LABORATORY MANUAL

Experiment 5
VGA Controller

UNIVERSITY OF TEHRAN

School of Electrical and Computer Engineering

Digital Logic Laboratory
ECE 045

Fall 1396

Prepared and developed by Katayoon Basharkhah under the supervision of Professor Z.Navabi

EXPERIMENT 5: session 12

VGA Controller

INTRODUCTION

Displaying text is an important function of a video controller. Dedicated circuits are often used to facilitate the display of text characters on a screen.In this experiment, you will learn how to display text or color on a computer type monitor using the VGA output of DE1 board as Figure 1 shows the experiment set-up.

You have to create a character generator circuit for displaying ASCII text characters on the VGA display. The VGA driver in this experiment is capable of displaying characters from a display RAM on a standard VGA monitor. After discussing how a VGA monitor works, we show hardware for driving it. The design methodology presented here uses Verilog blocks, megafunctions, memories and schematic capture. Below is the topics that are explained in the following of this experiment:

- VGA driver operation
- Monitor synchronization hardware
- Character display
- VGA driver implementation on DE1 board

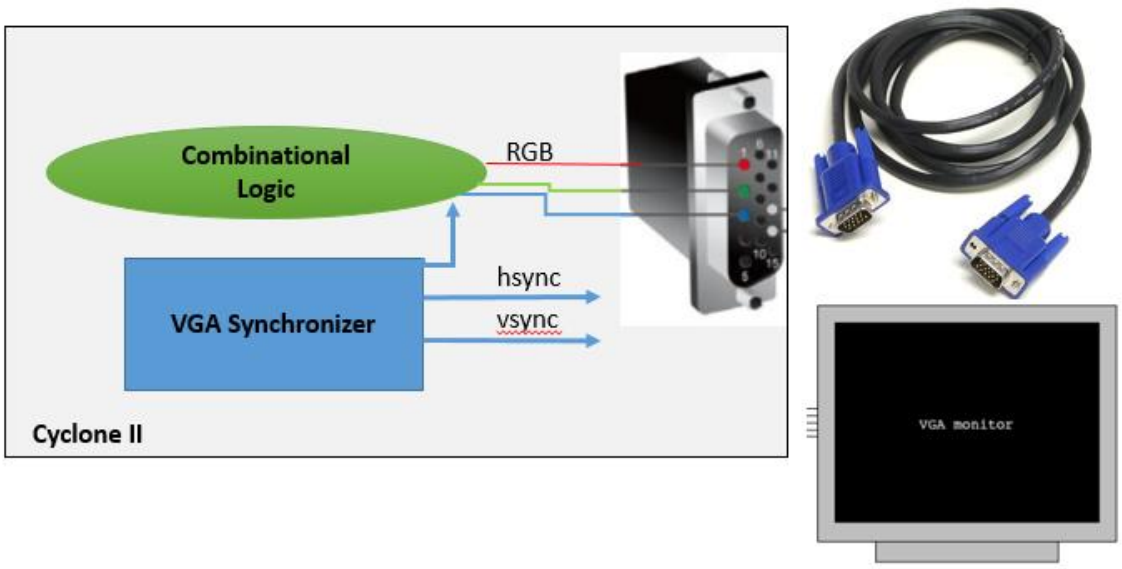


Figure 1: experiment set-up

Prepared and developed by Katayoon Basharkhah under the supervision of Professor Z.Navabi

PART I
1-VGA Driver Operation

A standard VGA monitor consists of a grid of pixels that can be divided into rows and columns. A VGA monitor contains at least 480 rows, with 640 pixels per row. Each pixel can display various colors, depending on the state of the red, green, and blue input signals. These signals are analog signals and determine the intensity of their corresponding colors. In a VGA monitor pixels are refreshed with proper color one-by-one from location (0, 0) to (640, 480). This is done from upper left of the monitor to its lower right. To eliminate any screen flickering, the refreshing must be done such that the entire screen is refreshed, i.e., all screen is sweeped in less than 0.02 second. For a better than maximum refresh time, of we choose a refresh cycle of 1/60 of a second. With this rate, and considering the overhead time of moving scan from one line to another, and one screen to next, a frequency of 25.175 MHz is required for refreshing each pixel.

2- Pixel sweeping

For the VGA monitor to work properly, it must receive data at specific times with specific pulses. Horizontal and vertical synchronization pulses must occur at specified times to synchronize the monitor while it is receiving color data.

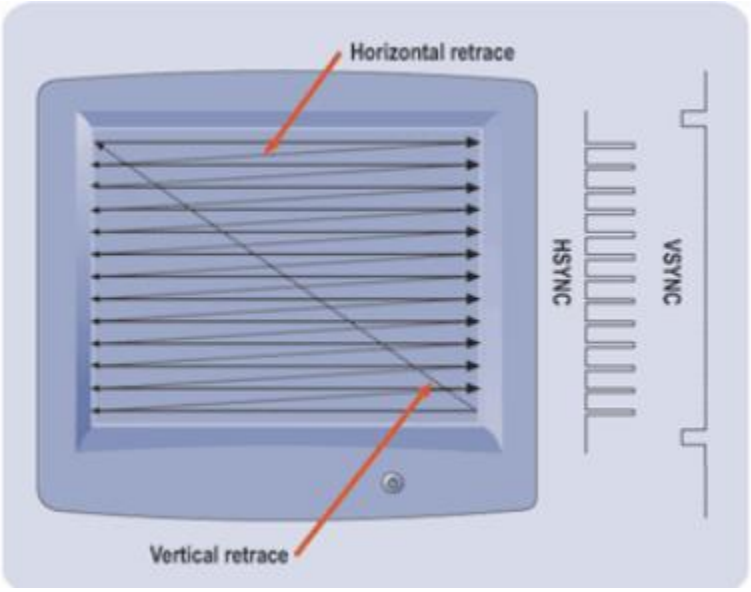


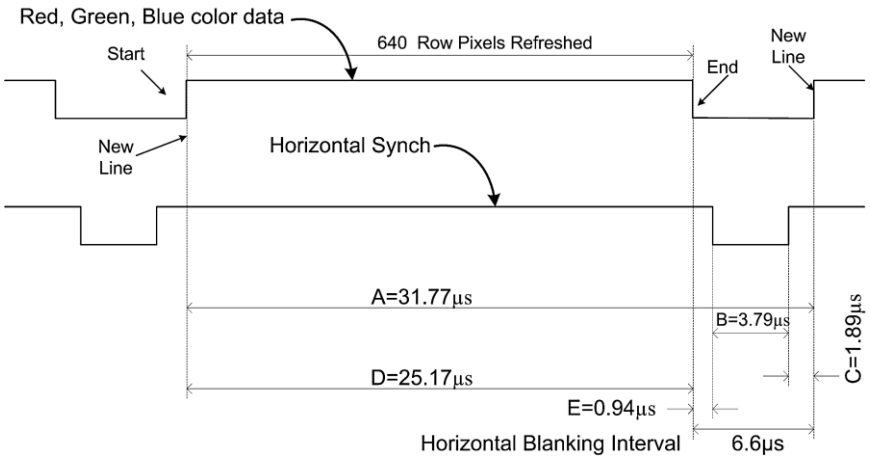
Figure 2: Pixel sweeping

You can make a generous eight colors on the screen by turning on combinations of the R, G, and B. Figure 3 shows the colors you can get with this simple interface.

VGA_RED	VGA_GREEN	VGA_BLUE	Resulting 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

Figure 3: RGB colors

Figure 4 shows the timing waveform for the color information with respect to the horizontal synchronization and vertical synchronization signal. Times shown are for the standard pixel frequency of 25.175 MHz for 640×480 pixel resolution.



Prepared and developed by Katayoon Basharkhah under the supervision of Professor Z.Navabi

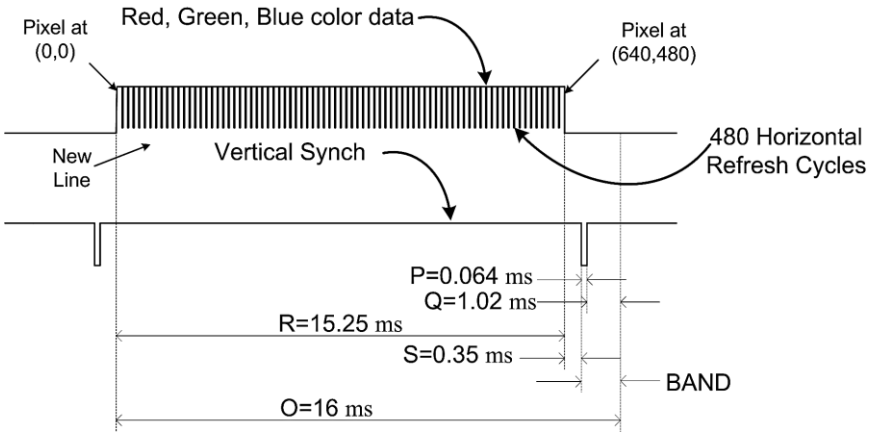


Figure 4: Horizontal and vertical Refresh Cycle

The hardware required for VGA signal generation must keep track of the number of clock cycles (equivalent pixels), and issue signals according to the timing waveforms of Figure 4. The Verilog code of Figure 5 uses the SynchClock clock signal to generate Hsynch (Horizontal Synch of Figure 4), Vsynch (Vertical Synch of Figure 4), and Red, Green, and Blue color data.

```
module MonitorSynch (
   input PixelOn,
   input RedIn, GreenIn, BlueIn, Reset, SynchClock,
   output Red, Green, Blue, Hsynch, Vsynch,
   output [9:0] PixelRow, PixelCol);

   reg [9:0] Hcount, Vcount;

   always @(posedge SynchClock) begin
      if (~Reset) Hcount = 0;
      else
         if (Hcount == 799) Hcount =0;
         else Hcount <= Hcount + 1;
   end
   always @(posedge SynchClock) begin
      if (~Reset) Vcount = 0;
      else
         if (Vcount >= 525 && Hcount >= 756) Vcount = 0;
         else if (Hcount == 756) Vcount <= Vcount + 1;
   end

   assign Hsynch = (Hcount>= 661 && Hcount <= 756) ? 0 : 1;
   assign Vsynch = (Vcount>= 491 && Vcount <= 493) ? 0 : 1;
   assign {Red, Green, Blue}=(Hcount<=640 && Vcount<480) ?
      {PixelOn&RedIn, PixelOn&GreenIn, PixelOn&BlueIn} : 0;
   assign PixelCol = (Hcount <= 640) ? Hcount : 0;
   assign PixelRow = (Vcount <= 480) ? Vcount : 0;
endmodule
```

Figure5 Monitor Synchronization Hardware

The code shown uses color specifications from *RedIn*, *GreenIn* and *BlueIn* input signals and during the time periods specified by parameter *D* and *R* in Figure 4, puts them on the *Red*, *Green* and *Blue* output signals. At any pint in time, the Verilog code of Figure 5 outputs the position of the pixel being updated in its 10-bit *PixelRow* and *PixelCol* output vectors.

Two always blocks that are synchronized with *SynchClock* keep track of horizontal and vertical counts (*Hcount* and *Vcount*). *Hcount* is associated with parameter *A* of **Error! Reference source not found.** and *Vcount* with parameter *O* of **Error! Reference source not found.**. The Verilog code of Figure is defined as a block that will be used in our implementation of a character display design.

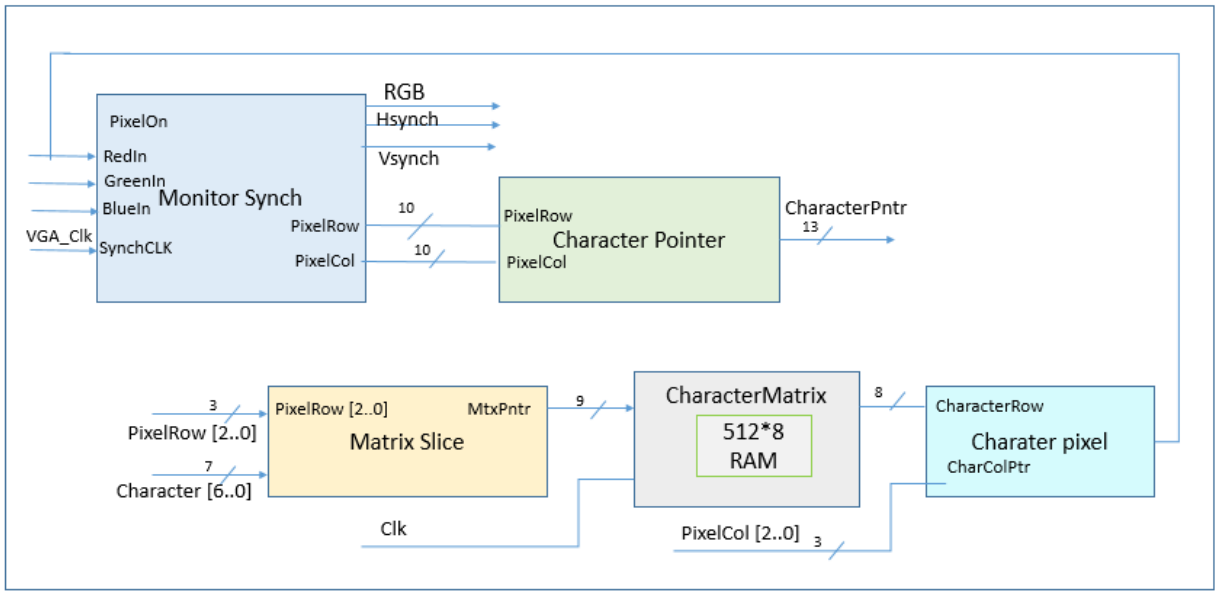


Figure 6: Character Display Hardware

A- Character Pointer

The character pointer hardware is the block shown in upper right part of Figure 6. MonitorSynch provides X and Y pixel coordinates for this hardware. CharacterPointer takes these coordinates and generates a 13 bit address pointing to one of the 4800 characters at the screen location. Because each character consists of 8 row and 8 column pixels, 64 different X and Y coordinates map to the same character address. The Verilog code for generating this mapping is the CharacterPointer module of Figure 7.

```
module CharacterPointer (
   input [9:0] PixelRow, PixelCol, output [12:0] CharPntr);
   wire [6:0] ScreenLine, ScreenPos;

   assign ScreenLine = PixelRow [9:3];
   // 60 Lines=480/8 (8 is Character Pixel Hight)

   assign ScreenPos = PixelCol [9:3];
   // 80 Positions=640/8 (8 is Character Pixel Width)

   assign CharPntr =  ScreenLine*80 + ScreenPos;
endmodule
```

Figure Error! No text of specified style in document.**: Finding Character Position from a Pixel Position**

B- **Pixel Generation Hardware**.

The lower part of Figure 6 is responsible for generation of a specific pixel value (1 or 0) for the spe-cific X-Y position of screen and the character being displayed. Inputs to this part are ASCII code of the character being displayed (Character[6..0]) and coordinates within the 8*8 pixel area of the character (PixelRow[2..0] and PixelCol[2..0]). The three parts of pixel generation are MatrixSlice, CharacterMatrix and CharacterPixel.

The MatrixSlice module (Figure 8) takes the ASCII code of the input character and subtracts 32 from it to make printable character codes begin from 0. It then appends three bits of PixelRow to its right to form an address for the pixel row of the corresponding character.

```
module MatrixSlice (input [2:0] PixelRow, input [6:0] Char,
                    output [8:0] MtxPntr);
   assign MtxPntr = {Char - 32, PixelRow[2:0]};
endmodule
```

Figure 8 Matrix Slice Verilog Code

The output of this module looks up a row of the character being displayed from the CharacterMatrix component. In our simple design we use 8*8 character resolution and only support ASCII characters from 32 to 95. With these 64 supported characters, our character ma-trix becomes an 8-bit memory of 512 words, in which every 8 consecutive words define a character. For example, as shown in Figure 9, pixels for character “5” with ASCII code of 53 decimal, begin at address 0A8 Hex that is (53-32)*8.

```
0A8  : 01111110 ; %   ******      %
0A9  : 01100000 ; %   **          %
0AA  : 01111100 ; %   *****       %
0AB  : 00000110 ; %       **      %
0AC  : 00000110 ; %       **      %
0AD  : 01100110 ; %   **  **      %
0AE  : 00111100 ; %    ****       %
0AF  : 00000000 ; %               %
```

Figure 9: Character Matrix for Character "5"

The CharacterMatrix component is a RAM that is implemented with an Altera LPM and is mapped into the on-chip memory of our FPGA. This component is called LPM_RAM and is available under the storage category of Altera megafunctions. Using the megafunction wizard we configure this component as an 8-bit memory with nine address lines. During the configuration process we are asked to enter the Memory Initialization File name (.mif), for which we use CharMtx.mif. Using the mif format, pixel values (similar to those shown in Figure 9 for character "5") are defined for ASCII characters from 32 to 95. Figure 10 shows the beginning and end of this file, from which its complete format can be seen.

The output of CharacterMatrix is q[7..0]. This output has a slice of the character that is being displayed. For example for row 2 of character "5", q[7..0] is 01111100.

The last component shown in Figure 6 that is responsible for pixel generation is CharacterPixel. This component takes a character row and its column pointer (PixelCol [2:0]) and looks up the pixel to be displayed. The Verilog code of CharacterPixel is shown in Figure 11. This code uses CharColPntr as index to look into CharacterRow in reverse bit order.

The complete schematic of our character display hardware is shown in Figure 6. The MonitorSynch module continuously sweeps across the 640*480 pixel screen and refreshes pixel with colors specified by its three color inputs. At the same time it reports the position of the pixel being refreshed to CharacterPointer. Based on these coordinate, this module calculates the address of the character that is being displayed. After the character is looked up refreshed is found. This pixel value allows color inputs to be used by the MonitorSynch module for painting the pixel. The complete design shown in Figure 6 is referred to as CharacterDisplay.

```
DEPTH = 512;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
% Character Matrix ROM,                %
% addressed by PixelGeneration module %
CONTENT
BEGIN
  % ASCII 0010_0000 to 0010_1111  %
  000  : 00000000 ; %              %
  001  : 00000000 ; %              %
  002  : 00000000 ; %              %
  003  : 00000000 ; %              %
  004  : 00000000 ; %              %
  005  : 00000000 ; %              %
  006  : 00000000 ; %              %
  007  : 00000000 ; %              %
  . . .
  1F8  : 00000000 ; %              %
  1F9  : 00010000 ; %      *       %
  1FA  : 00110000 ; %     **       %
  1FB  : 01111111 ; %   *******    %
  1FC  : 01111111 ; %   *******    %
  1FD  : 00110000 ; %     **       %
  1FE  : 00010000 ; %      *       %
  1FF  : 00000000 ; %              %
END;
```

Figure 10: Character Matrix mif File

```
module CharacterPixel (
   input [7:0] CharacterRow,
   input [2:0] CharColPntr, output Pixel);
   wire [2:0] indx =
          {CharColPntr[2], CharColPntr[1], CharColPntr[0]};
   wire [7:0] Vector =
          {CharacterRow[0], CharacterRow[1],
           CharacterRow[2], CharacterRow[3],
           CharacterRow[4], CharacterRow[5],
           CharacterRow[6], CharacterRow[7] };
   assign Pixel = Vector [indx];
endmodule
```

Figure 11: Looking up Character Pixel

- Use the codes provided to you for the character display module and construct the circuit of Figure 6 in a block diagram file.

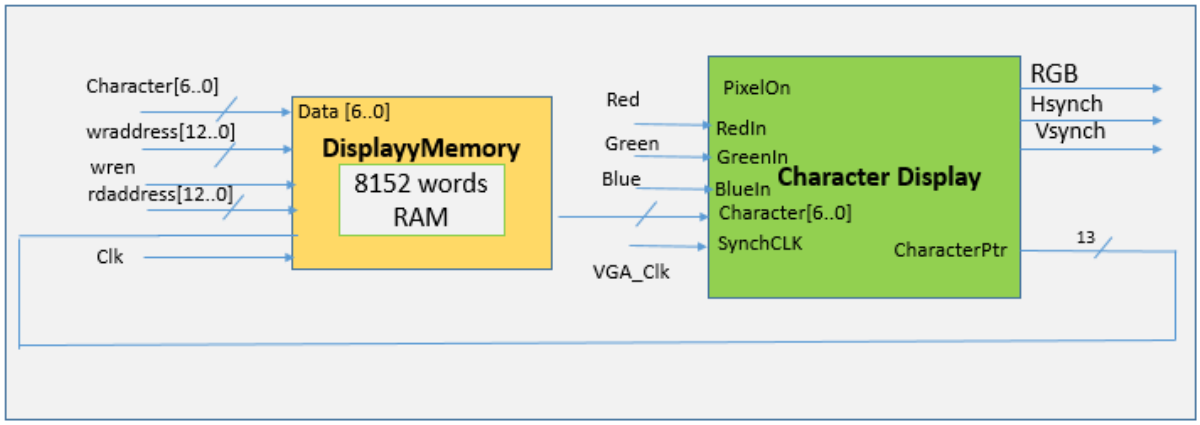


Figure 12: VGA Driver with Display Memory

3- VGA Driver for Text Data

The previous section discussed the complete design of *CharacterDisplay* hardware. This hardware outputs the address of one of the 4800 characters that is to appear on the screen, looks up its ASCII code, and generates pixel colors and horizontal and vertical synchronization signals. This section shows a simple VGA driver that provides data to be displayed to our *CharacterDisplay* hardware.

The complete schematic of our *VGA_Driver* is shown in Figure . On the right hand side is *CharacterDisplay* that generates character address, monitor synch, and pixel information. On the left hand side is *DisplayMemory* that is a dual-port read/write memory. Character address from *CharacterDisplay* goes to its read address. The write address, data to be written, and the write enable of this memory are provided externally. While *CharacterDisplay* is displaying the current contents of the memory, new data for display can be written into this memory.

Our VGA-Driver has two clock inputs, *VGAclk* and *Memoryclk*. The latter must be a faster clock so that address output from *CharacterDisplay* can be used to lookup the display character from the Display Memory.

- construct the circuit of Figure 12 in a block diagram file in QuatusII. Use the block diagrams of the *character display* design.

4- VGA Driver test

Figure 13 shows a simple tester for our VGA driver. This circuit causes the initial data in DisplayMemory to be displayed on the VGA monitor. Changing display data can only happen by changing contents of DisplayRAM.mif file which is loaded into this memory.

Note the use of the T-type flip-flop for generating a slower clock for the VGA driver than that of the memory. This circuit is implemented on a UP3 development board and verifies the operation of our VGA adapter. A more elaborate testbench would have a counter to set memory display memory locations to desired characters. We leave this as an exercise.

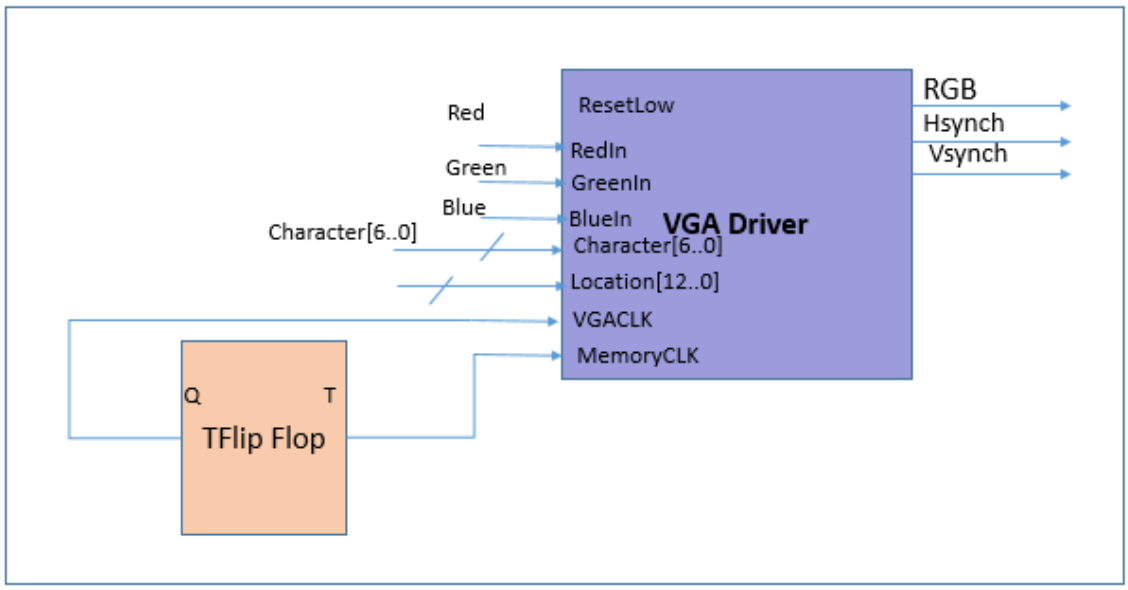


Figure 13: VGA driver tester

- Construct the circuit of Figure 13 in a block diagram file. Use the block diagram of the *VGA Driver* design. Make the proper pin assignment. Althogh the design has the capability of storing and displaying data from the input ports, you do not have to demonstrate this part. Just show the characters that are stored in the mif file. So you do not need to assign the character or location ports. Use the SW[2:0] to change the RGB color and KEY[0] to reset. For the MemoryClk use the 50MHz FPGA Clock. After programming the design on the specified Cyclone device, connect the VGA port to the monitor using the VGA cable available to you.
- Test your design and record the results. You should observe some text on the monitor.

Design Verification:

- ✓ **Demonstrate the result of this part to your TA .**
- ✓ **The TA should verify the accuracy of your results.**
- ✓ **Have your TA sign the part of your Lab-book corresponding to work you did for this part.**

PART II

After understanding the VGA operation and displaying the text on a monitor, now you have to choose one of the task below and complete it. Completing both of these tasks will have an extra point.

A) As you have seen in the previous part, the text on the monitor is displayed by 8*8 pixel that are small. Change the design to make the text larger but do not change the character pixel size or memory.

B) Use the Persian alphabet ASCII code instead of the English one and write them in *mif* file. Then do the same as part I and sho the characters on the monitor.

Design Verification:

- ✓ **Demonstrate the result of this part to your TA .**
- ✓ **The TA should verify the accuracy of your results.**
- ✓ **Have your TA sign the part of your Lab-book corresponding to work you did for this part.**

PRELAB ASSIGNMENT

Before coming to the lab, answer these questions. The pre-lab needs to be handed in at the start of the lab.

1- Read about the *.MIF files structure on http://quartushelp.altera.com/15.0/mergedProjects/reference/glossary/def_mif.htm

2- Explain the MonitorSynch module based on the Figure 4 synchronization timing. For more detailed information you can refer to "Embedded Core Design with FPGAs", McGraw-Hill Electronic Engineering, 2007.