

Multi Image VGA Display

Using Verilog

Team Execution Time



CAL POLY POMONA

Using Nexys 4 DDR

To Implement multiple image display via VGA.

GitHub Link:

<https://github.com/haileymag/VGA-multi-image-Display>

I. Project Introduction

The intended purpose of the verilog project was to display two separate images with different color palettes by use of a switch. Using the Nexys 4 DDR with Atrix-7, it already had an on board VGA port. Many people have used FPGA boards to process images, with some examples being compression, darkening, lightening, and removing saturation. The image is converted to two separate .mem files, compressing the photo from its original size. Compression became the focus, as most photos today are compressed by social media sites due to file limitations and the sheer load of requests on uploading/downloading/storing images. VGA can display the image of our choice, as long as it has been translated into a “bitmap”, or a file that translate each pixel to a specific color in hex, as well as the pixel’s position.

II. Mathematics

With displaying images from an FPGA board to a monitor, the image has to be to a certain size that the board can handle in it’s memory. We utilized VGA timing graphs as well as a very detailed explanation of displaying at 640x480 [1] [2].

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29
1024x768, 70Hz	75.000	1024	24	136	144	768	3	6	29
1024x768, 75Hz	78.750	1024	16	96	176	768	1	3	28
1024x768, 85Hz	94.500	1024	48	96	208	768	1	3	36

Source: Rick Ballantyne, Xilinx Inc. TABLE 1 VGA CORE VIDEO MODE

"640 x 350 (EGA on VGA)"	"640 x 400 VGA text"	"VGA industry standard"
Clock frequency 25.175 MHz	Clock frequency 25.175 MHz	Clock frequency 25.175 MHz
Line frequency 31469 Hz	Line frequency 31469 Hz	Line frequency 31469 Hz
Field frequency 70.086 Hz	Field frequency 70.086 Hz	Field frequency 59.94 Hz
One line:	One line:	One line:
8 pixels front porch	8 pixels front porch	8 pixels front porch
96 pixels horizontal sync	96 pixels horizontal sync	96 pixels horizontal sync
40 pixels back porch	40 pixels back porch	40 pixels back porch
8 pixels left border	8 pixels left border	8 pixels left border
640 pixels video	640 pixels video	640 pixels video
8 pixels right border	8 pixels right border	8 pixels right border
---	---	---
800 pixels total per line	800 pixels total per line	800 pixels total per line
One field:	One field:	One field:
31 lines front porch	5 lines front porch	2 lines front porch
2 lines vertical sync	2 lines vertical sync	2 lines vertical sync
54 lines back porch	28 lines back porch	25 lines back porch
6 lines top border	7 lines top border	8 lines top border
350 lines video	480 lines video	480 lines video
6 lines bottom border	7 lines bottom border	8 lines bottom border
---	---	---
449 lines total per field	449 lines total per field	525 lines total
per field		
Sync polarity: H positive, V negative.	Sync polarity: H negative, V positive.	Sync polarity: H negative, V negative.
Scan type: non interlaced.	Scan type: non interlaced.	Scan type: non interlaced.

Fig.2 VGA Timing (by Martin Hinner)

The image is preset to be 640x360, of 6 bit color (RGB, red, green, and blue). This is due to FPGA limitations at 4Mb of BRAM, while two images use around 2,700Kb of space.

$$360 * 640 * 6 = 1,350Kb$$

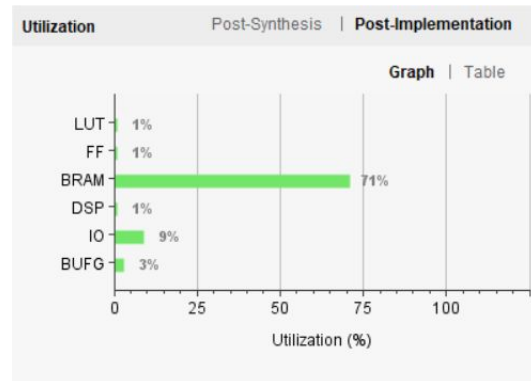


Fig. 3-Utilization of BRAM with 4 .mem files

An image needs to be mapped according to its color palette and it’s hex code pixel positioning. Using a Python program [3]

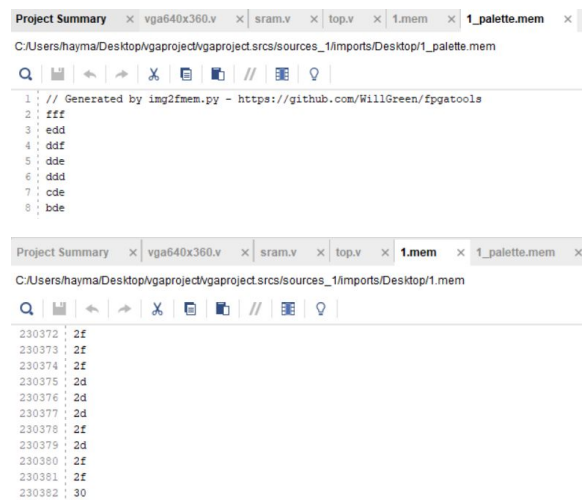
```

1#!/usr/bin/env python
2
3# img2fmem.py - image to FPGA memory map converter
4# By Will Green - https://timetoexplore.net
5# Copyright (c) 2018, Will Green, Licensed under BSD 3-Clause License
6# For latest version and docs visit https://github.com/WillGreen/fpgatools
7
8import os
9import sys
10from PIL import Image
11
12if len(sys.argv) != 4:
13    print("usage: python img2fmem.py image_file colour_bits output_format")
14    print("       image_file: source image file name")
15    print("       colour_bits: number of colour bits per pixel: 4, 6, or 8")
16    print("       output_format: mem or coe")
17    sys.exit()
18
19MESSAGE = "Generated by img2fmem.py - https://github.com/WillGreen/fpgatools\n"
20
21input_file = sys.argv[1]
22base_name = os.path.splitext(input_file)[0]
23
24colour_bits = int(sys.argv[2])
25if colour_bits == 4:
26    pal_size = 16
27elif colour_bits == 6:
28    pal_size = 64
29else:
30    pal_size = 256 # default to 8-bit
31    colour_bits = 8 # explicitly assign a value so we can use in COE format
32
33output_format = sys.argv[3]
34
35# Read image data

```

Fig. 4 Part of img2fmem.py

to get the output sample image, color palette, and memory hex values, \$readmemh can be used in the Verilog program. Loading the .mem file as a memory design, it will be saved to the board for use.



```

1 // Generated by img2fmem.py - https://github.com/WillGreen/fpgatools
2 fff
3 edd
4 ddf
5 dde
6 ddd
7 cde
8 bde

```

```

230372 2f
230373 2f
230374 2f
230375 2d
230376 2d
230377 2d
230378 2f
230379 2d
230380 2f
230381 2f
230382 30

```

Fig 5&6-Example of palette and photo .mem

\$readmemh works by reading the hex values in the memory files that are loaded in the board with the argument presented as such:

`$readmemh("img_palette.mem", palette);`

Adding the .mem files of the palette and translated pixels to be pulled from the

memory, it becomes translated to the output to be displayed.

III. Process and Issues

The first issues were figuring out how to properly translate an image of our choosing. After finding a tool created by Will Green [3], the use of a Python program properly translated our image. Setting up the Python code to output the correct translation was a difficulty, as we quickly discovered the Python code could only properly convert images under 100Kb, otherwise it would not display after referencing .mem files. While using less sized files does increase pixel artifacts, not wanting to risk the FPGA failing due to overload on the memory was more of a concern.

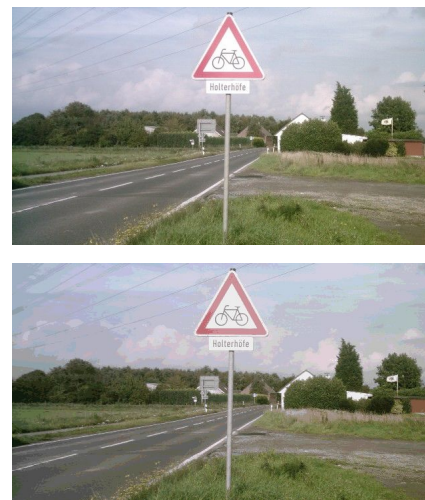


Fig 7&8- Image before and after conversion

Properly filling out the constraints, and fixing the clock to be 25MHz to make sure it would work on a normal monitor. When implementing multiple images, there was an issue with the default palette staying as the palette for the other image. This was fixed to make sure every image had the correct

palette while displaying using a case statement.

IV. Block Diagram

Will be shown on last page, due to size. The clock is sent to be divided by 4 to create the 25MHz clock. Our reset button is set to our display, that is outputting one of the images in their own RAM, with it's individual .mem file and .mem color palette that is already set as a memory file. The VRAM is sent to a MUX to be chosen by the switch to display. Each of the RGB channels of the photo are sent to the respective channel for output.

V. Encryption/Decryption Method

The method of encrypting and decrypting the image that was going to be implemented was the Advanced Encryption Standard (AES). The reasoning for deciding on implementing this type of encryption/decryption was due to our professor pressing us to implement it using an FPGA. The encryption of the image would run through a number of cycles depending on the number of bit encryption that was used either 128-bit or 256-bit encryption. The 128-bit encryption was the one going to be implemented in this project, since the project was not going to be demanding a higher level of encryption. The encryption process for AES is broken down into four stages, Substitute Bytes, Shift Row MixColumns, and AddRoundKey. The Substitute Bytes includes non-linear byte substitution, operating on each of the bytes independently, and is done using an

substitution table better known as an S-box. The table contains number between (0 - 255) and their corresponding result values.

The Shift Row shifts rows 2,3, and 4 by 1,2, and 3 bytes respectively all whilst leaving the first row intact.

The Mix Columns portion considers the columns of the state to be polynomials over $GF(2^8)$ and multiplied by module $x^4 +$ with a fixed polynomial

$$c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

The AddRoundKey takes in a Round Key that is added to the state which is the result of the MixColumns by a simple bitwise XOR operation. The Round Key of each round is derived from the main key using the Key Expansion algorithm.

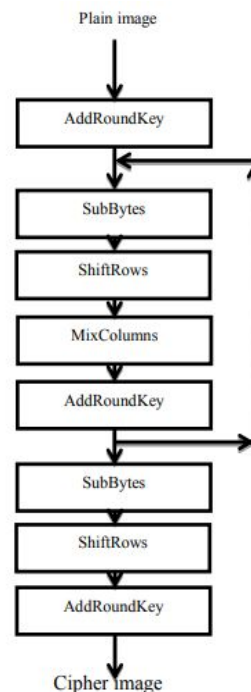


Fig 9- AES Encryption

The decryption portion of the image needs four stages as well, and they are the Add

Round Key, Inverse Shift Row, Inverse Substitute Byte, and Inverse Mix Columns. The Add Round Key is its own inverse function because the XOR function is its own inverse, thus the round keys have to be chosen in reverse order.

The Inverse Shift Rows does the opposite task as the Shift Rows, meaning that row 1 doesn't shift, while rows 2,3,and 4 each right shift 1,2, and 3 bytes respectively.

The Inverse Substitute Byte is done using a substitution table called the InvS-box. The InvS-box table contains numbers (0-255) and their corresponding values.

In the Inverse Mix Columns, the polynomials of degree less than 4 over GF (28), which coefficients are the elements in the columns of the state are multiplied by module (x4 +1) by a fixed polynomial $d(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}$, where {0B}, {0D}, {09}, and {0E} denote hex values.

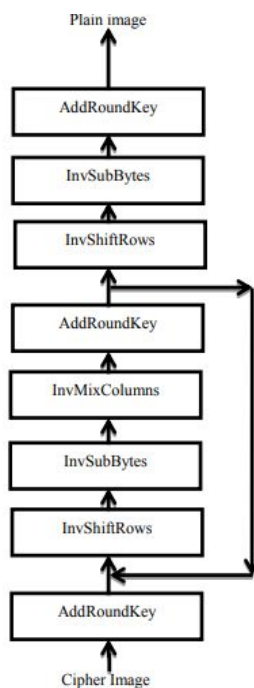


Fig 10- AES Decryption

VI. Conclusion

Figuring out the proper timing, correct file size allowed for photos to be displayed, and encryption proved to be the most difficult things. Choosing a smaller frame size with less bits sacrificed on a quality level, but allowed multiple images to be displayed via the BRAM.

VII. Resources

- [1]"VGA Timing Information." *Martin Hinner's Homepage*, martin.hinner.info/vga/timing.html.
- [2]Green, Will. "Video Timings: VGA, SVGA, 720P, 1080P." *Time to Explore*, timetoexplore.net/blog/video-timings-vga-720p-1080p.
- [3]Green, Will. "FPGA VGA Graphics in Verilog Part 2." *Time to Explore*, timetoexplore.net/blog/artty-fpga-vga-verilog-02.
- [4] Ghoradkar, Sneha; Shinde, Aparna. "Review on Image Encryption and Decryption using AES Algorithm." <https://pdfs.semanticscholar.org/714f/df79dacb11979701a702d6169de18b7f88bc.pdf>

The image displays three screenshots of a Verilog RTL schematic for a VGA controller, showing the internal logic and signal flow.

Top Screenshot: This view shows the main logic of the controller. It includes registers for color (color_reg[11:0]), address (address_reg[17:0]), and pixel (pxl_rgb_reg). Multiplexers (RTL_MUX) are used to select between different data paths. The logic is connected to the VGA output signals: VGA_B[3:0], VGA_G[3:0], VGA_R[3:0], VGA_HS[0], and VGA_VS[0].

Middle Screenshot: This view shows the display block (display) and the address register (address_reg[17:0]). The display block is connected to the address register and the pixel register (pxl_rgb_reg). The address register is also connected to the color register (color_reg[11:0]).

Bottom Screenshot: This view shows the color register (color_reg[11:0]) and the output multiplexers (RTL_MUX) for the VGA signals. The color register is connected to the output multiplexers, which are also connected to the VGA output signals: VGA_B[3:0], VGA_G[3:0], VGA_R[3:0], VGA_HS[0], and VGA_VS[0].

Hailey Magana

010424245

Researched more on VGA output, found python code that could be used for translating our images to .mem files for display. Discovered way to make RAM that would be able to display images, properly setting the constraint file for VGA output, as well as figuring out the best display option for multiple images (640x360 vs 800x600). Created presentation, wrote paper other than encryption section.

Raul Munguia

010329865

Researched the AES algorithm and its implementation on an FPGA for image encryption/decryption. Tried to replicate the results that were seen in videos of others implementing the algorithm but was unsuccessful in doing so on the FPGA. Wrote on encryption within the paper.

Johnny Estrada

011687442

Researched Image processing on FPGA using bitmap format, dependent on conversion of bmp to a hex. Ran into issue of BRAM instantiation as the implementation was simulation based only. Used the implementation found by Haley for BRAM to implement multiple images by using switches. Adjusted the palette collisions for the image colors. Set up a small discord server for team communication. Attempted to perform bit level encryption by logical operators, got Yoshi & Mario to disappear.

