Laboratory IV

LCD Framebuffer

Introduction

In this laboratory you will explore different ways of creating video images. There are four projects. In the first one you will create an image "on the fly" using PAL macros to tell your code which pixel is about to be drawn. The second one will examine the values of some of the parameters that will affect the next two projects. The third one will create an image on the fly, but will perform its own synchronization with the video refresh cycle. The last project will create an image by reading pixel values from RAM in real time, thus implementing a classic Framebuffer design. All four projects will be written so they operate both in simulation and as FPGA configurations.

Lab Activities

1. **Draw an Image Using PAL Macros for Synchronization**
2. **Display VideoOut Parameter Values**
3. **Draw a Test Pattern by Synchronizing with HBlank**
4. **Draw an Image From a Framebuffer**
5. **Submit a Report of Your Lab Activities**

Draw an Image Using PAL Macros for Synchronization

Create a workspace named "Laboratory IV" and create a project in it named "SevenSeg Display." Delete all configurations for the project except Debug and EDIF. Set up those configurations in the usual way. You won't need anything but the standard header files and libraries for this project[1]. You won't need any conditional compilation in your code for this project, or any of the other projects in this Laboratory, either.

For this project you are to write a Handel-C program named *sevenseg.hcc* that reads from the keyboard and writes their ASCII codes, in hexadecimal, on two seven segment displays. The special feature about this project is that you are to draw the seven segment displays on the LCD screen in addition to the real seven segment displays. The LCD display would look something like Figure 1. This is a crude image for representing seven segment displays, but as we shall see, the fact that all the edges of the segments are either horizontal or vertical will make this project easier to manage.

One thread is to read characters and to use their hex values to assign values to two eight-bit variables that tell which segments to illuminate. You can do this by table lookup from a ROM: `rom unsigned 8 hex2segments[16] = { 0b00111111, 0b00000110, … };` Name the segments of a seven segment display as in Figure 2, which corresponds to the naming convention in the documentation for the RC200E. You won't be drawing the decimal point for this project, but it's included in Figure 2 to be consistent with the *PalSevenSegWriteShape()* macro (see below). To draw the character '0' you need to illuminate all segments except the middle one, Segment *g*. So set hex2segments[0] to

[1] When you do keyboard and console I/O you need the standard "Pal Cores" headers and libraries for those devices, of course. But be sure to omit the console files when doing projects that use *PalVideoOut* macros.

0b00111111, with the convention that bit position 0 tells whether to turn Segment a off or on, bit position 1 tells whether to turn Segment b off or on, etc. The leftmost bit (bit position 7) controls the decimal point on the RC200E seven segment displays.

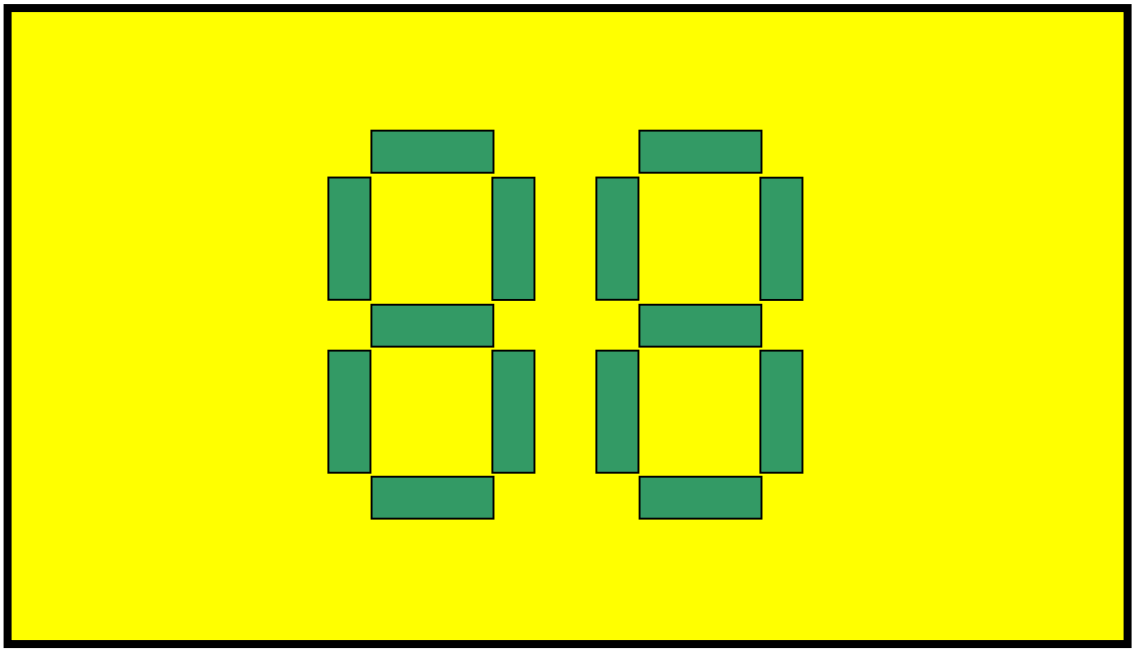


Figure 1. LCD display showing 0x88. Your colors may vary!

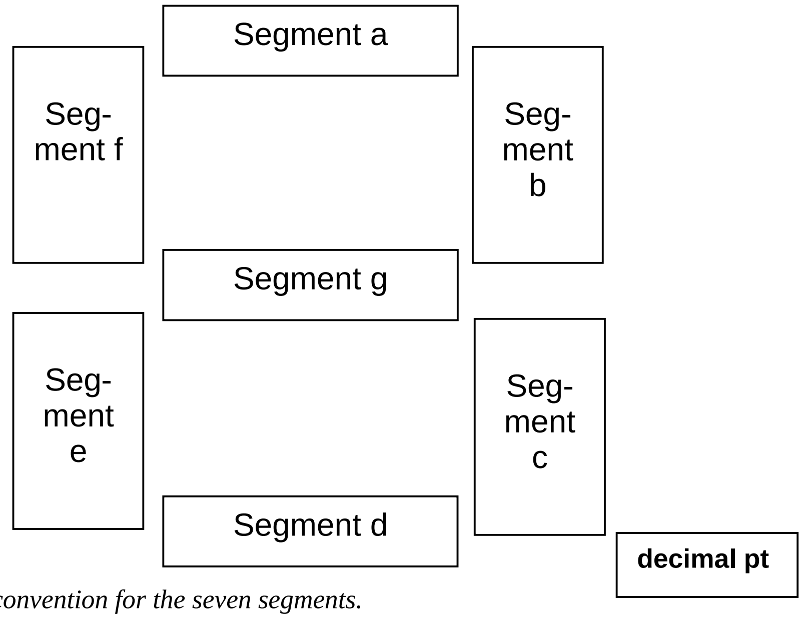


Figure 2. Naming convention for the seven segments.

The idea is that you can do two table lookups from *hex2segments* to determine what segments to turn on (or not) each time you read a character from the keyboard. One thread of your project should update sixteen bits of global information (8 bits each for the two displays) each time the user presses a key on the keyboard. Use the state of these bits as arguments to two calls to *PalSevenSegWriteShape()* to display the ASCII codes for the characters typed.

Moving on to the video part of the project, code another thread that repeatedly determines the coordinates of the pixel currently being refreshed on the LCD, and decides what color to display there. For this project, you need to synchronize your code with the LCD refresh cycle by writing to the screen on the next clock cycle after you determine the current coordinate position. Later projects will deal with the issue of synchronizing your code with the specific position in the refresh cycle more carefully. The macros, *PalVideoOutGetX(handle)* and *PalVideoOutGetY(handle)* return the current X and Y coordinate values, and you can use them repeatedly in a single expression without using any clock cycles. Your code will undoubtedly consist of one big *if* statement that tests if the coordinates are inside one of the currently "on" segments of either seven-segment display or not:

```
if ( /* fourteen tests OR'd together */ )
  PalVideoOutWrite( foreground color )
else
  PalVideoOutWrite( background color )
```

You probably want to make the sides of each segment vertical or horizontal to make it easy to test whether a point is inside a particular segment or not. With proper macro expressions defined, your fourteen tests would include code like, " … ((left_seg_a == 1) && (x > left_seg_a_left) && (x < left_seg_a_right) && (y > left_seg_a_top) && y < (left_seg_a_bottom)) || … ." In this code, I've assumed that *x* is a macro expression equivalent to *PalVideoOutGetX()* and *y* is equivalent to *PalVideoOutGetY()*.

Be sure your code works correctly both for Debug and EDIF configurations.

Display VideoOut Parameter Values

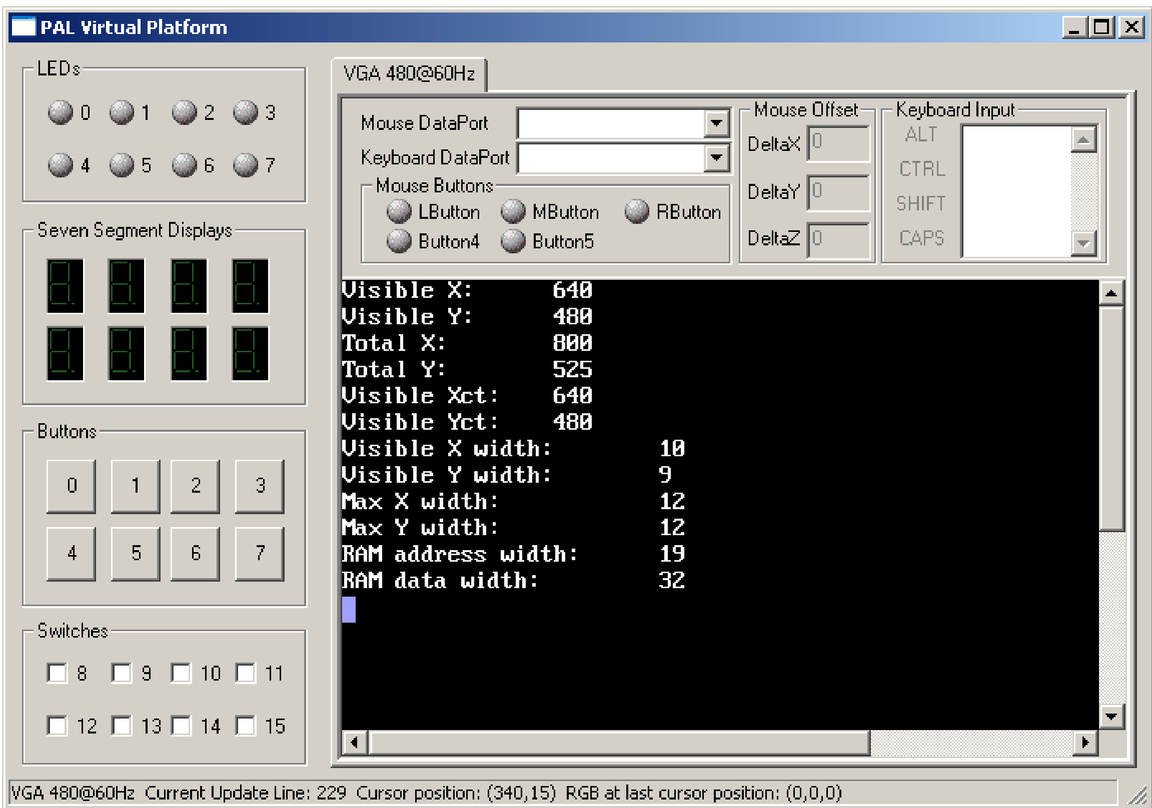
Create a second project called VideoOut_Parameters and configure it for both the RC200E and for Simulation. You will need to include the *pal_console.hch* header file in your source code and the *pal_console.hcl* library file in your Linker list for this project because you will be writing output using the LCD screen as a console. Write a Handel-C program called *parameters.hcc* that displays the following parameters on the PalConsole at run time:

- The number of visible pixels per scan line, determined at compile time.
- The number of visible scan lines, determined at compile time.
- The number of visible pixels per scan line, determined at run time.
- The number of visible scan lines, determined at run time.
- The total number of pixels per scan line.
- The total number of scan lines.
- The width (number of bits) needed for a variable that holds the X coordinate of a visible pixel.

- The width needed for a variable that holds the Y coordinate of a visible pixel.
- The width needed for a variable that ranges over all X coordinates on a scan line.
- The width needed for a variable that ranges over all Y coordinates on the display.
- The width needed for a variable that holds a RAM memory address.
- The width needed for a variable that holds a word of RAM data.

Be sure to use the appropriate PAL macros to determine all these parameter values. For example, some of the widths are not what you might expect them to be. Run the program both in simulation mode and on the RC200E, and note the differences between the two sets of values. Be sure to report these results in the Discussion section of you lab report, perhaps as tabular data.

Sample Output from parameters.hcc



Draw a Test Pattern by Synchronizing with HBlank

Your third project is to be named TestPattern, and the Handel-C source file should be called *test_pattern.hcc*. This project will use the PalVideoOut macros for writing to the screen instead of the PalConsole macros.

Use *PalVideoOutGetHBlank( )* to synchronize your code with the beginning of each scan line. Draw a white pixel in the first column of each scan line, and draw vertical bars of alternating colors across the remainder of each line. The requirements for this project can be accomplished quite easily because (1) It's all right to write to the LCD during the invisible portions of the refresh cycle (nothing happens), and (2) if you write white pixels during all the clock cycles when HBlank is true, the last one you write will show up in column one of the next scan line.

An easy way to draw alternating colors on a scan line is simply to test one bit position of the register you use to keep track of the current X position on the scan line. If you test bit 0 (the rightmost bit), the bars will be one pixel wide; if you test bit 1, the bars will be two pixels wide, etc.

Be sure your program works equally well when for both the Debug and EDIF build configurations.

Draw an Image From a Framebuffer

The fourth project is to synchronize your code with both the horizontal and vertical synchronization signals for VideoOut and to use one of the PL1 RAM banks on the RC200E as a framebuffer to hold an image to be displayed. To make debugging possible, the project draws a single, static image on the LCD display. Create a project named Framebuffer, and add *framebuffer.hcc* to it. Configure it both for Debug and EDIF.

Use one of the RC200E's PL1 memory banks as a framebuffer. Decide how to map pixel coordinates to memory addresses, and write a test pattern into RAM. The test pattern is to consist of a one-pixel wide white border along all four edges of the visible part of the display. Inside the border, draw vertical bars of alternating colors. Use the simulated version to verify that the correct pixel values are being stored in the correct memory locations. (*Note: The PAL virtual console shows byte addresses, not word addresses.)*

Now write code that continuously reads from the framebuffer and draws the pixels to their proper locations on the screen. Because of the delays involved in reading from the memory and writing to the screen, you will have to start processing each scan line during the end of the HBlank period of the previous scan line.

Use macro expressions to parameterize your code so that it does not directly use any of the numerical values you looked at in the Parameters project. But be sure your code will work correctly regardless of whether it is being simulated or configured for the FPGA. You are to meet this requirement using the proper macros to determine parameter values for Debug and EDIF rather than using conditional compilation.

You will need to pipeline your operations, staring 3 clocks before the beginning of a scan line. If *t(0)* refers to the clock period during which a pixel is displayed in the leftmost column of the display (*i.e.*, the first clock cycle after HBlank goes false), the first few stages of the pipeline can be represented as shown in Table 1. In this table, "set address" refers to setting the address for reading from the PL1 RAM, "read pixel" means to read the binary number giving the RGB values for a pixel from the RAM, "write pixel" means to write the RGB values for a pixel to the LCD, and "pixel *n* displays" means that the new pixel color becomes visible.

t(-3)	Set address for pixel 0			
t(-2)	Set address for pixel 1	Read pixel 0		
t(-1)	Set address for pixel 2	Read pixel 1	Write pixel 0	
t(0)	Set address for pixel 3	Read pixel 2	Write pixel 1	Pixel 0 displays
t(1)	Set address for pixel 4	Read pixel 3	Write pixel 2	Pixel 1 displays

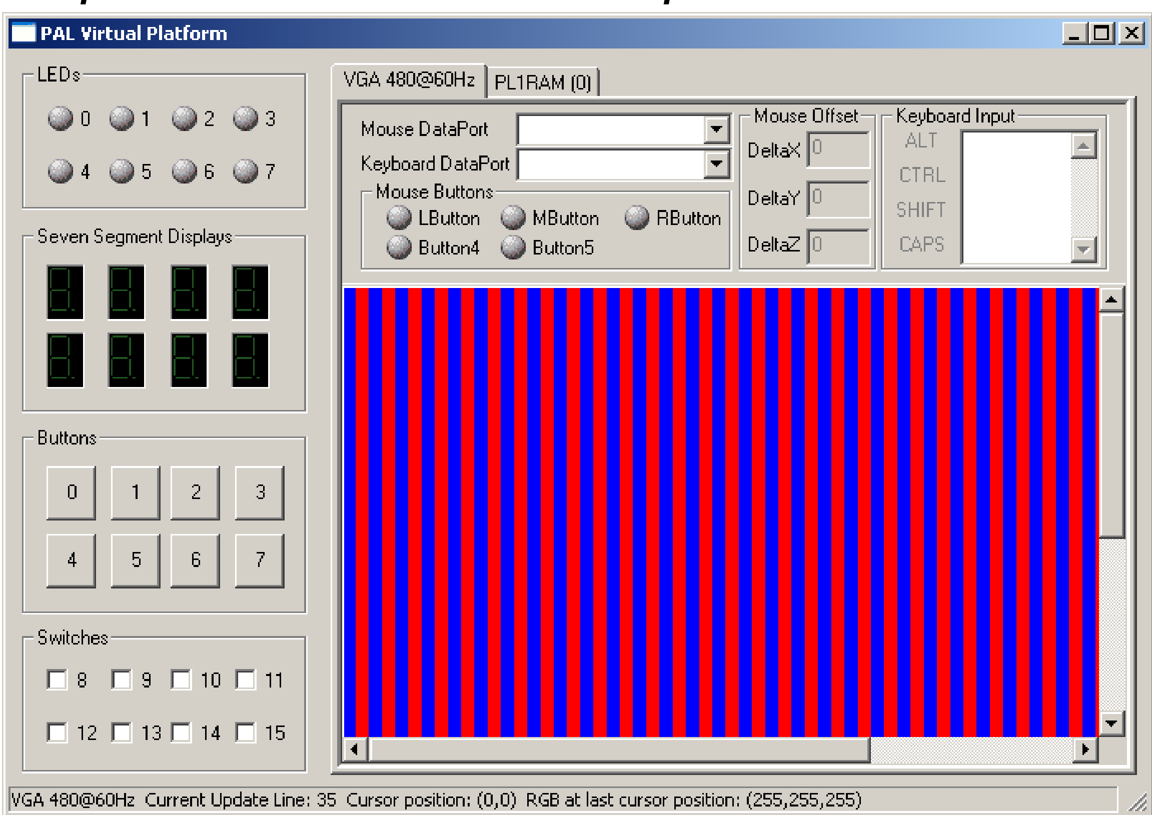
Table 1. Pipeline timing.

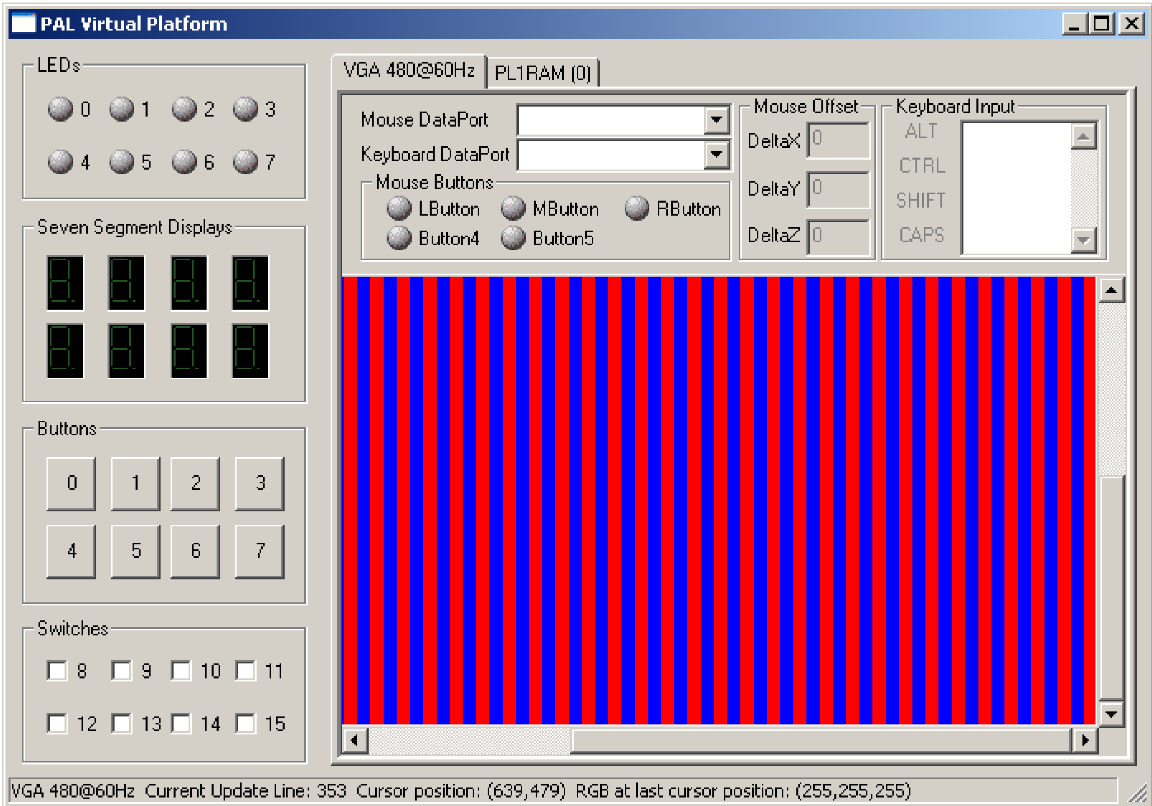
When you test your code on the simulator, you can position the mouse over the simulated display to see what pixel value has been written to each position. Verify that all the pixels in the first and last rows and first and last columns -- and no others -- are white.

> *There is a bug in the simulator. If you make the Pal Virtual Console big enough to see the entire screen at once, it chops off the right and bottom edges so you can't see those borders. However, if leave the window small and use the scroll bars to look at those edges, you can see them.*

Be sure your code generates "pixel perfect" images on both the simulated console and on the actual RC200E! Note that the screen shots below don't show the cursor crosshair, but the status lines show that the pixel values the cursor was on do have the correct values.

Sample Framebuffer Test Pattern Output





Optional: Write a macro expression or function named *write_pixel (x, y, pixel)* that writes pixel values into the framebuffer. This code must block during periods when the memory is being used to update the display to avoid mangling the contents of the RAM. Use *write_pixel( )* to show the ASCII codes for keyboard characters by drawing the images of two seven-segment displays on the console, as in the SevenSeg project.

Submit a Report of Your Lab Activities

Use a word processor to write a report of your lab activities that follows the format for lab reports discussed in class. Email it to me by midnight of the due date.