Video Graphics Adapter (VGA) Controller

In this lab, we implement a VGA controller in VHDL and test it on the FPGA board. This VGA controller will be used to control the monitor used for the Pong game, as shown in Figure 1.

To this end, we will provide you with a PMOD VGA interface, as shown in Figure 2a, to add a VGA port to your FPGA board. While the Pynq-Z2 board is equipped with an HDMI port, implementing an HDMI controller is outside the scope of this course. The VGA interface is simple and comprises three analog signals for the three RGB color components (R=Red, G=Green, B=Blue), a digital vertical synchronization signal `VSync`, and a digital horizontal synchronization signal `HSync` as shown in Figure 2b.

Hand-in instructions: Please note you should not hand in any report on this lab, instead, this lab will form a part of the final project.

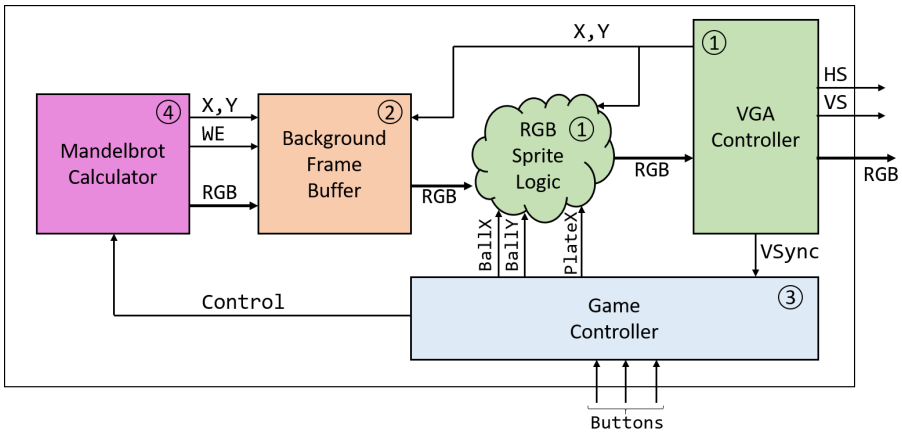
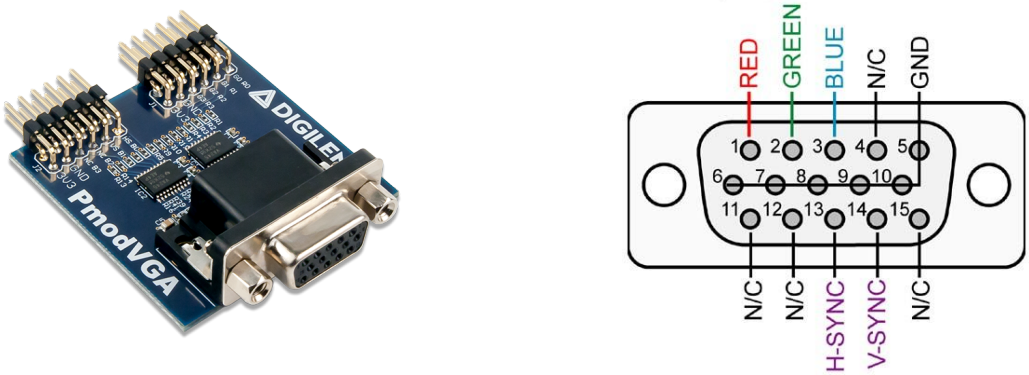


Figure 1: Full project schematic with background frame buffer receiving X- and Y-coordinates from the VGA controller to output the correct pixel values for the PONG background.



(a) Digilent PMOD VGA interface.

(b) VGA connector pins.

Figure 2: The PMOD module in Figure 2a converts the digital signals from the FPGA to analog signals for the pins shown in Figure 2b.

Important information ⚠

Please always check these things before you start a lab or when you have issues!

FPGA:

- Remember to use the reset button/switch on the FPGA to reset your design and to turn this off again if using a switch.
- Connect the FPGA board's Ethernet port to a computer, router, or any other device with an Ethernet port that can power the Ethernet chip on the FPGA board (no internet connection is needed). See the Lab 2 manual for an explanation.

VHDL:

- For combinational logic, use `process(all)`. Do not write your own sensitivity lists! Please remember to change files using `process(all)` to VHDL 2008. This is done by selecting the file in the `Source` tab. Look at the `Source File Properties` tab and change the type to `VHDL 2008` by clicking on the 3 dots in the `Type` field.
- For defining registers, use the clocked process style `process(CLKxCI, RSTxRI)`. Do not define combinational logic like `CNTxDP <= CNTxDP + AxDI` inside a clocked process! See Task 4 in Exercise 3 and its solution for further explanation.
- Never write to the same signal in multiple concurrent statements! This means that if you assign to a signal in a process that signal can only be assigned to in that process and nowhere else. The only place you are allowed to assign multiple times to a signal is inside the (single) process where it is assigned to.

Virtual machines:

- EDA server users must start Vivado with `vivado -source load_board_files.tcl` as described in Lab 1. If you do not see the board files in the Vivado GUI, you have likely used the command with a spelling error or something similar.

Windows users:

- Avoid spaces and special characters in your filepaths! Vivado projects can become corrupted if you have spaces and special characters in your filepaths.
- You may have to disable your antivirus tool before running simulations in Vivado.

For common questions/hints to this lab, please see the last page of this document which contains various hints and best-practices.

Task 1: VGA Controller Design

Download the provided `.zip` file from Moodle and use it as a starting point for your project using the directory structure from the project description PDF. This time, we provide you with the following files (under the `src` directory of the downloaded `.zip` file):

- `vga_controller.vhdl`: Template for the VGA controller.
- `vga_controller_top.vhdl`: Top-level containing the component instantiations and declarations for the clock circuit generator and VGA controller. For now, we do not consider the top-level and focus only on the VGA controller.
- `vga_controller_tb.vhdl`: Testbench for `vga_controller.vhdl`. You can run this testbench with `vga_controller.vhdl` (not `vga_controller_top.vhdl`) once you have implemented the VGA controller (Task 2).
- `dsd_prj_pkg.vhdl` Package containing constants that define the various VGA timing parameters for a screen resolution of 1024×768 and the ports of the controller. This package will be updated with more constants as the project progresses.

Furthermore, there is the `.xdc` template file and the board-files (when using the servers) as usual. **Please use the file-hierarchy described in the project description PDF!**

You should then proceed as follows:

1. Familiarize yourself with the content of `vga_controller.vhdl`.
2. Draw a block diagram of a VGA controller based on the provided background material in the lecture and using the same ports as in `vga_controller.vhdl`.
3. Implement your VGA controller in `vga_controller.vhdl`.

A controller can be implemented with the following components:

- A counter for the current pixel/column in a line. Note that with a $75\,\text{MHz}$ clock one pixel has a duration of one clock cycle exactly!
- A counter for the current line/row in the image.
- Logic that generates the horizontal synchronization signal from the pixel counter.
- Logic that generates the vertical synchronization signal from the line counter.

Please remember the following:

- The timing parameters given in `dsd_prj_pkg.vhdl` for the vertical synchronization signal are based on the number of horizontal lines and not the number of pixels (clock cycles).
- **When a horizontal or vertical blanking period starts, the RGB output should be all black**. The blanking period includes the front porch, sync pulse, and the back porch.
- You should register the synchronization and RGB outputs of your VGA controller, i.e., there must be a flip-flop for those outputs.
- For this step, you can assume that you have a $75\,\text{MHz}$ clock, so one clock cycle is equal to the duration of a pixel. This is provided directly in the testbench. The details of getting this clock on the FPGA is left for Task 3.

Hints and common errors

Note that when you generate the clocking circuit, you may get the output files in Verilog instead of VHDL. This is perfectly fine! However, if you want to get the output files in VHDL, you should set the target language to VHDL in the project settings as shown in Figure 3. Note in some cases Vivado may still generate Verilog output files.

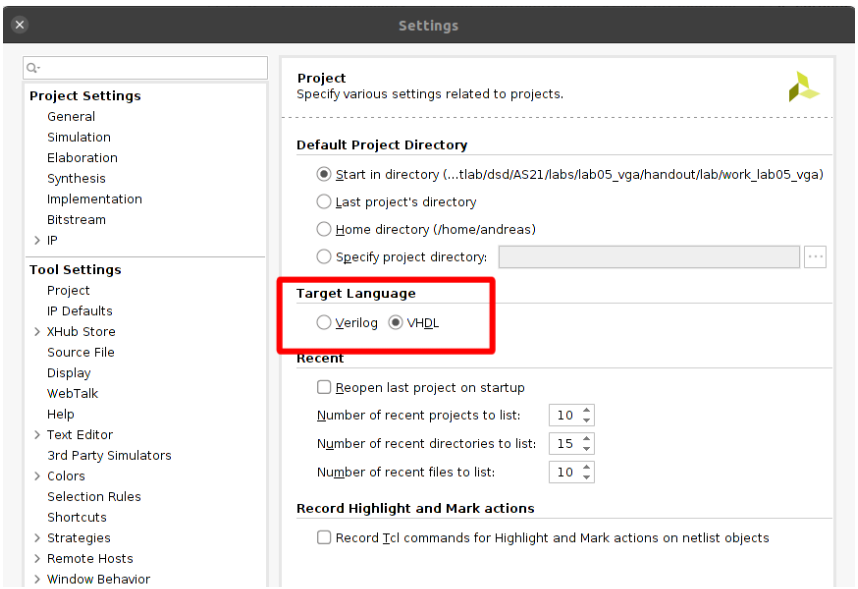


Figure 3: To get VHDL output files, select the target language as VHDL in the project settings.

Circuit designer's toolbox

In our class, we often seen the use of lengthy and repeated conditions in if-statements and other conditional blocks. This approach leads to some issues:

- **Redundancy:** Repeatedly writing the same condition is inefficient.
- **Obscurity:** Conditions often represent meaningful concepts, like start/stop signals. When these are buried within complex if statements, understanding and debugging the code becomes difficult.

A solution is to encapsulate conditions into well-named signals which clarifies the purpose of each condition. Debugging is also easier, since the conditions can be directly observed during simulation through the signals. This is beneficial in projects like the VGA and Pong game controllers, where the control logic relies on multiple, sometimes intricate, conditions.

As an example, we see in Listing 1 the same conditions repeated in multiple if-statements. To determine during simulation which if-statement will be evaluated, you have to consider combinations of values of `AxS`, `BxS`, `CxS`, and `DxS`. In Listing 2, we have instead made the conditions into meaningful signal names. Even if a condition is only used once, making a signal with a meaningful name can be beneficial just for clarity.

Listing 1: VHDL code showing how conditions repeated in if-statements.

```
process (all) is
begin
  ...
  if (AxS = '1' and BxS = '1') and not (CxS = '1' and DxS = '0') then
    ...
  end if;
  if (AxS = '1' and BxS = '1') then
    ...
  end if;
  if (CxS = '1' and DxS = '0') then
    ...
  end if;
end process;
```

Listing 2: VHDL code showing the 2 conditions mapped to signals. These signals can now directly be observed in Vivado when simulating and their meaning is clear.

```
startxS <= AxS = '1' and BxS = '1';
stopxS  <= CxS = '1' and DxS = '0';

process (all) is
begin
  ...
  if startxS = '1' and stopxS = '0' then
  ...
  end if;
  if startxS = '1' then
  ...
  end if;
  if stopxS = '1' then
  ...
  end if;
end process;
```

Task 2: Simulation

Verify the controller using the provided testbench `vga_controller_tb.vhdl`. The testbench will check the following:

- Horizontal synchronization pulse width
- Horizontal line width
- Vertical synchronization pulse width
- Vertical frame width

An example output from the testbench is shown below. A warning may be given in the beginning in the Tcl Console (next to Messages and Log at the bottom of Vivado), as shown below, since the testbench is not aligned to the synchronization signals. **This can be ignored as long as it only happens once in the beginning.**

```
Warning:
Horizontal line width is 1184 CC, expected 1328
Horizontal line width is 15787456 ps, expected (approx) 17707552 ps

Time: 15835125 ps  Iteration: 0  Process: /vga_controller_tb/p_verify_hsync
Warning:
Vertical frame width is 1031856 CC, expected 1070368
Vertical frame width is 13758767904 ps, expected (approx) 14272286912 ps
```

At the end of the simulation, a small report is printed in the Tcl Console and the number of incorrect signal widths encountered during simulation is displayed as shown below. Note that we can have one error in the beginning for both horizontal sync check and vertical sync check, but this can safely be ignored if it does not happen again. **So, as long as you only get one error for the first Vertical sync and/or frame it is fine. Same for the horizontal sync, it is okay to have one error for the first Horizontal sync and/or line. This is because the simulator needs to align itself to the first pulses.**

```
Time: 13758815573 ps  Iteration: 0  Process: /vga_controller_tb/p_verify_vsync
Note:
**********************************************************************
HORIZONTAL SYNC CHECK COMPLETE
    Got 0 errors for horizontal pulse width (1 is okay initially)
    Got 1 errors for horizontal line width (1 is okay initially)
    Got 2823 correct for horizontal pulse width
    Got 2822 correct for horizontal line width
**********************************************************************

Time: 50000000875 ps  Iteration: 0  Process: /vga_controller_tb/p_verify_hsync
Note:
**********************************************************************
VERTICAL SYNC CHECK COMPLETE
    Got 0 errors for vertical pulse width (1 is okay initially)
    Got 1 errors for vertical frame width (1 is okay initially)
    Got 3 correct for vertical pulse width
    Got 2 correct for vertical frame width
**********************************************************************
```

If the output is similar to above, and you have checked that the signal waveforms look as expected in the simulation (the testbench does not verify everything), proceed to the next task.

Task 3: FPGA Implementation

Now that you have verified your design using the testbench, you can implement the VGA controller on the FPGA. For this, you have to add the following to the Vivado project:

- `vga_controller_top.vhdl`: This is the top-level, containing your VGA controller and the clock generation circuit.
- Clock generation circuit: Use the Clocking Wizard in Vivado to generate a clocking circuit that provides a $75\,\text{MHz}$ clock from the $125\,\text{MHz}$ system clock. This means that one clock cycle is equal to the duration of a pixel. The clocking circuit is already defined as a component and instantiated in `vga_controller_top.vhdl`, however, you need to generate the files defining this clocking circuit in your project. See the moodle video at tube.switch.ch/videos/1db542d7

After adding the top-level and the clock generation circuit, you can proceed to generate the bit-file and verify your implementation with a real screen after programming the FPGA and connecting an Ethernet cable and the VGA cable as shown in Figure 4. The output should generally be all white, as seen in Figure 5.

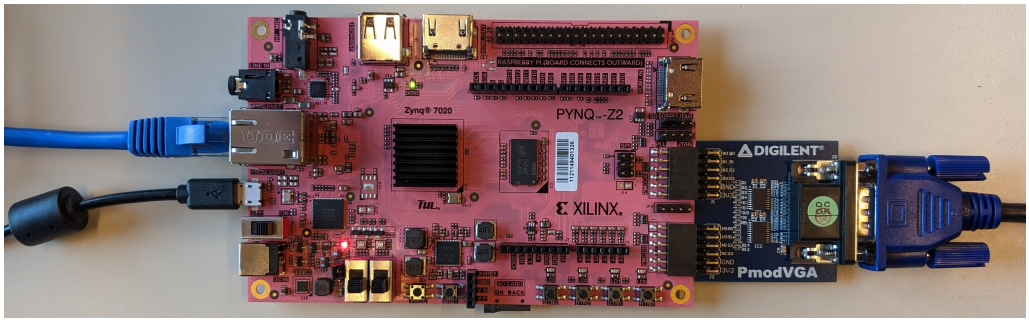


Figure 4: FPGA board connected with PMOD VGA, USB, Ethernet, and a VGA cable. For those using VGA to HDMI adapters, the setup is the same with the adapter connected to the PMOD VGA module.

IMPORTANT! Please read the following points

- With an HDMI adapter, flicker may be observed on the screen, this is okay.
- With an HDMI adapter, the output brightness may be lower than shown on Figure 5.
- When using an HDMI adapter, you also have to connect the USB cable of the adapter to your computer or another power source.
- Black borders may appear depending on your screen size, as seen in the third example in Figure 6. A small border is okay, but if the image position is excessively to the left or right this indicates that the sync signal position or the porches are not set correctly.
- An Ethernet cable has to connected to the FPGA board as the FPGA clock comes from the Ethernet chip. The Ethernet cable has to be connected to a computer or a router.
- Please fix any synthesis/implementations warnings for latches before proceeding. The Vivado simulation may look correct with latches in your code, but the implementation on the FPGA likely will not work.

Figure 5: A successful implementation of a VGA controller should have an all white video output from the FPGA.

Task 4: Color Test Patterns

With the VGA controller tested on the FPGA, we will now generate some more interesting output to the screen. Use the X- and Y-coordinates from your VGA controller to generate different color patterns as shown in Figure 6.

1. First, to make sure that you correctly get coordinates out of your VGA controller, you need to divide the screen into 4 equally sized tiles and color them as red, green, blue, and white. Note that depending on where your horizontal/line and vertical/frame counters start from (display, front porch, pulse, or back porch) you may have to subtract/add an offset to your counters to correctly output the X- and Y-coordinates of the pixel being displayed on the screen. If you cannot use your X- and Y-coordinates from your VGA controller to do divide the screen into four squares you should fix this issue as we need the correct X- and Y-coordinates for later labs.
2. Generate some more interesting patterns like the ones shown in Figure 6. It is up to you which patterns you want to generate, and they do not have to match the ones in Figure 6 exactly.

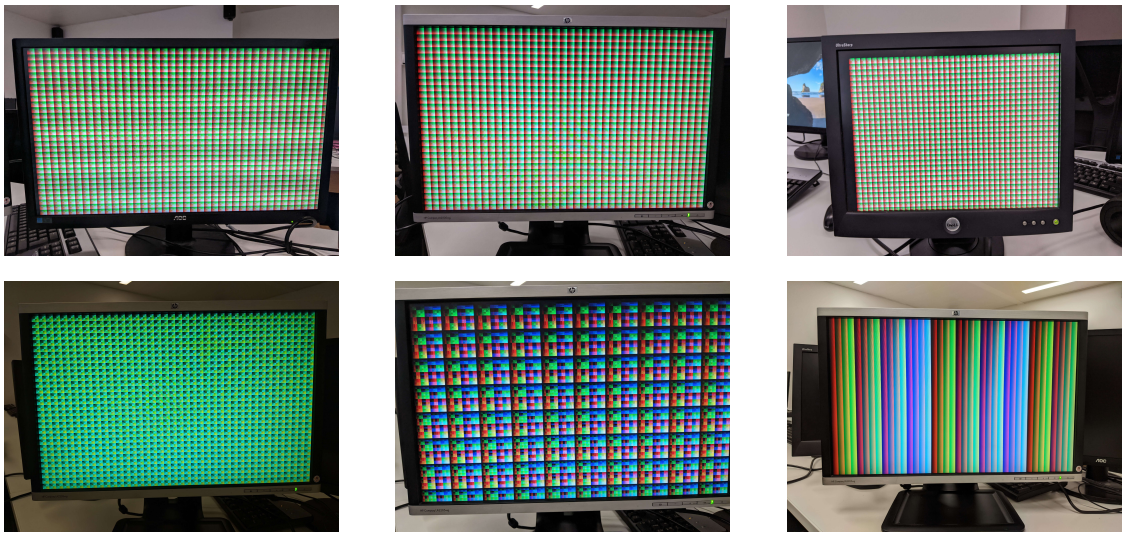


Figure 6: Various color patterns generated using the X- and Y-coordinates on different screens.

Common Questions

Common questions/remarks for this lab are:

- **How do I create a VHDL file?** After creating a project in Vivado you can click `File` → `Add Sources` → `Add or create design sources`. Alternatively, just use your normal code editor and create new files with the `.vhdl` (recommended) or `.vhd` extensions.
- **How do I resolve the warning 'The PS7 cell must be used in this Zynq design ...?** This warning can be safely ignored as it's unrelated to what we do on the FPGA.
- **How do I resolve the error 'Unconstrained Logical Port'?** While VHDL itself is case-insensitive, **`.xdc` constraints are case sensitive** and your port names should match those in the `.xdc` file in case as well.
- **Outlook does not allow `.vhd` files:** You cannot send `.vhd` files in Outlook, try renaming to `.vhdl` as the `.vhd` extension is also used for **V**irtual **H**ard **D**isk on Windows.
- **Remember that order matters in processes!** Since the order of assignments are done sequentially in a process, meaning that in the example below `DxSO` is only assigned `AxSI and BxSI` and never `AxSI or BxSI`.

  Listing 3: This implementation ignores the line `AxSI or BxSI` as it is always overwritten by the final assignment to `DxSO`.

```
process(all)
begin
  if (CxSO = '0') then
    DxSO <= AxSI or BxSI;
  end if;

  CxSO <= not AxSI;
  DxSO <= AxSI and BxSI;
end process;
```

- **Remember to separate the description of the flip-flops from the combinational logic!** Use a single process for updating the flip-flops and a separate process or concurrent assignments for updating the adder as shown below. This is really important! We also discuss this in Exercise 3.

  Listing 4: VHDL code to show how to define flip-flops for a counter.

```
CNTxDN <= CNTxDP + 1; -- Increment outside clock-process

process(CLKxCI, RSTxRI)
begin
  if (RSTxRI = '1') then
    CNTxDP <= (others => '0');
  elsif CLKxCI'event and CLKxCI = '1' then
    CNTxDP <= CNTxDN;
  end if;
end process;
```

- **Remember to never write to the same signal in multiple concurrent statements!** When assigning to a signal in a process, you can only assign to that signal in that (single) process. The code below in Listing 5 shows the signal `CNTxDN` being assigned to in two different concurrent statements, which is not allowed. With this code, you will see an 'X'

for `CNTxDN` in the waveform viewer. The solution is to put the default assignment in a process like shown in Listing 6.

Listing 5: VHDL code which shows how not to assign to a signal!

```
CNTxDN <= CNTxDP;

process(all)
begin
  if (In0xSI = '1') then
    CNTxDN <= CNTxDP + 1;
  end if;
end process;
```

- **Use default values in your process!** To avoid introducing errors in your code from missing assignments to signals, you should always use a default value for all signals assigned to in a `process(all)` when describing combinational logic as shown in Listing 6. This is done as the first thing in a process.

Listing 6: VHDL code which shows the assignment of a default value.

```
process(all)
begin
  -- Default values
  CNTxDN <= CNTxDP;
  AxD    <= (others => '0');
  BxD    <= (others => '0');

  -- Actual logic after default values
  if (In0xSI = '1') then
    CNTxDN <= CNTxDP + 1;
    AxD    <= In1xSI;
  elsif (In1xSI = '1') then
    CNTxDN <= CNTxDP - 1;
    BxD    <= In1xSI;
  end if;
end process;
```