Third lab report

Athanasiadis Lazaros

Student No.: 2488

03/12/19

Abstract

The third lab assignment's aim was the development of a VGA (Video Graphics Array) driver capable of displaying an 128×96 pixels picture, scaled to 640×480 pixels. Three digital signals are used for red, green and blue, resulting in 8 possible colors. In order to store the displayed image, I used three $16KB \times 1$ block RAM modules , with each module storing the data of one color's bits. After my driver could display a static image successfully, I attempted to display an animation.

Introduction

The displayed colors

The VGA standard defines the signals for red, green and blue as analog signals, with 0.7V peak-to-peak voltage. Since using a digital to analog converter was beyond the scope of this assignment, I used digital signals for each color, resulting in 8 possible colors, as seen in the table below.

Red	Green	Blue	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

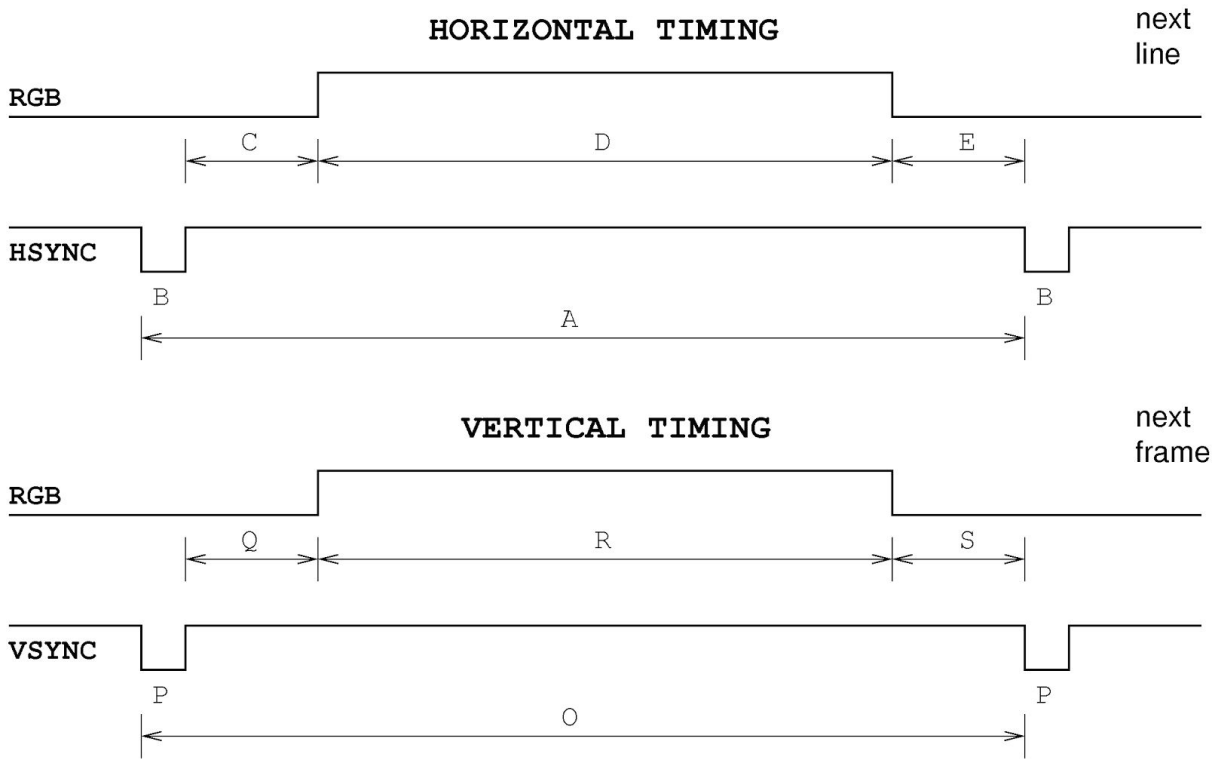
Image scaling

While the original VGA standard was addressed at 640×480 monitors, the image that I'll display has ⅕ of the expected height and width. This was done in order to avoid having to use more than three block RAMs to store the image. For a 640×480 image, $640 \cdot 480 = 307200B = 300KB$ of memory space is needed for each color, while for a 128×96 image, only $128 \cdot 96 = 12288B = 12KB$ are needed, which means one block ram module of 16KB × 1 will suffice.

The displayed image is scaled by showing each bit more than one time (25 times, to be exact; each pixel in the original image is replaced by a 5×5 square), essentially mapping the smaller image to the bigger one.

Signal Timings

In order to display the correct pixel at each given time, the synchronization signals HSYNC and VSYNC, alongside the three signals for the colors, must be driven according to the schematic below:



Span	Name/Description	Duration for which the signal is active
A	Scanlight Time	32μs
B	HSYNC Pulse Width	3.84μs
C	HSYNC Back Porch	1.92μs
D	Display Time	25.6μs
E	HSYNC Front Porch	0.64μs
O	Total Frame Time	16.67ms
P	VSYNC Pulse Width	64μs
Q	VSYNC Back Porch	928μs
R	Active Video Time	15.36ms
S	VSYNC Front Porch	320μs

During each *display time,* the pixels of one row are shown—the VGA color signals (RGB) change only when *display time* and *active video time* overlap. *HSYNC Front Porch, HSYNC Pulse Width* and *HSYNC Back Porch* follow, in which the RGB signals are held low; after these time spans, the pixels of the next row are shown and so on. When this happens 480 times, $480 \cdot \{scanlight\ time\} = 480 \cdot 32\mu s = 15.36ms$ have passed, which equals to the span of an *active video time*. After *VSYNC Front Porch, VSYNC Pulse Width* and *VSYNC Back Porch* have passed, the pixels of the first row will start being displayed again.

Part A—Video RAM

In order to store the static image that will get displayed, I used three 16KB $\times$ 1 block ram modules, each storing the data for one color. The image's dimensions are 128×96, which means that out of the 16KB available, only $128 \cdot 96 = 12288B = 12KB$ are being used. The contents of the memory modules can be initialized with the INIT parameters, so there was no need to write to the memory.

Testbench

I tested the video RAM by filling each memory module as seen in the table below, and checking the outputs for each address range.

	Red Memory	Green Memory	Blue Memory
Addresses 0 to 4047	FFFF...FFFF	0000...0000	0000...0000
Addresses 4048 to 8191	0000...0000	FFFF...FFFF	0000...0000
Addresses 8192 to 12287	0000...0000	0000...0000	FFFF...FFFF

Part B—Horizontal Timing

Cycling through *Pulse Width*, *Back Porch*, *Display Time* and *Front Porch* is accomplished by using a finite state machine (FSM) with 4 possible states. Each state has its own counter, in order to keep track of the time elapsed at each state. Clock cycles are calculated with a 50MHz clock in mind.

State	Time Span	Clock Cycles
Pulse Width	3.84us	192
Back Porch	1.92us	96
Display Time	25.6us	1280
Front Porch	0.64us	32

At each state, the counter of the previous state is reverted to zero. If at any time the reset signal becomes high, all the counters are reverted to zero and the state reverts back to the first one (pulse width). During the *display time* state, the *enable* output is held high, to signal that the VGA color signals should stop being held at low.

The hpixel counter

The hpixel counter is an output that stores the column of the currently displayed pixel. During each *display time (DT)*, the pixels of one row are displayed. For images with a width of 640 pixels, this means that each pixel's colors are displayed for $1280\ (DT)/640\ =\ 2$ clock cycles. My image has a width of 128, so each pixel's colors must be displayed for 5 times the previous amount. Therefore, *hpixel* gets incremented every 10 clock cycles.

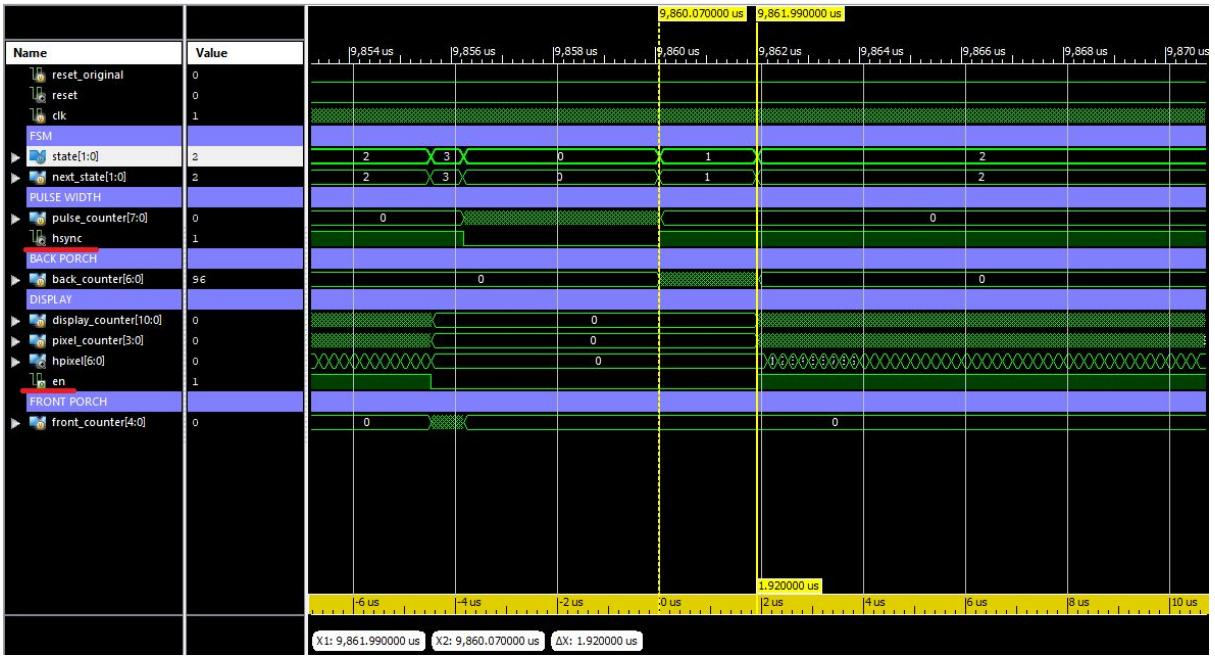
A problem I encountered with the hpixel counter was that it was originally synthesized as a latch. The reason was that I had modelled it as a FSM output, which was a combinational circuit and its value changed in certain occasions, which meant that it latched for the rest of them. I solved this by moving the logic for calculating hpixel in a separate, sequential always block.

Testbench

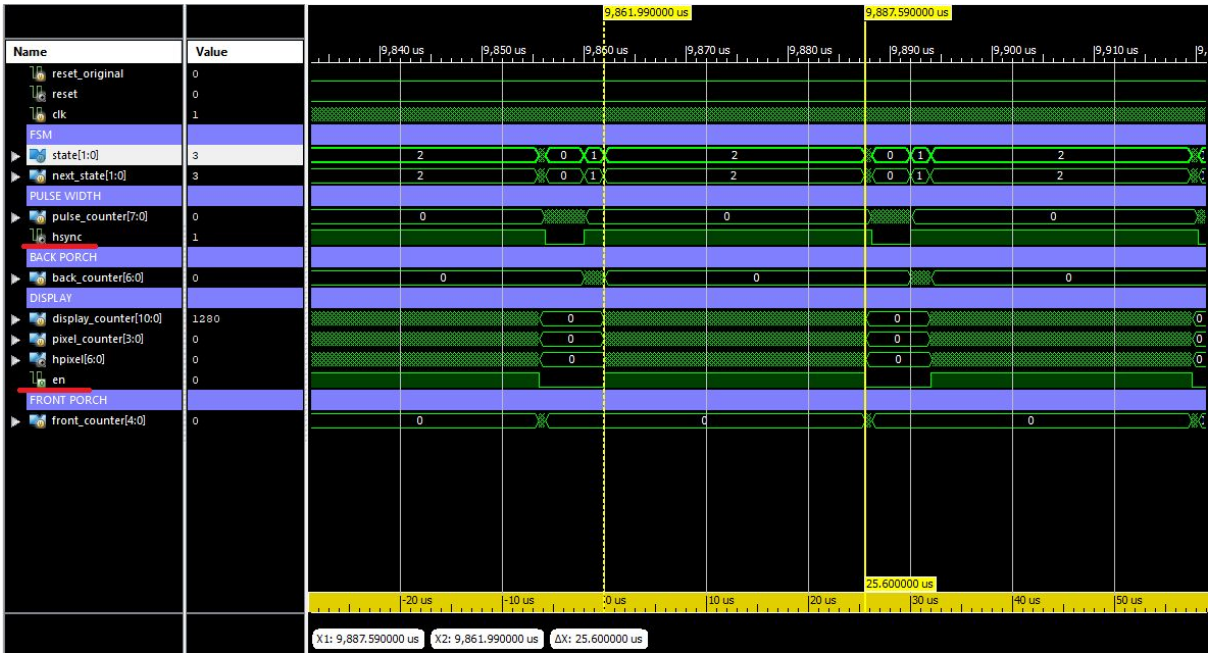
Testing the horizontal timings was straightforward since I only had to check if the *HSYNC* and *enable* outputs were high/low for the specified amount of time.

1. Pulse Width state: 3.84us duration, both hsync and enable (en) are low

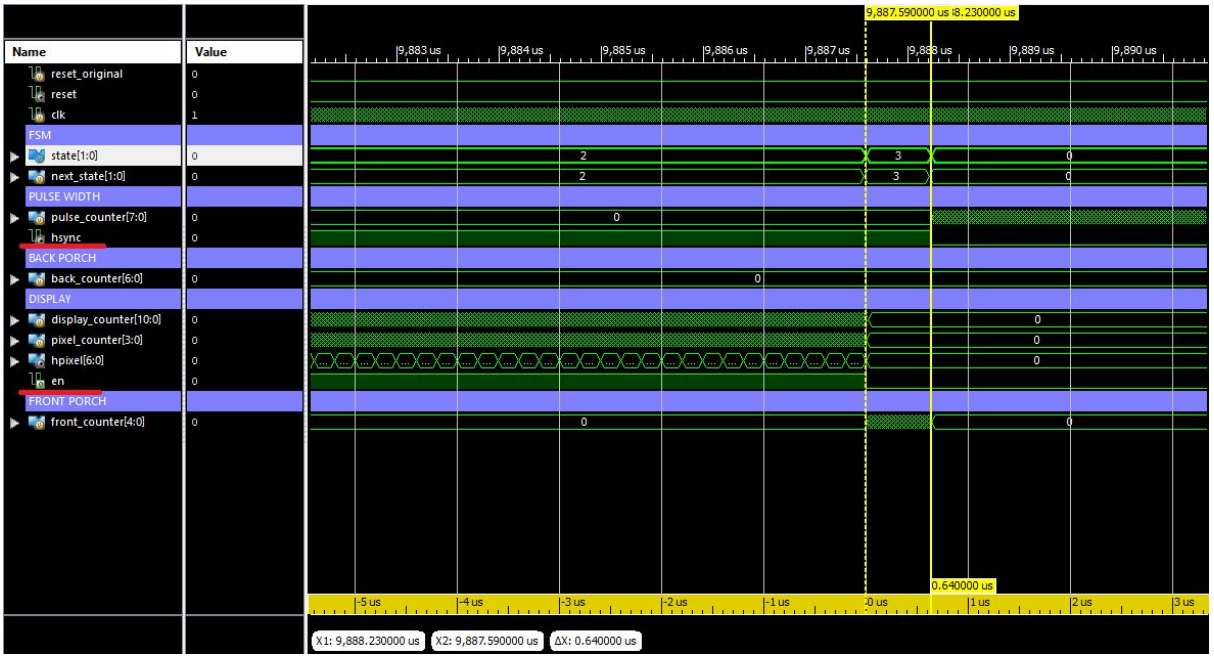
2. Back Porch state: 1.92us duration, hsync is high but enable is low



3. Display Time state: 25.6us duration, both hsync and enable are high



4. Front Porch state: 0.64us duration, hsync is high but enable is low



Part C—Vertical Timing

Driving the vertical timing signals is accomplished in an identical way. Four counters in four FSM states calculate elapsed time and drive the signals *VSYNC* and *enable* accordingly. Clock cycles are calculated with a T=20ns clock in mind.

State	Time Span	Clock Cycles
Pulse Width	64us	3200
Back Porch	928us	46400
Active Video Time	15.36ms	768000
Front Porch	320us	16000

The respective counters behave like the previous ones; at reset the are all reverted back to zero, and at each state the previous state's counter is reverted to zero. During *active video time*, the *enable* output is held high, to signal that the VGA color signals should stop being held at low.

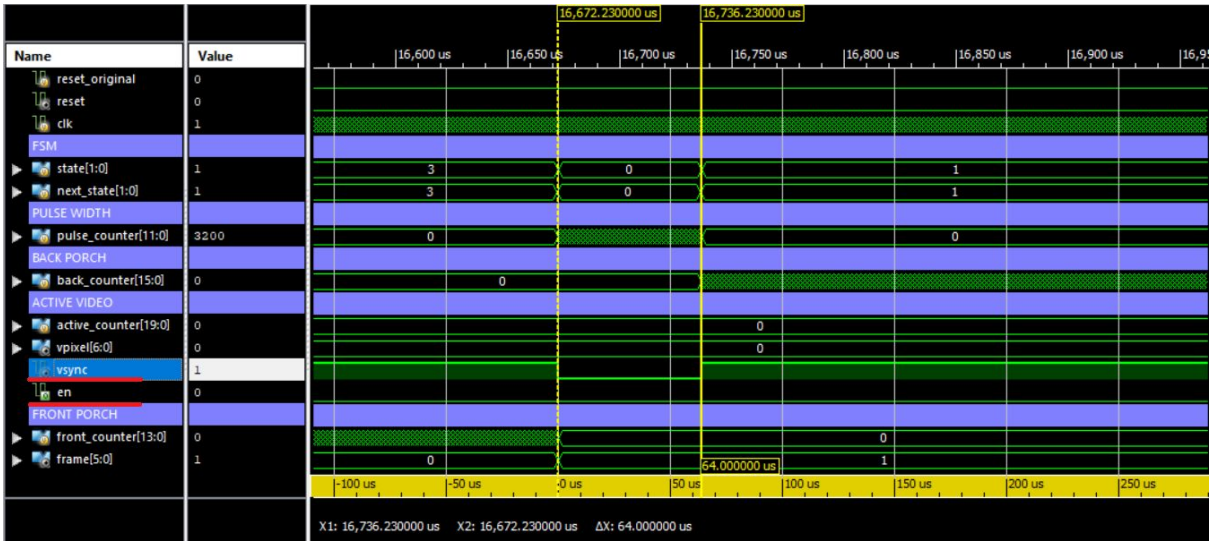
The vpixel counter

The vpixel counter is an output similar to hpixel; it stores the current pixel's row. The 15.36ms of active video time are enough to scan 480 vertical lines one time, since $480 \cdot \{scanlight\ time\} = 480 \cdot 32us = 15360us = 15.36ms$. My source image only has ⅕ of those lines, so instead of changing vpixel's value once per scanlight time span (32us), I change it once for every 5 scanlight time spans (160us).

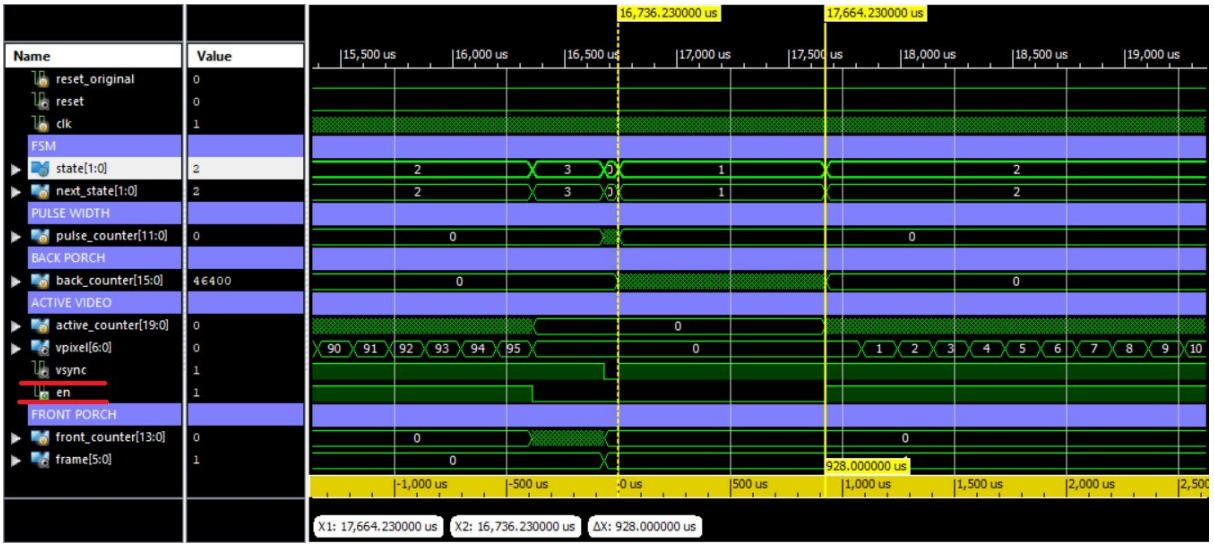
Testbench

The testbench for the vertical timings is identical to the one for the horizontal timings; all I had to do was make sure that the *vsync* and *enable* signals were driven according to the specification.

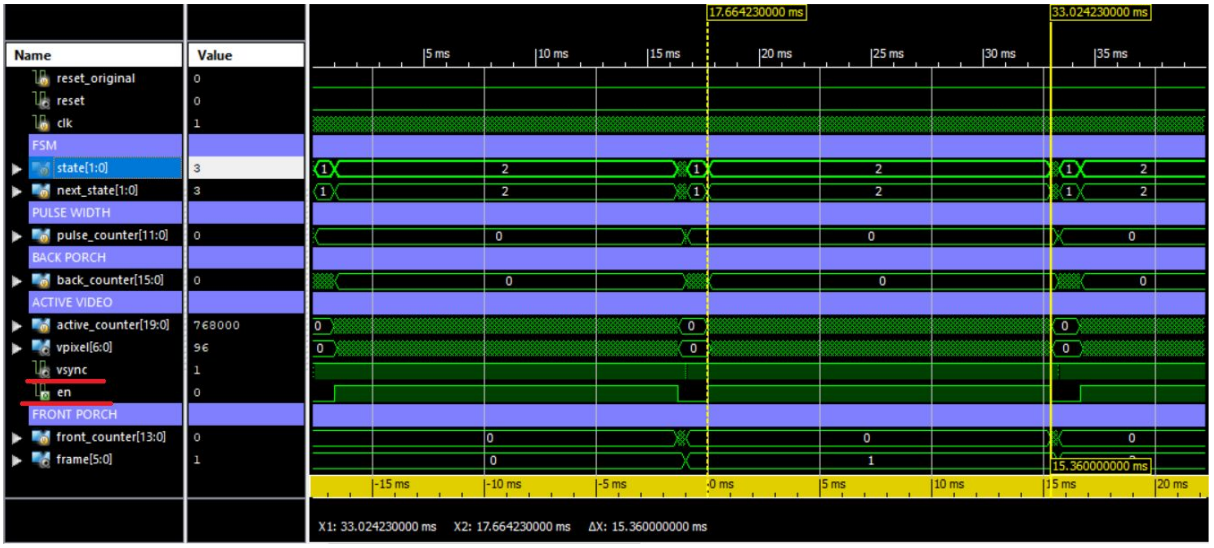
1. Pulse Width state: 64us duration, both vsync and enable are low



2. Back Porch state: 928us duration, vsync is high but enable is low



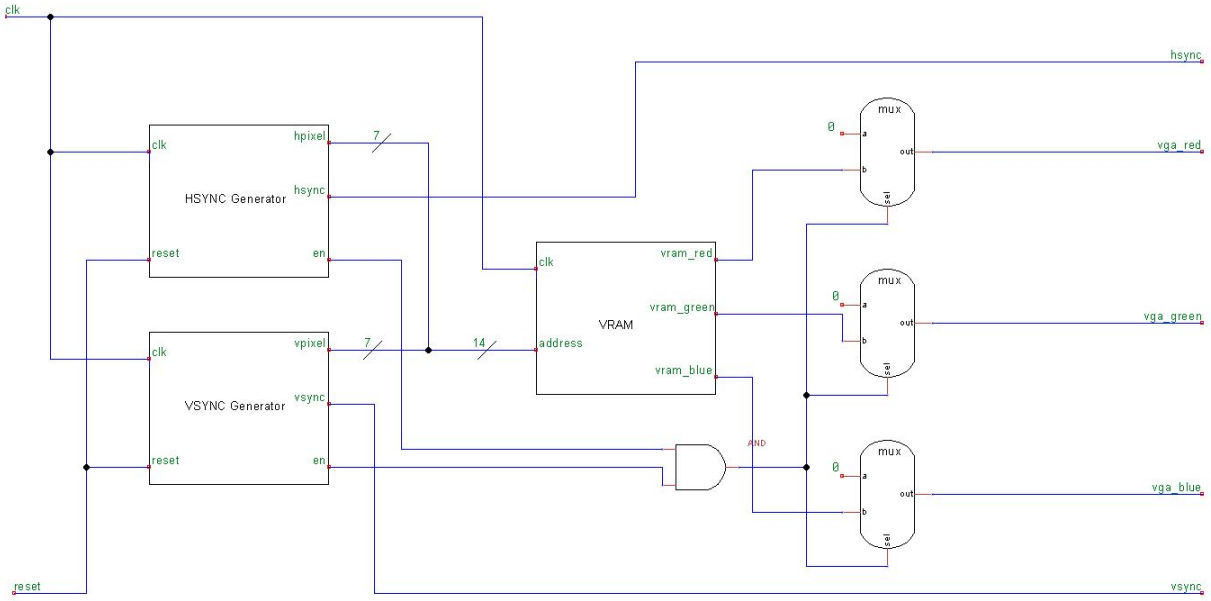
3. Active Video Time state: 15.36ms duration, both vsync and enable are high



4. Front Porch state: 320us duration, vsync is high but enable is low

Part C (cont.)—Completing the VGA driver

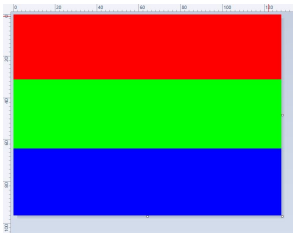
After I was finished with the above modules, completing the VGA driver was just a matter of instantiating them in a top level one and connecting them.



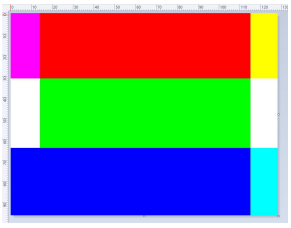
I use three multiplexers to assure that the R,G,B signals (vga_red, vga_green and vga_blue) are not held low only when *display time* and *active video time* overlap (since that's when the enable outputs are high). For simplicity, I didn't add the debouncer and synchronizer modules to the RTL diagram.

First test on the FPGA

In my first test on the FPGA, the following image was displayed, because of the way I initialized the video ram:



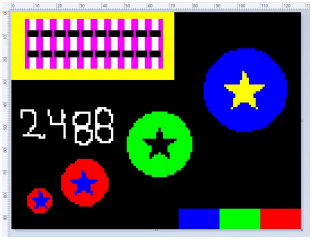
While my driver worked and the image was displayed without any problems, I changed the memory contents, adding some boxes to the displayed image, in order to make sure that the vertical timings also worked. The image that was displayed is the following:



I didn't encounter any problems when I tested my design on the FPGA.

Displaying any 128x96 image

The displayed image is stored in the video RAM, the contents of which are initialized by the INIT parameters. By changing these parameters manually, there's no way I would be able to display a complex image. To combat this, I developed a python script that accepts an 128x96 bitmap file as input and outputs three text files which contain the INIT parameters that will initialize the memory in such a way, that the bitmap file input will be displayed in the monitor (scaled to 640x480). So, all I have to do to change the displayed image is replace the correct INIT parameters after they are generated by my script. After toying a bit, I created the following image:



Testing the complex image on the FPGA

When I tested the above image on the FPGA, the display was mirrored and garbled. The cause of this was that I had initialized the memory the wrong way. Each INIT parameter is 256 bits, which means it holds the data for 2 image rows. Each parameter constant initializes the memory big-endian style; this is the opposite of how someone would intuitively map an image to a memory block, if he scanned it left to right, top to bottom.

By swapping the first 128 bits with the last 128 bits, I fixed the garbled display problem; now the image rows are mapped sequentially in memory, whereas before they were interleaved (the 2nd line was displayed first, then the 1st line, then the 3rd and so on).

By reversing the bits of each 128 bit part (each one representing one image line), I fixed the mirrored display problem: now the image pixels where mapped the correct way in the memory.

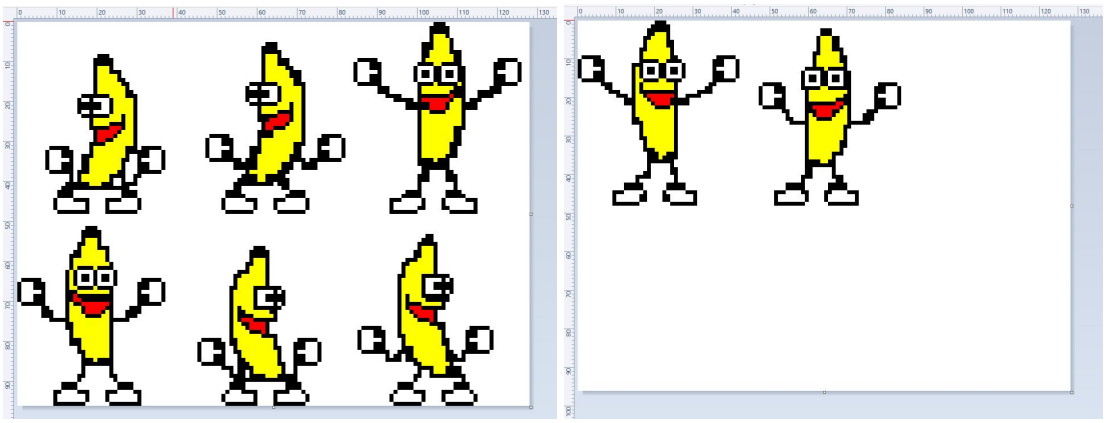
After I fixed my python script and reloaded the block ram contents, the image was displayed with no further problems.

Extra—Displaying an animation

After my driver could display a static image properly, I attempted to display a simple animation with 8 sprites, each one being 42×48 pixels. The 16KB $\times$ 1 block RAM modules could only store 6 whole sprites, so I added three more block ram modules instead of splicing the last sprites and "fitting" them in the original three block rams. I added a *select* input to the VRAM module, in order to choose between the original three block RAMs and the new ones.

My driver expects an 128×96 image to be displayed, while each sprite is only 42×48 pixels. To avoid scaling the image even further, I displayed each sprite in the middle of the monitor while keeping the rest of the pixels white. To accomplish this, I used a new module, *image_source*, that determines if the source of the color signals should be the

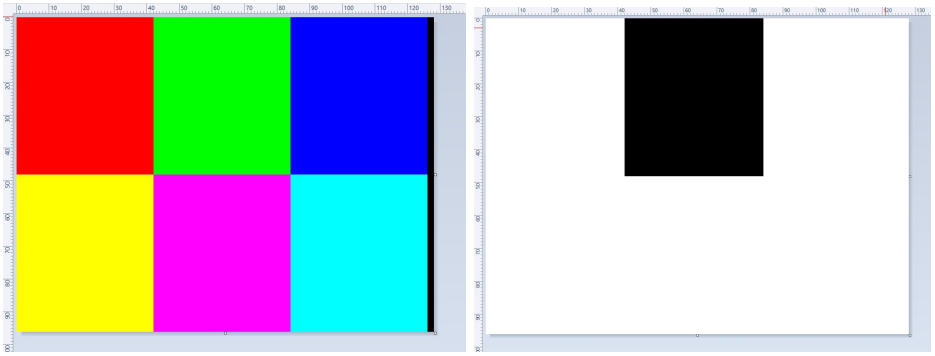
VRAM or if they should be assigned to 1 to display white. The selection is done based on the current pixel being displayed. If the image's source is the VRAM, another new module, *addres_generator*, determines the correct memory address (to display the right sprite) and the value of the *select* signal (0 for the first 6 sprites and 1 for the last 2). My animation runs at 10 frames per second, so I display each sprite for 6 frames before moving to the next one. The duration of each frame is one *total frame time* span, which is 16.67ms.



The first 6 sprites are stored in the original three block rams, while the remaining two are stored in the new ones. To display the correct sprite, I add an offset at the memory address according to the current sprite. For example, in order to display the second sprite, I add 42 to the original hpixel value, since each sprite is 42×48 pixels and the first sprite was already displayed. Finally, I subtract 42 from the hpixel value and 23 from the vpixel value to align the start (point 0,0) of the memory contents with the start of the box at the middle of the screen in which the sprites are being displayed (point 23,42).

Testbench

I used the following "sprites" to test that the animation worked correctly:



The black bar in the first image should never be displayed, since it is not part of any sprite. I didn't encounter any problems neither while simulating the design nor when I tested it on the FPGA.

Conclusion

I didn't encounter any major issues while designing the VGA driver, but some of the calculations where confusing enough that I didn't get them right the first time. I didn't find

the assignment particularly hard and since I finished early I delved deeper into it, developing the parameters generation script and extending the driver in order to display an animation. I also learnt more than I imagined I would; I put my basic python knowledge in use for the first time in an assignment and I designed a small tinyCAD library for the multiplexers in my RTL diagram.