TSIU03: Lab 3 - VGA

Petter Källström, Mario Garrido

September 8, 2021

Abstract

In the initialization of the DE2-115 (after you restart it), an image is copied into the SRAM memory. What you have to do in this lab is to read the image from the SRAM to the FPGA and then send it to the screen through the VGA port.

Contents

1 System Overview

Fig. 1 shows the overview of the system. It includes the SRAM, the FPGA and the digital to analogue converter (DAC), which provides the analogue video signal for the VGA.

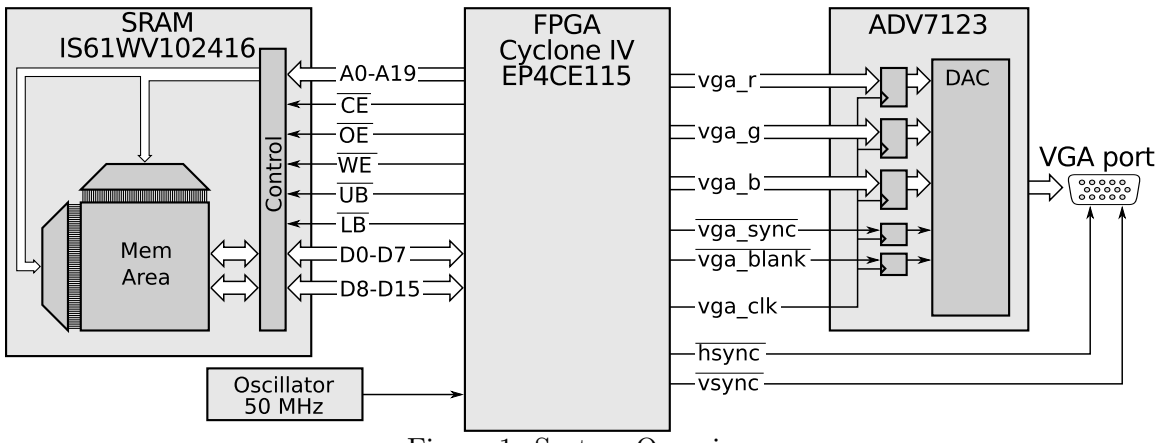


Figure 1: System Overview.

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

1.1 About VGA and ADV7123

The VGA is explained in Section 4.10 of the user manual of the DE2-115 board that you can find in `K:\TSIU03\DE2_115_Resources\DE2_115_User_Manual.pdf`

The screen that we have connected to the VGA port works with a resolution of 640 × 480 pixels, and a frame update frequency of 60 Hz, i.e., it displays 60 frames per second. Therefore, consider only this configuration among those provided in the manual of the DE2-115.

The VGA port has two synchronization signals `hsync` and `vsync`, and three analogue colour signals (`R`, `G`, and `B`). The colour signals come from a digital to analogue converter (DAC) named ADV7123. The image on the screen is drawn from left to right, line by line. `hsync` is the horizontal synchronization, which must be done after each line. The `vsync` is the vertical synchronization, which must be done after the entire image has been displayed in order to return to the beginning of the image. `hsync` and `vsync` are active low.

The digital versions of the colour information are sent to the ADV7123 from the FPGA, together with `vga_clk` and `vga_blank`. The `vga_clk` must run in 25 MHz. We also provide the `vga_sync` = '0' (what this signal does can be read in the ADV7123 manual).

Figure 2 shows the timing model for the VGA (640x480 @ 60 Hz). Note that it includes the visible image plus a part for blanking and synchronization. The `vga_blank` signal (or `blank`) must be active (='0') outside the visible picture (regions d, a and b). This indicates that no colour is sent. The synchronization signals must be active (='0') in the corresponding regions a, as shown in the figure.

Table 1 shows the number of lines and pixels that the different regions take. From Table 1 you can read 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.

Figure 3 shows the timing relation between the `blank` and the `hsync` signals around one line of visible image. The surrounding blanking and synchronization timing intervals are also included. Note that the two blanking and hsyncing periods "belongs" to two different lines.

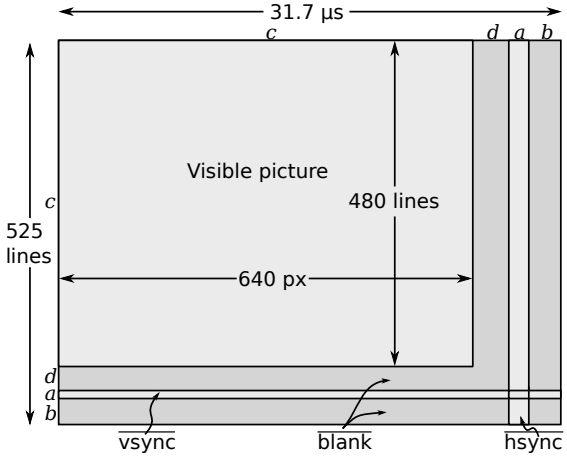


Figure 2: Timing model for the VGA (640x480).

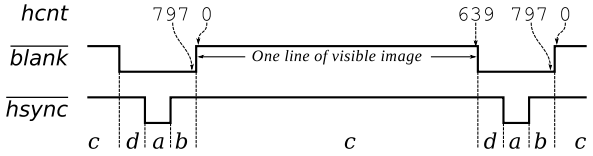


Figure 3: Relation between the `hsync` and `blank` signals.

Fig. 4 show a shorten version of an entire frame. The `vsync` is expected to start/stop when the `hsync` starts, which makes Fig. 2 actually slightly wrong. The rightmost a and b regions should be one pixel up (why?).

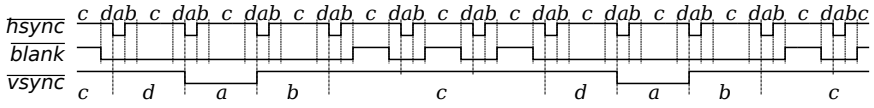


Figure 4: An entire frame, but with vertical $c = 3$, $d = 1$, $a = 1$, $b = 1$ for simplicity.

1.2 About SRAM interface

You can read about the SRAM in the manual of the DE2-115, that you can find in: `K:\TSIU03\DE2_115_Resources\DE2_115_User_Manual.pdf`. The SRAM is introduced in Section 4.19 and the pins for the SRAM are in pages 67 and 68.

The explanation of the control signals of the SRAM and other technical details are provided in the SRAMs datasheet: `K:\TSIU03\DE2_115_Resources\DE2_115_Datasheets\Memory\SRAM`. The control signals are explained in pages 1 and 3 of the datasheet and the levels of those signals are in page 4. In this lab, we only read from the SRAM (we do not write), and we read the 16 bits of the data (upper and lower part) simultaneously. Based on this, select the values of the control signals according to the datasheet.

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

Table 2: Pixels stored in the SRAM.

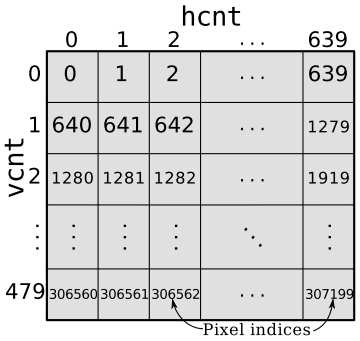


Figure 5: Pixel indices on the screen.

1.3 About the Image in the SRAM

Table 2 shows how we have stored the image in the SRAM memory. Each address of the memory stores two pixels and each pixel is represented with 8 bits. The corresponding pixel on the screen is shown in Figure 5. Therefore, pixel 1 corresponds to the second pixel of the first line of the image and is stored in upper bits (D15-D8) of the first address of the SRAM.

The 8 bits of each pixel hold the colour information, as shown in Table 3. There are two colour modes, defined by the MSB x_7. If $x_7 = 0$, then the other bits represent a grey scale: $grey \in [0,\ 127]$, where 0 is black and 127 is white. This information must be sent to all three colour channels in order to get a grey scaled image. If $x_7 = 1$, then the remaining bits are divided into three parts. $Red \in [0,\ 3]$, where 0 is no red and 3 is max on the red channel. The same holds for $Green \in [0,\ 7]$ and $Blue \in [0,\ 3]$.

Bits	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
Grayscale mode	0	$Grey_{6..0}$						
RGB mode	1	$Red_{1..0}$		$Green_{2..0}$			$Blue_{1..0}$	

Table 3: Colour coding in the SRAM.

2 Your Task

Your task is to implement the FPGA application, that continuously show the image stored in the SRAM on the VGA screen.

In order to help you, we have created a Quartus project for this lab, to start from. It includes a graphical top module (finished), all required sub modules (mostly empty), and the input and output signals. The pin placement is already done. This is described in section 3.

▶ Copy the entire folder `K:\TSIU03\Labs\Lab3_VGA` to e.g. `X:\TSIU03\Lab3_VGA`, and open the project.

Then change the group number, and complete all uncompleted modules, with the help given in section 3. As a result, you will

- Generate address and control signals to the SRAM
- Read and decode pixel data from the SRAM
- Generate the control signals for the VGA and the video DAC

- Send the pixel data to the video DAC
- Make all this work in a pipelined implementation.

When done, you must simulate the design using a predefined test bench. This is explained in section 4. Finally, you have to verify on the FPGA hardware (section 5).

3 The FPGA Application

When you open the provided project you will see that the system looks like Fig. 6.

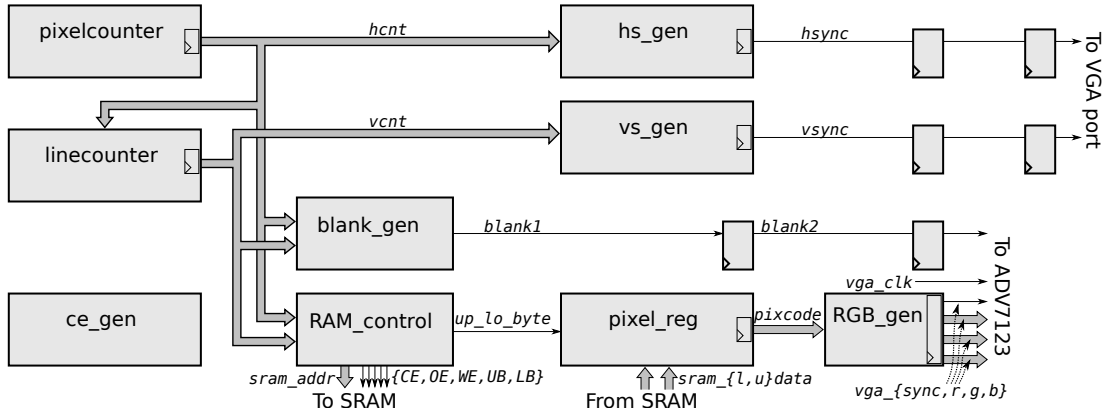


Figure 6: The entities and signal names in the VGA lab.

Apart from the lab group number, you should not do any change at all in the block diagram.

The following sections describe the functionality of the modules of the system.

All modules have a completed entity, and most have empty architectures. Do not change the entity. If you do, you also have to create a new symbol for module (remember from Lab1), update the symbol in the block diagram, and regenerate a VHDL representation of the block diagram (used for simulation, and explained in the Simulation section).

3.1 Group Number

In the "Lab group" module in the top of the block diagram, set your group number in the Parameter of that module. Click around until you find out how to do that (don't edit the VHDL file).

3.2 Clock Enable Generator (CE_gen)

The clock that we use in the DE2-115 has a frequency of 50 MHz. However, the VGA pixel clock is 25 MHz (25 million pixels per second). Hence, we must handle one pixel every other clock cycle. In order to do so, we need a module that sends out a "clock enable" (CE) signal every other clock pulse. The rest of the system must take this signal into account. It is especially important for the counters.

The implementation of `ce_gen` is a very simple state mashin.

3.3 Pixel Counter

Fig. 7 shows the `pixelcounter`. This module generates a counter `hcnt` that counts the pixels of each line. To get one pixel every second clock cycle, it must count only when the CE is '1'. According to Table 1 we have $640 + 15 + 95 + 48 = 798$ pixels horizontally. Therefore, the counter must count 798 columns (from 0 to 797). After reaching 797, the counter restarts from 0 (on the next line). As the counter counts from 0 to 797, how many bits does it need?

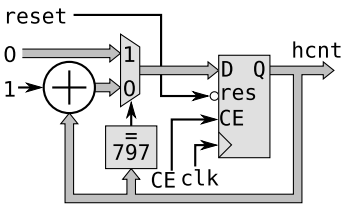


Figure 7: Pixel counter.

3.4 Line Counter

The line counter counts rows on the screen. It works just like the pixel counter, with the only difference that it should increase once per line, not every pixel. The screen expect the vsync to update around the start of the hsync pulse, so the counter should be updated then as well. Fig. 8 shows this.What happens if you forget the CE input signal?

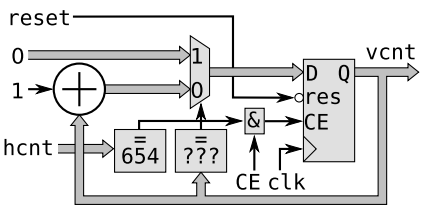


Figure 8: Line counter.

3.5 Blank Generator

The module `blank_gen` generates the `blank` signal. Remember that, according to Figure 2, the `blank` signal must be '0' outside the visible area of the screen. Note also that this signal is generated using combinational logic and no sequential logic is needed.

3.6 Hsync and Vsync Generator

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

The `{h|v}s_gen` modules generates the `{h|v}sync` signals. Use the description in Section 1.1 to identify the value that they must have depending on the value of the counters.

Those are registered, so we may for instance implement them to *turn* on or off at specific moments, rather than when to *be* on or off.

3.7 RAM Control

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

The values of `hcnt` and `vcnt` for the different pixels in the screen are shown in Figure 5. Note that the index of the pixel on the screen can be calculated from the values of the counters. Then, the index of the pixel is related to the address where it is stored, as shown in Table 2. Based on this information, in this module you have to generate the address of the SRAM from `hcnt` and `vcnt`. Note that the SRAM address only needs to change every second pixel, because we retrieve two pixels every time that we read. As we read both upper and lower pixels in the same address simultaneously, the `up_low_byte` signal is sent to the `pixel_reg` module to indicate which of the pixels needs to be selected at each clock cycle. For the preparation of the lab, calculate the relation between the counters and the SRAM address, and design the circuit that you are going to use for the RAM Control. Decide the number of bits that you are going to use for the signals.

3.8 Pixel Register

The `pixel_reg` module has two tasks: Select the proper byte from the SRAM, and store that value in a pipeline register. These two tasks are simple to merge in a small process.

3.9 RGB Generator

The `rgb_gen` has the responsibility to decode the eight bits from the SRAM into an RGB coded colour, as well as pipelining the resulting RGB colour.

The colour coding is described in Section 1.3. Note that the fields *Grey*, *Red*, *Green* and *Blue* have 7, 2, 3 and 2 bits respectively. However, the `ADC7123` chip requires 8 bits per channel, i.e., a number from 0 to 255. To generate the 8 bit signals, repeat the bits in the colour code and merge them together. Figure 9 shows how to obtain the 8 bits for green ($G_{7..0}$) from the 3 bits in the colour code ($x_{4..2}$).

3.10 Pipelining

If the processing of a signal takes too long time, it might not have time to finish during one clock cycle. A solution to this is to split the computation into two or more stages, with registers in between. This is called

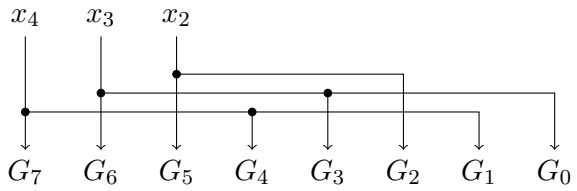


Figure 9: Example of how to scale up a three bit number (x) to eight bits (G).

"pipelining", and the registers are called pipeline registers. Now each stage is shorter, so it is easier to finish the computation in time. After the first stage calculates its first pixel (in our case), it starts to calculate the second pixel. In the same time, the second stage starts to calculate the first pixel.

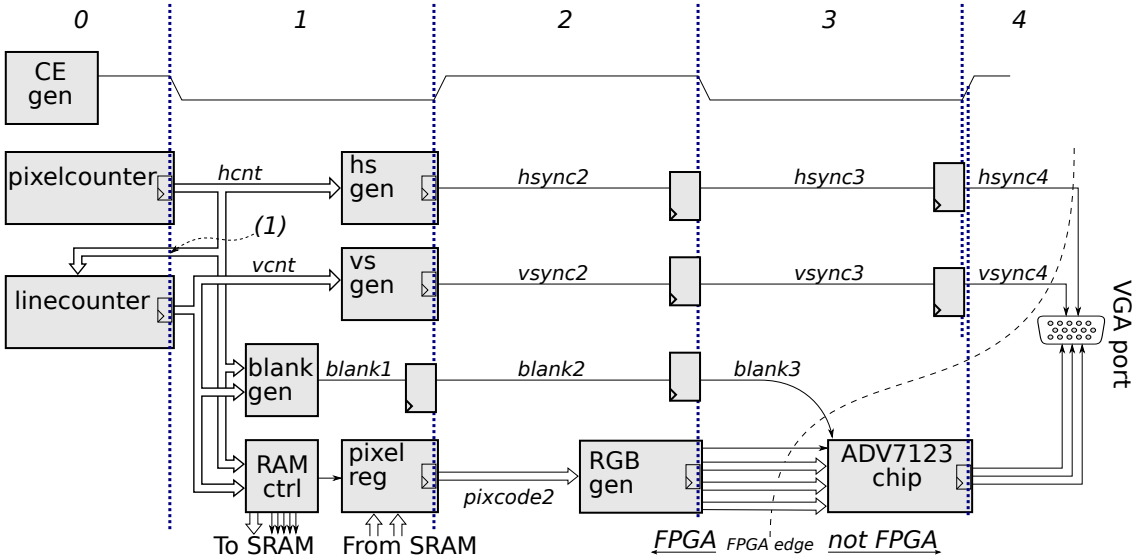


Figure 10: The pipelining of the system, as a combined block diagram and timing diagram.

Figure 10 illustrate how our system is divided into four stages (one clock cycle per stage), meaning the output to the VGA port is four clock cycles later than our counters. Note that RGB_gen outputs when CE *is* high (that is, outputs just around the falling flank of CE). This guarantees that it's output is stable around the CE rising flank, used to clock the ADV7123 chip.

In general, each stage must only read registers from the stage before, or combinational signals from the same stage. An exceptions, marked with "(1) in the figure, are discussed in section 3.10.1.

Clock cycle	`CE`	**Stage 0** (counters)	**Stage 1** (RAM, syncr)	**Stage 2** (`rgb_gen`)	**Stage 3** (ADV7123)
0	0	Count pixel 0	-	-	-
1	1	Count pixel 0	Look up pixel 0	-	-
2	0	Count pixel 1	Look up pixel 0	Decode pixel 0	-
3	1	Count pixel 1	Look up pixel 1	Decode pixel 0	Convert pixel 0
4	0	Count pixel 2	Look up pixel 1	Decode pixel 1	Convert pixel 0
5	1	Count pixel 2	Look up pixel 2	Decode pixel 1	Convert pixel 1
...	...	...	...	...	...

Table 4: The pipeline stages and the pipelining. Stage 0 are only updated after `CE` = 1. Stage 3 are clocked on `CE`.

The ADV7123 has its own pipeline stage with registers, and we must take this into account. Those registers must, however, be clocked with the pixel clock - 25 MHz. From Table 4, we see that the output from the ADV7123 stage changes when `CE` changes from 0 to 1. So we can feed them with the `CE` as a new

clock. You must always be careful when mixing two clock domains. In this case, the input data to the ADV7123 are stable around the positive flanks of the new clock, so there should be no problem.

With this schedule pipeline stage N will always be N clock cycles "later" than pipeline stage 0, causing it to be in another "time domain".

3.10.1 Problems with Pipelining

As mentioned before, there is an exception/problem in Fig. 10, pointed out with **(1)**.

The problem 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 would be to add -1 DFF on the input, which is of cause not possible. Instead the solution is to compensate for this. The line counter shall increase when `hsync` starts, i.e. when `hcnt`$= 640 + 15 = 655$ (see Tab. 1). But with this compensation, linecounter must instead look for the value pixelcounter had one pixel before, which is 654.

4 Simulation

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

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

- Compile and simulate a given test bench and all your VHDL files.
- Correct errors and resimulate until you have no "NOKs".

4.1 Convert Quartus Schematic to VHDL

Your system includes a block diagram, which Modelsim does not understand.

From Lab 1 we remember that we can create a VHDL representation of the block diagram. This does, however, use `std_logic_vector` for all multi bit signals. Hence, each time we generate a new or updated VHDL file, we have to change those to unsigned in the VHDL file.

To save you some time, we have created this for you.

4.2 Do the Simulation

▶ Open the Modelsim project `MSim/Lab3_TB.mpf` (File → Open, select proper file type).
▶ Compile all (run `vlib work`, if Modelsim complains about missing library).
▶ Load ("Simulate") `tb_top`. Do *not* select "without optimization" — you need some speedup.
▶ 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 (especially the clock is slow to show).
▶ Add dividers, change the signal radix, colour the signals etc. Then save the waveform format — it can be good to have.
▶ Run the simulation, `> run 100us` or `> run -a`.

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

4.2.1 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.

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
```

Then you just need to focus the transcript window, and press Up arrow + Enter.

4.2.2 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.

4.2.3 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 colours.
- Error in SRAM address generation: Can cause bad colours.
- 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 colour decoding: Can cause bad colours.

4.3 The Test Bench

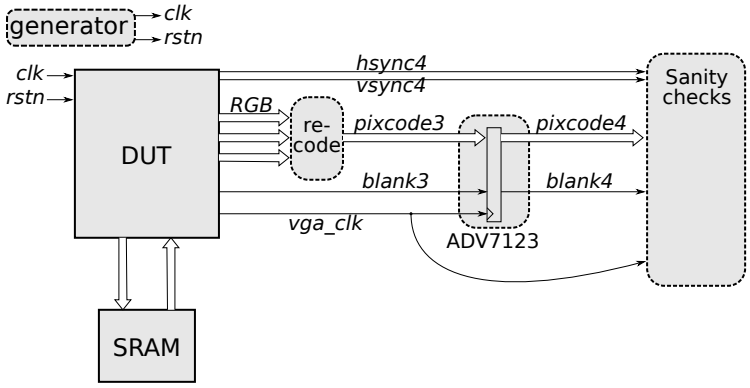


Figure 11: The test bench design.

The provided `TB_top.vhd` is written to test your design for timing errors (that it follows the VGA specification), pipeline errors, colour handling errors etc. This makes it rather complex.

This section explains how the test bench 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 `TB_top` first creates the 50 MHz `clk` and the `rstn` signals. Then it instantiates the design under test (DUT), which is your VGA controller. It also instantiates the SRAM model, and connects it to the memory interface of the DUT. The RGB values are encoded back into the colour code, `pixcode3`. The video

DAC (`ADV7123`) is "simulated" by pipelining the pixcode and blanking to `pixcode4` and `blank4`. Finally, there are plenty of sanity check sections, that verifies the behaviour of the DUT. This is depicted in Fig. 11.

4.3.1 Sanity Check: Constants and Clock

The process `sanity_const_p` verifies that some of the constants sent out from the FPGA have the right values. The process `sanity_clk_p` verifies the timing of the VGA clock, so a clock cycle is actually 40 ns.

4.3.2 Sanity Check: Horizontal Timing

The horizontal timing process looks for:

1. Rising edge of the blanking (`blank4`). The actual time is saved as variable "`t0`" of type *time* (VHDL have such a data type).
2. Falling edge of the blanking. This is "`t1`". `t1-t0` is part c (visible image), should be 25.4 µs.
3. Falling edge of the hsync. This is "`t2`". `t2-t1` is part d.
4. Rising edge of the hsync. This is "`t3`". `t3-t2` is part a (sync pulse).
5. Rising edge of the next blanking. This is "`t4`". `t4-t3` is part b.

The process computes those `t1-t0` etc, and compares with the adjusted definition (see table 1). Any pipeline discrepancy between the blanking and the hsync signal is detected here.

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

Note: `t0` to `t4` is of type `time`, which is an actual data type in VHDL.

4.3.3 Sanity Check: Vertical Timing

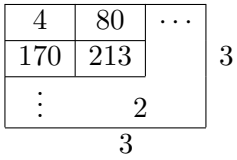
It is rather complex to check the vertical timing. First we need a line counter, `line_cnt`, that increases with each hsync pulse. Then we need to detect when the blanking is inactive *within* each line. For this, we let `hsync_del` be the hsync signal, delayed to be active inside the image (to catch the blanking). Now we can look for blank and vsync along `hsync_del`, and record the `line_cnt` as y0, ..., y4.

1. Wait for vsync pulse.
2. Next `hsync_del` pulse with blank = 1. This is "`y0`", and indicates the start of the second frame.
3. Next `hsync_del` pulse with blank = 0. This is "`y1`".
4. Next `hsync_del` pulse with vsync = 0. This is "`y2`".
5. Next `hsync_del` pulse with vsync = 1. This is "`y3`".
6. Next `hsync_del` pulse with blank = 1. This is "`y4`".

E.g. `y1` − `y0` is region "c", and should be 480 lines. "`OK`" or "`NOK`" is reported.

4.3.4 Sanity Checks: Colours

To verify the colour handling, the model of the SRAM contains the values 4, 80, 170 and 213 for the top left most four pixels, and 2 for all other pixels, as illustrated below. Addresses outside the image gives 3.



The 4 and 80 tests different levels of the grey scale, while 170 and 213 tests different RGB values. Three sanity check processes checks the colour system.

The process `sanity_col0_p` verifies that the recreated pixcode has the expected values. Any pipeline discrepancy between the colours and the blanking, will be detected as colour check error. "`OK`" or "`NOK`" is reported.

The process `sanity_col1_p` verifies that no "out of image" pixels (with values 3) are sent to the screen while blanking is high. This is also likely to detect pipeline errors.

The process `sanity_col2_p` verifies that no other colours than 2, 3, 4, 80, 170 and 213 are ever provided.

4.3.5 Simulation Time

There are some "`done`" signals. The clock generator stops when all those are true. In this way, the simulation halts by itself when everything is done.

The Vertical timing will run two frames. With 60 frames per second, this means about 33 *ms* of simulation. This might take 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). This should be enough for the horizontal timing to report, and for you to manually verify the colours of the top left pixels.

5 Physical Verification

▶ Try your design on the FPGA board.

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 grey 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.

6 Requirements to Pass

In order to pass the lab:

- The code must be well written. Make sure the indentation is good. Use comments where appropriate.
- The simulation must show that everything works.
- All pixels must be visible on the screen (including top left pixel).
- You must understand the implementation (especially how the address is generated). No need to understand the testbench.

When you want to demonstrate, be ready with the programmer, waveform, code and understanding.

To show your understanding, you might be asked questions like "if we add +1 here, what will we see on the screen?", "how many bits are used for this?", "can this overflow?", "how does the hardware of this look?" or something else. Each student need to prove understanding. You might get a poser (swedish: kuggfråga), where your argumentation is important.

Understanding the test bench is *not* required.

Appendix A Common Errors

A.1 Syntax Error (compilation errors)

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

A.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.

A.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 analysing the different signals. See more details in Sec. 4.2.3.

A.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.

Arbitrary error	
Manipulated test bench	Copy a fresh version of the test bench from `K:`.
Changed top module	If you (e.g. by mistake) changed the top module, and did not update the VHDL code for it, then you synthesize and simulate different things. Look through the top module, and regenerate a new VHDL file for it.