Electrical and Computer Engineering
University of Thessaly (UTH)

ECE333 - Digital Systems Lab

Fall Semester — Educational year 2023-2024

Lab03

Implementation of a Video Graphics Array (VGA) Driver Controller

Georgios Kapakos - AEM: 03165

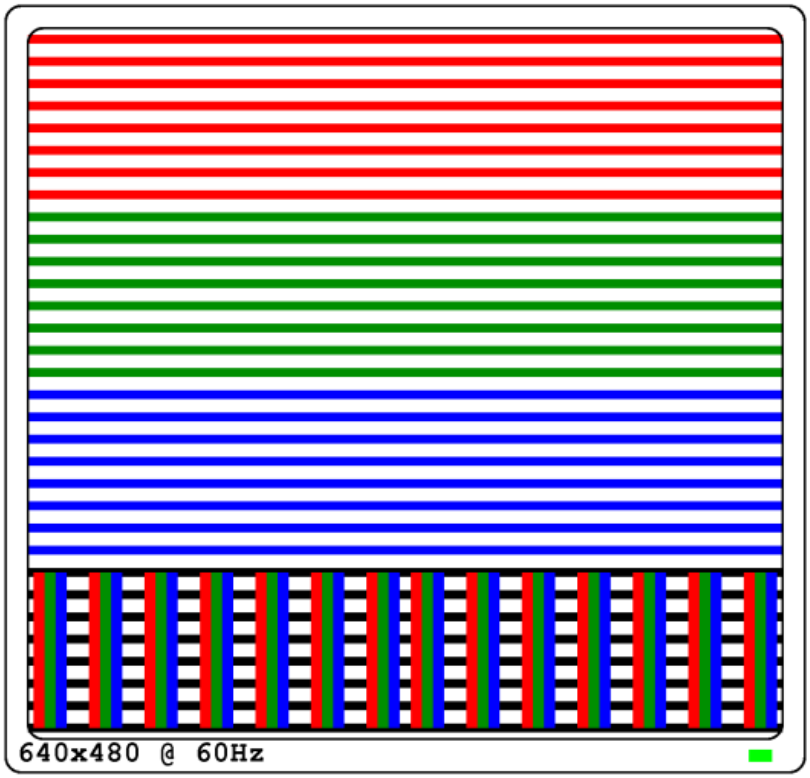
Contents

Abstract

The goal of the 3rd laboratory assignment is the implementation of a controller for the VGA (Video Graphics Array) display port. By programming this port, we achieve the driving of an image onto a screen. This laboratory report describes the method by which we defined the VRAM (Video RAM) to enable continuous display on the screen through persistent VGA driving. We aim to reproduce a pattern similar to that shown in Figure 1. The specific image has the following specifications:

- Resolution: 640 × 480 pixels (picture elements)
- Refresh Rate: 60 Hz

Figure 1: Control Image for VGA Controller Verification



1 Introduction

The objectives of this assignment are the implementation of a VGA port, a port that continuously drives an image onto a screen. The user stores this image in a VRAM unit, which consists of 3 individual BRAMs (Block RAMs) to support 8 colors—one for each primary color (RGB = {Red, Green, Blue}).

The VGA port comprises 15 pins, of which we use 5. We utilize 3 pins for displaying the three primary colors (RGB) and the other 2 to drive the HSYNC and VSYNC signals, which control horizontal and vertical timing, respectively. Although these signals are defined as analog with a pulse height of 0.7V, we treat them as digital.

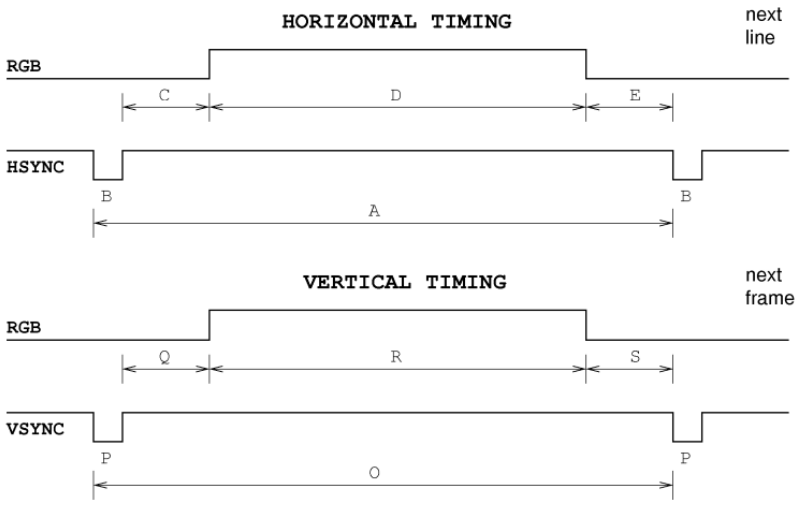
We map the colors according to the table in Figure 2, representing 8 colors based on the values of the Red, Green, and Blue signals:

Figure 2: Representation of Digital Color Mapping

Κόκκινο (Red)	Πράσινο (Green)	Μπλε (Blue)	Συνισταμένη Χρώματος
0	0	0	Μαύρο
0	0	1	Μπλε
0	1	0	Πράσινο
0	1	1	Κυανό
1	0	0	Κόκκινο
1	0	1	Μοβ
1	1	0	Κίτρινο
1	1	1	Άσπρο

In CRT monitors, the two timing signals (HSYNC, VSYNC) control the beam, while in Liquid Crystal Display (LCD) monitors, they manage the writing of new pixels. When the image is active, the values of the Red, Green, and Blue signals determine the output color. As shown in Figure 3, the waveforms for HSYNC and VSYNC are similar. We describe horizontal scanning: after the image activation ends, there is a front porch (E), followed by a pulse (B), whose width and frequency indicate the horizontal resolution to the screen. Then, for a duration (C), the back porch occurs, where the image remains inactive. Finally, the next line appears for a duration (D), where the image is active. Vertical scanning operates similarly based on this description.

Figure 3: Horizontal and Vertical Synchronization of the VGA Driver



The timing of these intervals depends on the resolution (640x480) and refresh rate (60 Hz). Thus, the table in Figure 4 illustrates the timing values for the vectors in Figure 3.

Figure 4: Timing Interval Values of the Waveforms

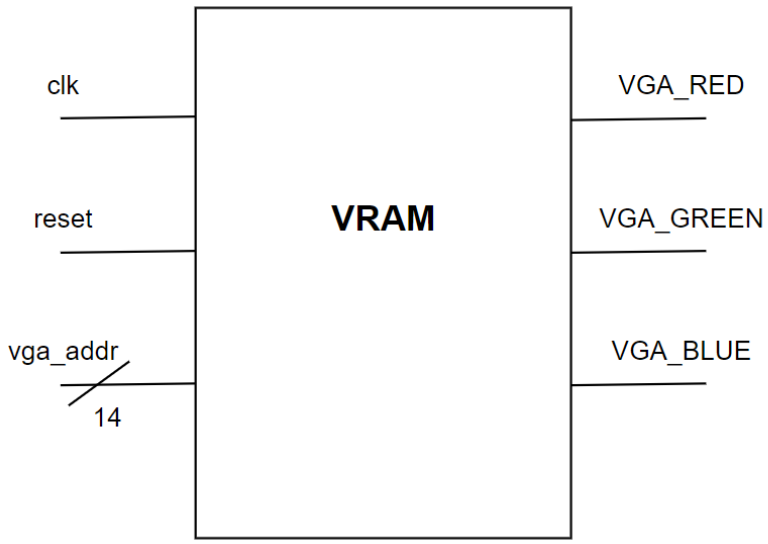
Διάστημα	Περιγραφή	Περιγραφή στα Αγγλικά	Τιμή
A	Χρόνος Σάρωσης Γραμμής	Scanline Time	32 µsec
B	Πλάτος Παλμού HSYNC	HSYNC Pulse Width	3.84 µsec
C	Πίσω Όψη	Back Porch	1.92 µsec
D	Χρόνος Απεικόνισης	Display Time	25.6 µsec
E	Μπροστινή Όψη	Front Porch	0.640 µsec
O	Συνολικός Χρόνος Εικόνας	Total Frame Time	16.67 msec
P	Πλάτος Παλμού VSYNC	VSYNC Pulse Width	64 µsec
Q	Πίσω Όψη	Back Porch	928 µsec
R	Χρόνος Ενεργής Απεικόνισης	Active Video Time	15.36 msec
S	Μπροστινή Όψη	Front Porch	320 µsec

Image activation depends on the HSYNC and VSYNC signals. If these signals are inactive, the image is disabled. Additionally, if either one is inactive, the screen turns off to save power in the VGA system.

The VRAM is a memory continuously displayed on the screen. Its size directly correlates with the screen resolution. For a 640 x 480 pixel resolution, we need $640 \times 480 = 307,200$ bits for monochrome bits. For colored bits, we require $3 \times 307,200 = 921,600$ bits, or approximately 115.2 KB of memory. We use BRAM, an internal memory, to create the VRAM, mapping 3 BRAMs of 16K x 1 into a VRAM of 128 x 96.

2 Part A - Implementation of VRAM

2.1 Implementation



In this part, we implement a VRAM, a central component of the VGA controller. The VRAM accommodates an image of 128 x 96 pixels with 8 colors. It consists of 3 individual BRAMs, each initialized separately for the primary colors Red, Green, and Blue.

We obtain the BRAMs from Vivado, following the instructions, and initialize them appropriately. When a color appears on the screen, the bit representing that color at that point must be active. For example, to display red, the red BRAM's bit should be 1, while the green and

blue BRAMs' bits should be 0 at that position. We initialize the BRAMs via the `.INIT_xx` parameters, as described above. We define the color combinations from the end to the beginning of each `.INIT_xx` line, as this reverse order ensures correct display on the screen.

We describe the verification image pattern using 48 lines, each with 256 bits. Thus, we define each image line up to the middle of each `.INIT_xx` row. After the 48th line, the remaining `.INIT_xx` values are irrelevant and set to zero, since our array is $128 \times 96 = 12,288$ bits, requiring $48 \times 256 = 12,288$ bits.

For the VRAM implementation, we created the following module:

It consists of the inputs:

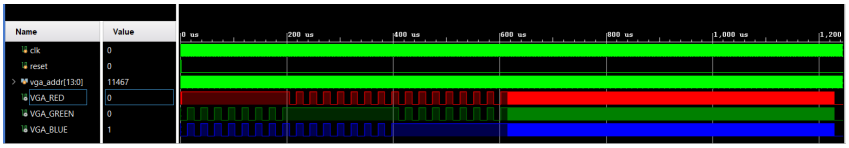
- **reset**: Asynchronous reset of our circuit.
- **clk**: Clock with a frequency of $f_{clk} = 100\,\text{MHz}$.
- **vga_addr**: The address of the element within the VRAM, stored as a combination of the color addresses of the 3 BRAMs.

It consists of the outputs:

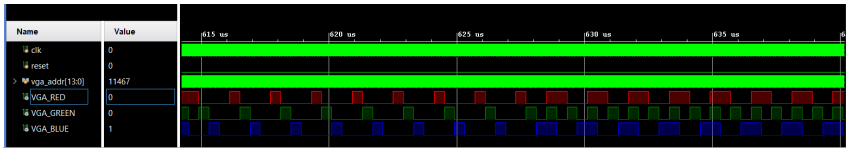
- **VGA_RED**: The value of the element regarding the red color.
- **VGA_GREEN**: The value of the element regarding the green color.
- **VGA_BLUE**: The value of the element regarding the blue color.

2.2 Verification

For verification, we created a testbench to examine the output signals `VGA_RED`, `VGA_GREEN`, and `VGA_BLUE` for different `vga_addr` values.



In the above figure, we observe that the colors form a pattern as close as possible to the image we aim to describe. We start with line-by-line alternations of red and white, followed by green and white, then blue and white. Finally, we have column-wise alternations of red, green, blue, and black, with line-by-line alternations of red, green, blue, and white.



The second part of our description, shown in the figure above, begins with alternations of red-green-blue-black, followed by red-green-blue-white.

3 Part B - Implementation of HSYNC and Horizontal Pixel Counter

3.1 Implementation

In this part, we implement the HSYNC signal and the horizontal pixel counter (HPIXEL) for a 640 x 480 resolution and 60 Hz refresh rate. Alongside HSYNC, we use the HPIXEL counter to appropriately select pixels stored in the VRAM, from address 0 (first pixel) to address 12,268 ($128 \times 96 = 12,268$), where 128 x 96 represents the BRAM dimensions composing the VRAM.

For the horizontal counter, we calculated clock cycles for the respective times and states, as shown in the table below:

State	Symbol	Parameter	Time	Clocks
Sum of all states	T_s	Sync pulse	32 µs	800
State D	T_{disp}	Display time	25.6 µs	640
State B	T_{pw}	Pulse width	3.84 µs	96
State E + State A	T_{fp}	Front porch	640 ns	16
State C	T_{bp}	Back porch	1.92 µs	48

Figure 5: Signal Timings for 640-Pixel by 480-Row Screen Resolution with 25 MHz Pixel Clock and 60 Hz Refresh Rate for the Horizontal Counter

For the Hcounter (horizontal pixel counter) implementation, we created the following module:

It consists of the inputs:

- **reset**: Asynchronous reset signal of the circuit.
- **clk**: Clock of our circuit.

It consists of the outputs:

- **VGA_HSYNC**: Horizontal synchronization signal for the VGA port.
- **HPIXEL**: Address of horizontal bits (6 bits), later used to compose `vga_addr`.

- **hcount**: 10-bit counter tracking the time intervals from the table above, aiding state transitions based on its value.
- **HRGB_EN**: Signal indicating when RGB can take values according to what we want to transmit.

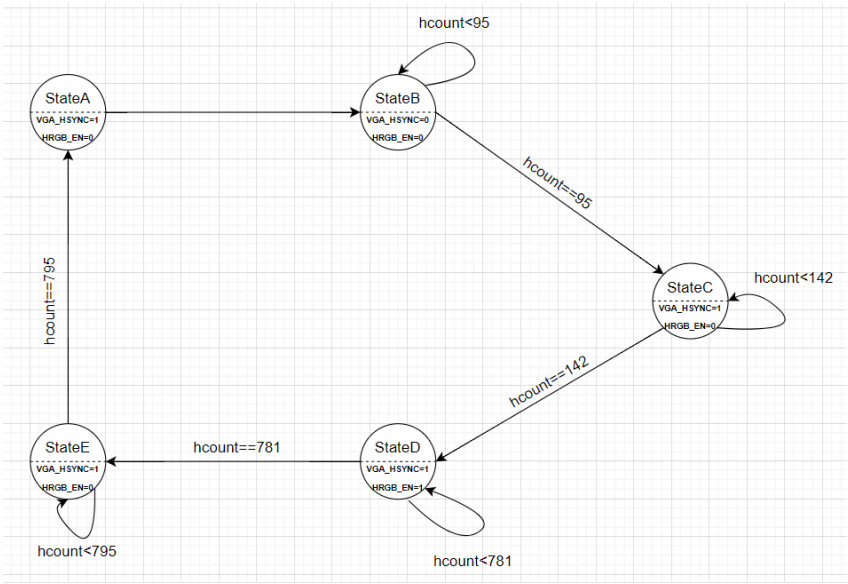
We implement an FSM within the module to perform the appropriate actions for signal transmission:

1. **stateA (0)**: Counts one cycle, initializes `hcount` to 0, sets `HRGB_EN = 0` and `VGA_HSYNC = 1`, and transitions directly to the next state.
2. **stateB (1)**: `hcount` counts up to 96 cycles, then transitions to the next state. `VGA_HSYNC` becomes 0, while `HRGB_EN` remains 0.
3. **stateC (2)**: `hcount` counts another 48 cycles, then transitions. `VGA_HSYNC` becomes 1, `HRGB_EN` remains 0.
4. **stateD (3)**: Counts 640 cycles for `hcount`. Every 5 bits, we increment `hcount_ins` using a `flag` variable set to 4, then reset and repeated until the state ends. `hcount_ins` drives `HPIXEL`, updating every 5 bits due to the total count (640/5 = 128, matching VRAM row size). `VGA_HSYNC` remains 0, `HRGB_EN` becomes 1.
5. **stateE (4)**: Counts 15 cycles with `hcount`, returns to the initial state, setting `HRGB_EN = 0` and `VGA_HSYNC = 1`.

For the FSM implementation, we use two always blocks:

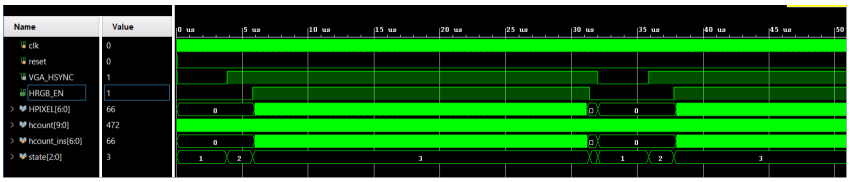
- A sequential always block assigns values to the current state by equating it to the next state. Each variable change updates the current state; otherwise, reset initializes all values to 0.
- A combinational always block modifies the next states.

Below, we present the FSM model schematically as a Moore model:



3.2 Verification

In the testbench, we verify if the states and timings align with the table above (Figure 5):

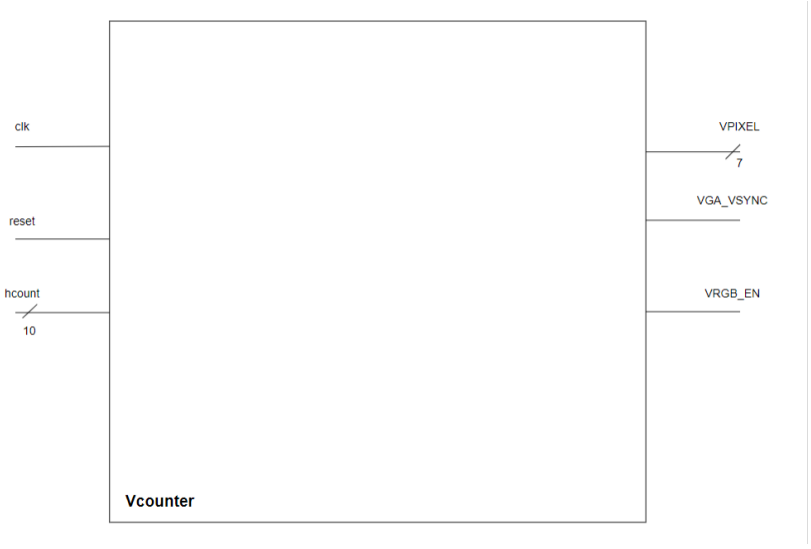


The timings appear correct, and the `HSYNC` and `HRGB_EN` signals activate at the appropriate moments.

4 Part C - Implementation of VSYNC and Vertical Pixel Counter - Completion of the VGA Controller/Driver

4.1 Implementation

4.1.1 Implementation of VSYNC and Vertical Pixel Counter



In this part, we implement the VSYNC signal and vertical pixel counter (VPIXEL) for a 640 x 480 resolution and 60 Hz refresh rate. Alongside VSYNC, we use the VPIXEL counter to select pixels stored in the VRAM, from address 0 to 12,268 ($128 \times 96 = 12,268$), matching the BRAM dimensions. `VPIXEL` increments each time a row changes, ranging from 0 to 95 (96 rows total), traversing the VRAM's data vertically in combination with the horizontal counter to form `vga_addr`.

For the vertical counter, we calculated clock cycles for the respective times and states, as shown below:

State	Symbol	Parameter	Time	Clocks	Lines
Sum of all states	T_s	Sync pulse	16.7 ms	416,800	521
State R	T_{disp}	Display time	15.36 ms	384,000	480
State P	T_{pw}	Pulse width	64 µs	1,600	2
State S + State O	T_{fp}	Front porch	320 µs	8,000	10
State Q	T_{bp}	Back porch	928 µs	23,200	29

Figure 6: Signal Timings for 640-Pixel by 480-Row Screen Resolution with 25 MHz Pixel Clock and 60 Hz Refresh Rate for the Vertical Counter

For the Vcounter (vertical pixel counter) implementation, we created the following module: It consists of the inputs:

- **reset**: Asynchronous reset signal of the circuit.
- **clk**: Clock of our circuit.
- **hcount**: 10-bit counter tracking the entire Hcounter pulse duration, used as a control signal to increment `vcount`, the counter used across all Vcounter states.

It consists of the outputs:

- **VGA_VSYNC**: Vertical synchronization signal for the VGA port.
- **VPIXEL**: Address of vertical bits (6 bits), later used to compose `vga_addr`.
- **VRGB_EN**: Signal indicating when RGB can take values based on VRAM data.

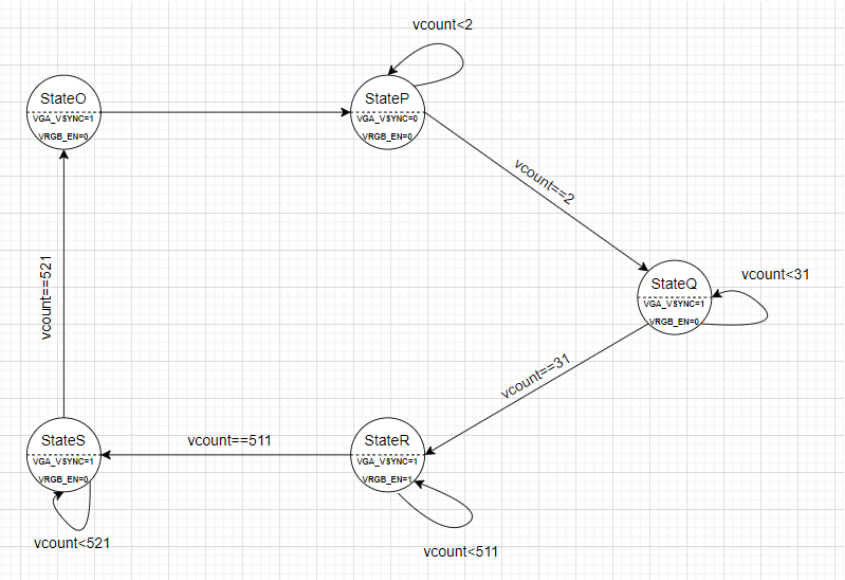
We implement an FSM within the module to perform the appropriate actions for signal transmission:

1. **stateO (0)**: Counts one cycle, initializes `vcount` to 0, sets `VRGB_EN = 0` and `VGA_VSYNC = 1`, and transitions directly to the next state.
2. **stateP (1)**: `vcount` counts 2 lines, then transitions. `VGA_VSYNC` becomes 0, `VRGB_EN` remains 0.
3. **stateQ (2)**: `vcount` counts 29 lines, then transitions. `VGA_VSYNC` becomes 1, `VRGB_EN` remains 0.
4. **stateR (3)**: Counts 480 lines for `vcount`. Every 5 bits, `vcount_ins` increments using a `flag` variable set to 4, then resets and repeats until the state ends. `vcount_ins` drives `VPIXEL`, updating every 5 bits (480/5 = 96, matching VRAM column size). `VGA_VSYNC` remains 0, `VRGB_EN` becomes 1.
5. **stateS (4)**: Counts 10 lines minus one cycle with `vcount`, returns to the initial state, setting `VRGB_EN = 0` and `VGA_VSYNC = 1`.

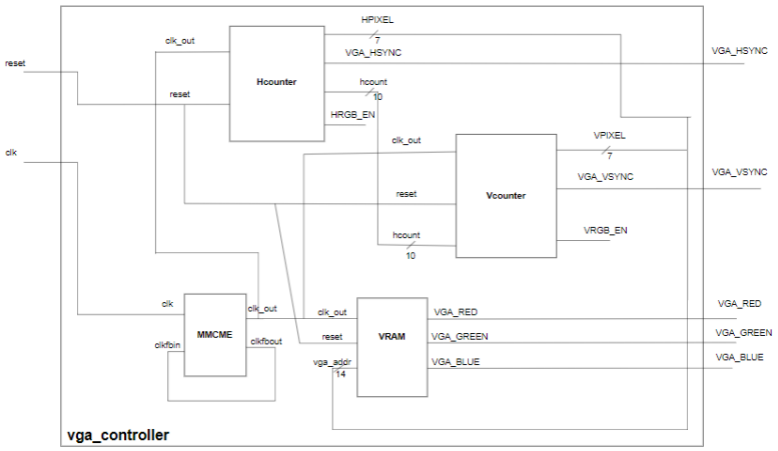
For the FSM implementation, we use two always blocks:

- A sequential always block assigns values to the current state by equating it to the next state. Each variable change updates the current state; otherwise, reset initializes all values to 0.
- A combinational always block modifies the next states.

Below, we present the FSM model schematically as a Moore model:



4.1.2 Implementation of VGA Controller/Driver (vga_controller)



We connect the modules described above to create the VGA driver. Additionally, we implemented an MMCME unit to convert the fast clock to a slow one, aligning the timing with the clock periods in our tables.

We implemented an MMCME unit, setting `CLFBOUT_MULT_F` to 12, `CLKIN1_PERIOD` to 10, `CLKOUT1_DIVIDE` to 48, and `DIVCLK_DIVIDE` to 1. According to the manual's formula: $F_{out} = F_{clkin} \times M/(D \times O)$, where:

- F_{out}: New frequency with a slow clock (40 ns period, $F_{out} = 25\,\text{MHz}$).
- F_{clkin}: Fast clock frequency (10 ns period, $F_{clkin} = 100\,\text{MHz}$).
- M: `CLFBOUT_MULT_F`, multiplying all `CLKOUT` clocks to the desired frequency.
- D: `DIVCLK_DIVIDE`, dividing `CLKIN` effectively with `CLFBOUT_MULT_F`.

- *O*: `CLKOUT1_DIVIDE`, dividing only this specific clock.
- `CLKIN1_PERIOD`: Input clock period (10 ns).

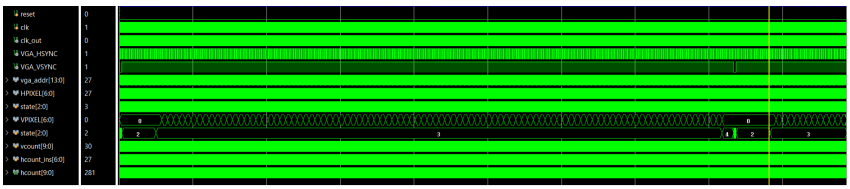
We also used a constraints file to connect our Verilog variables to the board's pins:

```
## This file is a general .xdc for the Nexys A7-100T
### Clock Signal
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name clk -period 10.00 -waveform {0 5} [get_ports { clk }];
### RGB Display
##RED
set_property -dict { PACKAGE_PIN A3 IOSTANDARD LVCMOS33 } [get_ports { VGA_RED }];
set_property -dict { PACKAGE_PIN B4 IOSTANDARD LVCMOS33 } [get_ports { VGA_RED }];
set_property -dict { PACKAGE_PIN C5 IOSTANDARD LVCMOS33 } [get_ports { VGA_RED }];
set_property -dict { PACKAGE_PIN A4 IOSTANDARD LVCMOS33 } [get_ports { VGA_RED }];

##GREEN
set_property -dict { PACKAGE_PIN C6 IOSTANDARD LVCMOS33 } [get_ports { VGA_GREEN }];
set_property -dict { PACKAGE_PIN A5 IOSTANDARD LVCMOS33 } [get_ports { VGA_GREEN }];
set_property -dict { PACKAGE_PIN B6 IOSTANDARD LVCMOS33 } [get_ports { VGA_GREEN }];
set_property -dict { PACKAGE_PIN A6 IOSTANDARD LVCMOS33 } [get_ports { VGA_GREEN }];

##BLUE
set_property -dict { PACKAGE_PIN B7 IOSTANDARD LVCMOS33 } [get_ports { VGA_BLUE }];
set_property -dict { PACKAGE_PIN C7 IOSTANDARD LVCMOS33 } [get_ports { VGA_BLUE }];
set_property -dict { PACKAGE_PIN D7 IOSTANDARD LVCMOS33 } [get_ports { VGA_BLUE }];
set_property -dict { PACKAGE_PIN D8 IOSTANDARD LVCMOS33 } [get_ports { VGA_BLUE }];

set_property -dict { PACKAGE_PIN B11 IOSTANDARD LVCMOS33 } [get_ports { VGA_HSYNC }];
set_property -dict { PACKAGE_PIN B12 IOSTANDARD LVCMOS33 } [get_ports { VGA_VSYNC }];
### Button(s)
set_property -dict { PACKAGE_PIN N17 IOSTANDARD LVCMOS33 } [get_ports { reset }];
```

4.2 Verification

4.2.1 Verification of VSYNC and Vertical Pixel Counter

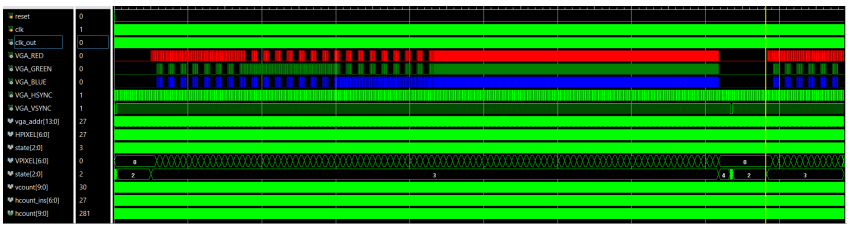
In the testbench, we verify if the states and timings align with the table above (Figure 6):



The timings appear correct, and the `VSYNC` and `VRGB_EN` signals activate at the appropriate moments. The `VPIXEL` counter takes correct values from 0 to 95 (96 cycles). The signals synchronize successfully, and the states described in the table are reflected in the waveforms.

4.2.2 Verification of VGA Controller/Driver (vga_controller)

In the testbench, we check if the colors (RGB) activate in the correct state and display in the order defined by the VRAM:



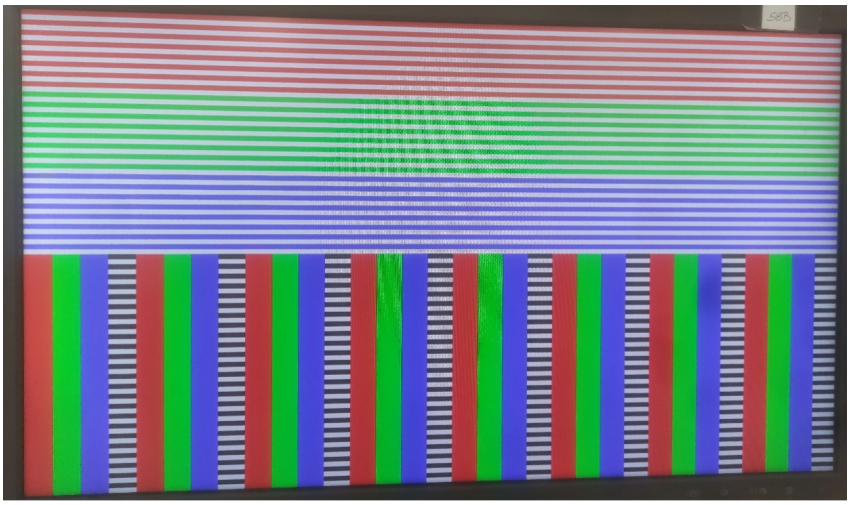
The colors display during the correct state activations and in the desired order, indicating that the color addresses have the expected values. The image also shows that when `HRGB_EN`

is zero, colors deactivate with dashed lines in some areas, and when `VRGB_EN` is zero, all color values are zero.

4.3 Experiment/Final Implementation

After generating the bitstream, we applied it to the board, connecting the VGA cable to the screen. In the first attempt, the pattern was correct, but the colors were incorrect. After setting the colors to zero when `HRGB_EN` and `VRGB_EN` are zero, the second attempt worked, displaying the correct order and colors on the screen.

We present the waveforms generated on the screen after implementing the circuit on the board in the second, correct implementation:



Additionally, we observe from the screens that the top half of the image alternates between white and red/green/blue lines, while the bottom half has columns of red/green/blue with line alternations between black and white, exactly matching the pattern stored in the VRAM.