DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
EECE 353 – Digital Systems Design

Lab 4: VGA Controller / Drawing Lines

Learning Objectives

1. Independently design a finite-state machine and datapath system in VHDL
2. Build complex arithmetic functions in VHDL
3. Gain more experience with complex processes and variables in VHDL

Overview

In this lab, you will be designing your own hardware to draw lines on the VGA screen. At first glance, you might think that drawing lines is really boring or you might think that it's not very useful for you to learn about. Turns out though, it is one of the most fundamental operations in graphics technology. And because it's so fundamental, line-drawing is often implemented with dedicated hardware. Anytime you look at a screen with any sort of rendered graphics, be it your desktop PC, smart phone, smart TV, Xbox, or even futuristic virtual reality technologies like the Oculus Rift or Microsoft HoloLens, you will see hardware-implemented line-drawing in action.

The top level diagram of your lab is shown below. The VGA Adapter Core is a component that is given to you. This is common practice in industrial design – taking predesigned components that are either purchased or written by another group and incorporating them into your own design.

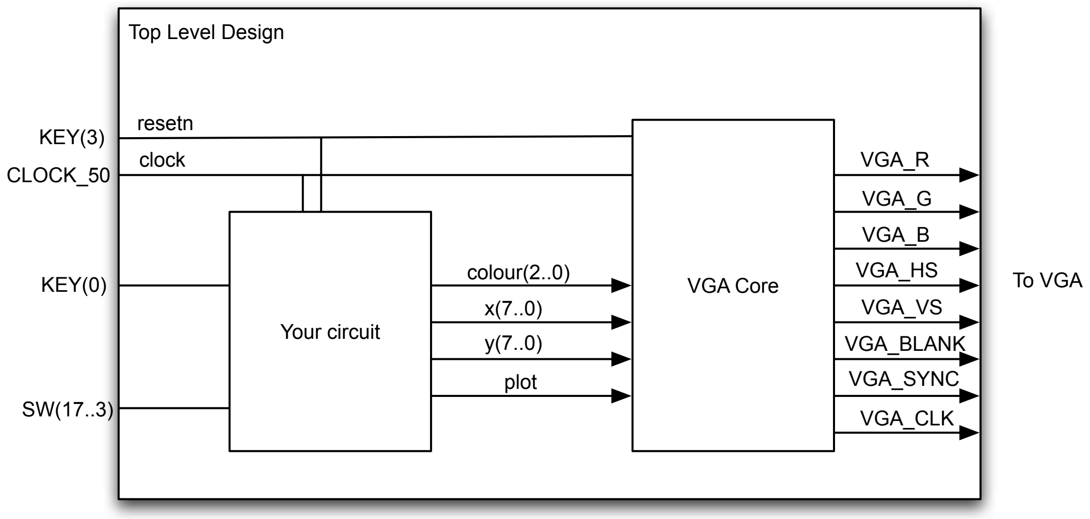


Figure 1: Overall block diagram.

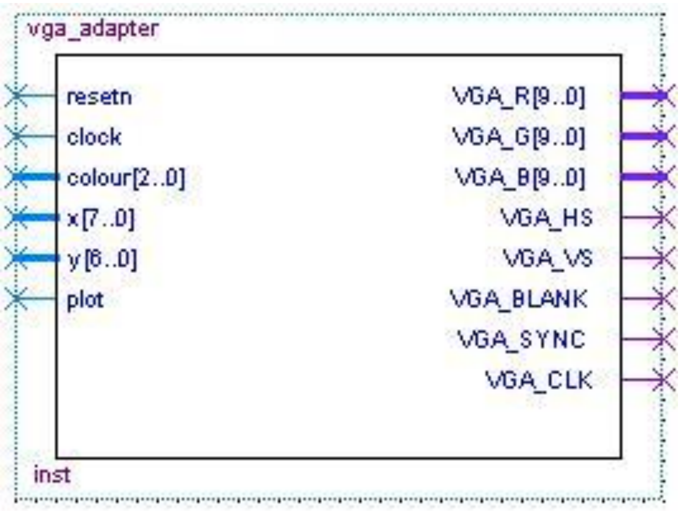
Task 1: Understand the VGA Adapter Core

The VGA Adapter core was created at the University of Toronto for a course similar to EECE 353. The following describes enough for you to use the core; more details can be found on University of Toronto's web page: http://www.eecg.utoronto.ca/~jayar/ece241_07F/vga

Some of the following figures have been taken from that website (with permission!).

In order to save on the limited memory on DE2 board, the VGA Adapter core has been setup to display a grid of 160x120 pixels, with the interface shown in Figure 2:

Inputs
From your circuit
(You will be working with these)



Outputs
To DAC & Monitor
(You do not need to control these)

Figure 2: VGA Adapter core as a black box

Inputs:

Resetn	Active low reset signal. Digital circuits with state elements should always contain a reset.
Clock	Clock signal. The VGA Adapter core must be fed with a 50MHz clock to function correctly.
colour(2 downto 0)	Pixel colour (3 bits). Sets the colour of the pixel to be drawn. The three bits indicate the presence of Red, Green and Blue components for a total of 8 colour combinations.
x(7 downto 0)	X coordinate of pixel to be drawn (8 bits) – supported values $0 \le x < 160$.
y(6 downto 0)	Y coordinate of pixel to be drawn (7 bits) – supported values $0 \le x < 120$.
Plot	Active high plot signal. Raise this signal to cause the pixel at (x,y) to be set to the specified colour on the next rising clock edge.

Outputs:

Note: You shouldn't have to worry about the outputs except that they need to be properly connected to your top-level ports (we already set this up in the starter vhdl file).

`VGA_CLK`	VGA clock signal.

`VGA_R(9 downto 0)`
`VGA_G(9 downto 0)`
`VGA_B(9 downto 0)`
	Red, Green, Blue components of display (10 bits).
	These signals are connected to the Digital-to-Analog Converter (DAC) on the DE2 board before transmitting to the monitor.

`VGA_HS` **`VGA_VS`** **`VGA_SYNC`** **`VGA_BLANK`**	VGA control signals.

Note that you will connect the outputs of the VGA Adapter core directly to appropriate output pins of the FPGA.

You can picture the VGA screen as a grid of pixels shown in Figure 3. The X/Y position (0,0) is located on the top-left corner and (159,119) pixel located at the bottom-right corner. The role of the VGA Adapter core is to continuously draw **the same thing on the screen** at the monitor's refresh rate, eg 60 Hz. To do this, it has an internal memory that stores the colour of each pixel. Your circuit will write pixel colours to the VGA Adapter core.

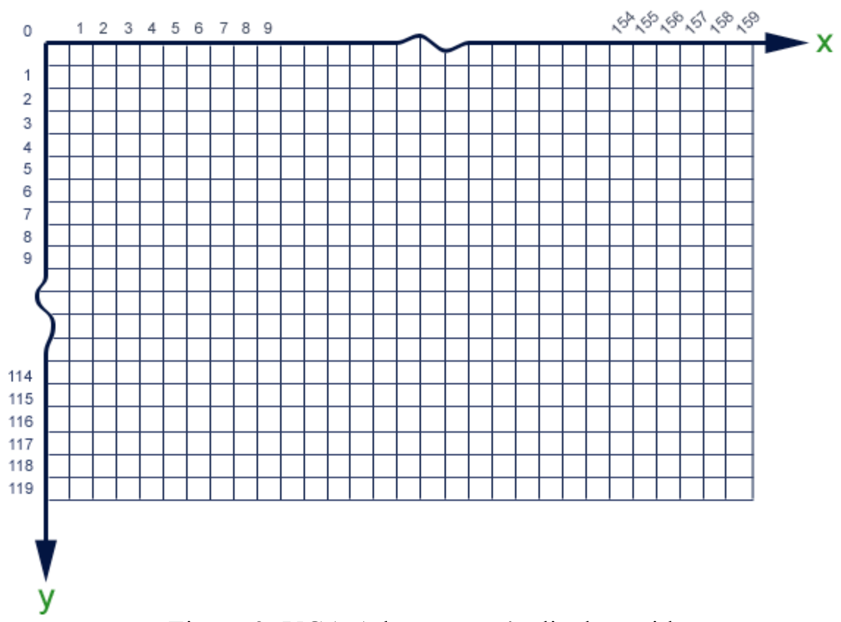


Figure 3: VGA Adapter core's display grid

To set the colour of a pixel, you first drive the VGA Adapter Core's X, Y and COLOUR inputs with the pixels' x coordinate, y coordinate, and desired color value, respectively. You then raise the PLOT input to high. You must keep these values driven until the next rising clock edge. At the next rising clock edge, the pixel colour is accepted by the VGA Adapter core's memory. Then, starting on the next screen redraw, the pixel will take on the new colour.

In the following timing diagram (from the UofT Website), two pixels are changed: one at (15, 62) and the other at (109,12). As you can see, the first pixel drawn is red and is placed at (15, 62). The second is a yellow pixel at (109, 12). It is important to note that, at most, *one pixel can be changed on each cycle*. **<u>If you want to change the colour of *m* pixels, you need *m* clock cycles.</u>**

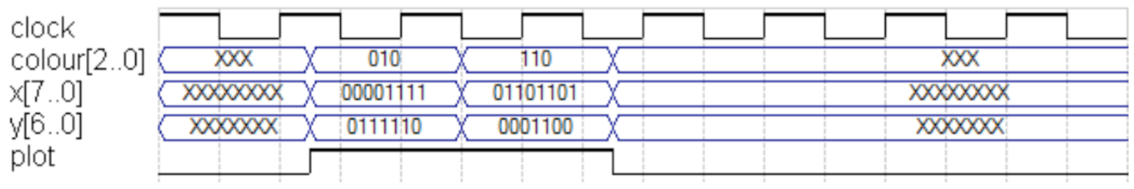


Figure 4: Timing Diagram

The source code for the VGA Adapter core consists of multiple files and is provided on the EECE 353 Connect site. This core is written in Verilog, not VHDL. That is ok: when making a structural description, you can include Verilog modules just as you can VHDL modules (this "mixed language" design approach is common in industry).

The Verilog files describing the VGA Adapter core can be included into Altera Quartus II project just like the VHDL files. If you read them, you would notice that the "module" definition in Verilog is similar to "entity" in VHDL. This means you can instantiate modules in the Verilog files as components in your VHDL files. You shouldn't be modify the VGA Adapter Core code.

To help you understand the VGA Adapter core, we have created a `vga_demo.vhd` file. This file does nothing but connect the VGA Adapter core I/O to switches so you can experiment. We suggest you download this file, understand it, and try it out. It will you understand how the inputs of the core work.

Note that this file is *only* used for you to understand the core; you will *not* use it when constructing your circuit for this lab. This task is not worth any marks, but you should do it to ensure that everything else is working (e.g. your VGA Cable is good) before starting the main task below.

Make sure you include the Adaptor Core files in your project: `vga_adapter.v`, `vga_controller.v`, `vga_address_translator.v` and `vga_pll.v.` And remember to set up your pin assignments

Task 2: Fill the Screen

You will create a new component that interfaces with the VGA Adaptor Core. It will contain a simple FSM to fill the screen with colours. This is done by writing to one pixel at a time in the VGA Adapter core. Each row will be set to a different colour (repeating every 8 rows).

Since you can only set one pixel at a time, you will need a FSM that does something like this:

```
for y = 0 to 119 {
    for x = 0 to 159 {
        set pixel (x, y) to colour ( y mod 8)
    }
}
```

Create an FSM that implements the above algorithm. A skeleton file lab4.vhd is available on the Connect site. Your design should have an asynchronous reset which will be driven by KEY(3). You don't need to use KEY(0) or any of the switches in this task. Note that your circuit will be clocked by CLOCK_50.

Test your design on the DE2. You need your DE2 board with a USB cable, a VGA cable, and a VGA-capable display. In the MCLD358 lab, a VGA cable and LCD display are provided. Use the VGA cable to connect your DE2 board to the VGA display. Most new LCD displays have multiple inputs, including DVI (digital) and VGA (analog). The LCD displays in MCLD358 use the DVI input for the PC, while the VGA input is connected to a VGA cable for you to use. You can switch between the inputs using the display's input select button. Note: the VGA connection on your laptop is an **OUTPUT**, so **<u>do not connect your laptop's VGA port to your DE2 board.</u>**

Hint: Modelsim will be very useful for debugging your component's outputs.

Task 3: Bresenham Line Algorithm

The Bresenham Line algorithm is a hardware (or software!) friendly algorithm to draw lines between arbitrary points on the screen. The basic algorithm is as follows (taken from Wikipedia):

```
function line_bresenham(x0, y0, x1, y1)
    dx := abs(x1-x0)
    dy := abs(y1-y0)
    if x0 < x1 then sx := 1 else sx := -1
    if y0 < y1 then sy := 1 else sy := -1
    error := dx-dy
    x := x0
    y := y0
    loop
        setPixel(x, y)
        if x = x1 and y = y1 exit loop

        e2 := 2*error

        if e2 > -dy then
            error:= error – dy
            x := x + sx
        end if

        if e2 < dx then
            error:= error + dx
            y := y + sy
        end if
    end loop
end function
```

The algorithm is efficient: it contains no multiplication or division (multiplication by 2 can be implemented by a shift-register that shifts right). Because of its simplicity and efficiency, the Bresenham Line Algorithm can be found in many software graphics libraries, and in graphics chips.

In this task, you will implement a circuit that behaves as follows:

1. The switch KEY(3) is an asynchronous reset. When the machine is reset, it will start clearing the screen to black. Hint: Task2 is basically clearing the screen if you set all pixels to black. Clearing the screen will take at least 160*120 cycles.

2. Once the screen is cleared, your circuit will idle. At this point, the user can set switches 17 downto 3, which indicates a point on the screen, and switches 2 downto 0, which indicates a colour. Specifically, SW(17 downto 10) will be used to encode the X coordinate of the point and SW(9 downto 3) will encode the Y coordinate of the pixel. Finally, SW(2 downto 0) will indicate one of 8 possible colours used to draw the line. **IMPORTANT: Restrict user entered coordinates to be within (0,0) to (159, 119).** If you don't you will see some unexpected behavior (strange patterns being drawn instead of a line). For example, if the user set the switches to indicate a value of 124 for X coordinate, just clip it to 119.

3. When the user presses KEY(0), the circuit will draw a line. Draw from the centre of the screen (location 80,60) to the position specified by the user. Of course, this will take multiple cycles; the number of cycles depends on the length of the line.

4. Once the line is done, the circuit will go back to step 3, allowing the user to choose another point and color. Do not clear the screen between iterations. At any time, the user can press KEY(3), the asynchronous reset, to go back to the start and clear the screen. The reset signal on the VGA Core **does not** clear the screen. That's why you need to do it manually in step 1. But this also means that you don't have to do anything special to retain previously drawn lines on the screen.

Note that you are using CLOCK_50, the 50MHz clock, to clock your circuit. You must clearly distinguish the datapath from the FSM in your VHDL code (i.e. don't write one giant process to do everything). The reason that this is a requirement is that we want you to practice manual partitioning of the datapath and FSM for now.

Suggestions and Hints

Unlike Lab 3, we are not designing the datapath or FSM for you here. But here are some hints and things to think about:

1. Think about how you will partition your FSM and Datapath before you start writing any VHDL.

2. You may consider a simplified drawing algorithm first while you work out your FSM. For example, when the user presses the "draw" button (KEY(0)), you may consider having your design draw some hardcoded pattern onto the screen. This will let you focus on the FSM and screen clearing. Once those parts seem to work, you can replace your hardcoded pattern with the line drawing algorithm.

3. Be mindful of your variable/signal vector sizes. Are they sized large enough? For example, in the Bresenham Algorithim, "e2 := 2*error". "e2" must be 1-bit larger than "error" to not overflow. Overflow can be a pain to debug unless you know to look for it.

4. Be careful when casting between UNSIGNED and SIGNED types. Remember that these types are still just vectors of bits. However, they have an **interpretation** placed upon the bits when used with arithmetic functions. Specifically, the SIGNED type interprets the bits as a two's-complement binary number (you should review twos-complement numbers if you've forgotten how they work). Casting between SIGNED and UNSIGNED does **not** perform twos-complement conversion! The bits are now simply just interpreted differently if you try to use them to do arithmetic. For example,

```
variable u : UNSIGNED(2 downto 0);        -- can represent decimal values 0 to 7
variable s : SIGNED(2 downto 0);          -- can represent decimal values -4 to 3
…
u := "111";          -- This is interpreted as decimal 7
s := SIGNED(u);  -- This is now interpreted as -1 whereas you may have thought it would be 7
```

Lab Demonstration

For this lab, **<u>demonstrate the circuit described in Task 3</u>**. You do **not** need to demonstrate Task 1 or Task 2. For this lab, also show and explain to the TA your VHDL code, and how you partitioned your FSM and datapath.

Challenge Circuit: (1 mark)

Challenge tasks are tasks that you should only perform if you have extra time, are keen, and want to show off a little bit. This challenge task is only worth 1 mark. If you don't demo the challenge task, the maximum score you can get on this lab is 9/10 (which is still an A+).

This challenge task is actually fairly easy.

A. In the original circuit, you always connect the centre of the screen (80,60) to the point specified by the user. Modify your circuit such that, instead of starting from the centre, it starts from the point specified by the user in the previous iteration. So for the first iteration, if the user specifies point (x_0, y_0), a line is drawn from the middle of the screen to (x_0, y_0). Then, in the second iteration, if the user specifies point (x_1,y_1), a line is drawn from (x_0,y_0) to (x_1,y_1). In iteration *i*, a line is drawn from point (x_{i-1},y_{i-1}) to (x_i,y_i).

B. (optional) Rather than taking the point from the switches, choose each point and colour pseudo-randomly (using a Linear Feedback Shift Register). Draw the lines quickly without waiting for KEY(0) between lines. Note that you will have to add some delay, or the screen will fill up too quickly for you to see. You only need to complete Part A for the Challenge mark.

Technical Grading

1 marks (initializing screen black after reset)
2 marks (structure of FSM and Datapath; clear separation of code)
5 marks (drawing lines works for all cases using Bresenham Algorithm)
1 marks (drawing multiple lines without reset works)
1 mark (challenge circuit)