TSIU03: Lab 3 - VGA

Petter Källström, Mario Garrido

September 7, 2015

Abstract

In this lab you will create an application that shows a picture, stored in the SRAM, onto a VGA screen. You will learn three major things: how the VGA protocol works, the concept of pipelining, and how the graphical schematic editor in Quartus II works.

For simplicity you will in the lab use a very basic resolution of 640×480 pixels, and a frame update frequency of 60 Hz.

Contents

1 Introduction

On the DE2 board the Flash memory contains an image, that is copied to the SRAM memory as soon as the board is turned on. This is done automatically, and nothing you have to do.

You will create a system that reads this image, and sends it to the VGA port together with some control signals that should be generated.

This is the first lab that uses modules to build the system. You will create those modules, and connect them in a graphical top module.

Sections 2 and 3 defines what's require from you. Sections 5 and 6 helps you design and simulate the system as required.

Appendix A describes the VGA interface.

Appendix B describes the used components on the DE2 board.

Appendix C describes the SRAM, and how the picture is stored.

Appendix D describes the pipelining that should be used in this lab.

2 Your Task

Create the system "`Lab3_VGA`". The system shall:

- Generate control signals for the VGA and the video DAC.
- Read and decode pixel data from the SRAM.
- Send the pixel data to the video DAC.
- Be pipelined.
- Print your group number on two 7-segment displays.
- Use the (`un`)`signed` data types from the package `ieee.numeric_std` where appropriate.
- Be simulated using a provided VHDL test bench (without NOKs).

There is a lab skeleton on `U:\da\TSIU03\Labs\Lab3_VGA\` – Copy this to somewhere on `H:\`. This contains an uncompleted versions of all modules. The pin placements for all buses are also done, but not for the one-bit signals.

3 Requirements to Pass

General requirements are given in the "Lab Demonstration" in the FAQ [2].

- The system must fulfill the list in section 2 ("Your Task").
- All pixels must be visible.
- You must understand the implementation.

4 Common Errors

Here are some common errors described. Many of the errors are mentioned in the section "Common Errors" in the FAQ[1], e.g.:

- **Mixing Combinational and Sequential Syntax** ⇒ Compilation errors.
- **Bad Pin Placement** ⇒ Malfunction, despite working simulation.

4.1 Syntax Error (compilation errors)

Consult any VHDL resource (the course books, "A small VHDL guide", or the internet).

4.2 Simulation Loading Error

If you get "Error loading design" when trying to simulate, the "failure" lines above will hint about why. Probably you didn't changed `std_logic_vector` to `unsigned` in the correct places in the generated VHDL file from the schematic.

4.3 Simulation Result Error

If the simulation gives a bad result (e.g., a "NOK" or other error message), you may locate the error in the design by analyzing the different signals, see more details in Sec. 6.5.

4.4 Bad Result

If the simulation works, but you don't get the correct image when running on the FPGA, here are some possible reasons.

First of all, verify on the HEX display that it is *your* system running on the FPGA.

The screen indicates no signal	
Error in pin placement	Check the pin placement, **and resynthesize** in Quartus.
Disconnected VGA cable	Solve it!

The screen gives a black image	
Error in SRAM content[1]	Restart the DE2 board and try again.

[1] If someone do something wrong, they can easily overwrite the SRAM content with zeros.

Flickering image	
Race Condition	Read Section 4.5, "Race Condition".

4.5 Race Condition

According to the ADV7123 data sheet the data input must not be changed from 0.5 ns before until 1.5 ns after the positive clock flank. Usually the data signals arrive after that. Sometimes, however, some data signals may travel faster or as fast as the clock signal to the ADV7123 chip from the FPGA, causing a so called *race*. A race can appear as flickering pixels, or cause strange colors, and it is affected by any change at all within the entire design. This problem is rare and not required to solve in this lab. It can however be solved by setting the VGA generator's DFFs to trig on the falling edge of the clock (test `falling_edge` rather then `rising_edge`). In this case the data will change half a clock cycle earlier, so the data will come in good time before the rising clock edge to the ADV7123 chip. If this solution is adopted, the pipeline correction for the sync signals must be corrected for only one missed pipeline level.

5 The FPGA Application

Within the FPGA you should build a system that handles all the signals depicted in Fig. 6 in App. B. The first thing you need is a clock divider that divides the clock from 50 MHz to 25 MHz. You also need to keep track of where, on the screen, the cathode ray is. To do so, you need two counters; one horizontal (called *pixelcounter*) and one vertical (called *linecounter*), producing the signals `hcnt` and `vcnt` respectively. You also need to keep track of when to generate the synchronization pulses and the blanking signal. All those parts should be built up by different modules (VHDL entities). The structure of your system is briefly depicted in Fig. 1.

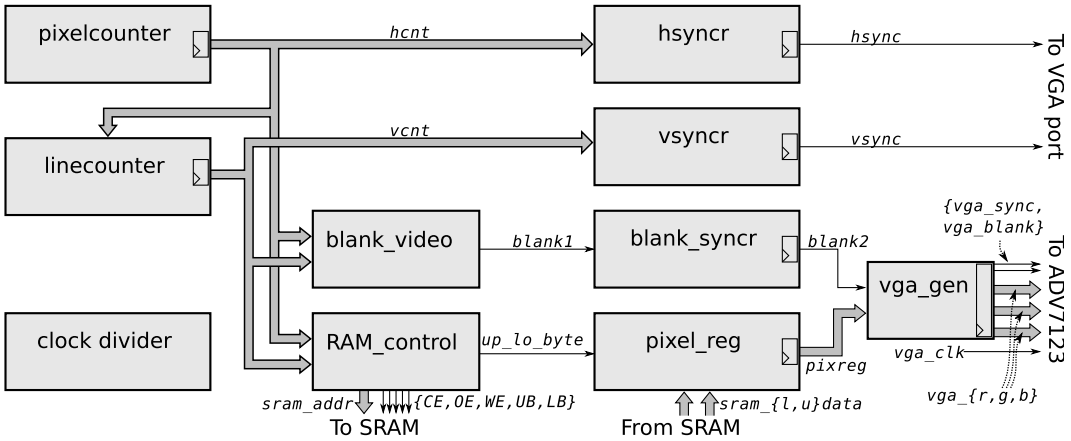


Figure 1: The entities and signal names in the VGA lab.

Hereinafter, the `hsync` and `vsync` signals are together referred to as the `{h|v}sync` signals. The same holds for the `{h|v}syncr` modules and the `{h|v}cnt` signals.

What is not shown in Fig. 1 is the clock and the reset signals. The clock divider module gets the FPGA clock (50 MHz), and produces the `vga_clk` (25 MHz) which is fed to the ADV7123 chip, and to all units having a register (together with the reset signal). The reset signal should be connected to *Key0*. Remember that *Key 0* is active low, i.e., it is `'1'` when it is not pressed, and `'0'` when it is pressed.

5.1 The VHDL Units

First a gentle reminder: there are many standards for indentation. Keep to whichever you want, but keep to a standard. If you are reckless with the indentation, you may loose an "`end if;`", and the lab assistant will probably not help you if the code is not readable. Also write comments where appropriate.

5.1.1 Clock Divider

The Clock Divider contains a DFF and an inverter. Each clock cycle (of the system clock), it inverts its output (the `vga_clk`). In this way, every second clock cycle, the output will turn from zero to one. Since

the system clock runs in 50 MHz (50M 0 ↗ 1 flanks per second), `vga_clk` will run in 25 MHz (25M 0 ↗ 1 flanks per second).

You must have an internal signal for the clock. For simulation purpose, make sure this is initiated to `'1'`. Otherwise ModelSim will set it to `'U'` (uninitiated), which will not work.

5.1.2 Pixel Counter

The `pixelcounter` unit should hold the counter `hcnt`, which must be located in DFFs. In Table 2 (in App. A) you can summarize the pixels in each line to 798 in total. If the *1st* pixel has `hcnt=0` then the *798:th* pixel must have `hcnt=797`. So, after 797 the counter must restart from 0 (on the next line). Due to the maximum value of 797 this signal must be 10 bits (`9 downto 0`).

The structure of the counter is therefore as depicted in Fig. 2.

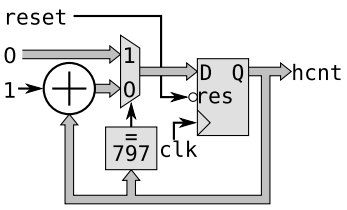


Figure 2: Pixel counter.

5.1.3 Line Counter

The line counter works just like the pixel counter, with the only difference that it should count up once per line only, typically when the horizontal sync starts. To do so you must detect this *before* it occurs (due to the pipeline stages). Therefore you must detect when the `hcnt=654`. The structure is depicted in Fig. 3.

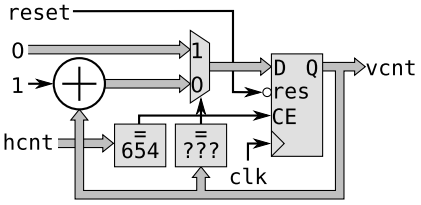


Figure 3: Line counter.

5.1.4 Blank Video

Since the blanking is active low (depicted by the inversion bars in Fig. 6 in App. B), and it is active outside the visible image (described in App. A), then this signal must be `'1'` inside, and `'0'` outside the visible image. "Inside image" is when `hcnt` $\in [0, 639]$ and `vcnt` $\in [0, 479]$.

Remember that this unit should be asynchronous. No clock is given, and no process is needed.

5.1.5 Hsync and Vsync Generator

The `{h|v}syncr` modules generates the `{h|v}sync` signals, and just as the blanking signal, they are active low. During pixels/lines *a* (see Fig. 5 in App. A) their output should be `'0'`, and `'1'` otherwise. Table 1 shows some `hcnt` values for different parts of the VGA timing, which can give a hint of how to use Table 2 from App. A. Fill in the corresponding values for the vcnt.

width	c = 640 px				d = 15 px			a = 95 px			b = 48 px		
hcnt	0	1	...	639	640	...	654	655	...	749	750	...	797
height	c =				d =			a =			b =		
vcnt													

Table 1: Some `hcnt` values for different parts of the VGA timing. Fill in the `vcnt` values if you want.

Do not forget to compensate for the two missing pipeline stages, described in section D.1 in App. D.

5.1.6 RAM Control

The RAM Controller should generate control signals and memory address to the SRAM. It will also generate a control signal, `up_low_byte`, used by `pixel_reg` to determine which input bytes it should use.

App. C describes where in the SRAM the different pixel data are stored.

The pixel indices can be generated in at least two ways. One way is a formula based on `{h|v}cnt`, which suits the given architecture in Fig. 1. This formula is quite easy to find, but more interesting to describe in VHDL.

The other way is to have a large counter, that counts the pixel index. The index should count when blank = 1, and can be reset, e.g., when `vsync` is active. Things to find out here are how this affects the pipelining, and which value the register should be reset to in order not to get a pixel skew in the picture. If

you solve the pixel index in this other way, the unit will need four inputs: `blank1`, `vsync`, `clk` and `rst`, and you can skip the `{h|v}cnt` signals. Instead of `vsync`, you can reset the counter using one bit from `vcnt`. How?

The pixel index is at most 307199, which means you need 19 bits to represent it.

The SRAM address is very easy to generate from the pixel address, as well as the signal `up_low_byte`. You can code the `up_low_byte` in any of two ways, but a suggestion is that you let `'0'` mean "lower byte", and `'1'` mean "upper byte".

At most the SRAM address is 153599, which means you will need 18 bits to access all used memory cells.

When it comes to the control signals to the SRAM, all of them are inverted (active low), and all can be constant in this lab. A few questions can be used to figure out what to set them to. They are active low, so a "yes" = `'0'` and a "no" = `'1'`. Do you want the SRAM to be enabled? Do you want the SRAM to send out values on the data bus? Do you need to write to the SRAM? Do you want to use the upper and/or the lower byte respectively?

5.1.7 Pixel Register

The pixel register has two tasks: select the proper byte from the SRAM, and store that value in a pipeline register. Those two tasks are simple to merge in a small process.

5.1.8 Blank Synchronizer

The Blank Synchronizer is just a DFF, with an asynchronous, active low, reset.

5.1.9 VGA Generator

The VGA generator has the responsibility to decode the eight bits from the RAM into an RGB coded color, as well as pipelining both the blanking signal and the resulting RGB color. It should also provide the constant

The color coding is described in App. C.

Note that the `ADC7123` chip requires 10 bits per channel, e.g. a number 0 to 1023. None of the fields *Gray*, *Red*, *Green* nor *Blue* is ten bits, and they represent much smaller numbers. The numbers must be scaled up, or you will have a very dark picture. For instance, the *Green* is three bits, where the value 7 should give max output (1023) on the green channel. This scale up can be done using a shift-and-merge operation, as illustrated in Fig. 4. In VHDL this can be expressed using the concatenation operator `&`.

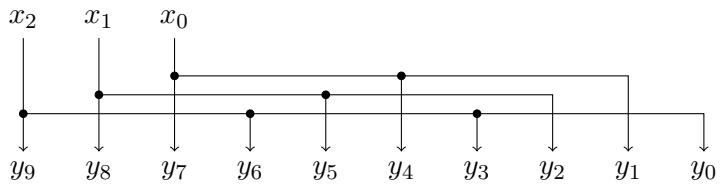


Figure 4: Example of how to scale up a three bit number (x) to ten bits (y).

5.2 Top Module

In the lab skeleton you got a top module, that is close to ready to use.

In the "Lab group" module in the top, you should set your group number in the Parameter of that module. Click around until you find out how to do that.

If you have made any changes in the entity of any module, you have to create/update it's symbol file, and then update the corresponding symbol in the schematic.

5.2.1 Warning

Quartus schematic editor have some pitfalls. Two wires can easily be short circuited if they are moved together, and they will try to follow any moving object they are connected to.

Make sure to do the mistakes in a sand box: copy some units at random in the schematics to an empty area. Draw some wires, move/cut/copy/paste the units and see what happens, try to make a good mess. Before compilation remove your mess and continue with the real design. Such a small exercise will cost you a few minutes now, but will probably save you an amount of time later.

5.2.2 Naming of Wires and Instances

For the simulation, you will generate a VHDL file from the schematic. Then it's vital for the understanding that you have relevant signal and instance names[1]. The default names are like "`inst4`" for instances, or "`SYNTHESIZE_WIRE_15`" for wires in the schematic. You do not want to analyze the waveforms with names like this.

You can **name a wire** by selecting it and start typing, then the wire will get that name. A bus can be named as, e.g., `hcnt[9..0]`, to note that this is a `hcnt : std_logic_vector(9 downto 0)` (or an `unsigned(9 downto 0)`), just as shown in the symbol ports.

If two or more different wires share the same name, they are assumed to be connected, which is handy for huge nets, like the clock and the reset signals. Select any clocked unit, draw two small wires to the clock and the reset input. Name those to `vga_clk` and `rstn`, respectively. Mark those, and hold down *ctrl* while you drag a copy of those to the next unit, to the next and next and so on. The drawback with such named nets is that it is very hard to see where the data comes from, and how the units are connected, so it should only be used for the clock and reset signals.

You can **rename an instance** by double click on the instance name at the bottom of the module symbol. You can also right click on it and select "Properties". Name the instances in a clever way, e.g. "`iclock_gen`" for the clock generator.

5.2.3 Pin Assignments

The pinout for all pins are found in the *DE2_UserManual_1.6.pdf*, mainly in chapter 4. Most of the signals are already set.

6 Simulation

You have to simulate your design, in a way that detects any kind of error you may do. The test bench is provided with the lab skeleton.

In order to do so, there are a number of things you must do:

- Generate a VHDL file for the top module schematic, see Sec. 6.1.
- Compile and simulate a given test bench and all your VHDL files.
- Correct errors and resimulate until you have no "NOKs".

[1] The module name is the name of the type. The module can be instantiated several times, with different names. Compare the class name "`string`" in C++, that you can instantiate in several variable with different names.

6.1 Convert Quartus Schematic to VHDL

Some preparations are required for the schematic file in Quartus, since Modelsim cannot read it:

- Generate a `.vhd` file from the schematic.
  - File→Create/Update→Create HDL Design...
  - Make sure the VHDL choice is selected. [Ok].
- Unfortunately, the schematic generates `std_logic_vector` of all `unsigned` signals, so you have to open the VHDL file (don't add it to the project) and change the required `std_logic_vector` to `unsigned`.
- Update `TB_top.vhd` so it matches the port declaration of your file.

6.1.1 Experimental alternative: Verilog netlist

An alternative to the VHDL unit is to generate a verilog file instead[2]. Modelsim is not so strict in the matching data type between VHDL and Verilog modules. This still needs some hand-on, but less so.

- Generate a Verilog `.v` file from the schematic.
- Open in, remove all lines containing `defparam` (should only be your group number).
- (In Modelsim, the command `vlog file.v` is used to compile verilog files).
- **You will get no help from the lab assistant on this.**

6.2 Do the Simulation

Create a Modelsim project, and include all relevant VHDL files. Compile all.

Load ("Simulate") `tb_top`. Do *not* select "without optimization" - you need all speedup you can get.

Add the waves you need. To make the waveform window faster, do not add more waves than necessary. Typically you can add all signals in the test bench and in the DUT top level. Then remove the signals you don't need, e.g. duplicates. You probably don't need the clk either (since everything is clocked on the `VGA_clk`.

Add dividers, change the signal radix, color the signals etc. Then save the waveform format — it can be good to have.

When done, run the simulation, `> run 100us` or `> run -a`.

Check for result. If you run for 100us only, then the horizontal sanity is reported. You have to manually check the colors (since the test bench check this in the second frame).

6.3 ModelSim Tricks

When you add signals to the wave, the option "All items in design" will add *all* signals in all modules. Feel free to try different ways/options, and see what happens. If you play around and get a good mess, clear it afterwards.

Suggestion: Add all signals in design. Manually add module dividers. Save the waveform.

When you compile the VHDL files from the ModelSim project, you just get a "success" or "error" message. In order to see more error information, you can double click on the error message.

To resimulate after a VHDL file change, the following one-liner is handy:

```
> vcom ../*.vhd; restart -f; run -a
```

6.4 Warnings

The first clock cycles, there are often many `'U'` or `'X'` among the signals. The arithmetic operators warns about this. All those warnings before 1 µs can be ignored.

[2] Verilog is a competitive hardware description language

6.5 Errors

You will probably not get all "OK" at once, so you have to debug your code, using the waveform as a tool.

Use the simulation to debug your system.
- **Zoom** to track individual signals.
- Use **Cursors** to measure timings.
- **Track** your signals. Why is a signal assigned a value? Compare with the VHDL code.
- Add **debug prints** to your code. See the `report`'s in the test bench. Warning: What will happened in the transcript window if you report something every clock cycle for 33 ms?
- The VHDL command "`integer'image(int)`" converts an integer to a string. This can be useful in reports.

The following procedure can help you debug (for each "NOK" you get):

1. Why was a NOK reported? E.g., the horizontal timing requires that all four periods (a to d) are correct. Measure which of the periods that failed.
2. At least one of the signals causing the error are wrong. Which one? How should the signal behave?
3. Find the assignment of the signal, and track down why it behaves as it does.
4. Correct the problem.

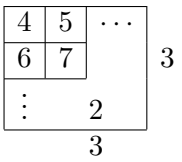
The following errors are likely to occur:

- Error in `{h|v}cnt`: Can cause really bad timing and bad colors.
- Error in SRAM address generation: Can cause bad colors.
- Error in pipelining: Can cause bad timing (just a pixel or line away) or bad pixels (correct pixels values, but on the wrong places in the screen).
- Error in color decoding: Can cause bad colors.

6.6 The Test Bench

This section will explain how the `TB_top.vhd` works, which you need to understand in order to debug your code. Feel free to check/modify the test bench after your need.

There are two files given for the test bench (in the folder `MSim`). The `TB_top.vhd` contains the test bench. The `TB_SRAM.vhd` contains a very simple model of the SRAM. The memory content is 4, 5, 6 and 7 for the top left most four pixels, and 2 for all other pixels, as illustrated below. Addresses outside the image gives 3.



Those bytes should result in grayscale colors at the output.

The video DAC (`ADV7123`) is "simulated" by a pipeline stage.

The SRAM are simulated by instantiating the SRAM model, and connect it to the memory interface of the DUT.

6.6.1 Sanity Check: Horizontal Timing

The horizontal timing process looks for:

1. Rising edge of the blanking.
2. Falling edge of the blanking.
3. Falling edge of the hsync.
4. Rising edge of the hsync.
5. Rising edge of the next blanking.

The process takes the difference in time between those occurrences.

"`OK`" is reported in the transcript if the timing are correct. Otherwise "`NOK`" is reported.

6.6.2 Sanity Check: Vertical Timing

There is a line counter, `line_cnt`, that increases with each hsync signal.

The vertical timing process looks on a vertical line in the middle of the picture. The vertical line is simply a copy of the hsync signal, delayed 10 µs.

1. First line with blank = 1 after a vsync pulse.
2. First line after that with blank = 0.
3. First line after that with vsync = 0.
4. First line after that with vsync = 1.
5. First line after that with blank = 1.

The process takes the difference in `line_cnt` between those occurrences.

"`OK`" or "`NOK`" is reported.

6.6.3 Sanity Check: Colors

The color verifier process looks for:

1. First rising edge of the blank after a vsync pulse. Measure color 4.
2. Wait one VGA clock pulse. Measure color 5.
3. Wait one VGA clock pulse. Measure color 2.
4. Wait until next rising edge of the blank. Measure color 6.
5. Wait one VGA clock pulse. Measure color 7.
6. Then for each VGA clock pulse:
   - Never a value other than values 2, 4, 5, 6, 7 when `blank = '1'`

"`OK`" or "`NOK`" is reported.

6.6.4 Simulation Time

There are some "`done`" signal, that terminates the clock when all tests are done.

The Vertical timing will run two frames. With 60 frames per second, this means about 33 *ms* of simulation. This will take really long time to simulate, and be really slow to show in the waveform.

When you have loaded the design, added signals to waves etc, you can "run all" `> run -a`.

One idea can be to run the simulation for just a few lines (say, 100 µs), which should be enough for the horizontal timing to report, and for you to manually verify the colors in the top left pixels.

7 The Result

The resulting picture is available in the file `HaveYouPassed.png`. When verifying your design, look extra carefully in the corners of the image. It contains some patterns in the corner. Each corner's pattern occurs twice, both in the outermost white border, and in the light gray border.

The same picture is shown in the default application (when you restart the DE2 board), but with mirrored text "You" (to distinguish it from *your* implementation). If the corners are not visible, when you are using the default application, then auto adjust the monitor.

References

[1] TSIU03: FAQ, section *Common Errors*.

[2] TSIU03: FAQ, section *Lab Demonstration*.

Appendix A VGA Interface

The VGA interface has five main signals: three analog color channels (`R`, `G`, `B`), and two synchronization signals (`hsync`, `vsync`).

In cathode ray tube (CRT) monitors, the cathode ray is controlled both horizontally and vertically by two controller units that sweep the ray over the screen. The horizontal controller sweeps fast from left to right (typically 10-100 thousand times per second). The vertically sweeps slow from top to bottom (typically 60 times per second). The color of the ray is controlled in real time from the three color channels. The horizontal synchronization signal (`hsync`) tells the horizontal sweep unit to reset to left, and corresponding with the vertically synchronization signal (`vsync`). All in all this gives a behavior where the picture is plotted line by line.

If the cathode ray is turned on (showing something else than black) while the sweep units are resetting to left or top respectively, it will plot an unwanted pattern on the screen. In order to avoid this, it is important that the signal is *blanked* just before, during and after the synchronization signals.

Figure 5 shows the timing model for the VGA interface, especially for the resolution 640×480 @ 60 Hz (which is the default VGA resolution). Originally the model "starts" with horizontal and vertical synchronization (setting the lines a to the left and in the top). However, putting the visible image in the leftmost top corner makes it easier to calculate the pixel index, so this model is used in the lab.

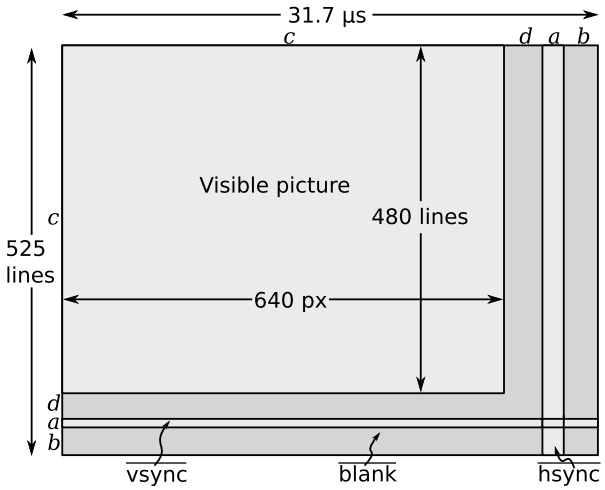


Figure 5: Timing model for the VGA resolution.

Table 2 shows the detailed timings for the resolution used in this lab. As can be seen, the 640 pixels (for each line) should be sent during 25.4 µs, which gives 39.69 ns per pixel. This period can however be stretched to 40 ns (corresponds to 25 MHz) without problem.

	c	d	a	b	**Units**
Horizontally[1]	25.4	0.6	3.8	1.9	µs
Vertically	480	10	2	33	lines
Horizontally[2]	640	15	95	48	pixels

[1] Definition
[2] Adjusted to 25 MHz pixel clock

Table 2: VGA signal timing for 640×480 @ 60 Hz.

The c part is the visible area. The blanking is active (low) during the horizontal or vertical d, a, and b parts. The `hsync` and `vsync` pulses are active (low) during the a part of horizontal and vertically axes respectively.

From Table 2 you can read out that, for instance, the `hsync` signal should be active during 95 pixels of each line, and the `vsync` signal should be active during 2 entire lines.

Appendix B DE2 Overview

On the DE2 FPGA board there are a number of components. Most important is the FPGA, but in this lab, you shall also use two other components; the SRAM and the video DAC. The SRAM contains the image you are going to plot on the screen, and the video DAC is used to produce the three analog color channels which are sent to the VGA monitor. Those components and the signals between them are depicted in Fig. 6.

Worth to note is the block of registers in the video DAC (part number `ADV7123`). This register level will be discussed in Section D, *Pipelining*.

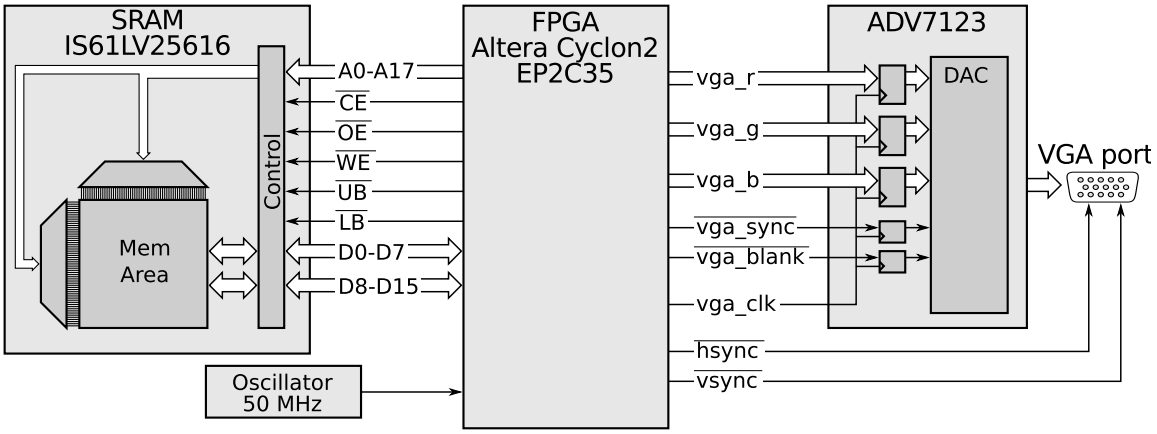


Figure 6: Illustration of some components on the DE2 board, and how they are connected.

There are a some control signals to the SRAM, which must be correctly set. You will never write data to the SRAM (that is solved by the default program), so those are very easily handled; all control signals are constant `1` or constant `0`. The SRAM has support for both 8 and 16 bits operations, thou the two data buses. Either if you read or write to/from the SRAM, you can select if you mean to do so to/from one of the bytes or to/from both. By simplicity reasons, you will read from both, all the time (more about that later). A manual for the `IS61LV25616` chip is found in `U:\da\TSIU03\Docs\Datasheets\Memory`.

The main feature of the video DAC circuit (`ADV7123`) are to convert the three digital channels to analog channels. It also supports some other functions, like composite sync. You will not use this, so you should assign `vga_sync=0`, as recommended in the ADV7123 data sheet. The `vga_blank` signal should however be used to control the blanking. A manual for the `ADV7123` chip is found in `U:\da\TSIU03\Docs\Datasheets\VGA DAC`.

Appendix C The SRAM and the Image

An important part in this lab is how the VGA image is stored in the SRAM. The image is stored using 8 bits (one byte) per pixel.

C.1 Addressing

Figure 7 illustrates a way of index the pixels on the screen[3]. The gray box corresponds to the screen and the small squares corresponds to the pixels. The numbers in the pixel squares are the pixel indices. The variables `hcnt` and `vcnt` are corresponding the x and y variables, as described in section **??**, *The Implementation*.

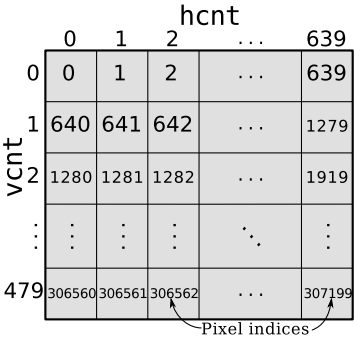


Figure 7: Pixel indices on the screen.

The SRAM does however return *two* bytes for each address. The conclusion is that each address corresponds to two pixels on the screen. As depicted in Tables 3 (a) and (b), the first memory cell in the SRAM (which has address 0) contains pixel index 0 and 1. The next cell (address 1) contains pixels 2 and 3, and so on. Pixels 638 and 639 (the two rightmost in the up most row) are placed on address 319. Address 320 contains the two first pixels on the next row, and so on. The upper byte (`D15-D8`) contains the odd pixel indices, and the lower byte (`D7-D0`) contains the even pixel indices.

The issue with the two bytes from the SRAM must be solved in the FPGA, which is a part of your task. How this is solved is discussed in section **??**, *The Implementation*.

C.2 Color Coding

Each byte in the SRAM defines the color of one pixel on the screen. There are two color codes, defined by the MSB in the byte. Table 4 describes this.

[3]This system is just a local notation within this lab manual only.

Pixel	Address	Byte
0	0	lower
1	0	upper
2	1	lower
3	1	upper
4	2	lower
...	...	...
307198	153599	lower
307199	153599	upper

(a) From a pixel index perspective.

Address	D7–D0	D15–D8
0	pixel 0	pixel 1
1	pixel 2	pixel 3
2	pixel 4	pixel 5
3	pixel 6	pixel 7
4	pixel 8	pixel 9
...	...	...
153598	307196	307197
153599	307198	307199

(b) From an SRAM perspective.

Table 3: Pixel index in the SRAM.

Bit index	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Grayscale mode	0	$Gray_{6..0}$						
RGB mode	1	$Red_{1..0}$		$Green_{2..0}$			$Blue_{1..0}$	

Table 4: Color coding in the SRAM.

If the MSB is `'0'`, then the remaining bits represent a grayscale. The $gray \in [0,\ 127]$, where 0 is black and 127 is white. This information should be equally copied to all three color channels, in order to get a grayscale image.

If the MSB is `'1'`, then the remaining bits are divided into three bits. The $Red \in [0,\ 3]$, where 0 is no red and 3 is max on the red channel. The same holds for $Green \in [0,\ 7]$ and $Blue \in [0,\ 3]$.

Appendix D Pipelining

You *can* build the application as depicted in Fig. 1, and use only D-type flip flops (DFFs, or *registers*) for the line and pixel counters. The problem is that it takes a certain time for the signal from the DFFs in the counter units to calculate the address, travel out from the chip to the SRAM, find the correct memory cell, return that value to the FPGA, handle it there and finally send it to the `ADV7123` chip, and its registers. This will probably take more than one clock cycle, and then you have problems because you cannot know when the signals reach the `ADV7123` chip, which will cause an unpredictable system.

The solution is to introduce pipelining, a technique where you insert pipeline register sets in the design, letting the different parts act in different "time domains" (or more accurately *pipeline stages*). This causes the output to be some clock pulses later, but in this case it does not matter if the picture is shown a little later (less than one µs).

Divide your application into the four pipeline stages 0, 1, 2 and 3, as depicted in Fig. 8. It is important to point out that the clock divider defines the clock, and does *not* belong in any pipeline stage. Note the series of registers that are put into all units crossing the pipeline borders, which means that their output is in the "next" pipeline stage.

By inserting register levels this way, the longest signal path between two registers is reduced. The time it takes from one register to another is now less than the clock cycle time (or you have to insert another pipeline stage where necessary).

The concept of pipelining are depicted in Table 5. Take for instance the fifth clock cycle (clock cycle 4) in the table, which can be described as follows:

- In **stage 0**, the counters calculate location for pixel 4, but send out pixel 3 to the next stage.
- In **stage 1**, the synch signals, blanking signal and SRAM address for pixel 3 are calculated. Pixel 3 is read from the SRAM, and pixel 2 is sent out to the next stage.
- In **stage 2**, the module `vga_gen` manages pixel 2, and sends out pixel 1 from the FPGA chip.
- In **stage 3**, the `ADV7123` chip reads pixel 1, and feed pixel 0 to the VGA connector.

With this schedule pipeline stage N will always be N clock cycles "later" than pipeline stage 0, causing

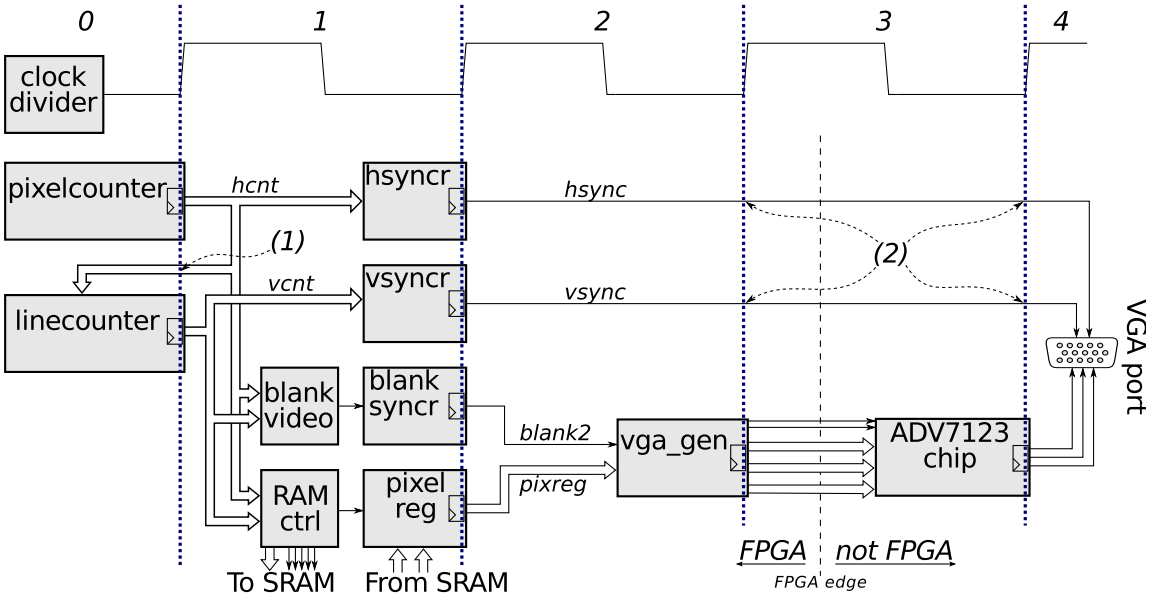


Figure 8: The pipelining of the system.

Clock cycle	Stage 0	Stage 1	Stage 2	Stage 3
0	Handles pixel 0	-	-	-
1	Handles pixel 1	Handles pixel 0	-	-
2	Handles pixel 2	Handles pixel 1	Handles pixel 0	-
3	Handles pixel 3	Handles pixel 2	Handles pixel 1	Handles pixel 0
4	Handles pixel 4	Handles pixel 3	Handles pixel 2	Handles pixel 1
...	...	...	...	...

Table 5: The pipeline stages and the pipelining.

it to be in another "time domain".

D.1 Problems with Pipelining

The alert reader will notice two problems with the solution depicted in Fig. 8. They are pointed out with **(1)** and **(2)** respectively (in the figure), and they are discussed below:

The problem in **(1)** is that the line counter (which is in pipeline stage 0) reads values from pipeline stage 1. This means that all values on its input port are one clock cycle later than they should be. The correct solution to this should be to add -1 DFF on the input, which is of cause not possible. Instead the solution is to compensate for this. Normally, the line counter shall increase when the pixel counter is zero. But with this compensation, linecounter must instead look for the value pixelcounter had before zero, which is 796 (described in App. A).

The problem in **(2)** is that the `{h|v}sync` signals are not at all delayed the last two stages, so they will reach the VGA monitor two pixels (80 µs) earlier than the color information. The solution should be to put two DFFs on each signal, delaying them two clock pulses. The easiest solution is however to compensate the equations in `{h|v}syncr` modules. Since you want the `{h|v}sync` signals to be two clock cycles *later* than they should be in pipeline stage 1, you should *increase* the compared values with 2 in the `hsyncr` unit (the `hcnt` signal increases its value with one each clock pulse). The `vsync` cannot be handled like this, but it does not matter exactly on which pixel it is updated, as long as it is on the correct line, so `vsyncr` can be left unchanged.