PROJECT3 REPORT

1. **TITLE**

VGA CONTROLLER - Video Graphics Array (VGA) Driver

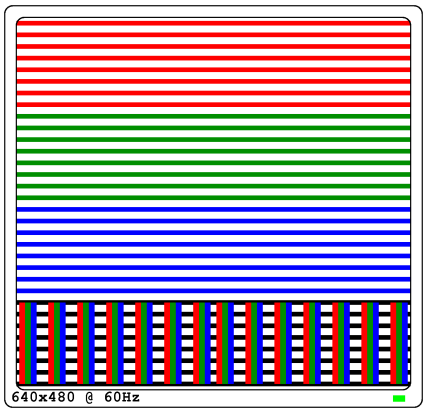
GAROUFALI MARIA NEFELI, AEM: 3129, 12/12/22

1. **SUMMARY**

The 3rd lab project of the course is presented in this report. This project concerns the construction of a VGA Controller/Driver. Its implementation took place in stages, which will be detailed. Each stage is described by Implementation, Verification, and Experiment.

1. **INTRODUCTION**

The goal of the 3rd lab work is to implement a VGA Controller/Driver (Video Graphics Array) display port to drive a conventional screen and display an image on it. To achieve this, we need to set part of the board's internal memory as the driver’s VRAM and initialize it to the image we want to display. The recommended and simple display image that can be used to verify the correct functionality of the controller is shown below.



The display of the control image on the screen should be stable and distinct. The monitor driver specifications are:

• Resolution 640 × 480 pixels (picture element — screen elements - dots) and

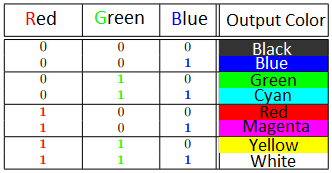
• Refresh rate (refresh rate) 60Hz.

The project goal was achieved by properly initializing the controller's synchronization signals for each line and each frame. Later, will be presented a more extensive analysis of the synchronization of the lines and the frames.

**4. PART A – Video RAM (VRAM)**

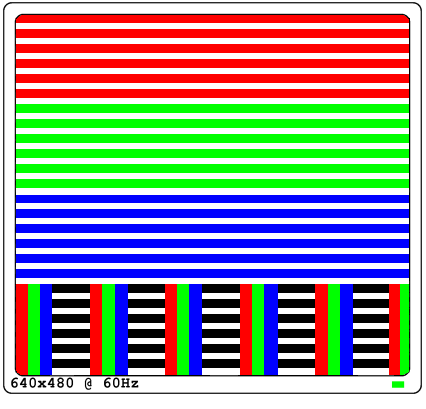
* **IMPLEMENTATION**

In this part, the memory to be displayed had to be implemented. Memory is a key internal part of a controller. Memory size is a function of resolution as it defines the number of displayed pixels and the number of colors per pixel. Thus, 640 × 480 resolution requires 640 × 480 = 307200 bits of memory, for each monochrome pixel. For the three colors allowed by the FPGA per pixel, 3 bits are required, so the total memory needed for color pixels is 307200 × 3 = 115.2 Kbytes memory. Most FPGAs include internal RAM (Block RAM - BRAM), which can be used either as a 1-bit memory, or as more than 1-bit memory, and can be mapped to the VGA Controller implementation in VRAM. Due to a lack of easily accessible BRAM in the device, however, and reasons for simplicity of implementation, it is suggested that you use three BRAMs, one for each color, and of size 16K × 1, to implement the required 128 × 96 VRAM. As said earlier, the VRAM should fit 128x96 image pixels and eight colors. The input of VRAM will be the address of a pixel, while the output will produce the color values of that particular pixel. the memory was not built from scratch. For the memory, an appropriate template was used which is supplied by Vivado. The only thing we should do is initialize the size of the memory we want to use to the color we want to display. Before we discuss the amount of memory, we should use it is good to mention the basic digital color matching of 8 colors, in terms of Red, Green, and Blue signal values. A table is shown below.



Now let’s discuss the amount of memory we want to initialize. As said before we want to use 3 BRAMS (one for each color) to present an image of 128 x 96 pixels. The total number of pixels is 12288. Every line of the memory has a size of 256. Dividing the 2 sizes we conclude that we must initialize 48 lines of each memory. This number corresponds to 2F as it is shown in hexadecimal. As that was all the memory I needed, I put everything else into comments. At this point, we should comment that every line of the memory corresponds to 2 lines of the image.

The image I made was based on the prototype but is not exactly the same. I preferred to spend less time and design something simpler. More specifically, I divided the image horizontally into 4 equal parts. So, each of these parts consists of 96/4 = 24 lines. To create the repeating pattern line-by-line I initialized the first line of the 3 memories suitably so that the output color would be the color red. Then I repeated this procedure to create the other lines too. At this point, we should mention that this initialization creates 2 same lines followed by 2 different lines. That is because, as mentioned earlier 1 line of the memory corresponds to 2 lines of the image. At the fourth horizontal part of the image, the initialization should create vertical lines. The lines I created are sparser than the prototype’s. A detailed image of the desired outcome of my initialization is shown below.



As I placed all 3 memories in the same file, I needed to change the instance names to differ. I also had to instantiate some values necessary for the functionality of the BRAMS. To do that I read the manual. More specifically, for the memory size, I wanted to use I set the read and write width to 1. From these widths, I knew the DO and DI sizes. Then I initialized the memories with my input address and with my outputs each for every color. Finally, as I only wanted to read from the memory, I set WRITE\_MODE(“NO CHANGE”) and WE(write enable) to 0. Every other enable signal was set to 1. I also put my clock and my reset to the memories.

* **SIMULATION**

This part was checked with the simulation method. I want to make sure that the memory has been properly initiated and functions. So, I made a test bench. Then I created the clock and initiated it. I also made a reset pulse to initiate the memory. Then, I gave with some delay various values to the address that would be input into the memory. Finally, I instantiated the module and run the simulation. The results are shown below and will be commented on.

Εικόνα που περιέχει κείμενο, ηλεκτρονικές συσκευές, στιγμιότυπο οθόνης, εμφάνιση

Περιγραφή που δημιουργήθηκε αυτόματα

As we see above, a reset is done to run the memory. With a 50ns delay, the address changes values pointing every time to a different bit of the memory. The values of red, green, and blue for each color are shown as output. As it is shown in the screenshot, all these pixels belong to the first line of the image, which is red. This means that only the red color will be 1 while green and blue will be 0. This is confirmed by the simulation. We also notice that there is a delay from the time the address enters the memory until the time the result comes out. This delay is the time needed from the memory to produce the output.

* **EXPERIMENT**

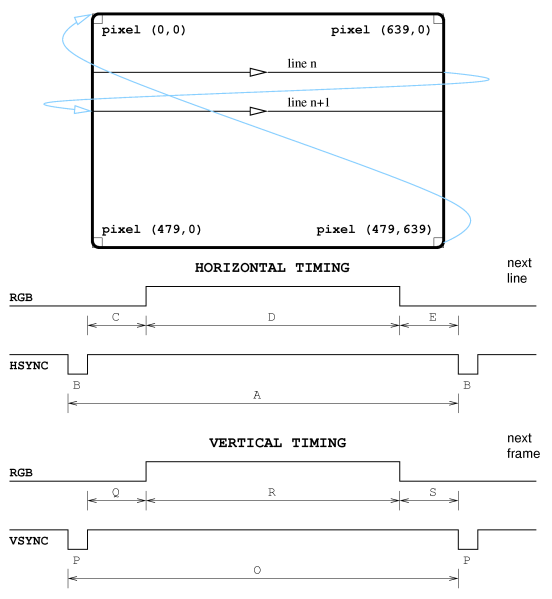
This part couldn’t be autonomously checked experimentally in the FPGA.

**PART B & PART C – HSYNC, VSYNC & Display Image**

* **IMPLEMENTATION**

In this part, the HSYNC and VSYNC signals had to be implemented, with the appropriate timing, for a resolution of 640 × 480 and a 60Hz Refresh Rate. Before analyzing the implementation of the HSYNC pulse we must see how it works.

In addition to colors, the VGA port includes two timing signals, HSYNC and VSYNC, which time the horizontal and vertical coordinate of the screen respectively. In Cathode Ray Tube (CRT) displays, these control the beam, while in the Liquid Crystal Display (LCD) type, they control the recording of new pixels. Figure 2 shows the waveforms of the timing signals, HSYNC and VSYNC, which are identical regardless of resolution, and the horizontal and vertical inversion process. The HSYNC signal controls horizontal timing, while VSYNC controls vertical timing. The image is active in the intervals D horizontally, and R vertically. As long as the image is active, the values of the red, green, and blue signals determine the color of the display pixels in relative screen position. So, they should change accordingly. The horizontal and vertical scanning, as shown by their waveforms, are identical to each other. We're focusing here on horizontal scanning, and it works similarly to the timing of the vertical. For time E after the end of the active image, the so-called front view (Front Porch — before the pulse), the image display is disabled. Next is the B pulse, the width, and frequency of which are shown on the screen in the horizontal analysis. Then, for time C, the back view (Back Porch — after the pulse), the image remains off. After C, the next line appears for time D. Similarly, vertical scanning works. A Figure of this analysis is shown below



Now we should compute the duration of each stage for both Horizontal and Vertical Timing for the specific resolution we want to achieve. Although the times are given, we can do some calculations to verify these values and understand how they arise. We start by analyzing the 60Hz refresh rate. That means we refresh (display again) the image 60 times per second. From that, we can compute the average time of the vertical timing (O). This will be 1/60 = 0.016667. We round the result to 0.0167s which is 16,700,000,000ns. The corresponding time for the horizontal is given and is equal to 32,000ns. The values of the pulse width, the back porch, the display time, and the front porch are also given and will be viewed in detail in the table below. This table is from the board’s manual for the VGA System Timing. Having all the values for the horizontal timing we can verify the display time value of the vertical. As we want to display 480 lines and each line needs an overall of 32000ns to be displayed we compute that for the whole image or 480 lines we need 480\*32000 = 15,360,000ns. This is the same value shown in the table. We can finally confirm that the individual times for the horizontal and vertical if added together give the total time (A and O). While for the horizontal timing the result of the sum is the total time (A), for vertical timing the result is 16.672ms instead of 16.7ms. So, we ignore the rounding and keep the most accurate value because otherwise there will be problems with the synchronization. So finally, the total time for the vertical (O) is equal to 16.672ms or 16,672,000,000ns.

Εικόνα που περιέχει πίνακας

Περιγραφή που δημιουργήθηκε αυτόματα

In the board’s manual, it is suggested to use a new clock of 25MHz instead of the 100MHz clock of the board. So, all the clock’s pulses shown in the table are computed with this clock. Initially, I decided to use this clock too. For this reason, I used an MMCM to divide the 10ns clock and generate the new 40ns clock. Although I didn't need this code since I decided to implement it with the board’s clock, I have left this code in the comments inside the top module. As we use the board’s 100MHz clock I made the calculations again by dividing each value (in ns) by 10ns which is the period of the clock. The following table is the edited table of the manual with my calculations and with the Symbols shown in the figure of the timing analysis.

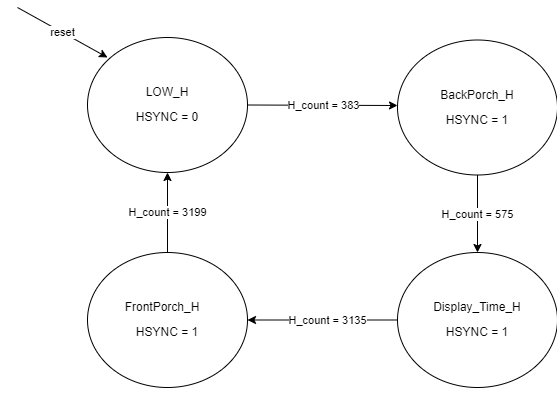
Εικόνα που περιέχει πίνακας

Περιγραφή που δημιουργήθηκε αυτόματα

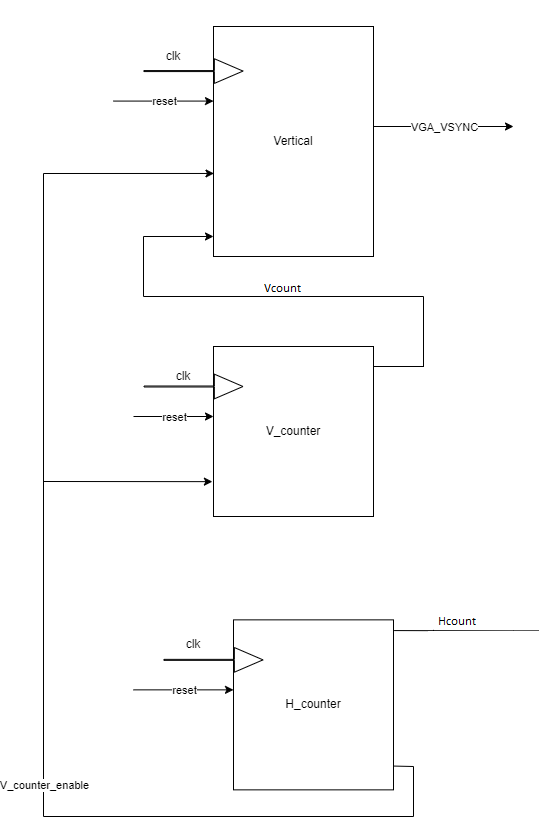
Both tables have an additional column for vertical sync. This column expresses the vertical sync timing concerning the lines. Since I used the specific times for the vertical synchronization let's analyze them more. If we divide the total time (O) of the vertical with the total time (A) of a line (16,672/32) we take as result 521 lines. In other words, a frame will change at the time 521 HSYNCS will have passed. Continuing the same process (dividing the individual times of the vertical with the total time of the horizontal) we take the rest values shown in the table. We can notice that the number of lines corresponding to the display time of the vertical is 480, exactly the number of lines we want to display.

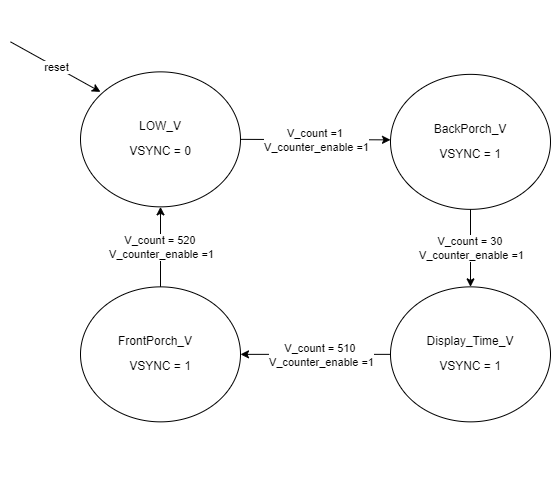
Having analyzed the times and the theory behind Synchronization, let's now see the implementation in detail. To implement horizontal synchronization, I need to make a counter and an FSM. The counter counts till the number of clock pulses I calculated above. At certain values of the counter, the FSM changes its state, and the output HSYNC changes (or does not) its value. So, the first unit is H\_counter and is a sequential counter unit. As it is a sequential system it is implemented with non-blocking assignments. It takes as input the clock and the reset and it gives as output the value of the counter. Since the max value of the counter is 3199, we need 12 bits. So, the width of the output is [11:0]. This value of the output is the input of the FSM. The FSM is implemented inside the unit Horizontal. This module takes as inputs the clock, the reset, and the value of the counter as we previously said. Inside the module, there is implemented a MOORE FSM. The output of the FSM is the pulse HSYNC which is also the output of the module. Let’s see the FSM in more detail. The FSM has 4 states. The first state is the state LOW\_H where HSYNC is 0. The next state is the state BackPorch\_H where HSYNC is 1. Moving on we go to the Display\_Time\_H state where HSYNC is also 1. Finally, there is the state FrontPorch\_H. There, HSYNC is also 1. Below we see the dataflow of the counter (H\_counter) and the FSM (Horizontal).



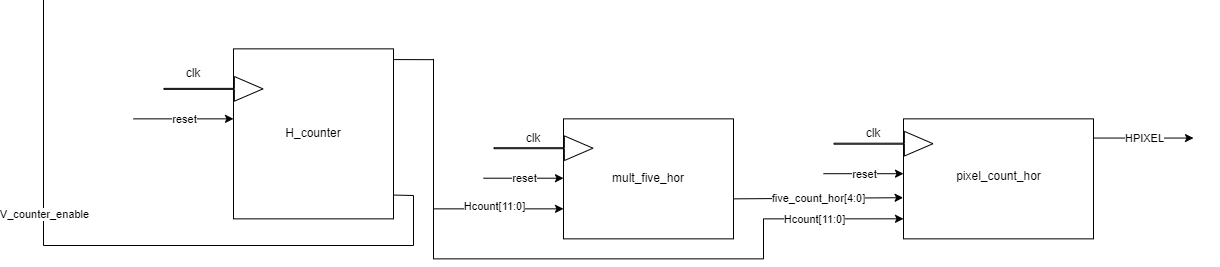


We notice one more output from the H\_counter. As mentioned earlier I made the Vertical Timing according to the lines. So, each time a line (HSYNC) has passed it is generated a pulse that is an enable signal for the vertical counter. Let’s deepen into the Vertical Timing. As in Horizontal Timing here we have a counter too. This counter increases its value concerning the lines and functions as a signal for the change of states of the FSM. More specifically, the sequential system of the counter takes as inputs the clock, the reset, and the enable signal from the horizontal counter (V\_counter\_enable). As it is a sequential system it is implemented with non-blocking assignments. This value of the output is the input of the FSM. The FSM is implemented inside the unit Vertical. This module takes as inputs the clock, the reset, the value of the counter as we previously said, and the enable signal from the H\_counter (V\_counter\_enable). Inside the module, there is implemented a MOORE FSM. The output of the FSM is the pulse VSYNC which is also the output of the module. Let’s see the FSM in more detail. The FSM has 4 states. The first state is the state LOW\_V where VSYNC is 0. The next state is the state BackPorch\_V where VSYNC is 1. Moving on we go to the Display\_Time\_V state where VSYNC is also 1. Finally, there is the state FrontPorch\_V. There, VSYNC is also 1. Below we see the dataflow of the counter (V\_counter) and the FSM (Vertical).

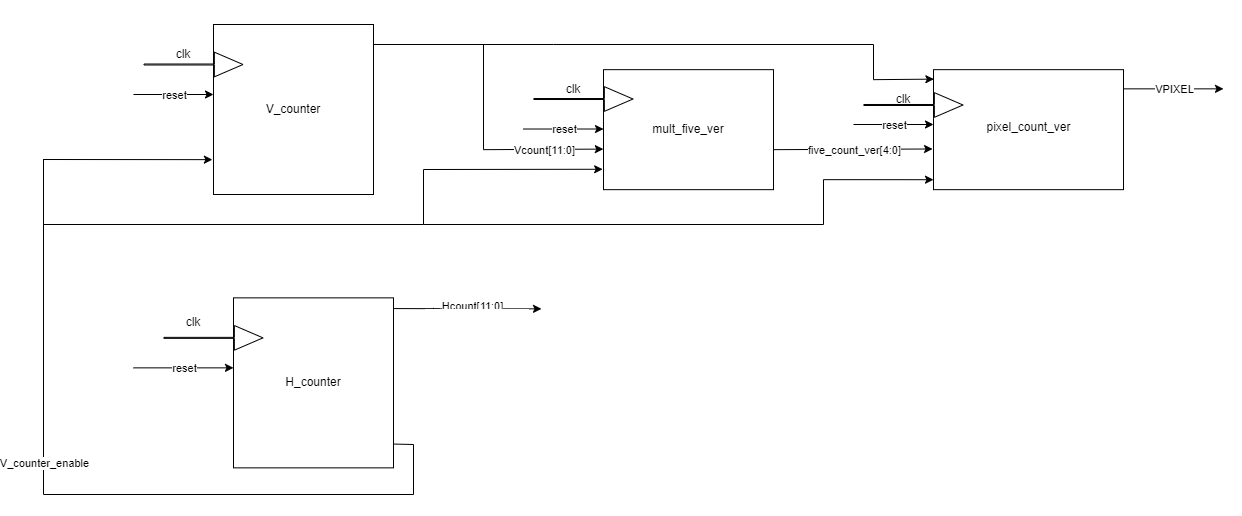




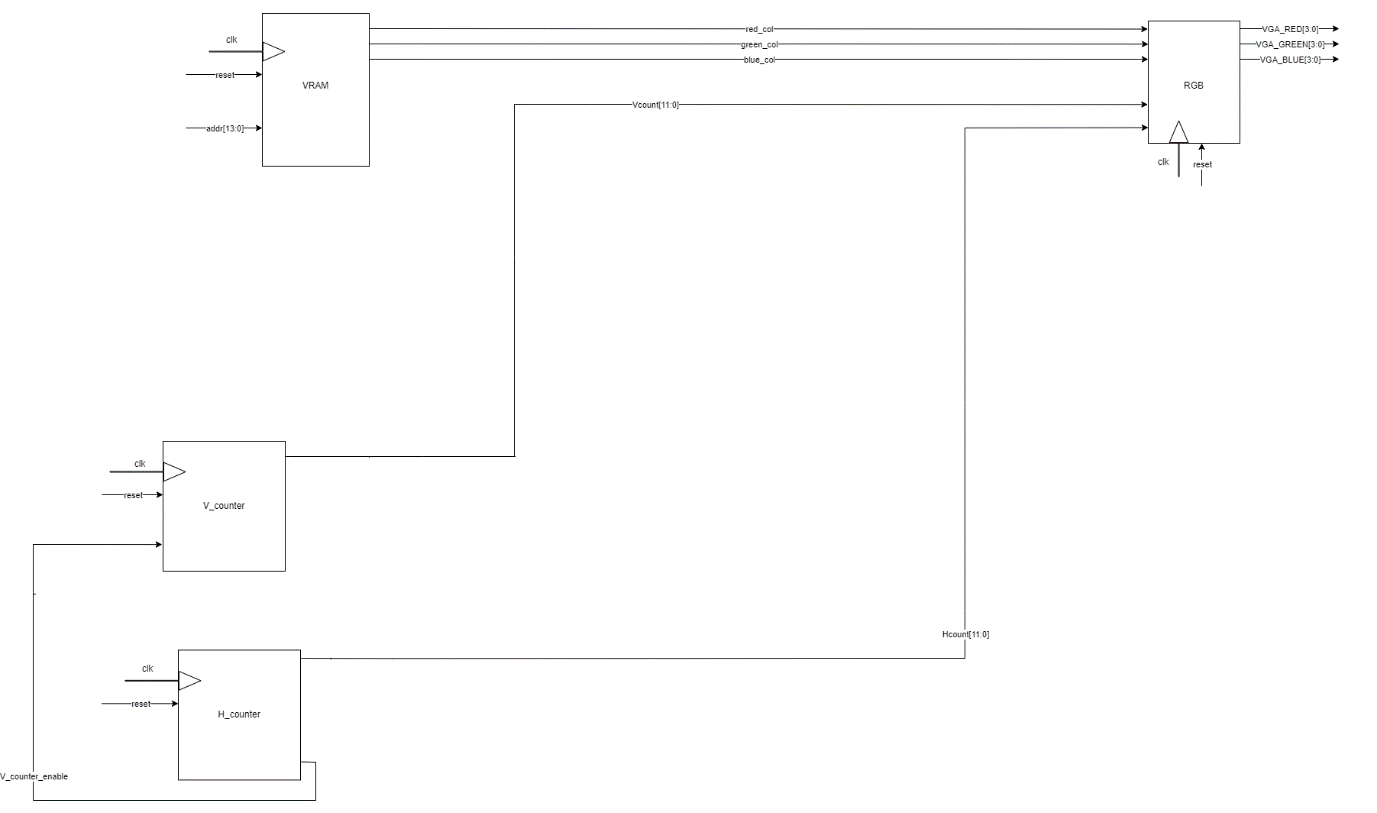
At this point, we have achieved both horizontal and vertical synchronization. This is necessary for the activation of the screen and the display of the image. However, the image we have saved in memory is 128X96 pixels, while the image we want to display is 640x480. That means that The VRAM should be scaled by the Controller, to fill a 640 × 480 resolution screen. To achieve that scaling we must implement some more modules both for lines and columns. Let’s start with the columns. First, by making a simple division we conclude that we want to scale the image by 5 times. To achieve this, it is necessary to implement a fivefold counter. Then we must also implement a counter which will count the pixels of each line. The logic is that for each Pixel in the line the quintuple counter will make a “cycle”. Until this cycle is done, we will display the same pixel. In this way, we will finally multiply the pixels of a line since we will have displayed 640 instead of 128. Let's delve deeper into the implementation of this logic. First, we must notice that this whole process takes place in the display time of the horizontal timing. Now we must calculate how much time we should display each pixel as well as, the duration of the quintuple counter’s cycle. Looking back at the times we have we see that the display time for the horizontal timing is equal to 25600ns. In this 25600ns we want to display 640 pixels. So, the counter will increase its value every 25600/640 = 40ns. However, the 640 pixels are per five identical. This means that the counter’s cycle to change the address of the pixel and move to the next one is equal to 25600/128 = 200ns. This time corresponds to 20 clock pulses. So, let’s deepen the implementation. The mult\_five\_hor Unit is the sequential unit of the quintuple counter. As a sequential unit, it is constructed by non-blocking assignments. This module takes as input the clock, the reset, and the H\_count and has as output the counter’s value. The max value of the counter is 19 so the width of the variable is 5 pixels. If reset or if we are one cycle before the display time for horizontal, we reset the counter. We do the same if the counter has reached its max value (aka 19). In any other case, the counter increases its value at the posedge clock. The value of the counter goes as input in the other sequential unit pixel\_count\_hor. As a sequential unit, it is constructed with non-blocking assignments. This unit takes as inputs the value of the counter as we mentioned before, H\_count, the clock, and the reset. The output of this module is the HPIXEL, a very important variable necessary for the address. Now let’s deepen the implementation of the module. If reset or if we reach the end of the line, HPIXEL is set to 0. If we are in the display time of the horizontal timing, and if the quintuple counter is in its last value HPIXEL increases its value by 1 in the posedge clock, else it reserves its previous value. In any other case, (BackPorch, FrontPorch) HPIXEL is set to 0. Below is shown the dataflow of these modules with every other module needed.



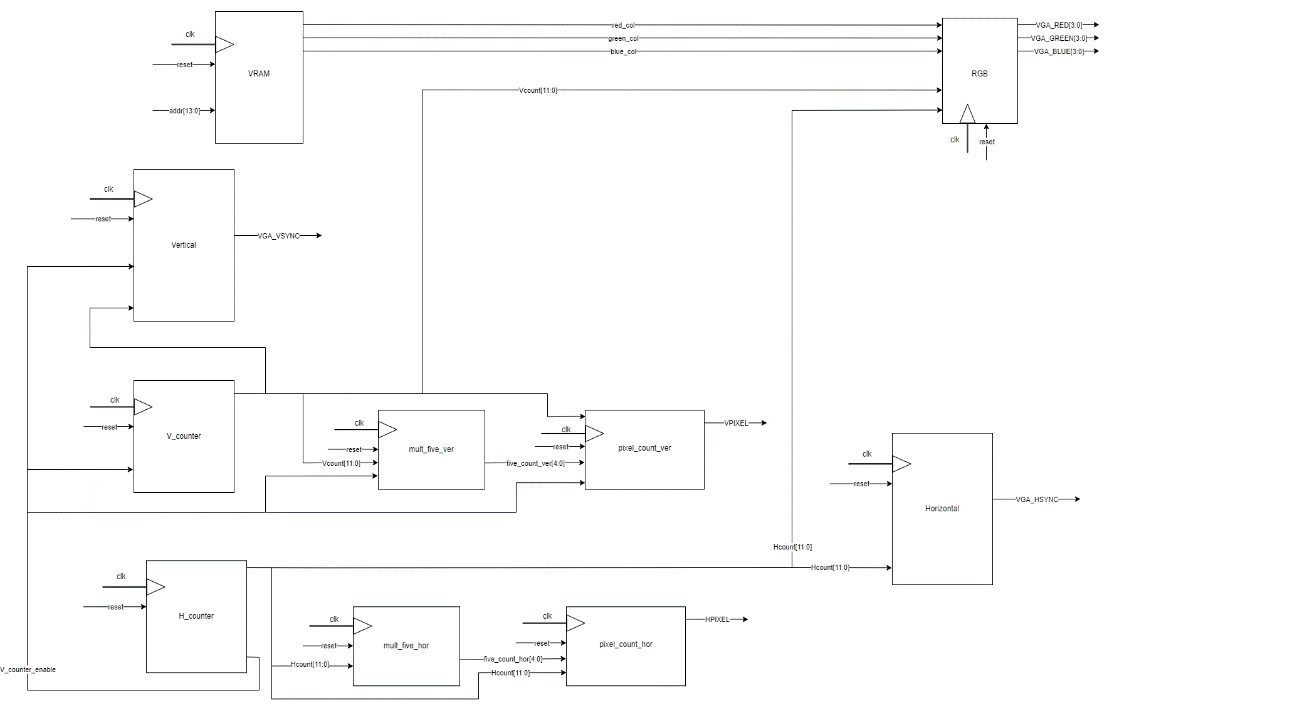
Now let’s analyze what happens with the lines. As it happens with the columns, we want each line to be displayed 5 more times to get the final image of 640x480. To achieve this, it is necessary to implement a fivefold counter. Then we must also implement a counter which will count the lines. The logic is that for each line the quintuple counter will make a “cycle”. Until this cycle is done, we will display the same line. In this way, we will finally multiply the lines since we will have displayed 480 instead of 96. Let's delve deeper into the implementation of this logic. First, we must notice that this whole process takes place in the display time of the vertical timing. Since the vertical timing is done according to the lines, there is no need to calculate clock cycles or any other time. The modules will take as input the V\_counter\_enable which will synchronize them with the end of each line. Starting with the quintuple counter is a sequential unit constructed by non-blocking assignments. It takes as inputs the clock, the reset, the V\_count, and the V\_counter\_enable. The output of this module is the value of the counter. Since the max value of the counter is 4 (0 -> 4) we need width [2:0] for this variable. If reset or if we are 1 clock cycle before the display time the value of the counter is set to 0. If the counter has reached its max value and we are at the end of the line (V\_counter\_enable = 1) we reset the counter. If, however, we are at the end of a line and the counter hasn’t reached the final value we increase its value by 1. In any other case, the counter maintains its value. The value of the counter goes as input in the other sequential unit the counter of the lines. This module (constructed with non-blocking assignments) takes as inputs the clock, the reset, the V\_counter, the V\_counter\_enable, and the value of the quintuple counter as we previously mentioned. The output of this module is the VPIXEL, a very important variable necessary for the address. Now let’s deepen the implementation of the module. If reset or if we reach the end of the frame, VPIXEL is set to 0. If we are in the display time of the vertical timing, and if the quintuple counter is in its last value, and we are at the end of this line, VPIXEL increases its value by 1 in the posedge clock, else it reserves its previous value. In any other case, (BackPorch, FrontPorch) HPIXEL is set to 0. Below is shown the dataflow of these modules with every other module needed.



The HPIXEL and the VPIXEL are connecting to create the address. This is the assignment that happens to the top module. Address = {VPIXEL, HPIXEL}. This variable has a width of 14 bits since it is constructed from variables with a width of 7 bits. At this point, we theoretically are able to activate the screen and display an image with a resolution of 640x480 and a refresh rate 60Hz. We have the image saved in the memory. We also have the implement horizontal and vertical synchronization to activate the screen and finally, we have implemented the tools to multiply the small image of the memory and display it in the dimensions we want. The only thing that is missing is to combine all these modules to display the image. To achieve that we create an extra module called RGB. This module is a sequential unit (constructed with non-blocking assignments). It is a flip flop that assigns the values of the VGA\_RED, VGA\_GREEN, and VGA\_BLUE at the posedge clock whenever we are in the display time of both the horizontal and vertical timing. According to the manual of the board, the width of the variables VGA\_RED, VGA\_GREEN, and VGA\_BLUE that will connect to the board have a width of 4 bits. The unit takes as inputs the clock, the reset, the colors red\_col, green\_col, and blue\_col coming out from the memory, and both the counters for the Horizontal and Vertical Timing (H\_count, V\_count). The output is the 3 colors VGA\_RED, VGA\_GREEN, and VGA\_BLUE. Now let’s see more specifically what happens inside this module. If reset the output (aka 3 colors) is set to 0. Else while H\_count and V\_count indicate that we are simultaneously in the display time of the Horizontal and Vertical Timing or one clock cycle before we set the output colors at the corresponding values coming from the memory. In any other case (BackPorch, FrontPorch) we set the output colors to 0. Note that we assign every bit of the 4 bits of the output variables with the corresponding value of the input. A detailed dataflow of this implementation is shown below.

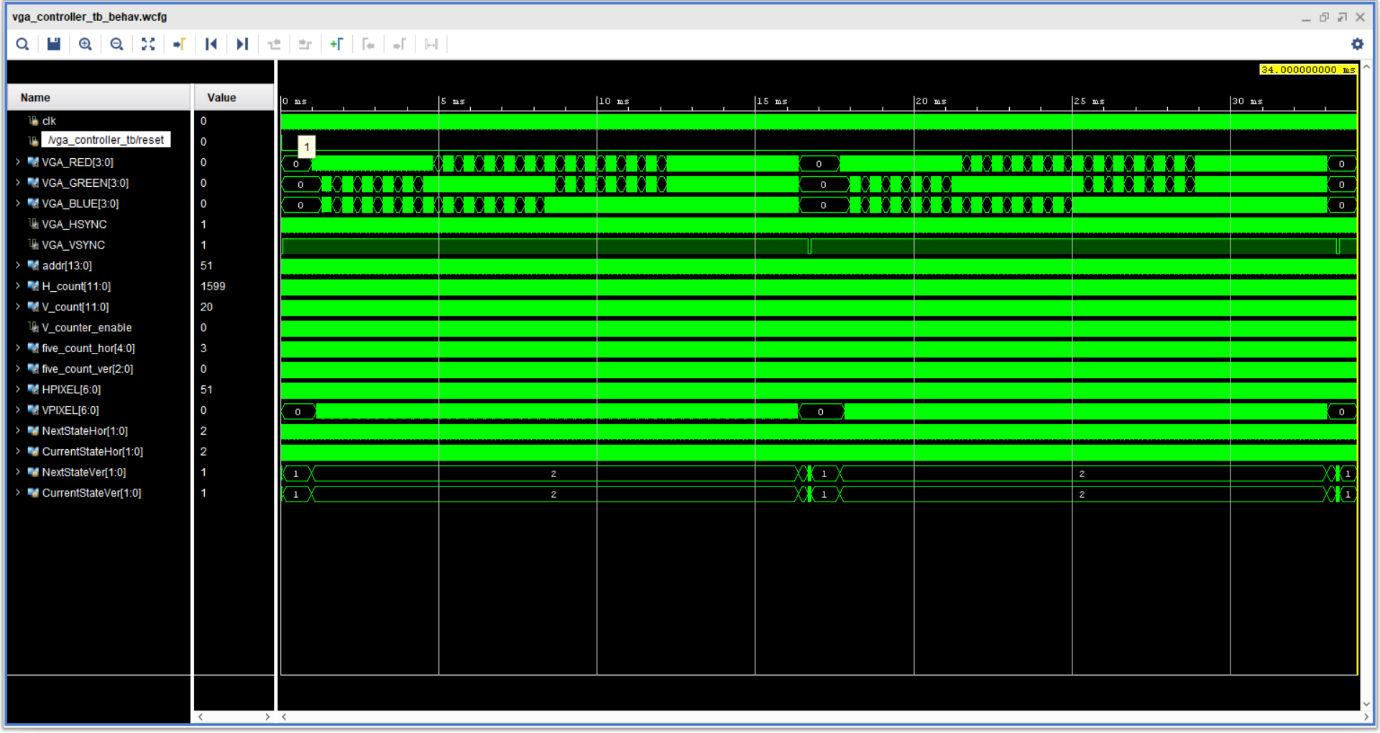


All the above modules are instantiated in the top module. We broke them into parts to make easier the analysis. In fact, there is only one dataflow that includes all these modules we reviewed. This data flow is shown below.

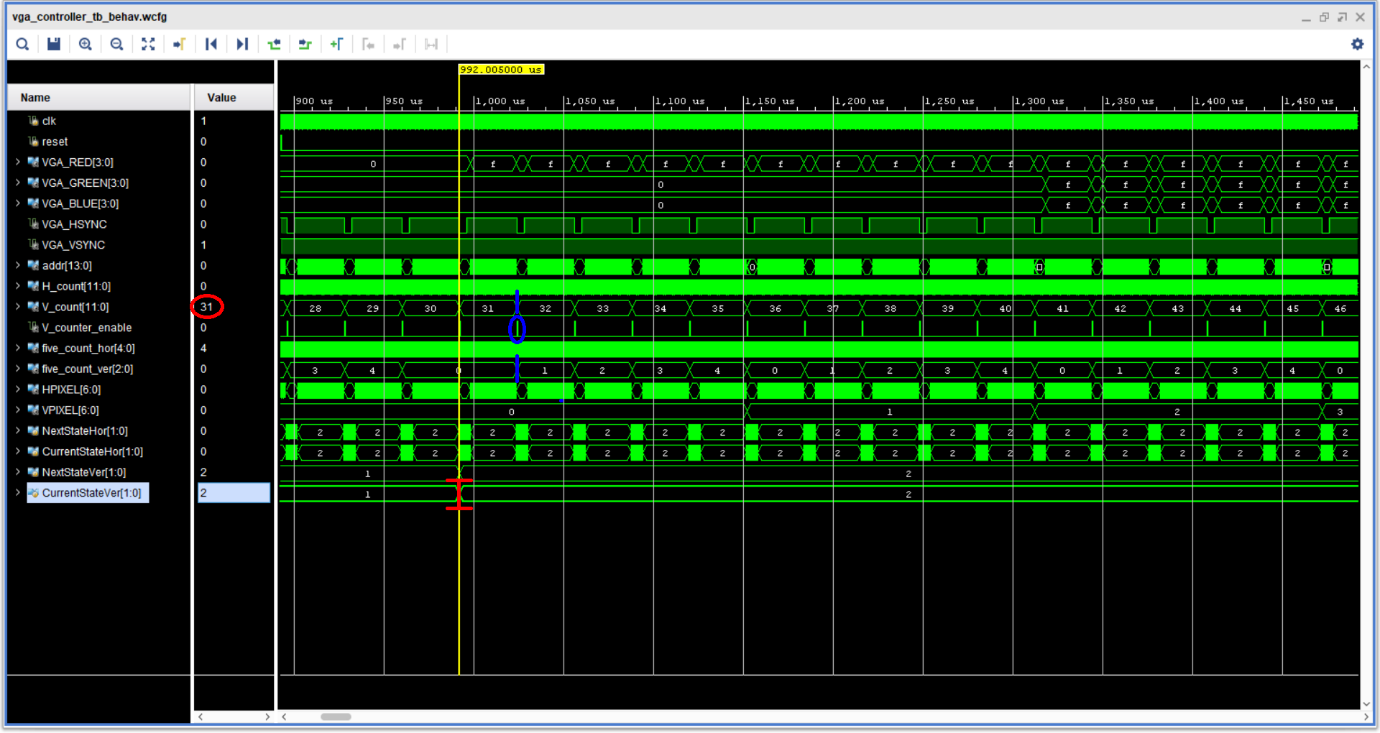


* **SIMULATION**

This part was checked with the simulation method. So, I made a test bench. Then I created the clock and initiated it. I also made a reset pulse to initiate the modules. Finally, I instantiated the module and run the simulation. As I wanted to check almost every module and every part of the implementation except the memory, I added all the necessary signals to the waveform. The results are shown below and will be commented on in parts, gradually. Note that in order to confirm the correctness of the project I run for 2 frames. So, let’s start with the general picture. Below is shown the result of the simulation for the total time without any zoom or edit.



At first glance, we notice that the pattern the colors make resembles the image we wanted to display. We also notice that the colors are displayed inside the display time of the vertical timing. The states of the vertical timing seem also to be right. Another thing we see is that VPIXEL is set to 0 except for the display area, as we wanted. This screenshot is just a first reading. It is zoomed out and we can’t distinguish in detail all the information we want. So, let’s deepen the analysis. Below are shown more detailed screenshots of the simulations with a more extensive explanation.



In this edited screenshot, we see (marked with red) that the display time of the vertical timing starts at the value 31 of the V\_count counter. This is the right value as we show in the FSM diagram before. Another piece of information we take from this screenshot is marked with blue. More specifically, we see that both the five\_count\_ver (quintuple counter) and V\_count change to their values with a posedge clock when V\_counter\_enable is 1. Now it is clearer the usage of this enable. As a state of these counters lasts many 10ns clock cycles, this enable signal is necessary to point out the end of the state which is the end of the current line. If there was not that signal all the states would last one 10ns clock cycle and that would be wrong. The counter that changes the VPIXEL also uses this enable signal. Now let’s continue the analysis with another screenshot.

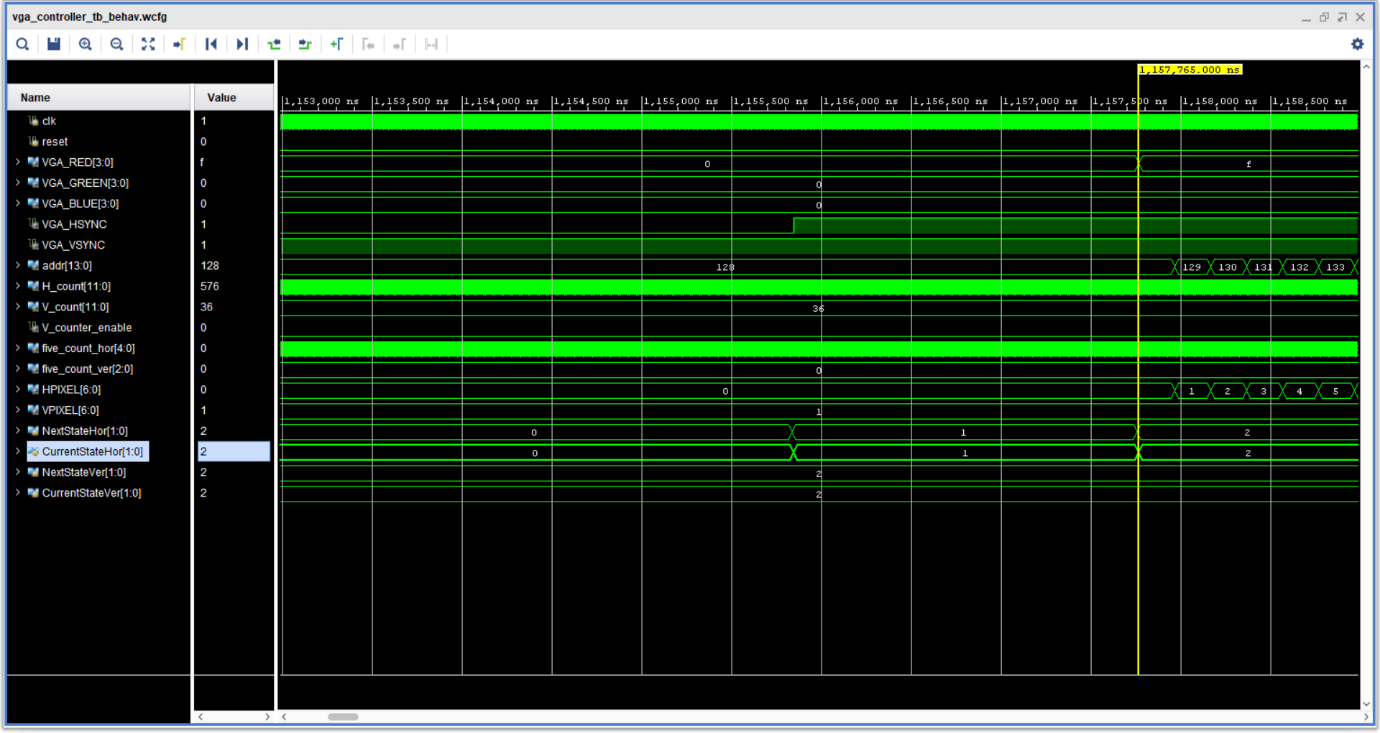


In this screenshot, we see marked in red the reset that happens at the five\_count\_ver as we enter the display time of the vertical. This is necessary to count properly. We also see that if this reset hadn’t been done the implementation would probably face problems and mistakes. We also see marked in orange that VPIXEL changes its value only after the quintuple counter will have make a full cycle. During this time, we notice that 5 full HSYNC happen which corresponds to five lines. This seems to function well. The goal was to multiply by five the current line we wanted to display and that happens as we see with the cyan marking. More specifically in cyan, we see the 2 first red lines to be displayed 10 times instead of 2 times. In other words, the quintuple (2\*5). Finally, another piece of information we get from this screenshot is that the pattern of the image may be the desired one. We conclude that because we see the red lines at the beginning of the image followed by the white lines and then again, the red lines. Of course, we cannot be sure about that, but it is a good sign. Now that we have finished the analysis of all the vertical modules, let’s make a deeper analysis with the following screenshot. It is zoomed in to see what happens with the address.

Εικόνα που περιέχει κείμενο, υπολογιστής

Περιγραφή που δημιουργήθηκε αυτόματα

First, we can notice clearer the HSYNC states. In red color, we observe that in any other state but the 2nd state, which is the display time or the horizontal, there are no colors displayed or to be accurate it is displayed a black color as all the color variables are set to 0 in these states. From this screenshot, we can also notice (marked with blue color) that the pulse HSYNC is set to 0 periodically at state 0. So, we conclude that this works as we wanted. Finally, once again we can see that the value of VPIXEL changes after the quintuple counter has made a cycle and reached its final value. Another piece of information we get from this screenshot has to do with the V\_counter\_enable. We explained this also previously but here it is clearer. This enable pulse happens at the end of the 3rd state just before the state 0 of the horizontal timing. Finally, we can also observe with the orange color that in the same states the address maintains its previous value. This sounds logical because since nothing is displayed the address has no reason to change value. This will be explained in more detail with the following screenshot which is more focused.



First, we can notice once again that the display of colors is done once we move to the 2nd state of the horizontal AND if we are simultaneously in the 2nd state of the vertical. In this screenshot, the second line is going to be displayed. We can see that the address has maintained the previous value and starts counting from that value. On the contrary, HPIXEL starts counting from 0 each time we want to display a new line. At this point, the only thing we haven’t explained yet is the HPIXEL counter and the quintuple counter for the horizontal. The final screenshot will explain these in detail.

Εικόνα που περιέχει κείμενο, ηλεκτρονικές συσκευές, υπολογιστής

Περιγραφή που δημιουργήθηκε αυτόματα

We are at the start of the display time for the horizontal and inside the display time for the vertical. First, we can see that the value of the H\_count is right. Then we see that the display of the red line begins at the time we enter the display time of the horizontal. At this point, we notice that the quintuple counter starts counting from 0. So, we expect that this counter functions well. After the counter reaches the value 19, we see both the address and the HPIXEL increase their values. This is logical and is a sign that the project has no mistakes there. In other words, HPIXEL maintains the same pixel’s value until the quintuple counter makes a full cycle. In this way, we display each pixel 5 times and multiply the image as it was explained above. The reason why this counter is implemented to count 20 times has been explained earlier too.

* **EXAMPLE**

This part is actually the whole project and was tested with the experimental method. More specifically, to experiment on the board with a real board I needed to make the xdc file before. This file contains the constraints and describes the clock as well as the pins and the buttons of the board that will be used as well as the variables that will connect to these. To do that I searched the manual to see what pins of the FPGA correspond to the ports of the VGA. Having this information, I wrote the xdc file. After that, I run synthesis and implementation to make sure there are no latches in the design or errors. The results of the synthesis showed some signals that were not in the sensitivity list as they should. There were no latches in the design. I corrected that and run the implementation, which resulted in no serious errors. After all these, I generated the bitstream to check the project on the FPGA. The first try was quite near the wanted result. More specifically, the screen turned on, and the image was presented with vivid colors. There was no video or loss of the image’s information. The only problem was a difference in the pattern of the resulting image with the image I wanted to present. I checked the VRAM again and I noticed that in a point of the memory, I forgot that 1 line of memory corresponds to 2 lines of the image. The mistake I made was that 2 continuous lines of the image were different. So, I changed the initialization, and the problem was fixed. The second try in the FPGA was totally successful. This time the image was clear. Below is shown the result I got the second time.

Εικόνα που περιέχει κείμενο, εσωτερικό, ηλεκτρονικές συσκευές, εμφάνιση

Περιγραφή που δημιουργήθηκε αυτόματα

As we see the result is the desired one. I also checked the image on the reset button, and it worked as expected.

**5. Conclusion**

This project was about the implementation of a VGA Controller display port to drive a conventional screen and display an image on it. The milestones of the implementation of this project are the Synchronization of Horizontal and Vertical Timing, the creation of the Block RAM / VRAM that will keep the image we want to display, and finally the scaling we may need to do if the image is saved in memory is smaller than the one, we want to display. The creation of the memory is easy for the most part and does not have any tricky points. Someone has however to read the manual to make the initialization right. The most important and perhaps difficult part of the project is Synchronization. If someone does not use the clock suggested in the manual of the board has to be very careful with the calculations. Any clock cycle loss will be translated into a lack of synchronization. Special attention should also be paid to the rest of the project's counters related to scaling. Since these counters determine the address of the displayed pixel, an error in them could mean a distorted image display. For this reason, it is a good practice to reset the scaling counters one clock cycle before their use. That way they will be ready to count from 0. Another good practice is to describe what should happen in every case. Although this is unnecessary, because no latches will be created, since we talk about sequential units, by doing this someone will make the debugging easier and will avoid any undefined behavior of the counters. A final tricky part of this project that I faced either, is the change of a counter with non-blocking assignments. More specifically, if we want a counter to change its value with a different clock from that we use, we should create an enable signal to make that change. In any other case, this counter will change its value with the clock we use, and its states will not last as we wanted them to.