ECE333 - Lab 3
Video Graphics Array (VGA) Driver

Nikas Ioannis-Iason

10/11/2024

Contents

1 Introduction **1**

2 Part A - VRAM module **2**
2.1 Implementation . . . 2
2.2 Pixel controller . . . 2
2.3 Simulation . . . 2
2.4 Notes . . . 2

3 Part B & C - HSYNC and VSYNC controller **2**
3.1 Implementation . . . 2
3.1.1 GSYNC (General-SYNC) controller . . . 2
3.1.2 GSYNC Finite state machine . . . 3
3.1.3 Top-level module . . . 4
3.2 Simulation . . . 5
3.3 Live testing . . . 9

4 Optional feature: Displaying an interactive sprite **11**
4.1 Implementation . . . 11
4.1.1 Renderer . . . 11
4.1.2 Sprite controller . . . 11
4.2 Simulation . . . 11
4.3 Live testing . . . 11

1 Introduction

The goal of this project was to implement a VGA (Video Graphics Array) driver and test it on an FPGA. The project is divided into four main components:

- A. VRAM module - Video-RAM is the memory used to store the intensity of the colors (R, G, B) of each pixel that is displayed.
- B. HSYNC and VSYNC controller - Controls the timing for HSYNC and VSYNC signal of the VGA
- C. Pixel controller - Controls the color output based on the HSYNC and VSYNC signals and handles prefetching from the BRAM
- D. Top level module - Connects the previously mentioned modules

2 Part A - VRAM module

2.1 Implementation

For the VRAM module the IP `RAMB18E1`, which is a 18Kbytes BRAM provided by the FPGA, was instantiated 3 times, 1 for each color (R, G, B). The read width was set to 18 bits and the write width to 1 bit (used in the optional feature). Additionally, port A was used for reading data and port B for writing. For initialization the `.INIT_xx` parameters were used, which were generated using a Python script that converted an 8-bit color image to 1-bit color and created the contents of the parameters.

2.2 Pixel controller

The BRAM on the FPGA needs one clock cycle to complete a read operation, so to ensure the correct timing on our read operations for the pixels, we have to set the read address one cycle before we actually need to send the contents of the memory to the VGA. In our case, because the read width of the memory is 18 bits (16 data and 2 parity bits), prefetching is only needed once per 16 bits read. The pixel controller handles this prefetching process and also determines whether to display a blank pixel or actual pixel data, based on signals generated by the HSYNC and VSYNC controllers described below.

2.3 Simulation

To simulate the VRAM module, an almost empty instantiation of the BRAM IP was used that only had the addresses 16-31 set to 1 and every other to 0. A simple testbench was used that performed a reset and after some time a read operation, by changing the read address.

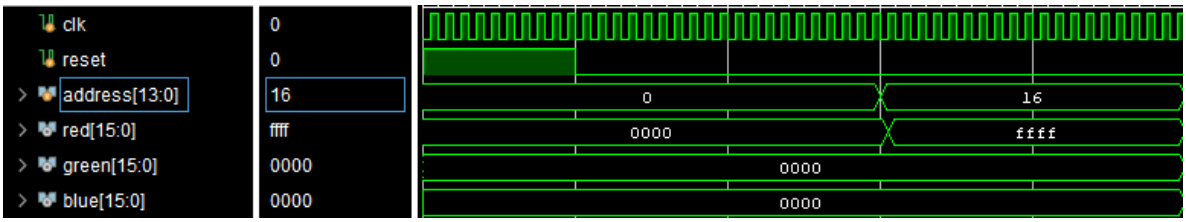


Figure 1: Simulation of the VRAM module showing the output changing after a read operation

2.4 Notes

When using asynchronous reset on signals that driven to the BRAM IP a warning appears when running implementation in Vivado which explains that using asynchronous reset or set signals for the `RDADDR` input can lead to the corruption of the memory's contents. Although this is correct, in our case there are multiple reasons why this warning can be ignored. First, we use the same reset signal for the BRAM so whenever the `RDADDR` resets, so does the BRAM. Additionally, we filter our reset signal with a debouncer and a synchronizer so it's not exactly asynchronous.

3 Part B & C - HSYNC and VSYNC controller

3.1 Implementation

The implementation of the HSYNC and VSYNC controllers uses a single parameterized module for both, as they fundamentally share the same states. The only difference between them is the number of cycles each must wait to transition between states.

3.1.1 GSYNC (General-SYNC) controller

The GSYNC controller instantiates the FSM used for creating the synchronization signals along with a `rgb_enabled` signal that shows when RGB values should be sent to the VGA. The controller itself is responsible for upscaling the memory contents by 5 times. This is done by displaying a single

pixel from the memory 5 times and because this controller is used for both the VSYNC and the HSYNC the upscaling is automatically done horizontally and vertically. This upscaling is necessary due to the limited memory resources available on the FPGA. We only used 3 BRAM (1 for each color) instantiations of 18Kbytes that could hold at max an image of resolution 128x96 which we can display to our desired 640x480 resolution by upscaling the contents of the memory 5 times.

3.1.2 GSYNC Finite state machine

The FSM used for the GSYNC is a Moore type FSM and consists of 9 states. The only input is an enable signal which was used to easily disable the display without resetting and just by using a switch. The FSM has the following 4 outputs:

- `sync` - The HSYNC or VSYNC signal used for VGA synchronization
- `rgb_enabled` - Used by the `pixel_controller` to decide if a blank pixel should be sent to the VGA or not
- `state_end` - Used by an internal counter to know when to reset for the next state.
- `frame_end` - Shows when a scanline time (or total frame time) has passed.

The states of the FSM are used to represent the different timing states we need to comply with the VGA protocol. In order to wait for each timing state to end we need a counter to wait a specific amount of clock cycles depending on the state. Using a `25MHz` clock and using the times provided by the VGA protocol we can figure out the cycles needed for each state. These cycles are passed as parameters to the `gsync_controller` by the top-level module and are shown in the table below:

Name	Time to Wait	Cycles to Wait
HSYNC Pulse Width	3.84μs	96
Back porch	1.92μs	48
Display time	25.6μs	640
Front porch	16.67ms	16
VSYNC Pulse Width	64ms	1600
VSYNC Back porch	928ms	23200
VSYNC Display time	15.36ms	384000
VSYNC Front porch	320ms	8000

Table 1: Timing states and their corresponding cycles to wait

Note that because we need to wait this many cycles and the counters start from zero in the parameters passed to the controller we subtract 1 from the cycles needed to wait the correct amount of clock cycles.

The states are the following:

- `IDLE` - Nothing is displayed on the screen and the `sync` signal stays high.
- `PULSE` - The `sync` signal goes to low and the `rgb_enabled` is still low.
- `PULSE_END` - Same as `PULSE` but the `state_end` is high so the counter resets.
- `BACK_PORCH` - The `sync` signal goes back to high and the `rgb_enabled` is still low.
- `BACK_PORCH_END` - Same as `BACK_PORCH` but the `state_end` is high so the counter resets.
- `DISPLAY` - The `sync` signal stays high and the `rgb_enabled` goes to high.
- `DISPLAY_END` - Same as `DISPLAY` but the `state_end` is high so the counter resets.
- `FRONT_PORCH` - The `sync` signal stays high and the `rgb_enabled` drops to low.
- `FRONT_PORCH_END` - Same as `FRONT_PORCH` but the `state_end` is high so the counter resets.

Below is a graph showing all the states of the FSM and how the transitions between them happens.

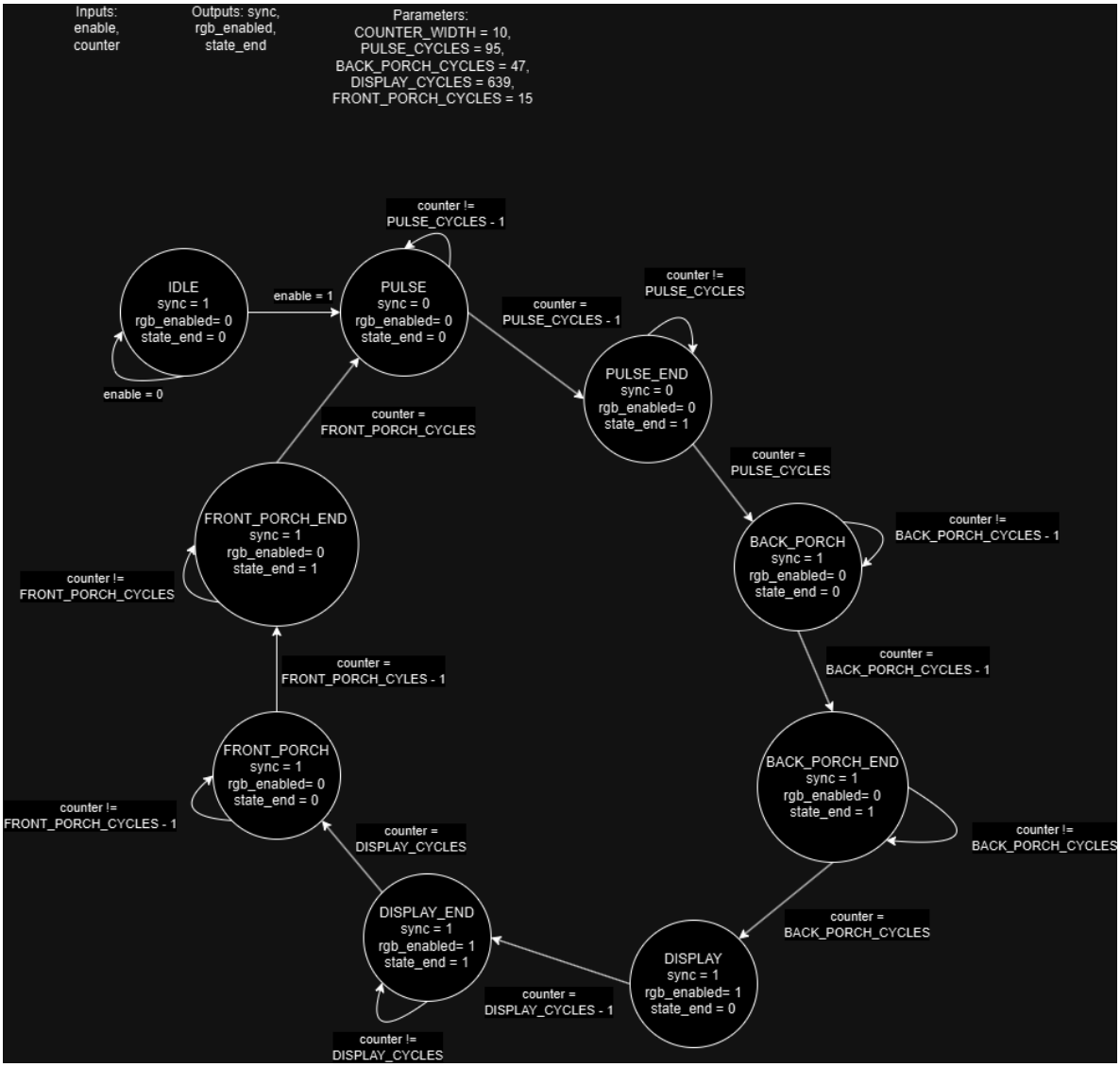


Figure 2: A graph describing the FSM used for synchronizing the VGA outputs

3.1.3 Top-level module

The top level module instantiates the clock divider which is a basic MMCM instantiation using the proper parameters to get a 25MHz clock, debouncers for the enable switch and the reset button, the `pixel_controller` described earlier and then the gsync controller 2 times, one for the HSYNC controller and one for the VSYNC controller. The only difference in the instantiations are the parameters passed to the controllers which describe the waiting times for each state. Below is a diagram displaying the top level module.

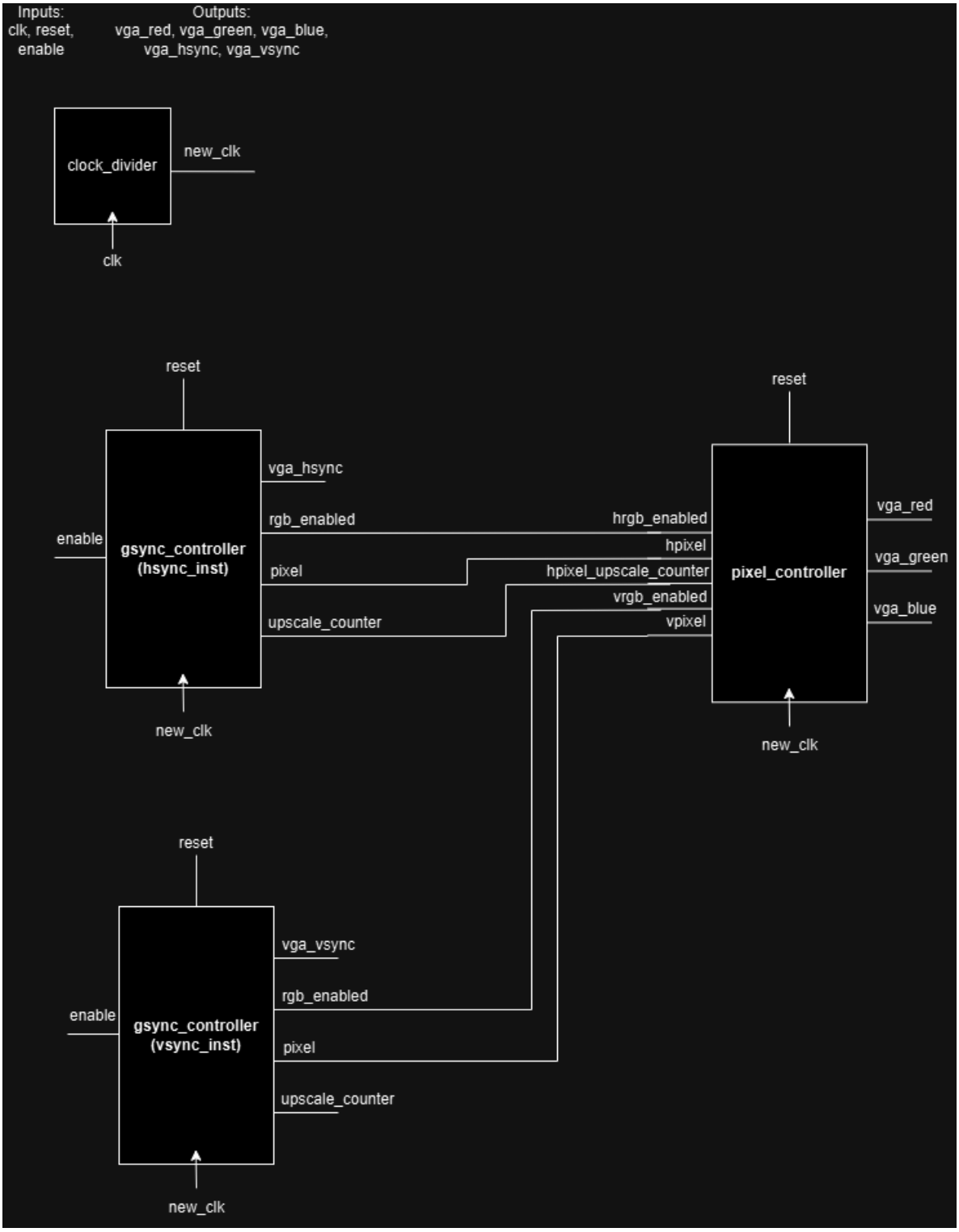


Figure 3: Top level dataflow

3.2 Simulation

For simulating, a testbench was used that performs a reset and enables the controller. By waiting enough time for a whole frame to be displayed we can figure out if everything went well. We can measure the time each state lasts and we can see that the pixel controller is at each given time outputting the correct pixel either it is a blank pixel (`hrgb_enabled` is low or `vrgb_enabled` is low) or

a pixel from the memory. The following image was loaded in the BRAM for testing:

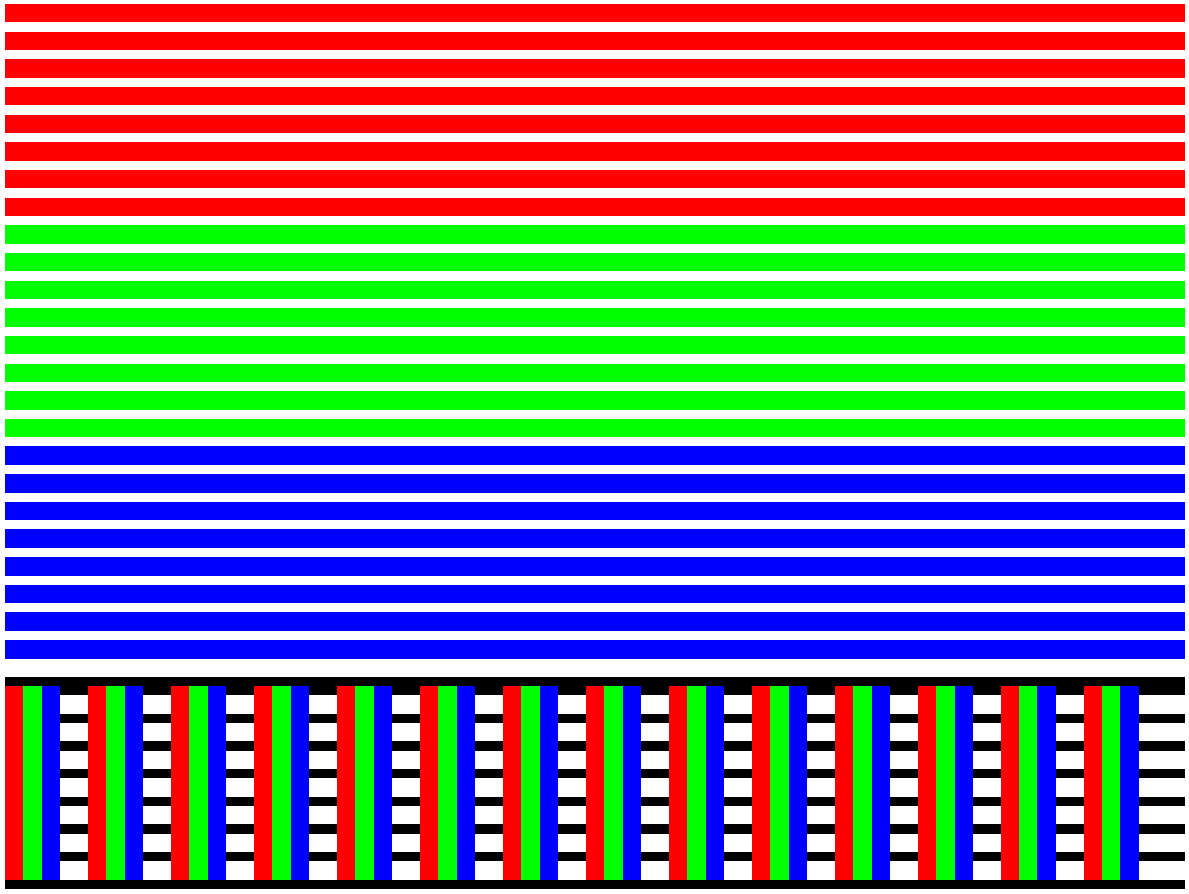


Figure 4: Image used for testing

Below are screenshots displaying the timings of each state and the pixels being displayed.

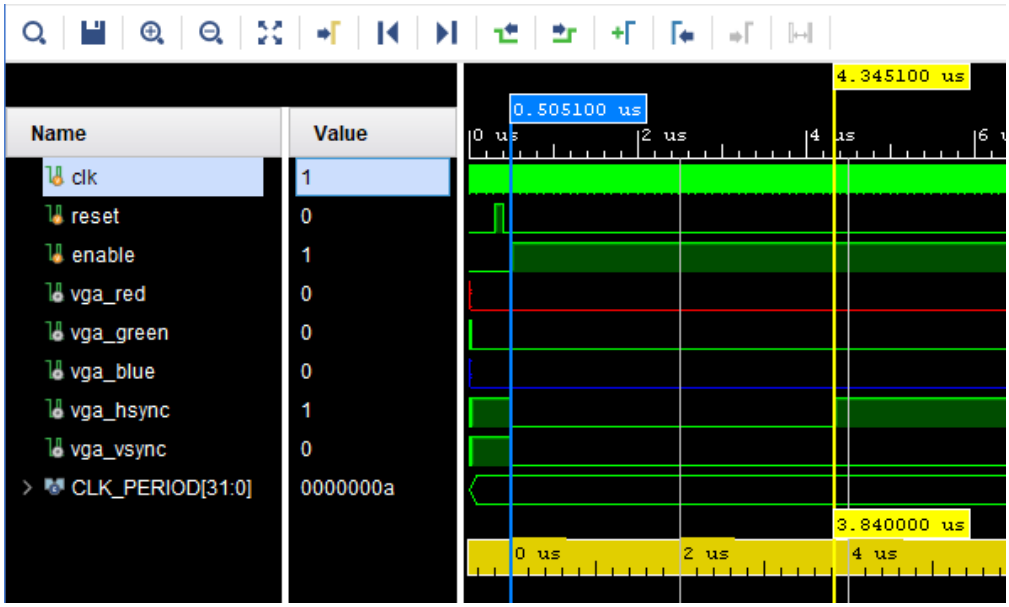


Figure 5: Measurement of the HSYNC Pulse time in simulation

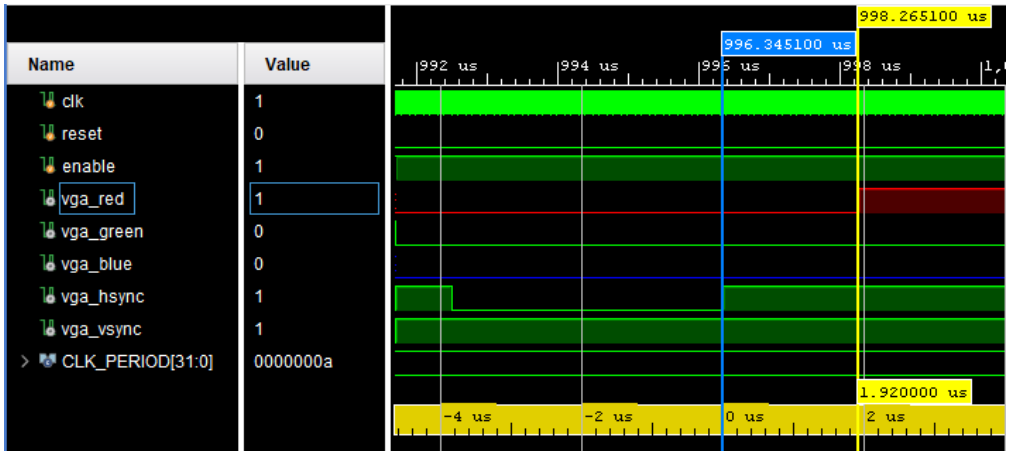


Figure 6: Measurement of the HSYNC back porch time in simulation

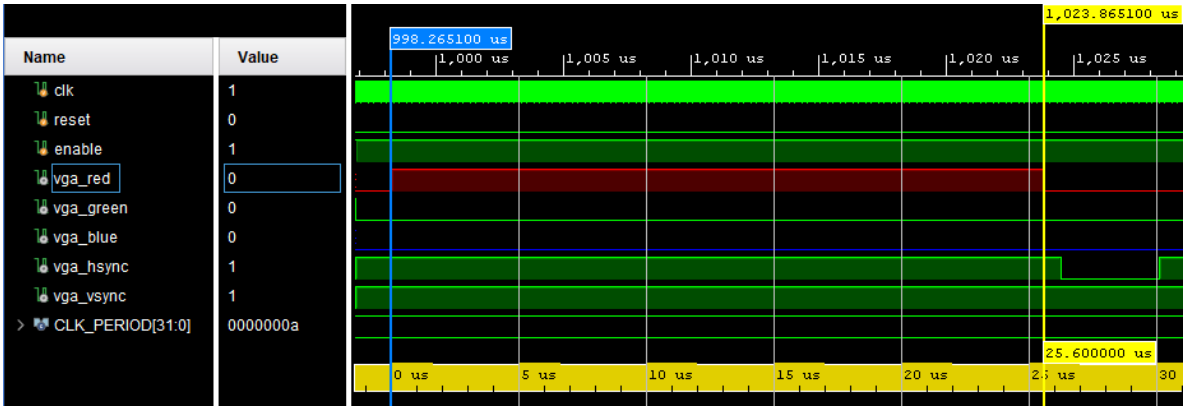


Figure 7: Measurement of the HSYNC display time in simulation

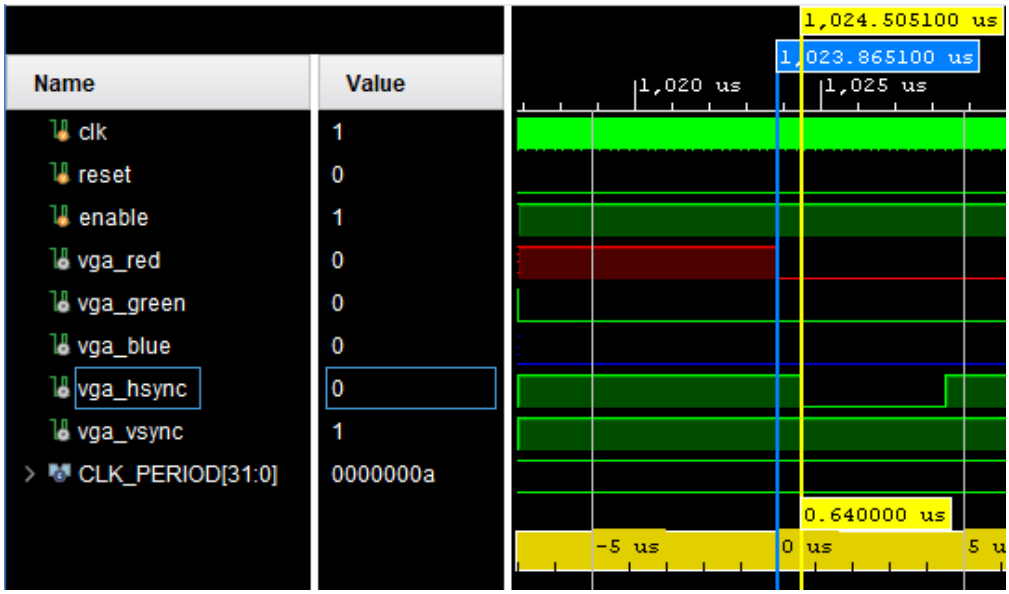


Figure 8: Measurement of the HSYNC front porch in simulation

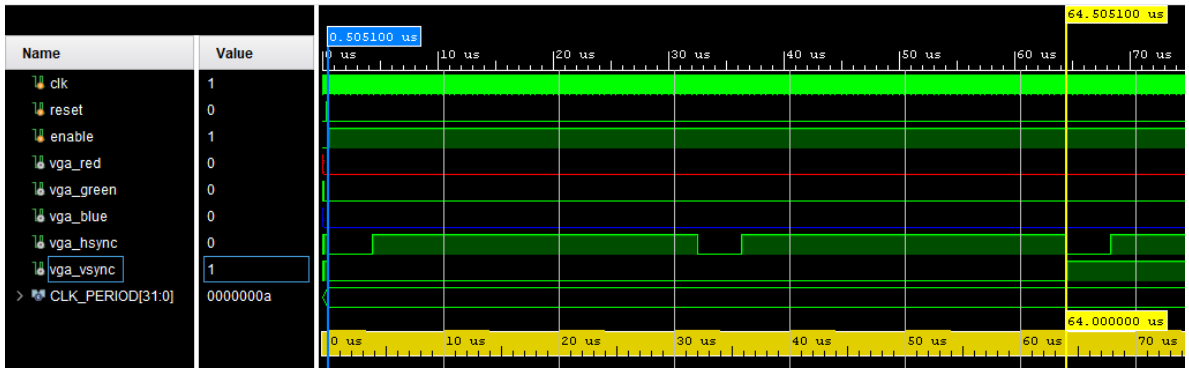


Figure 9: Measurement of the VSYNC pulse width in simulation

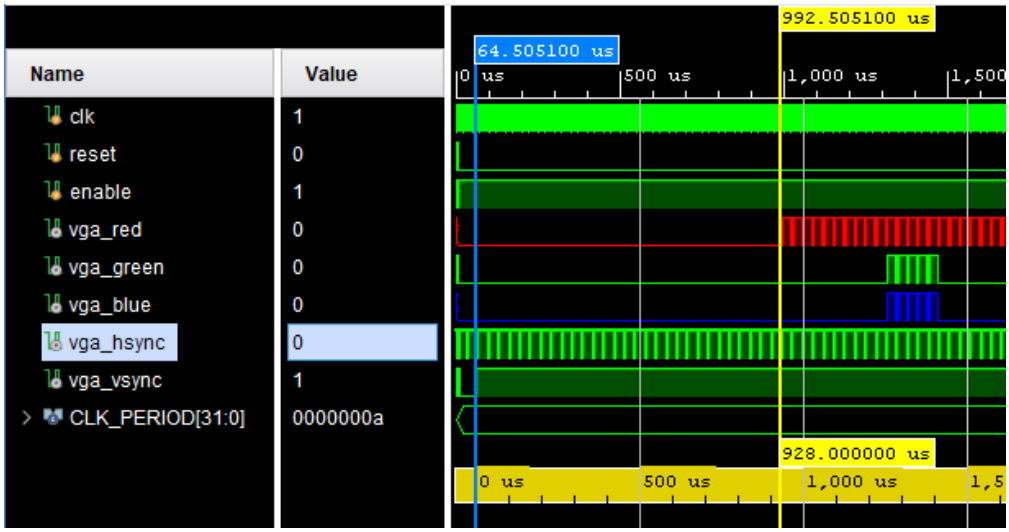


Figure 10: Measurement of the VSYNC back porch in simulation

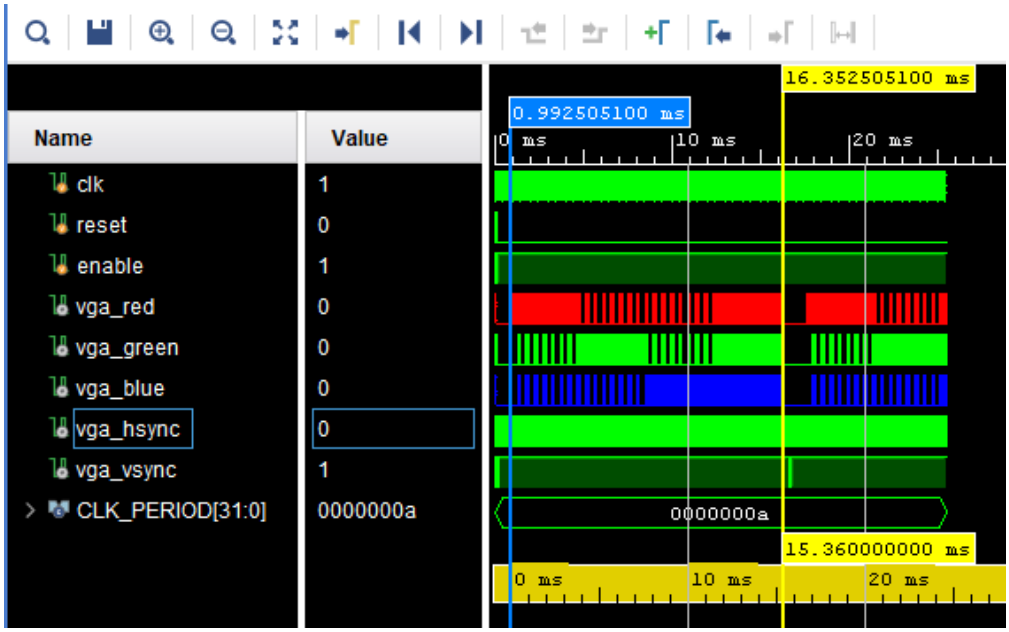


Figure 11: Measurement of the VSYNC display time (video time) in simulation

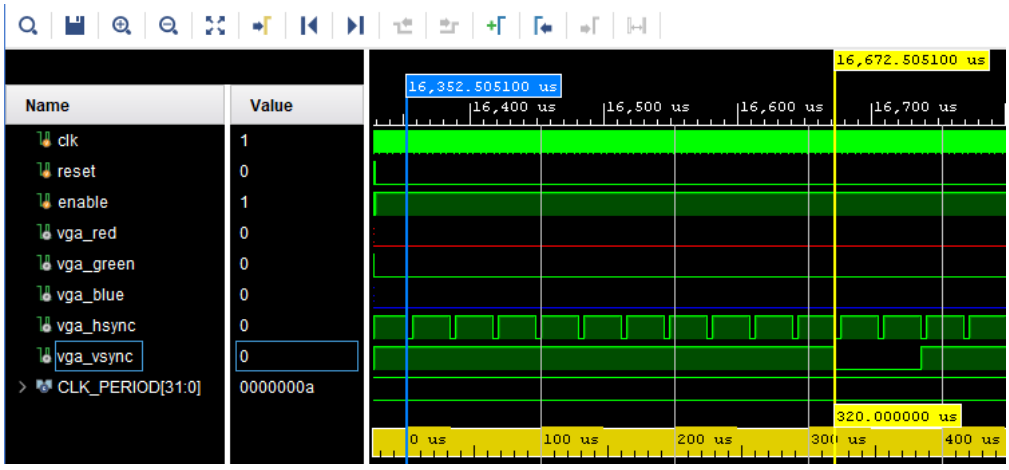


Figure 12: Measurement of the VSYNC front porch in simulation

3.3 Live testing

For live testing on the FPGA the same image as before was used. My only concern before testing it live was the functionality of the `pixel_controller` which implemented the prefetching done in the BRAM. But that was done correctly and the monitor displayed my image. No further fixes were needed since there wasn't any kind of flickering or misplaced pixels. Here is a screenshot of the image being displayed on the monitor:

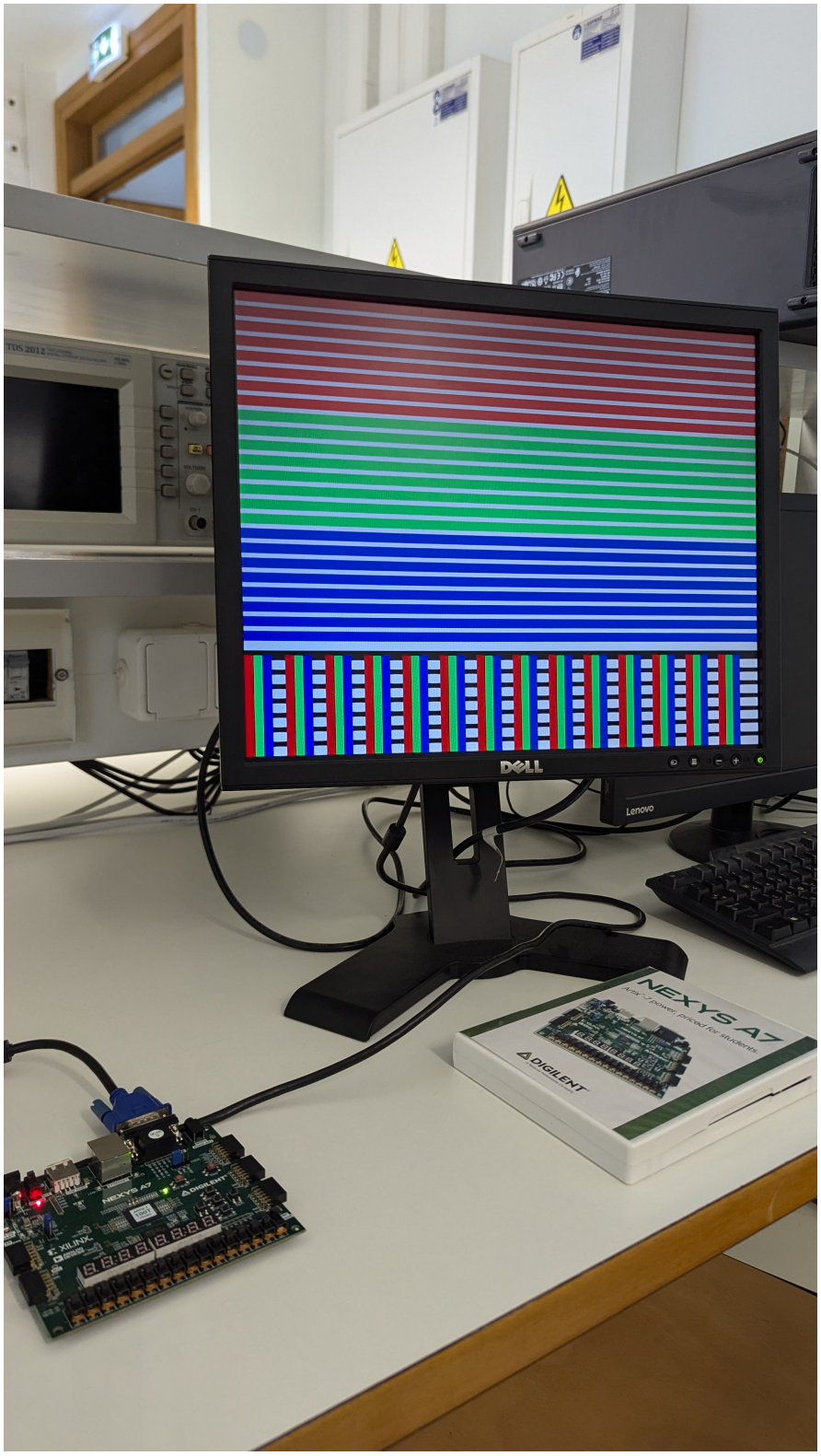


Figure 13: Live testing of the VGA driver

4 Optional feature: Displaying an interactive sprite

The idea behind this was to create the famous DVD screensaver animation that has a DVD logo bouncing around the screen. I quickly realized that this had many hidden difficulties I would soon face.

4.1 Implementation

First of all I needed to make a sprite that would be displayed on the screen and then make it move as time passes (or controlled by the user). For this I used another BRAM that only used a few bytes to store the sprite itself. This would help me "copy" the sprite from its BRAM and "paste" it to the BRAM the VGA driver uses at the correct position. Now the problem was that I needed to find the correct time at which to update the contents of the BRAM so I don't end up displaying half images to the VGA. I found out that between frames there was enough time to rewrite the whole BRAM, since to write one pixel we need one clock cycles and the time we have between frames is at least one VSYNC front porch, a back porch and a pulse width.

4.1.1 Renderer

The renderer functionality was to wait for the VSYNC to send a `frame_end` pulse then proceed to update the BRAM contents based on the new position of the DVD logo. This used the stored position of the sprite given by another module described below and based it on it copied the sprite on the VGA's BRAM to the desired location. It contains a simple FSM using 2 states: `WAIT` and `WRITE` in order to know when to write to the BRAM.

4.1.2 Sprite controller

After figuring out how to render the sprite into the display I needed to find a way to make it move. I decided to create another module that handled the movement of the sprite and output the current position for the rendered to use. I wanted to give it a constant velocity when moving across the screen and also make it collide with the edges of it. The constant velocity was relatively simple, I just had to move the sprite's position a specified amount of pixel diagonally every cycle. The collision detection on the other hand involved some simple maths and doesn't seem to work so well.

4.2 Simulation

For simulation I used the same testbench I have used for the static image and tried to understand if the sprite was moving frame by frame. This was a painful process since I needed to run really long simulations in order to get just 2-3 frames. The results seemed promising and because of limited time I didn't have the chance for more extensive testing.

4.3 Live testing

For live testing on the FPGA I tried both the constant speed sprite and the user-controlled sprite and although the DVD logo was moving, it didn't seem to correctly collide with the edges of the screen and it also wasn't moving at the correct speed. On the user-controlled mode things were a bit better because you could at least see the logo moving but once again the collision detection didn't work correctly. Here is a video demonstrating both the constant speed and user-controller modes of this project