12 VGA Adapter

The design of a VGA adapter capable of displaying characters from a display RAM on a standard VGA monitor is discussed in this chapter. The chapter discusses the basics of interfacing with a VGA monitor and develops an interface using the UP2 development board. Unlike the previous chapter that dealt with a top-down data/control design, this chapter designs small interfaces and puts a complete design together. The design methodology presented here uses Verilog blocks, megafunctions, memories, and schematic capture to complete the design of a display adapter.

12.1 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, as shown in Figure 12.1. Each pixel can display various colors, depending on the state of the red, green, and blue signals.

Each VGA monitor has an internal clock that determines when each pixel is updated. This clock operates at the VGA-specified frequency of 25.175 MHz. The monitor refreshes the screen in a prescribed manner that is partially controlled by the horizontal and vertical synchronization signals. The monitor starts each refresh cycle by updating the pixel in the top left-hand corner of the screen, which can be treated as the origin of an X–Y plane (see Figure 12.1).

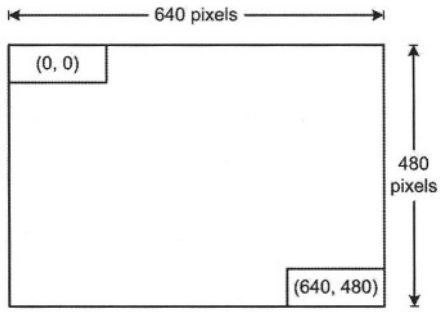


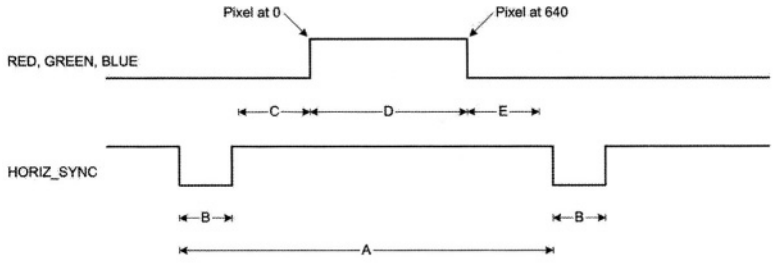
Figure 12.1 VGA Monitor

After the first pixel is refreshed, the monitor refreshes the remaining pixels in the row. When the monitor receives a pulse on the horizontal synchronization, it refreshes the next row of pixels. This process is repeated until the monitor reaches the bottom of the screen. When the monitor reaches the bottom of the screen, the vertical synchronization pulses, causing the monitor to begin refreshing pixels at the top of the screen (i.e., at [0,0]).

12.1.1 VGA Timing

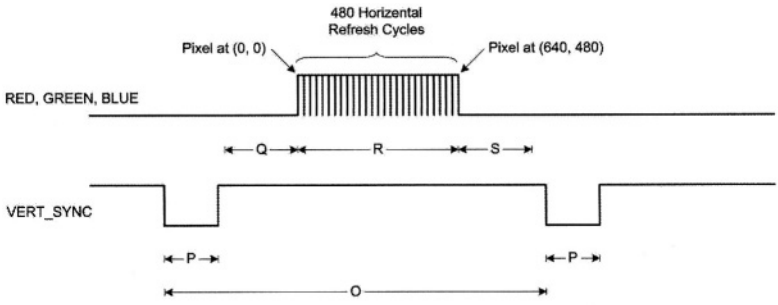
For the VGA monitor to work properly, it must receive data at specific times with specific pulses. Horizontal and vertical synchronization pulses must occur at specified times to synchronize the monitor while it is receiving color data.

Figure 12.2 shows the timing waveform for the color information with respect to the horizontal synchronization signal. Based on the clock frequency, these times translate to certain number of clock cycles shown in this figure. For example, a horizontal sweep (parameter *A*) that takes **31.77 μs**, translates to 800 clock cycles of 25.175 MHz.



Parameters	A	B	C	D	E
Time	31.77 μs	3.77 μs	1.89 μs	25.17 μs	0.94 μs
Clock Cycle	800	95	48	634	24

Figure 12.2 Horizontal Refresh Cycle



Parameters	O	P	Q	R	S
Time	16 ms	64 μs	1.02	15.25 ms	0.35 ms
Horizontal cycle	525	2	32	480	1

Figure 12.3 Vertical Refresh Cycle

Figure 12.3 shows the timing waveform for the color information with respect to the vertical synchronization signal. Based on the fact that a horizontal sweep takes 31.77 μs (800 clock cycles), the times shown take a certain number of horizontal refresh cycles that are shown in this figure. For example, a screen refresh cycle that takes 16.7 ms, translates to 525 horizontal cycles of 31.77 μs.

The frequency of operation and the number of pixels that the monitor must update determines the time required to update each pixel, and the time required to update the whole screen. The following equations roughly calculate the time required for the monitor to perform all of its functions.

$$T_{pixel} = 1/f_{CLK} = 40 \text{ ns}$$
$$T_{ROW} = A = B + C + D + E$$
$$= (T_{pixel} \cdot 640 \text{ pixels}) + \text{row} + \text{guard bands} = 31.77 \ \mu s$$
$$T_{screen} = O = P + Q + R + S$$
$$= (T_{ROW} \cdot 480 \text{ rows}) + \text{guard bands} = 16.6 \text{ ms}$$

Where:

T_{pixel}=Time required to update a pixel
f_{CLK}=25.175 MHz
T_{ROW}=Time required to update one row
T_{screen}=Time required to update the screen
B, C, E=Guard bands
P, Q, S=Guard bands

The monitor writes to the screen by sending red, green, blue, horizontal and vertical synchronization signals when the screen is at the expected location.

Once the timing of the horizontal and vertical synchronization signals is accurate, the monitor only needs to keep track of the current location, so it can send the correct color data to the pixel.

12.1.2 Monitor Synchronization Hardware

The hardware required for VGA signal generation must keep track of the number of 25.175 MHz clock cycles, and issue signals according to the timing waveforms of Figure 12.2 and Figure 12.3. The Verilog code of Figure 12.4 uses the *SynchClock* clock signal to generate *Hsynch* (HORIZ_SYNCH of Figure 12.2) and *Vsynch* (VERT_ SYNCH of Figure 12.3).

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

     reg Red, Green, Blue, Vsynch, Hsynch;
     reg [9:0] PixelRow, PixelCol;
     reg [9:0] Hcount, Vcount;

     always @(posedge SynchClock) begin
          if (Hcount == 799) Hcount =0;
               else Hcount = Hcount + 1;
          if (Hcount >= 661 && Hcount <= 756) Hsynch = 0;
               else Hsynch = 1;
          if (Vcount >= 525 && Hcount >= 756) Vcount = 0;
               else if (Hcount == 756) Vcount = Vcount + 1;
          if (Vcount >= 491 && Vcount <= 493) Vsynch = 0;
               else Vsynch = 1;

          if (Hcount <= 640) PixelCol = Hcount;
          if (Vcount <= 480) PixelRow = Vcount;

          if (Hcount <= 680 && Vcount <= 480) begin
               Red = RedIn;  Green = GreenIn;  Blue = BlueIn;
          end else
               {Red, Green, Blue} = 0;
     end
endmodule
```

Figure 12.4 Monitor Synchronization Verilog Code

The code shown, uses color specifications from *RedIn, GreenIn* and *BlueIn* input signals and during the time periods specified by parameter *D* in Figure 12.2 and parameter *R* in Figure 12.3, puts them on the *Red, Green* and *Blue* output signals. At any point in time, the Verilog code of Figure 12.4 outputs the position of the pixel being updated in its 10-bit *PixelRow* and *PixelCol* output vectors.

The *Hcount* variable in this Verilog code keeps track of the number of clock cycles in each row, and *Vcount* is the number of horizontal cycles in each screen. Considering the very first pixel at (0, 0) position, the counting of the horizontal pixels begins at the beginning of the *D* region of the waveform of Figure 12.2. Therefore as the Verilog code shows, *Hsynch* becomes **0** when *Hcount* is between 661 and 756 (this is the *B* region). Likewise, considering the beginning of region *R* as the 0 point, the *P* region in Figure 12.3 begins at *Vcount* of 491 and ends at 493. Therefore, as the code shows, *Vsynch* is **0** during such *Vcount* values. With the (0, 0) point defined as such, pixels are active while *Hcount* is between 0 and 640 and *Vcount* is between 0 and 480. During these count periods output colors are active and *PixelRow* and *PixelCol* outputs reflect *Hcount* and *Vcount* respectively.

The Verilog code of Figure 12.4 is defined as a block shown in Figure 12.5 to be used in our implementation of a character display design.

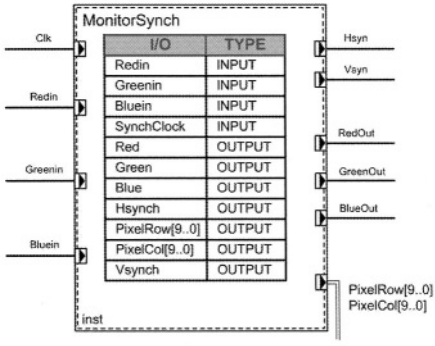


Figure 12.5 Monitor Synchronization Block Specification

12.2 Character Display

The design we are considering in this section is a character display hardware that outputs an address to a character display memory and inputs an ASCII code representing the character to display. We assume the display memory has 300 ASCII characters that will be displayed in 15 rows of 20 characters. Considering the 480 by 640 resolution, this makes each character occupy a matrix of 32×32 pixels.

In addition to the synchronization module of the previous section, the character display hardware has a character matrix and a pixel generation module. The character matrix defines active pixels of the supported characters and the pixel generation module reads this matrix and produces active color inputs (*RedIn, GreenIn,* and *BlueIn*) for the synchronization module.

12.2.1 Character Matrix

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 matrix becomes an 8-bit memory of 512 words, in which every 8 consecutive words define a character. For example, as shown in Figure 12.6, 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 12.6 Character Matrix for Character "5"

```
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 12.7 Character Matrix *mif* File

For implementing the character matrix we use the LPM_ROM megafunction of Quartus II. This component is available under the *storage* category of megafunctions. With the aid of the megafunction wizard, this component is configured as an 8-bit memory with 9 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 12.6 for character "5") are defined for ASCII characters from 32 to 95. Figure 12.7 shows the beginning and end of this file, from which the formatting can be seen.

The symbol of the *CharMtx* component defined in Quartus II is shown in Figure 12.8. The input of this ROM is a 9-bit address and its output is the horizontal slice of the display code of the character. The most significant 6 bits of the address are the ASCII code for the display character, and bits 2 to 0 of the address determine its slice number.

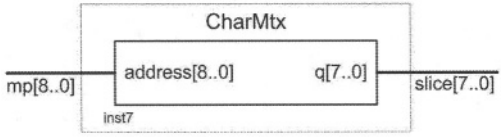


Figure 12.8 Character Matrix Symbol

```
module PixelGeneration (PixelRow, PixelCol, Clk, Char, MtxPntr, CharPntr);
    input [9:0] PixelRow, PixelCol;
    input Clk;
    input [7:0] Char;
    output [8:0] MtxPntr,
    output [8:0] CharPntr;

    reg [5:0] MtxStart;
    reg [8:0] MtxPntr;
    reg [8:0] CharPntr;

    wire [4:0] ScreenLine, ScreenPos;
    assign ScreenLine = PixelRow [9:5]; // 15 Lines=480/32
    assign ScreenPos = PixelCol [9:5]; //20 Positions=640/32

    always @(posedge Clk) begin
        CharPntr = ScreenLine*20 + ScreenPos;
        MtxStart = Char - 32;
        // Char resolution is 8 pixel rows, bits [4:2];
        MtxPntr = {MtxStart, PixelRow[4:2]};
    end
endmodule
```

Figure 12.9 Pixel Generation Verilog Code

12.2.2 Pixel Generation Module

Another component for our character display hardware is the *PixelGeneration* module. This module uses the 640×480 pixel coordinates from the *MonitorSynch* module and translates it to our low-resolution 20×15 character coordinates. Using the latter coordinates it generates an address for our display memory (that contains 300 ASCII codes) and reads the corresponding ASCII code. This code, offset by 32, and the pixel position that is being refreshed determine a pointer for the character matrix (*CharMtx*) discussed above. The Verilog code of this module is shown in Figure 12.9.

The output of the *CharMtx* ROM is an 8-bit slice of the character being displayed. The specific bit that is to be displayed is selected by pixel column position coming from the *MonitorSynch* module.

12.2.3 Character Display Hardware

The complete schematic diagram of the character display hardware is shown in Figure 12.10. This hardware is our VGA adapter that addresses a memory of 300 characters, reads the character and displays it in one of the 300 locations of the screen. This hardware uses *MonitorSynch, PixelGeneration, CharMtx,* and an 8-to-1 multiplexer.

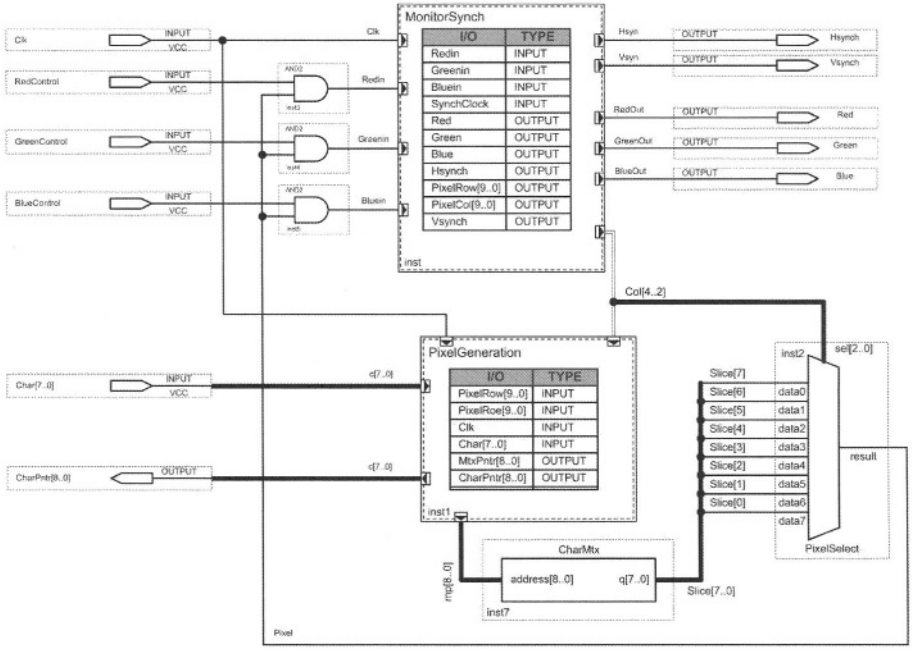


Figure 12.10 VGA Adapter Complete Schematic

The *MonitorSynch* module continuously sweeps across the 640×480 pixel screen and refreshes pixels with colors specified by its three color inputs. At the same time it reports the position of the pixel being refreshed to *PixelGeneration*. Based on these coordinate, this module calculates the address of the character that is being displayed, and using the *CharMtx* and the 8-to-1 multiplexer determines the value of the pixel in the screen position being refreshed. This pixel value allows color inputs to be used by the *MonitorSynch* module for painting the pixel.

12.3 UP2 Prototyping

The design of our VGA adapter of Figure 12.10 is complete in the sense that it addresses a character and displays it on the monitor. Testing this design requires a display memory with some data. Figure 12.11 shows a system utilizing this *Adapter* circuit to display character contents of *DisplayMem* while providing a mechanism of writing new characters into this memory.

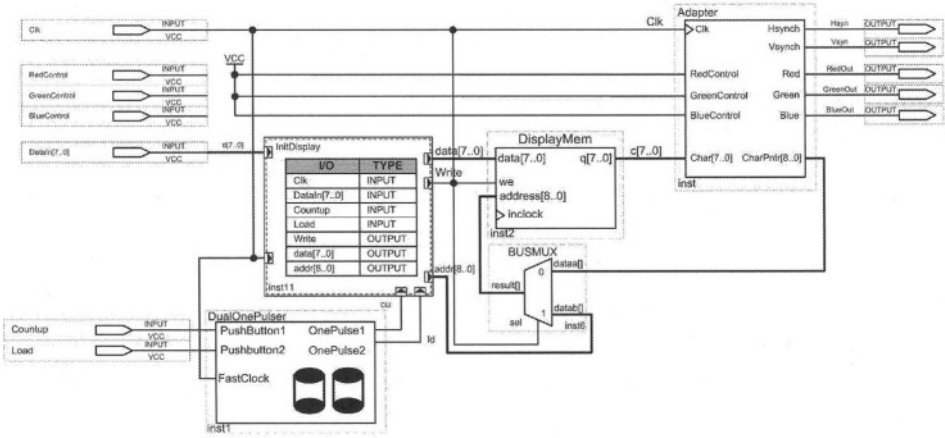


Figure 12.11 Prototyping the VGA Adapter (*Adapter*)

12.3.1 Display Memory

The *DisplayMem* block is a memory block of 512 8-bit words, of which only 300 are used. This memory is addressed by *Adapter* for reading from it, and by *InitDisplay* for writing into it. When writing into the memory, the *BUSMUX* multiplexer selects *addr* address output of *InitDisplay*. Writing into this memory is clocked, for which the main system clock is used, while reading is done asynchronous.

The display memory is designed by configuring the *LPM_RAM_DQ* megafunction of Quartus II. As with other memories, this megafunction is in the *storage* category of megafunctions. While configuring it, a memory initialization file in the *mif* format is specified. This file contains test data that

will be displayed. Also, during configuration of *DisplayMem,* input clocking, write-enable and other memory parameters will be specified.

12.3.2 Address Selection

The *BUSMUX* multiplexer that selects the read or write address of *DiplayMem* is a 9-bit 2-to-1 multiplexer. This unit is a megafunction in Quartus II.

12.3.3 Writing Display Data

The *InitDisplay* Verilog module of Figure 12.11 has a counter that counts between 0 and 299 with its *Countup* input and loads its *DataIn* input in its *data* output when *Load* is issued. We use this circuit to count to a location in *DisplayMem* and load data from UP2 switches into this memory. If *Load* is pressed twice in a row, the counter resets to location 0. Count-up and load input are taken from the UP2 pushbuttons.

```
module InitDisplay ( Clk, DataIn, Countup, Load, Write, data, addr );
    input Clk;
    input [7:0] DataIn;
    input Countup, Load;
    output Write;
    output [7:0] data;
    output [8:0] addr;

    reg Write;
    reg [8:0] addr;
    reg loaded; //two Loads in a row resets

    assign data = DataIn;

    always @(posedge Clk)
        if (Countup) begin
            addr = addr + 1;
            if (addr ==300) addr = 0;
            Write = 0;
            loaded = 0;
        end else if (Load) begin
            if (loaded) begin // perform reset
                addr = 0;
                loaded = 0;
            end else begin  // perform loading
                Write = 1;
                loaded =1;
            end
        end else Write = 0;
endmodule
```

Figure 12.12 Module for Manual Loading of Display Memory

InitDisplay is implemented in Verilog in a schematic block of Quartus II. As shown in the Verilog code of Figure 12.12, *DataIn* continuously drives *data* that

is the data to be written into the display memory. When *Countup* is issued, *addr* is incremented. When *Load* is issued, the output *Write* (write-enable) becomes active, that causes a write at the *addr* location.

12.3.4 Pushbutton Interfaces

The *DualOnePulsers* component of Figure 12.12 uses a dual debouncer (Figure 8.14) and two one-pulse generators (Figure 8.12) to generate synchronous one-clock duration pulses for every time a pushbutton is pressed.

12.3.5 Pin Assignments

In order to test out design, data and control inputs of Figure 12.11 are connected to UP2 switches and pushbuttons. The outputs of this circuit should be connected to FLEX 10K pins according to connections shown in Figure 6.36. Figure 12.13 shows these pin assignments.

12.3.6 Prototype Operation

Upon programming the FLEX 10K device, the monitor connected to VGA D-sub connector displays characters in the *DisplayMem* file. We can write any ASCII character on the display by setting its ASCII code on the FLEX switches, pressing the *Countup* pushbutton some number of times and then pressing *Load* to write the ASCII character in the counted location of the screen.

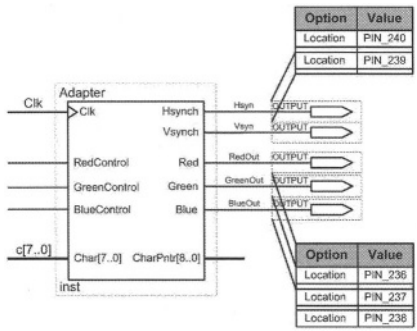


Figure 12.13 Adapter Pin Assignments to the VGA Connector

Note that our hardware only supports ASCII characters from 32 to 95, and large characters are displayed. To start from location 0, press *Load* twice.

12.4 Summary

This chapter showed a complete design by use of Verilog, schematics and megafunctions. Some features of Quartus II, such as use of memory blocks that were not discussed before, were presented in this chapter. If done

properly, the use of memory blocks is an Altera design uses FPGA memory bits that can free up a large number of logic elements for other uses. Dual-port memories cannot be implemented with FLEX memory bits and must be implemented using logic element flip-flops. Not only this is an inefficient use of logic elements, such memories cannot be initialized with memory initialization files.

In addition to presenting an elaborate use of Quartus II, this chapter showed the design of a VGA adapter. Understanding display monitors and being able to program them is important for logic designers and students in the digital field.