Lab 6: Finite State Machines and VGA Controller

Purpose

In this lab, you will learn about:

- The design of sequential circuits and finite state machines in VHDL
- The use of process blocks and behavioral description in VHDL
- The VGA standard and the design of a VGA controller
- The use the Xilinx IP core generator, DCMs, and global buffers

This lab will require you to design most of the modules in VHDL. If you are not comfortable with the concepts you have covered so far, you should review the VHDL lecture slides and modules you designed in earlier labs prior to coming to class.

Background Information

If you have used a desktop computer over the past two decades, chances are that at least one of the computers you used was connected to its monitor by a VGA cable. VGA, or Video Graphics Array, refers to the display technology created by IBM in the late 1980's for their personal computing products. Through its popularity, the cable protocol would eventually become the standard for interfacing with an analog display, and is still in widespread use today, though digital connections between computer and monitor have seen increased use over the past five years.

VGA is an analog connection between computer and monitor. This means that the pixel color values are transmitted as the analog voltage, instead of using voltage to represent binary values (as we would do between logic gates). Color is broken down into three components, red, green, and blue, and each color is given its own wire. On each wire there is a 0.7 V peak-to-peak signal that varies from 0V, which denotes black, to 0.7V, which is the highest intensity for that color. Varying values of these three colors are combined to generate the complete spectrum of visible color. This method of representing color is usually referred to as an RGB color scheme.

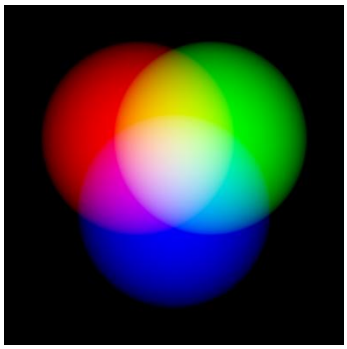


Figure 1: Example of Color Generation through RGB Combination

The RGB color value of a pixel is usually represented on a computer as three bytes: one for red, one for green, and one for blue. These values, however, are digital: in order to convert a digital value to an analog voltage, our VGA controller will use a digital-to-analog converter, or DAC. The VGA module on the Virtex II Pro (referred to as the XSGA module) has three special high-speed DACs capable of converting eight-bit digital color values to an analog voltage from 0V to 0.7V. In order to send video data over VGA to an external monitor, we will be designing a controller to interface with this XSGA module.

In order to differentiate between consecutive pixels, a VGA controller will use two signals to maintain synchronization with the monitor. These two signals are commonly referred to as ***HSYNC*** (horizontal sync) and ***VSYNC*** (vertical sync). The monitor starts drawing the image in the top left corner, which in computer graphics, is usually considered to be the origin (0,0) of the coordinate space. A pulse of the HSYNC signal instructs the monitor to move down to the next row/line. Between HSYNC pulses, the monitor uses the analog voltages for R, G, and B to determine the color of each pixel in a horizontal row. Once every row on the screen is drawn, the controller will send a VSYNC pulse, which instructs the monitor to restart at the origin (top left corner). The number of VSYNC pulses in a second is the same as the "refresh rate" of the video signal, as each corresponds to the display being redrawn.

The timings of the horizontal and vertical sync pulses is determined by the resolution of the video signal you are sending. The most common VGA mode is a 640-by-480 (horizontal-by-vertical) resolution with a refresh rate of 60 Hz, which will be abbreviated here as "640x480@60Hz". While VGA cables and monitors usually support many different resolutions, the 640-by-480 mode is usually referred to eponymously as "VGA mode" or "VGA resolution". There are eight timing parameters and a governing clock frequency specified for each display resolution. These components are shown in Table 1.

Horizontal			**Vertical**	
Name	**Duration**		**Name**	**Duration**
Video Active	640		Video Active	480
Front Porch	16		Front Porch	9
Sync Pulse	96		Sync Pulse	2
Back Porch	48		Back Porch	29
Total:	800		Total:	520

Table 1: VGA Resolution Timing Parameters

The clock frequency used for 640x480@60Hz video is 25.175 MHz. For the horizontal synchronization pulses, the duration numbers are the length of that phase in cycles of the 25.175 MHz clock. The video active phase, which corresponds to the time where the video signal is transmitting color data, lasts for 640 cycles (the horizontal resolution of the image). The front porch and back porch, which are inactive regions around the sync pulse during which there is no video, are 16 cycles before and 48 cycles after the sync pulse, respectively. Finally, the horizontal sync pulse itself is 96 cycles long, during which the HSYNC signal is active. It is important to note that for 640x480@60Hz operation, the HSYNC signal is ***active-low***, but this does vary by operation mode. An example of how the HSYNC signal is generated from these timing parameters is shown in Figure 2.

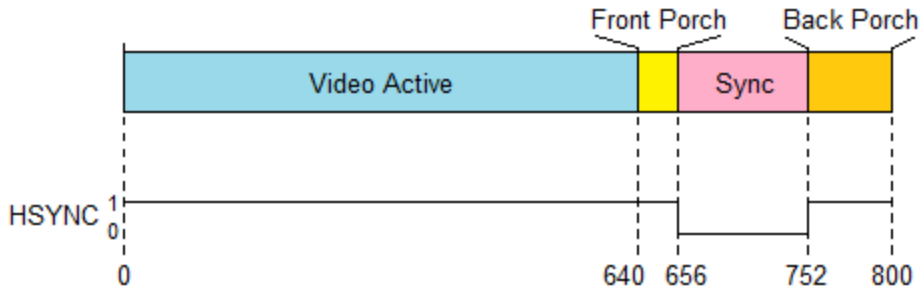


Figure 2: Horizontal Synchronization Pulse Example

The VSYNC timing parameters have similar meanings, except that instead of the durations counting clock cycles, they count the number of *complete* horizontal synchronization cycles that complete during that pulse. So, the video active phase of the VSYNC signal lasts for 480 HSYNC cycles, and the VSYNC pulse itself is active for two full HSYNC pulses. Recall that from Table 1, each HSYNC cycle is 800 cycles of the 25.175 MHz clock.

In order generate HSYNC pulses, we will be using a *modulo-800* counter, which is essentially just a counter that goes up to 799 and then resets back to 0. We can use this counter to keep track of where we are in the current cycle. If the value of this counter is between 656 (inclusive) and 752 (exclusive), the HSYNC pulse should be active (0). To track VSYNC pulses, we will use a modulo-520 counter. Every time the horizontal counter rolls over (the value passes 799), we want to increment the vertical counter by 1. To accomplish this, we will give each counter a "count enable" (CE) input. When this input is 1, the counter will increment, and if the input is 0, the counter will hold its current value.

In addition, whenever the VSYNC or HSYNC is active, the monitor will require that the RGB signals be at ground. In order to ensure that this is the case, we will be passing a one-bit output to the VGA DACs called "BLANK_Z". When this output is '1', the DACs function as normal, but when it is '0', the RGB outputs will be blanked, or set to 0V. This signal is active-low so that the VGA output will be inactive unless the user's design explicitly needs it (i.e. sets its value to '1').

An example circuit containing this functionality is shown in Figure 3.

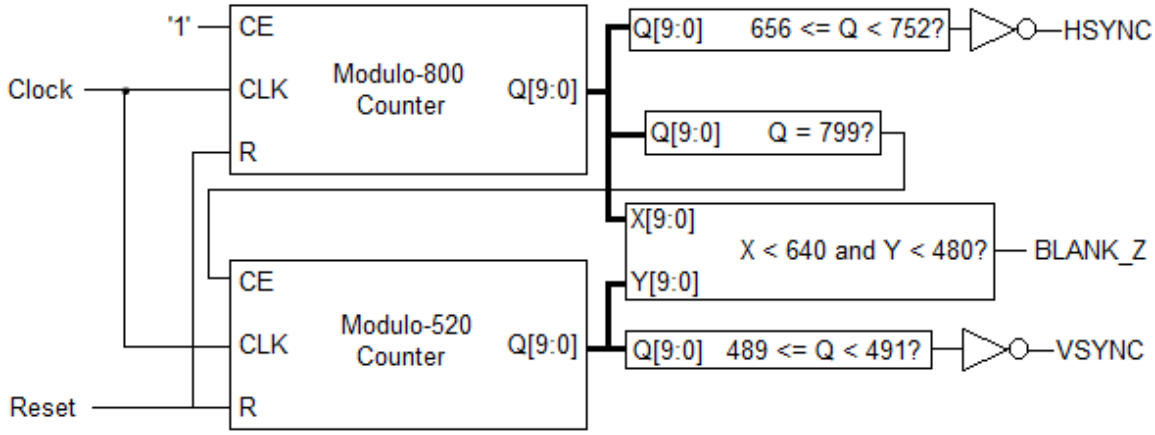


Figure 3: VGA Timing Generation Circuit

In order to generate proper timing, our VGA controller module is going to need access to a 25.175 MHz clock. Unfortunately, the only clocks available on the XC2VP30 run at 32 MHz and 100 MHz. In order to obtain a clock we can use, we will be dividing the 100 MHz system clock by four to generate a 25 MHz clock. Note that this is different than the 25.175 MHz clock specified by the VGA standard, and as a result the timing of our HSYNC and VSYNC pulses will be slightly off. However, most modern monitors (including the ones in Ketterer lab) are tolerant of slight imperfections in the timing pulses and will still display the video signal properly.

In previous labs, we have used a counter (the CB16CE) to divide clock signals. While this will work for slow clock signals like the one we use to switch the seven-segment display, this is not good practice when you need a faster clock for the entire system. In general, when a signal has a high *fan out* (it is used as an input for many other gates) this increases the load on that signal, causing it to respond more slowly to changes. If the load on a clock signal is too high, it may become *skewed*, which means that clock edges may arrive to different modules at different times. Obviously, this can have drastic consequences for a digital system.

We will do two things to prevent this. First, instead of building our own divider, we will use modules on the XC2VP30 designed for this purpose called Digital Clock Managers, or DCMs, to divide the 100 MHz system clock by four. Additionally, to prevent clock skew, we will use global buffers (BUFG) which have outputs that are specifically designed to tolerate the load of a large fan-out. Unfortunately, the techniques for interfacing with these modules can be obscure and complicated. Therefore, we will use the Xilinx IP (intellectual property) *core generator* to automate this process.

In order to verify that the VGA controller works as expected, we will be using it with a top-level schematic that generates RGB values to display. To generate RGB values, we will use a simple finite state machine that changes state every time a button is pressed. This Moore-type FSM will have a single input X, and a two-bit output S, whose value is simply the state number (so, this is basically just a four-bit counter). The state transition diagram is shown below in Figure 4.

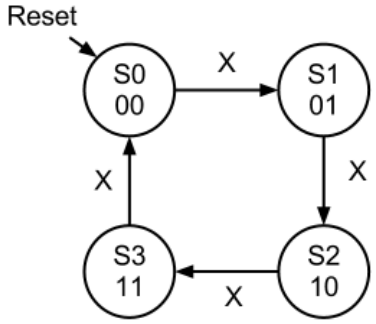


Figure 4: State Transition Diagram for Color Generation FSM

You will be given a module that generates RGB color values from the two-bit output S of this FSM. This color generation module also requires the values of the horizontal and vertical counters from the VGA controller in order to draw patterns on the screen.

Unfortunately, we cannot simply connect the X input of the FSM in Figure 4 to a push button from the DIO4 peripheral board. This is because the button on the board is being pressed by an actual person, and as a result, the button will likely be high for hundreds of milliseconds. Relative to the 25 MHz clock we will be using with this FSM, this button press will span millions of clock cycles. In order to prevent the FSM counting these (relatively) long pulses as multiple button presses, we will be designing a module to perform the conversion shown in Figure 5.

This module is called a one-pulse module. It ensures that whenever the input goes 1, the output is 1 for *exactly* one clock cycle and then becomes 0. The output will remain 0 until the input goes to 0 and then back to 1 again. You will design an FSM to implement this functionality as part of the pre-lab exercise.

Summary and Top-Level Schematic

To summarize, you will be designing and building the following modules:

1. One-pulse FSM
2. Color FSM
3. VGA Controller (timing signal generator)
4. Clock Generator (using IP Core Generator)
5. Top-level schematic

You will be provided with the following two modules to use with your code:

1. RGB Generator
2. VHDL test bench to verify the VGA controller

The top-level schematic for this design is shown in Figure 6. In order to use our design with the VGA port on the FPGA, we will be interfacing with the Virtex II Pro System's XSGA Module.

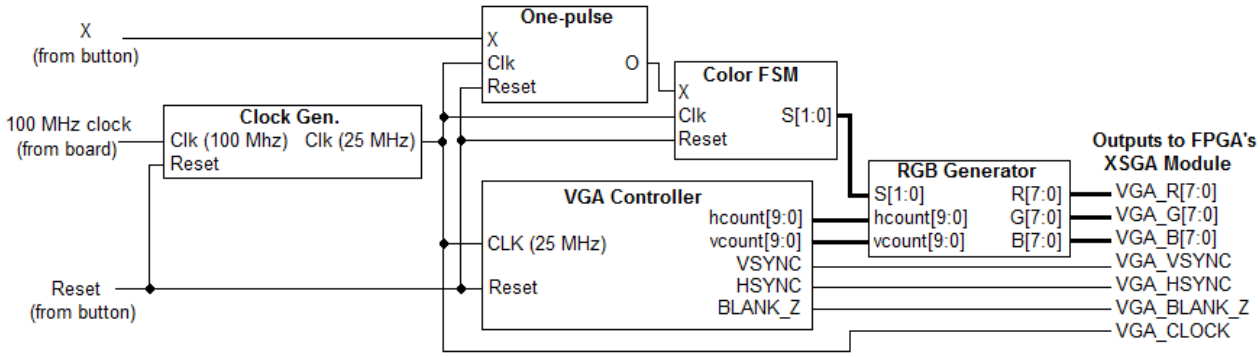


Figure 6: Top-Level Schematic

Notice that the XSGA module on the XC2VP30 requires a connection to the 25 MHz pixel clock, in addition to the RGB, sync, and blank signals.

Pre-Lab Questions

(Done individually)

*NOTE: We will be checking the prelabs **at the beginning of lab.** Any prelab not ready by the beginning of your lab section will not receive credit. All prelabs must be done individually, not in groups.*

1. Draw the state transition diagram for the one-pulse module.
2. Given the values in Table 1, calculate the length (in seconds) of the vertical and horizontal sync pulses for the VGA (640x480@60Hz) display standard, assuming the use of a:
    a. 25.175 MHz clock
    b. 25.000 MHz clock
3. Given the values in Table 1, calculate the number of frames that will be drawn in a second assuming the use of a:
    a. 25.175 MHz clock
    b. 25.000 MHz clock
4. Calculate the percent error between the answers in part a and part b for question 3. Treat the result for the 25.175 MHz clock as the accepted value.
5. Look up (i.e. Google) the values of the parameters in Table 1 for the XGA (Extended Graphics Array) display standard: 1024x768@60Hz.
6. Read the **VHDL Processes and FSM Tutorial** in preparation for the lab procedure and pre-lab quiz.

IMPORTANT: READ BEFORE YOU BEGIN

- Use **C:\user** (appears as *My Documents*) as your working directory
  - Do **NOT** use your ENIAC (S:) drive or a flash drive
  - Do **NOT** use C:\users (files **will be deleted** at logoff)
- Save regularly
- At the end of a work session, archive your project from Xilinx (Project → Archive) and save this .zip archive to YOUR ENIAC (S:) drive or a flash drive.

- This is a **one-week** lab. You must demo before your lab session next week.
- You will write a **lab report** for this lab. More information on this will follow.
- It is **very important** that you follow the instructions **carefully** in order to avoid as many issues as possible.

In-Lab Assignment

Part 1: Intro to FSM Design

First, you will design the two simple finite state machines you will be using in this lab: the one-pulse module and the color FSM.

1. Create a new project and give it the name **Lab6**. Place the project in the **C:\user\YourPennKey** folder.
2. Follow the directions in the **VHDL Processes and FSM Tutorial** to implement the color FSM module shown in Figure 4.
3. Conduct a simulation of the color FSM module. Save a screenshot of the simulation (as well as the code) for your report.
4. Implement the one-pulse FSM module that was designed as part of the pre-lab exercise.
5. Conduct a simulation of the one-pulse FSM module. Save a screenshot of the simulation (as well as the code) for your report.

Part 2: The VGA Controller

The next part of the lab involves developing the components of the VGA controller that generate the horizontal and vertical synchronization pulses. Because the VGA controller is difficult to verify via a test bench waveform, we will be providing a tester for your design.

Therefore, you do not have to take screenshots of simulations for any of the modules in this section.

Please note that while we have split the design of the VGA controller into separate modules, you are allowed to deviate from this design as you see fit.

Students who are more comfortable with VHDL may find it preferable to implement the entire controller in a single module.

6. Create a new VHDL module, and using a process, implement a modulo-800 counter. This module should have three one-bit **inputs**: *clock*, *reset*, and count-enable (*CE*), and a single ten-bit **output** *hcount* (10 bits are required to represent 799).

    a. On a positive clock edge where CE is '1', the value of count should increase by one.
    b. If count would increase to 800, roll back to 0 (when count is 799, if CE is '1', on the next cycle it should be 0).
    c. If reset is '1', the value of count should reset to 0.

    In order to make this operation simpler, you can make use of the `integer` data type in VHDL to represent the count internally.

    The `integer` datatype has a few advantages over the standard logic vector: specifically, it is easier to use with comparison and arithmetic operations.

When declaring an **integer**, you should provide a range and a default (starting) value as follows:

```
signal counter : integer range 0 to 1000 := 0;
```

The above line declares an **integer** signal named **counter** whose value will range from 0 to 1000, and whose value starts at 0.

Observe that in the declaration above, an **integer** can be assigned a value using a simple number (0, 12, 65536, etc.), instead of a binary string (like logic vectors). *You do not use quotes around the value of an integer*. See the following for more examples of interaction with integers.

```
counter <= counter + 1;

if(counter > 100) then
   counter <= counter - 100;
end if;
```

In order to assign the value of an **integer** to a standard logic vector (e.g. for use as an output from a module), you can do the following:

```
hcount <= conv_std_logic_vector(counter, 10);
```

The line above converts the **integer** value of the signal `counter` into a 10-bit standard logic vector, and assigns the result to signal `hcount`.

7. Create another VHDL module, and use a process to implement a modulo-520 counter. As with the modulo-800 counter, this should have three inputs: *clock*, *reset*, and count-enable (*CE*), and a single ten-bit output *vcount*.

8. Create a VHDL module to implement the necessary comparisons. This module should have two ten-bit inputs: *hcount* and *vcount*, and should have four one-bit outputs whose values are given by the following conditions (be careful about <= vs. <):

    a. HSYNC, false (active) when: 656 <= *hcount* < 752
    b. CE for modulo-520 counter, true when: *hcount* = 799
    c. VSYNC, false (active) when: 489 <= *vcount* < 491
    d. BLANK_Z, true when: *hcount* < 640 and *vcount* < 480

When implementing this module, you may find it useful to convert the standard logic vector inputs *back* to the convenient **integer** type. You can do that using the following function. Note that **hcount** is a logic vector input.

```
    signal hc : integer range 0 to 1000 := 0;

begin

   hc <= conv_integer(hcount);
```

You may also find the "when… else…" statement useful in this module. The syntax of this statement is as follows:

```
X <= value1 when (condition) else value2;
```

When condition evaluates to true, X is assigned value1. Otherwise, it is assigned value2. For a more concrete example, consider the following expression to generate the value of BLANK_Z:

```
hc <= conv_integer(hcount);
vc <= conv_integer(vcount);
blank_z <= '1' when (hc < 640 and vc < 480) else '0';
```

Unlike the "with-select" statement, you *can* cascade "when…else…" statements.

9. Combine all of the modules designed in this section to construct the VGA controller, following the schematic shown in Figure 3. You have two options for how to implement this module:

    a. In schematic, using the **VGA schematic template** and **VGA schematic wrapper** (*to download these and other files, click on the link, right-click, "Save Page As", and save the file to your project… do* ***not*** *copy the text into a module you create)*

    b. In VHDL, using the **VGA VHDL template**

    Whichever you choose, start with the provided template file and connect your other modules as shown in Figure 3. Be sure to add modules to your project using "Add copy of source"!

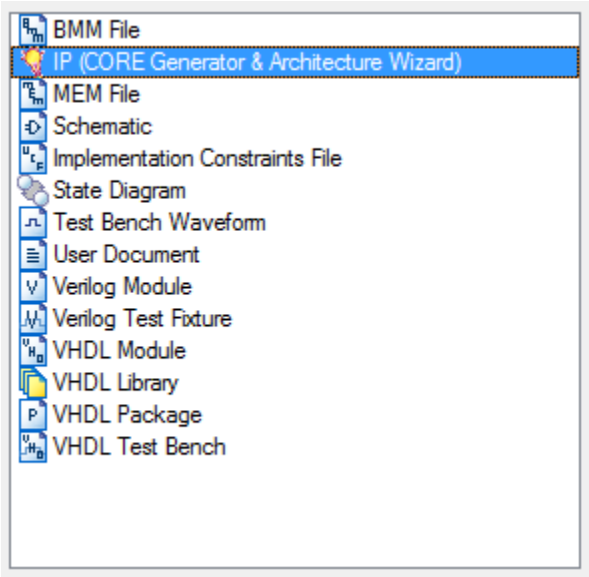
10. Download the **VGA controller VHDL test bench**, and use this to simulate your VGA controller design.

COMPLETE STEP 10 BEFORE CONSTRUCTING THE TOP-LEVEL MODULE!

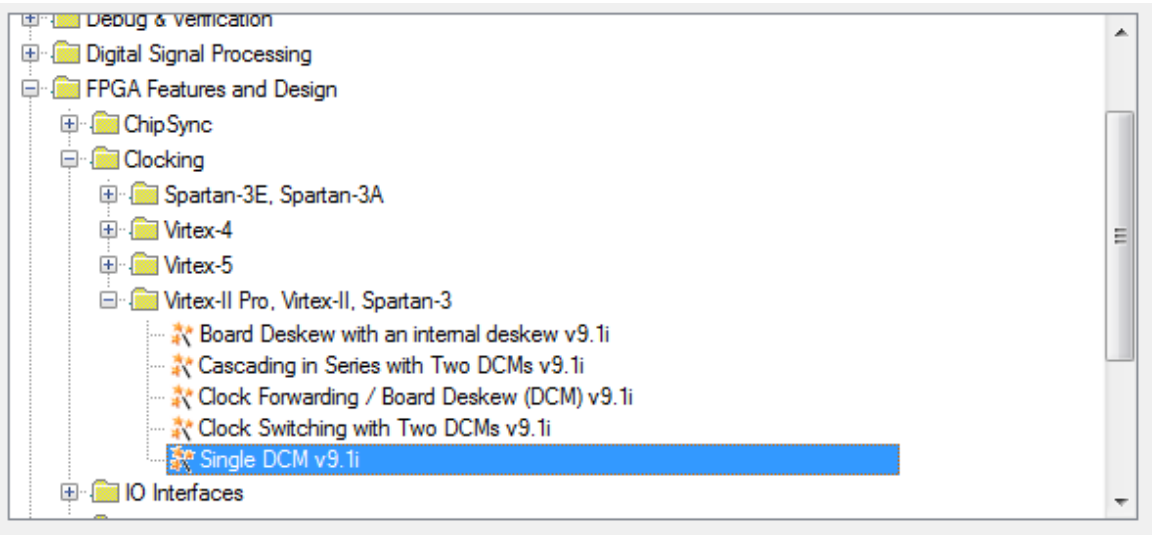
Part 3: Clock Generator Module

Next, you will use the IP core generator to generate the module to interface with the DCM and global buffers on the Virtex II Pro board.

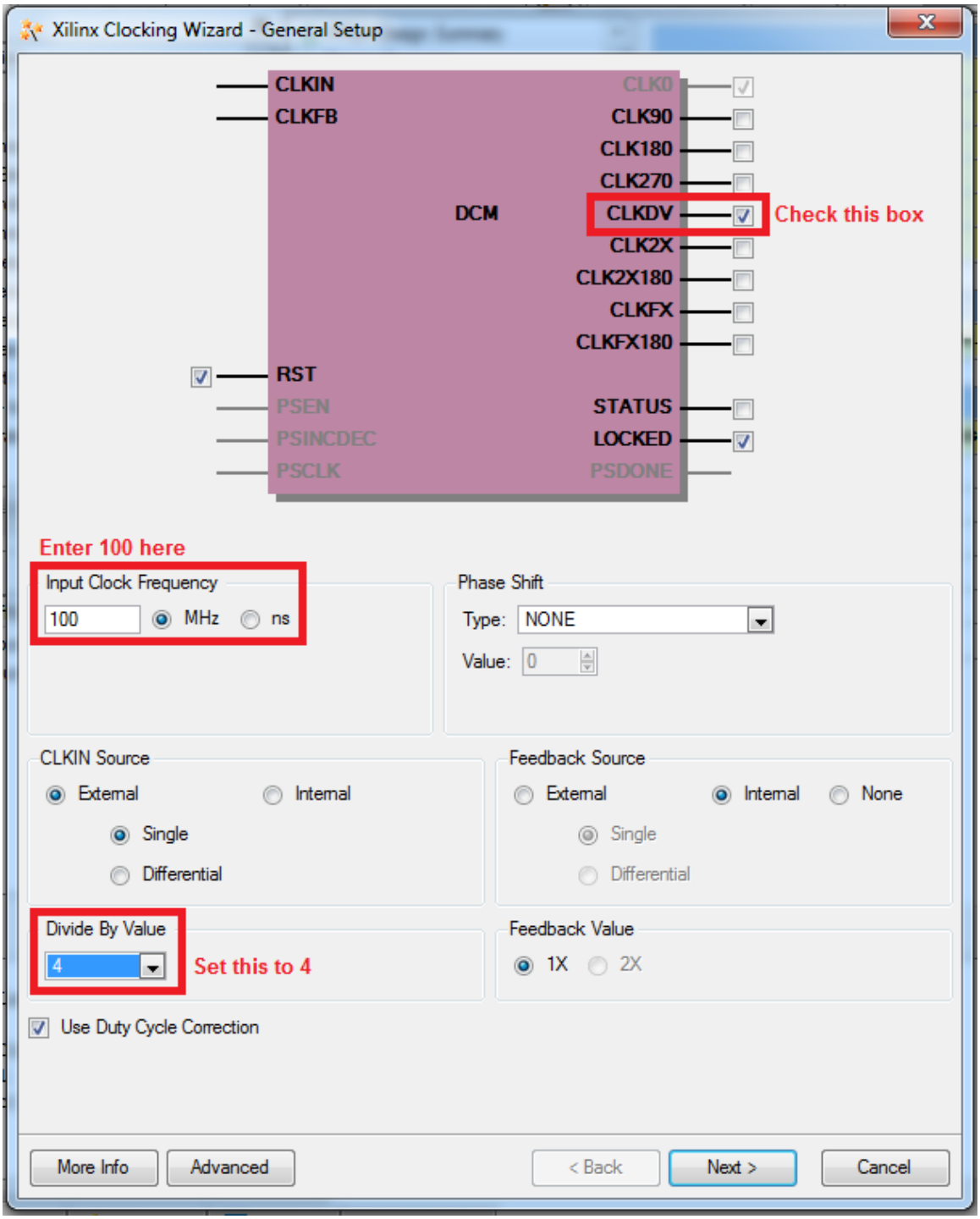
11. Create a new source, and in the New Source Wizard, select "IP (CORE Generator & Architecture Wizard)" as the source type.



12. Click "Next", and on the screen that appears, select "FPGA Features and Design -> Clocking -> Virtex-II Pro -> Single DCM v9.1i" as shown in the following screenshot. Once this is selected, click "Next" and then "Finish".



You should see the Xilinx Clocking Wizard appear, as shown in the following screenshot. Enter "100" under the Input Clock Frequency field as shown, and ensure that the "MHz" radio button is selected. Check the box next to the "CLKDV" output from the DCM module, and change the "Divide By Value" field to 4. *This instructs the setup wizard to configure the DCM to divide the input clock frequency by four*. See the following screenshot for an example. Select "Next" once you are done.



13. On the next screen, ensure that "Use Global Buffers for all selected clock outputs" is selected. Click "Next" again, and then click "Finish".
14. You should now see your newly created clock divider in the sources window. Select it, and from the processes window, choose "View HDL Source". Xilinx does not permit you to simulate Architecture Wizard/IP core modules, so (nicely) call over your TA to inspect the HDL source code and verify that it was configured correctly.
15. Create a schematic symbol for the clock generation (DCM) module.

Part 4: Top Level Module

Finally, you will construct the top level module and program

16. Download the **RGB generator module** and add it to your project using "Add copy of source".
17. Using the schematic in Figure 6 as a guide, construct your top-level module. As usual, you are not required to simulate the top-level module.
18. Create pin assignments for your design. The FPGA pins for the VGA outputs are given in Table 2-7 (XSGA Output Connections) in the Virtex II-Pro manual. For the button input, use a push button on the peripheral board. For the clock input, use the 100 MHz system clock on pin AJ15 (same as the usual system clock).
19. Generate a programming file and connect the VGA port of the FPGA board to a VGA cable from one of the monitors at your workstation. Ensure that the monitor is set to the proper (VGA/Analog) input, and configure the FPGA using iMPACT as usual.
20. Give a demo of your design to the TA. You must also run the tester to successful completion.