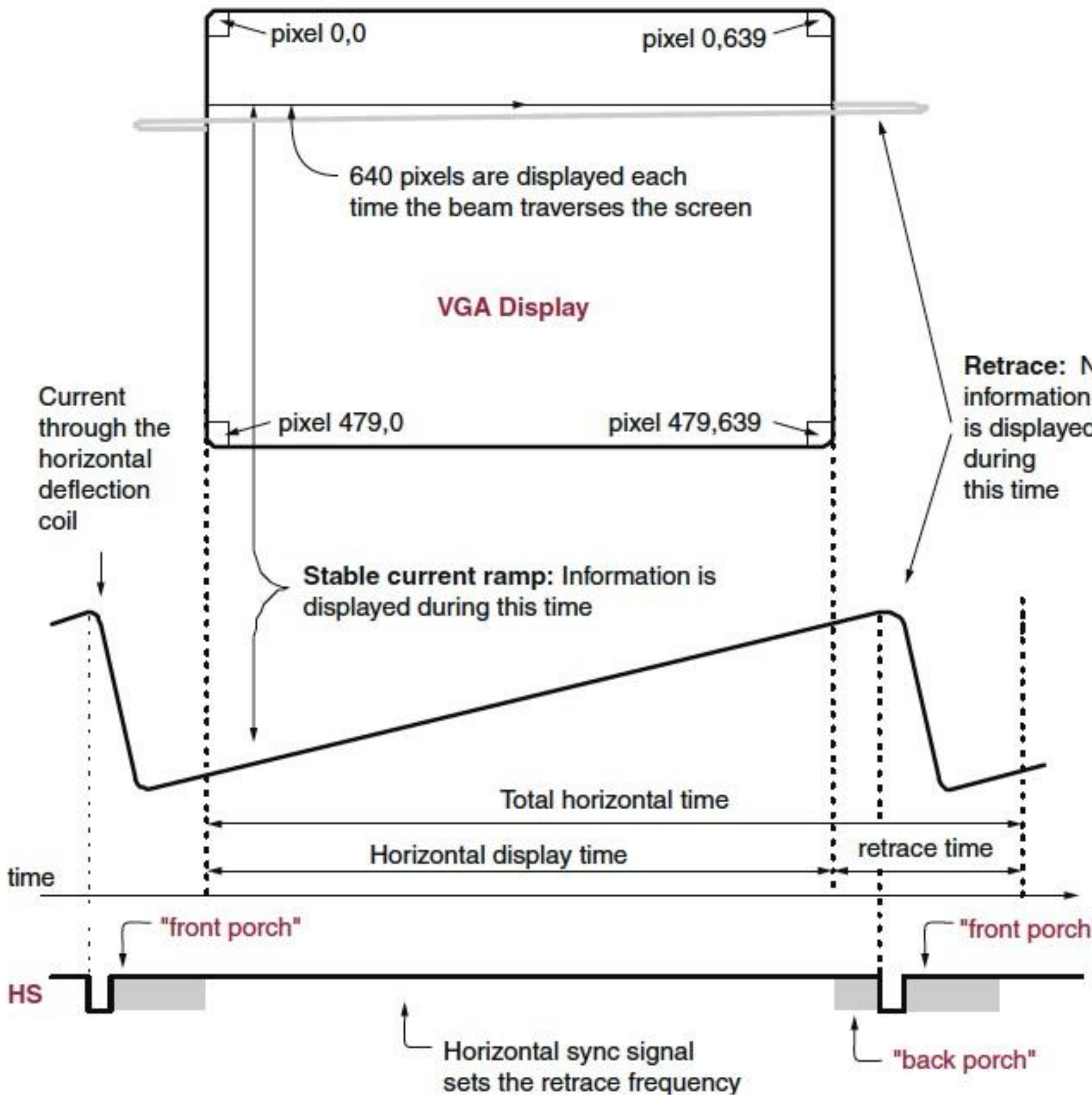

pixel 0,0
pixel 0,639
640 pixels are displayed each time the beam traverses the screen
VGA Display
Retrace: No information is displayed during this time
Current through the horizontal deflection coil
pixel 479,0
pixel 479,639
Stable current ramp: Information is displayed during this time
Total horizontal time
Horizontal display time
retrace time
time
"front porch"
"front porch"
HS
Horizontal sync signal sets the retrace frequency
"back porch"

Figure1: CRT VGA Horizontal Timing Example

The timing for all this depends on the monitor refresh rate. For a monitor with 60Hz refresh in 640x480 mode and a 25MHz pixel clock, you can use the timings in Figure 2. The timings in this figure define the display time (the time when the pixel is one of the 640 visible pixels in a line), pulse width (hSync or vSync), and the front porch and back porch timings. These times (in µs or ms) can also be measured in terms of the number of ticks of the 25MHz pixel clock, or in terms of the number of horizontal lines. That is, the vertical timing can be measured in terms of how many hSync pulses have been seen. The bottom line is that both the horizontal and vertical timing for the VGA_Controller are just counters. You may have to enable things or reset things or change things when the counters get to a certain value, but basically they're just counters.

Symbol	Parameter	Vertical Sync			Horizontal Sync	
		Time	Clocks	Lines	Time	Clocks
T_S	Sync pulse time	16.7 ms	416,800	521	32 µs	800
T_{DISP}	Display time	15.36 ms	384,000	480	25.6 µs	640
T_{PW}	Pulse width	64 µs	1,600	2	3.84 µs	96
T_{FP}	Front porch	320 µs	8,000	10	640 ns	16
T_{BP}	Back porch	928 µs	23,200	29	1.92 µs	48

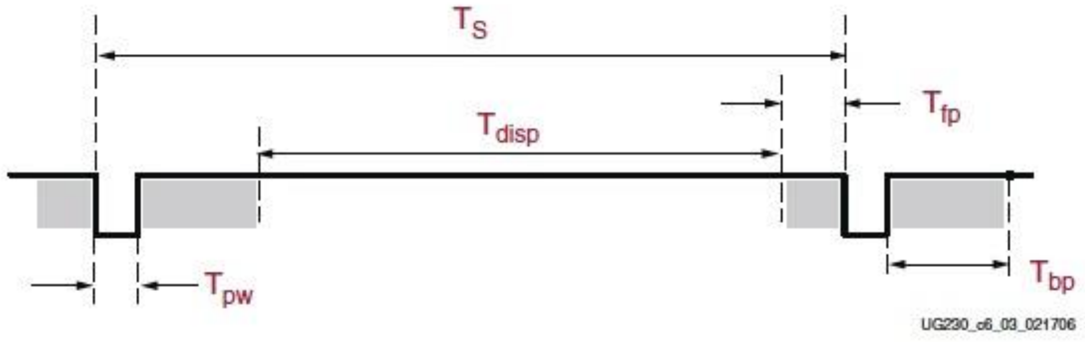


Figure2: VGA Control Timing at 60Hz refresh and 25MHz pixel clock

VGA_Controller circuit

I recommend designing a circuit that has two counters: one to keep track of horizontal location and one that keeps track of vertical location. The horizontal counter should count up in 25MHz ticks, and the vertical counter should count once for each full horizontal line.

Note that if you're going to use a purely synchronous design method (which I highly recommend!), you should NOT expect to have a 25MHz clock at your disposal. The system clock on the Nexys 3 board is 100MHz. You either need to setup the board's digital clock manager to give a phase locked divided clock (more accurate, but complicated) or you could divide the clock by four and use it as the 25MHz clock, but that would NOT be purely synchronous because you would be using a derived signal as your clock. Instead you should generate an enable signal (pulse) that is asserted every four 100MHz clocks, and use that enable signal in your horizontal pixel counter. For example, the horizontal pixel counter might look like:

```
// An example of a counter that only counts when En is high.
// This particular En is assumed to be high for a single clock
// cycle when it’s time to count
always@(posedge clk100MHz)
     if (clr)
          count <= 0; // clear count if clr is asserted
     else if (En)
          count <= count + 1; // If En is high, count
     // otherwise, hold previous value (default)
```

Of course, there are potentially lots of details missing here (like when to wrap around back to 0, for example), but this is an example of how you might count synchronously at 25MHz if there was a 25MHz pulse signal available.

For your vertical counter, I recommend generating an enable signal from your horizontal counter once per line and letting your vertical counter count up by one only when it’s enabled.

The main thing your VGA control circuit should do is generate hSync and vSync at the appropriate times based on the VGA timing. However, if you’re going to know where the beam is on the screen so that your Bitgen signal knows what color to put there, you also need to know which pixel and which line the beam is currently over. So, you also need to export the hCount and vCount signals so you can figure out which pixel is being painted. Finally, it’s a good idea to generate a signal from your VGA control circuit that is asserted whenever the beam is in a “pixel bright” area of the screen. That is, you are NOT in a front porch, back porch, or sync section of the timing.

I recommend that your VGA control circuit have the following interface:

```
module VGA_Controller(input              clock, clear,
                      output reg         hSync, vSync, bright,
                      output reg [9:0]   hCount, vCount);
```

The body of this module is, of course, up to you. It should generate the outputs based on the 640x480 60Hz refresh timing in Figure 2. You can validate the timing by simulating the VGA_Controller circuit in the ISE simulator. Some things to consider when you define this module:

- Remember that hSync and vSync are asserted low, high the rest of the time.
- hCount and vCount are used by your Bitgen circuit so that it knows where you are on the screen. It’s handy if the value of hCount is the value of the pixel. That is, if you start counting at 0 when you start the Horizontal Display Time from Figure 1, then the number on the counter will be the pixel number in the line (between 0 and 639).
- In the same way, if you start counting vertical lines in the Vertical Display Time portion of the vertical timing, then the value on the counter will be the line number (between 0 and 479)

- The bright signal can be either asserted high or low – it's used by your Bitgen circuit to tell that you're in a section of the screen where you want to turn on the pixel. If you're "not bright" you should force the RGB output to all zeros so that you're not driving the screen.

The VGA_Bitgen Circuit

This is the combinational circuit that takes the hCount, vCount, and bright signals, and decides for each pixel what color should be on the screen. At a minimum a Bitgen circuit would have the following interface:

```
module VGA_Bitgen(input              bright,
                  input      [7:0] pixelData,
                  input      [9:0] hCount, vCount,
                  output reg [7:0] rgb);
```

How does your Bitgen circuit know what the color should be? There are lots of ways, but the three general techniques are:

1. **Bitmapped graphics:** in this technique you have a "frame buffer" that holds the RGB values for every pixel on the screen. You can address this buffer using the hCount and vCount signals to retrieve the color at each of the 640x480 locations on the screen. Generally the VGA portion of the block RAM (frame buffer) is dual-ported so that, for example, a CPU can be putting data into the frame buffer and the VGA Bitgen circuit can be reading it out at the same time. This is the most memory-intensive technique and in 640x480 it has 307,200 pixel locations on the screen. If you pack two 8-bit rgb pixel values in each 16-bit word in memory, that's still 153,600 words of storage to hold the whole 640x480 screen (~300kBytes) and you are a limited to 32k of 16-bit words for your entire system using the Nexys 3 block RAM.

2. **Character/glyph graphics:** Instead of storing each of the pixels in the frame buffer, you can break the screen into chunks and store a pointer to a glyph in each screen chunk. For example, if you break the screen into 8x8 pixel chunks, then there are 80x60 chunk locations on the screen (this is 4800 locations total). If you have 256 possible 8x8 glyphs stored somewhere else, then you only need 4800 bytes (4.69kBytes) to store the screen (one 8-bit glyph pointer for each screen chunk), but at each location on the screen you are limited to one of the 256 8x8 glyphs. The glyphs themselves need to be in a separate memory that consists of 2k-4k locations of 8-bits each. So, the total storage requirement is at least 6848 bytes (6.69kBytes), a lot less than 150kBytes, but less flexible. It's common to have alphanumeric characters and some graphic glyphs in the glyph memory. You can use the hCount and vCount values not only to tell which chunk you're in, but also which pixel of the glyph you're currently in. Hints – If the glyphs are 8 pixels across, which bits of the hCount counter define the pixel within the glyph? How many bits does it take to count to 8 pixels? Which bits of hCount define which glyph you're in? You only want those bits to change once every 8 ticks of the pixel clock.

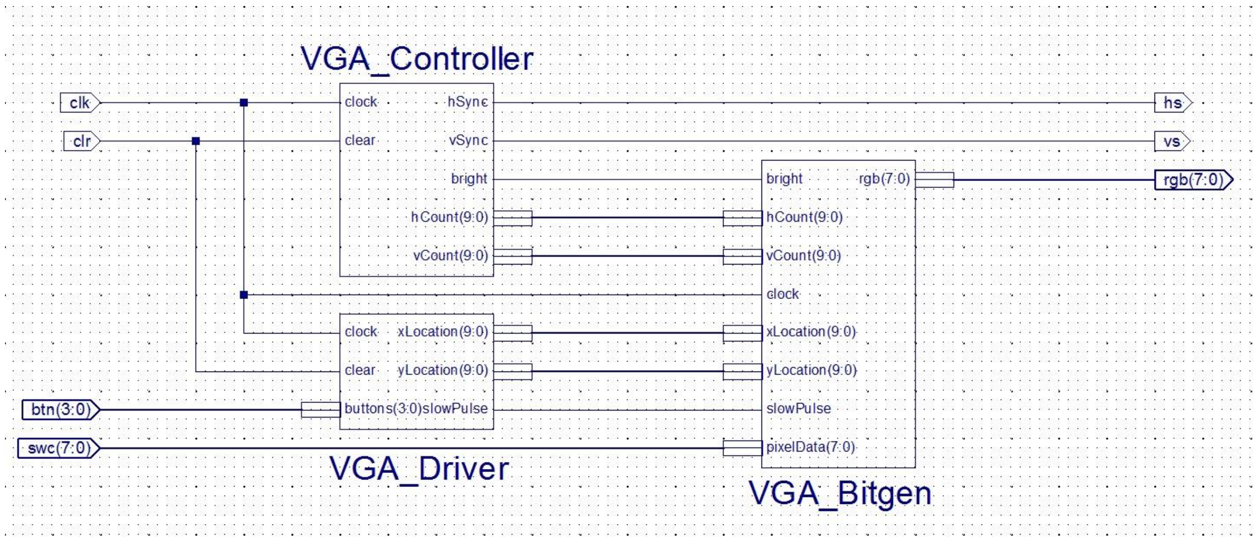
3. **Direct graphics:** Instead of storing information about what’s at each screen location, you can design a circuit that checks the current hCount and vCount and draws directly on the screen when you’re in the right spot. For example, if you want to draw a white box at a particular location on the screen, your Bitgen circuit might have the following Verilog code:

```
parameter BLACK = 8’b000_000_00;
parameter WHITE = 8’b111_111_11;
parameter RED   = 8’b111_000_00;
parameter GREEN = 8’b000_111_00;
parameter BLUE  = 8’b000_000_00;

always@(*) // paint a white box on a red background
     if (~bright)
          rgb = BLACK; // force black if not bright
     else if (((hCount >= 100) && (hCount <= 300)) &&
              ((vCount >= 150) && (vCount <= 350)))
          rgb = WHITE; // check to see if you’re in the box
     else
          rgb = RED; // background color
```

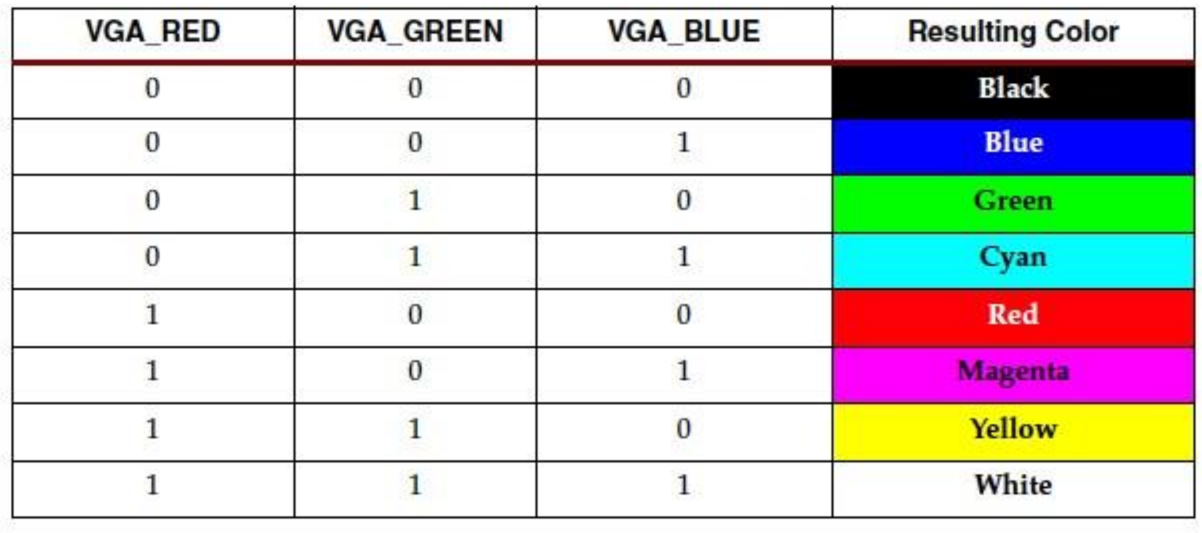
Simple VGA_Driver system

The image above is meant to show you one way of putting together the VGA circuit but not the only way for sure, and note that the Bitgen circuit was tailored to the demo using switches/buttons and drawing based on an [X,Y] location. Think about how this VGA_Driver circuit could be changed to generate memory addresses to read off pixel information from your frame buffer and feed the information to the Bitgen circuit instead of the switches.



RGB Colors

The Nexys 3 board has a VGA interface with 8 wires connected to the Xilinx part, 3 for the RED, 3 for the GREEN, and 2 for the BLUE signals. This means you can make a generous 256 colors on the screen by turning on combinations of the R, G, and B as shown in the color parameters in the code snippet above. Figure 3 shows the colors you can get with 3-bits of data.



VGA_RED	VGA_GREEN	VGA_BLUE	Resulting Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Figure 3: Eight colors from one--bit each of R, G, and B.

Now that you understand how 3-bit color with 8 different colors works so now you can expand it to 8-bits with 256 possibilities for some nicer shades. Simply give RED and GREEN three bits each and the BLUE two bits to get a wider range of colors.

VGA Connector

The standard VGA connector pin assignment can be found in the Nexys 3 board reference manual page 15. The pin assignments for using the connector are also in the User Guide but I'll repeat them here too in Figure 4. The HSYNC (hs) and VSYNC (vs) signals come from your VGA_Controller module. The RGB come from your VGA_Bitgen module.

```
NET "hs" LOC = N6 | IOSTANDARD = LVCMOS33;
NET "vs" LOC = P7 | IOSTANDARD = LVCMOS33;
NET "rgb[7]" LOC = U7 | IOSTANDARD = LVCMOS33;
NET "rgb[6]" LOC = V7 | IOSTANDARD = LVCMOS33;
NET "rgb[5]" LOC = N7 | IOSTANDARD = LVCMOS33;
NET "rgb[4]" LOC = P8 | IOSTANDARD = LVCMOS33;
NET "rgb[3]" LOC = T6 | IOSTANDARD = LVCMOS33;
NET "rgb[2]" LOC = V6 | IOSTANDARD = LVCMOS33;
NET "rgb[1]" LOC = R7 | IOSTANDARD = LVCMOS33;
NET "rgb[0]" LOC = T7 | IOSTANDARD = LVCMOS33;
```

Figure 4: UCF Constraints for the Nexys 3 board VGA connector.