LCD Backgrounds:
Image Creation Using FPGA Logic

Version 2.3   September 13, 2025

Textbook for the Course
Logic Systems and Processors

Richard Šusta

Department of Control
Engineering
CTU-FEE in Prague

Home page of this document and LCD source code: https://dcenet.fel.cvut.cz/edu/fpga/guides.aspx
GHDL installation manual: https://dcenet.fel.cvut.cz/edu/fpga/install_en.aspx
FPGA-LCD Tools mentioned here: https://github.com/cvut/FPGA-LCD_Utils

Author:	Richard Susta, richard@susta.cz, https://susta.cz/
Figures:	Richard Susta
Publisher:	Department of Control Engineering CTU-FEE in Prague, Technicka 2, 166 00 Prague 6 https://control.fel.cvut.cz/en
Datum of issue:	September 2025

Contents

Introduction

Configurable logic elements, the main components of FPGA circuits, can create some images more efficiently than when loading them from BMP, JPEG, or PNG files. On the other hand, these file formats store many more scenes better. However, if we are drawing the background of our LCD control panel, it is entirely up to us how we design it. We can therefore compose it from shapes in which logic excels.

The image below shows an example of background of 800x480 pixels, which would be stored in a 33,817-byte JPEG file at 80% quality or in a lossless compressed 7,384-byte PNG (Portable Network Graphics) file, whose methods are more refined for graphics with repeating motifs.

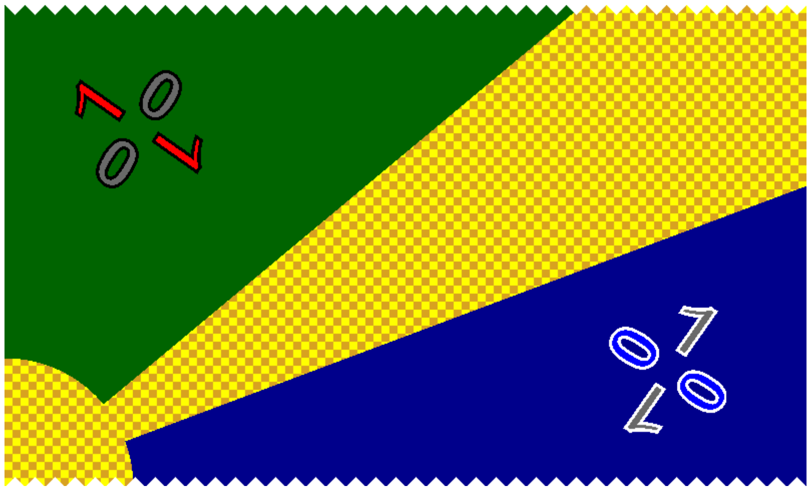


Figure 1 - Example of an LCD background created by logic

When the background is implemented using logic, it only requires 339 LE (logic elements). One LE stores 2 bytes, so it took up the equivalent of approximately 680 bytes. In addition, it needs another 4096 bytes of ROM memory to create symbols with 0 and 1 digits. In total, the background created by logic consumed the equivalent of approximately **4800** bytes, or roughly 2/3 of the size of a PNG file.

However, saving a third is not the decisive factor here. PNG and JPEG images are not encoded as continuous arrays of pixels. Their decompression involves several steps, in which various parts of the bitmap are filled in and rewritten, so the entire bitmap must be stored in memory. The test background in the figure above would require an additional 240 kilobytes of FPGA memory for its depacking, even with economical encoding colors as four-bit indexes into a palette. In total, it takes up 51 times more than the logic needs. And decompressing the image will slow down the processor on which its complex algorithm must be implemented.

In addition, the logical solution sends pixels as *a stream* of bits, which is precisely how an LCD panel works. We can transfer them directly to the panel.

For completeness, we must mention RLE (Run-Length Encoding), the sub-step of JPEG compression that can be easily implemented in hardware and output stream of bits as the LCD requires. Optimal RLE compression of the image above uses almost 36 kilobytes. Of course, we can apply the RLE method only to parts of our image if we have enough free space in the FPGA. And our assignment allows it! The future version V3.0 of the FPGA Utils will offer an option for converting an image to RLE.

However, RLE compression lacks any possibility of changes. The RLE reader can only display the image. If we create a motif using logic, we can dynamically modify it according to the input data.

We have created templates of graphic motifs together with VHDL codes that render them as inspiration, demonstrating some logic possibilities. All of them have been tested on the Veek-MT2 development board from Terasic, but they can also be adapted for other FPGAs and their boards.

LCD circuits version 2

Image creation was tested in the Quartus Lite development environment. The middle entity in the schema below, LCDlogic0, is analogous to drawing. It obtains synchronization signals and coordinates of pixels from the generator. It assigns a color to each coordinate and sends it to the register connected to the LCD panel.

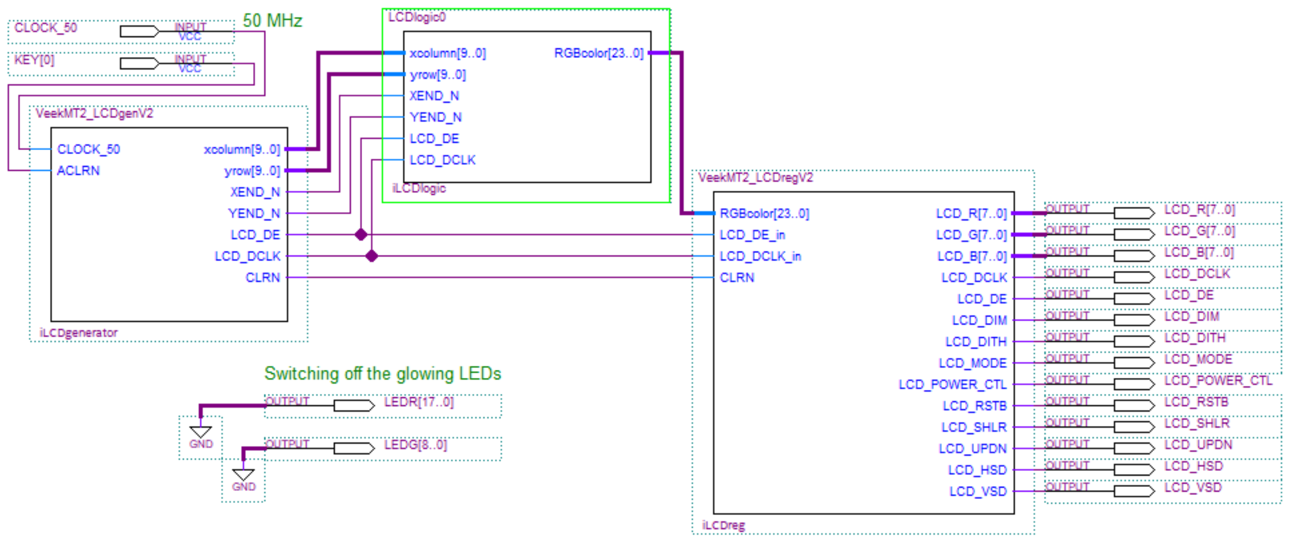


Figure 2 - Basic circuit for image creation

Note 1: This document has an attachment with sample code containing all three circuits in VHDL, namely VeekMT2_LCDgenV2, VeekMT2_LCDregV2, and the default LCDlogic0.

Note 2: The proposed schema works similarly to processor pipelines, where the clocks time their stages. The synchronization generator sends the coordinates x and y of a current pixel. In the next stage, this pixel is assigned a color in LCDlogic0. The color is loaded into the register and sent to the LCD panel. In the meantime, the two previous stages have already been completed for the next pixels.

Clocks	1: VeekMT2_LCDgenV2	2: LCDlogic0	3: VeekMT2_LCDregV2
Power-Up	-	-	-
Clock 1	Pixel coordinates [x,y]=**[0,0]**	-	-
Clock 2	Pixel coordinates [x,y]=**[1,0]**	assigns the color **[0,0]**	-
Clock 3	Pixel coordinates [x,y]=**[2,0]**	assigns color **[1,0]**	Color **[0,0]**→ LCD
Clock 4	Pixel coordinates [x,y]=**[3,0]**	assigns color **[2,0]**	Color **[1,0]**→ n LCD

LCDpackV2.vhd – definition library

LCDpackV2.vhd is the VHDL package. This document supposes its **version 2.1**, or higher with additional definitions. It is in its header, and V2.1 is upward compatible with V2.0. The package defines constants and functions for LCD panel geometry and converting colors. It is referenced in all subsequent VHDL codes.

Its main definitions:

```vhdl
constant LCD_WIDTH : integer := 800;          -- the visible part of LCD screen, the xcolumn axis
constant LCD_HEIGHT : integer := 480;         -- the visible part of LCD screen, the yrow axis
constant XCOLUMN_MAX : integer :=1023;        -- max. xcolumn lies in invisible part
constant YROW_MAX    : integer := 524;        -- max. yrow lies in invisible part
subtype  xy_t is unsigned(9 downto 0);        --xcolumn and yrow data sent by LCDgenV2
constant XY_ZERO : xy_t := (others=>'0');
subtype RGB_t is std_logic_vector(23 downto 0);  -- R G B color, R:23..16, G:15..8, B:7..0
function ToRGB(r, g, b:natural) return RGB_t;
-- + color constants for 16 named web colors, aka (i.e., also known as) 16 Windows colors:
--     AQUA, BLACK, BLUE, GRAY, GREEN, LIME,  OLIVE, MAROON,
--    NAVY, PURPLE, RED, SILVER, TEAL, VIOLET, WHITE, YELLOW
```

Note: Packages are explained in CircuitDesignWithVHDL_dataflow_and_structural_eng_V10.pdf, *Chapter 7, pages 58 to 62.*

VeekMT2_LCDgenV2 — Synchronization Generator

The VeekMT2_LCDgenV2 generator contains a pair of counters and comparators for their values. It has a simple connection, which mainly follows the timing according to the hardware specifications in the LCD panel manufacturer's catalog.

We would write an analogy of the generator in C using two cycles:

```
unsigned short int xcolumn, yrow;
unsigned int color; bool LCD_DE, XEND_N, YEND_N;
for (yrow = 0; yrow < 525; yrow++)
{ for (xcolumn = 0; xcolumn <1024; xcolumn++)
    { LCD_DE = xcolumn>=800 || yrow>=480 ? 0 : 1;
      XEND_N = xcolumn==1023 ? 0 : 1;     YEND_N = yrow==524 ? 0 : 1;
              color = LCDlogic0( xcolumn, yrow, XEND_N, YEND_N, LCD_DE);
    }     // In hardware, LCDlogic0 function is created by logic
}
```

Input

- CLOCK_50 – 50 MHz frequency input from the pin of the same name on the development board. The generator must be connected directly to it without any logic inserted, as required by the PLL (Phase-locked loop), which is embedded in it and changes the frequency from 50 Hz to 33 MHz. All electronics operating at higher frequencies usually contain some PLLs, and by adjusting their parameters, it is possible to overclock processors or graphics cards, for example.
- ACLRN — initialization after power-on. On the VEEK-MT2 board, it is connected to KEY[0].

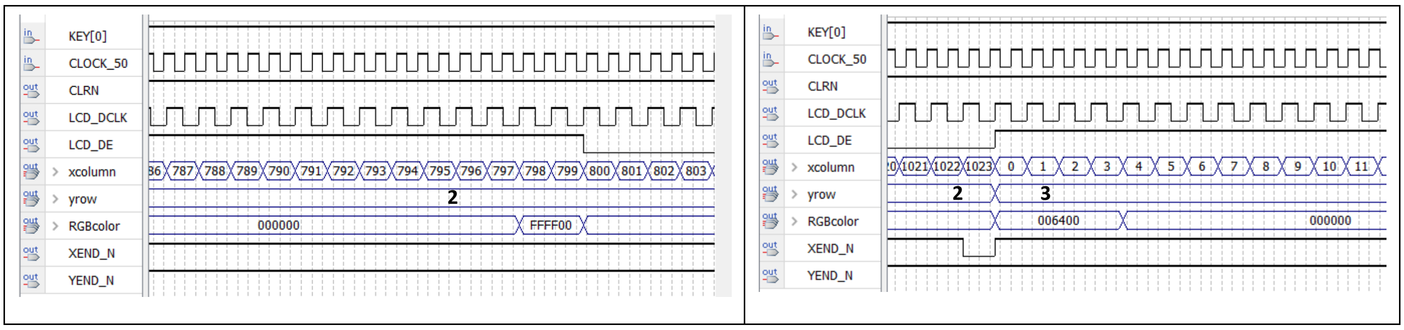


Figure 3 - Simulation of LCDgenV2 output

VeekMT2_LCDregV2 — sending color to LCD

At the rising edge of LCD_DCLK, the register stores the color assigned by the LCDlogic0 circuit. Its outputs are connected to the large rear LCD of the Veek-MT2 board. The register also crops the image so that LCD_R, LCD_G, and LCD_B outputs are at 0 (black) when LCD_DE='0', as the LCD panel requires.

*Note: The output pins of VeekMT2_LCDregister were generated automatically in the *.bdf schematic via the context menu of its symbol by selecting* Generate Pins for Symbol Ports.

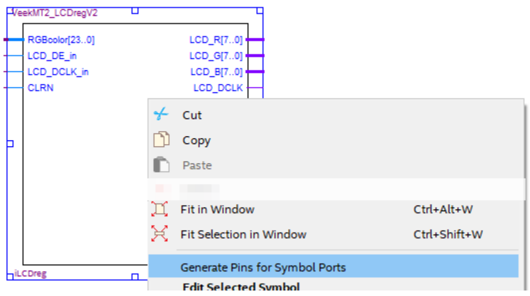


Figure 4 - Generate Pins for Symbol Ports

LCDlogic0 — the image drawing circuit

LCDlogic receives the unsigned **xcolumn** and **yrow** coordinates from the LCD synchronization generator, with the x and y axes orientations corresponding to Windows graphics.

It contains combinational logic that assigns an the RGBcolor variable for the current x, y pixel. Its prototype, LCDlogic0, can be found in the ZIP file along with the generator and register.

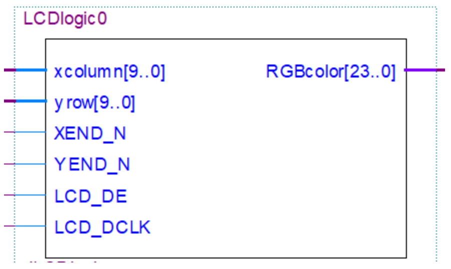


Figure 5 - LCDlogic combination circuit

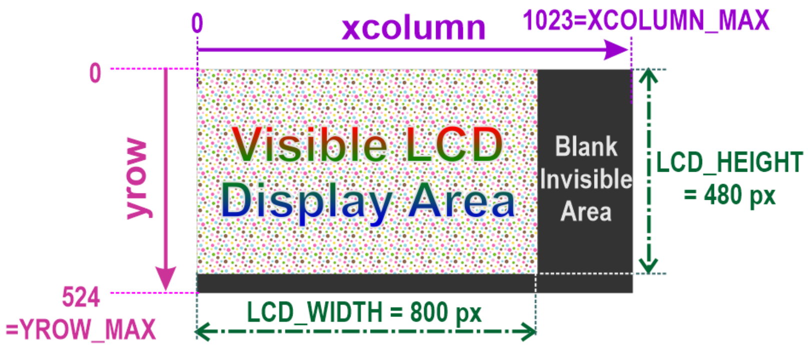


Figure 6 - Dimensions of the touchscreen LCD

LCDlogic inputs

xcolumn, yrow - 10-bit pixel coordinate signals have the unsigned type xy_t introduced in the LCDpackV2.vhd package, which defines the other constants listed below.

The **xcolumn** column varies from 0 to **1023=XCOLUMN_MAX**, but the visible image only lies in the range 0 to **799=LCD_WIDTH-1**.

The yrow row varies from 0 to **524=YROW_MAX**, but the visible part will only be in the range 0 to **479=LCD_HEIGHT-1**.

XEND_N is logical '0' when xcolumn=1023; otherwise '1'**.** It signals the last column.

It has a frequency of **32.2** kHz= 33 MHz/1024 = 33 MHz/(XCOLUMN_MAX+1)

YEND_N is logical '0' when yrow=524, otherwise '1'**.** It signals the last line of the frame.

...It has a frequency of **61.4** Hz= 33 MHz/(1024*525)=33 MHz/((XCOLUMN_MAX+1)*(YROW_MAX+1))

LCD_DE is the LCD Data Enable synchronization signal. On LCD_DE= '1', pixels belonging to the visible area are sent. In columns 800 to 1023 and rows 480 to 524, i.e., outside the visible region, the signal LCD_DE='0'. The LCD needs these invisible parts to write the image row and prepare to receive the following row or frame. The manufacturer's manual defines intervals in which LCD_DE must remain '0' and the color must be black. VeekMT2_LCDregister performs this cropping.

LCD_DCLK — LCD Data Clock has a frequency of exactly 33 MHz, with a duty cycle of 50%.

Output

RGBcolor - 24-bit std_logic_vector with 8-bit RGB color values. The R color is in the upper eight bits, and the B color is in the lower eight. *Note: RGB is without an alpha channel with opacity information. LCDs do not generally know transparency. Opacities are only used during graphic processing. Its results are without the alpha channel.*

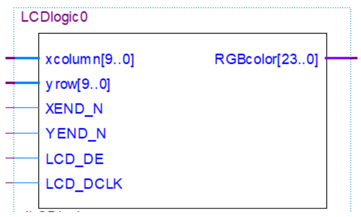
File testbenchV2_LCDlogic.vhd

The module simulates drawing and saves pixel colors in compressed form to a text file that can be loaded from Testbench Viewer from FPGA-LCD Utils to display the created LCD image.

The testbench contains its own synchronization generator in an optimized form for simulation and also its own register. Only LCDlogic* is inserted into it.

The code prototype

We will start by specifying the inputs and outputs using types from LCDpackV2. We suppose its version 2.1 and higher that contains assignIf functions.



We will use the following definitions from the LCDpackV2 package:

```
subtype  xy_t is unsigned(9 downto 0); --for data xcolumn and yrow
constant XY_ZERO : xy_t := (others=> '0');
subtype RGB_t is std_logic_vector(23 downto 0); -- R G B color, R:23..16, G:15..8, B:7..0
```

Let's write the entity and architecture:

```
library ieee, work; use ieee.std_logic_1164.all;  use ieee.numeric_std.all; -- for integer and unsigned types
use work.LCDpackV2.all;
entity LCDlogic0 is
   port(xcolumn, yrow  : in  xy_t  := XY_ZERO; -- x, y-coordinates of pixel (column, row indexes)
        XEND_N   : in  std_logic   := '0'; -- '0' only when xcolumn=XCOLUMN_MAX, otherwise '1; frequency
                                         -- 32,2 kHz = LCD_DCKL/1024 = LCD_DCKL/(XCOLUMN_MAX+1)
        YEND_N   : in  std_logic  := '0'; --'0' only when yrow=YROW_MAX, otherwise '1'; frequency
                              --61,4 Hz =LCD_DCKL/(1024*525) = LCD_DCKL/((XCOLUMN_MAX+1)*(YROW_MAX+1))
        LCD_DE   : in  std_logic   := '0';   -- DataEnable indicates the visible part of LCD
        LCD_DCLK : in  std_logic := '0'; -- 33 MHz exactly; LCD data clock
       RGBcolor : out RGB_t); --  defined in LCDpackV2; RGB_t = std_logic_vector(23 downto 0)
end entity;

architecture behavioral of LCDlogic0 is
  constant DARKBLUE: RGB_t := ToRGB(0, 0, 139); -- = X"00008B", adding the color not defined in LCDpackV2
  begin -- architecture

LSPimage : process( xcolumn, yrow, LCD_DE )
   variable RGB :RGB_t :=BLACK; -- the color of pixel
   variable x : integer  range 0 to XCOLUMN_MAX:=0;
   variable y : integer  range 0 to YROW_MAX:=0;
   begin -- process
    x := to_integer(xcolumn); y := to_integer(yrow); -- we convert unsigned inputs to integers
    ---------- our image ------------------------
    RGB :=DARKBLUE;
   -------------------------------------------------------------
    RGBcolor <= RGB; -- assigning the output signal
   end process;
end architecture;
```

Important notes:

1. Keep naming thinity, otherwise the code will become uncompileable. The file **LCDlogic0.vhd** contains the entity **LCDlogic0** with the behavioral architecture for **LCDlogic0**. The behavioral identifier is a local name, valid only within the entity. It can be used again in another entity.
2. The entity also contains inputs not yet referenced to increase its versatility. The compiler will omit everything unused during minimization. However, if we need another input in the future, we have it available and do not need to add it to the entity and regenerate its schematic symbol.
3. The initialization of signal and variable values in definitions is mainly for simulation. Synthesis is only performed for constant definitions or local variables in functions and procedures.
   The **process** needs initialization **with an assignment statement in its main code**.
4. The keyword **process** opens the VHDL sequential domain. LSPimage is an optional flag. It cannot be referenced in synthesis, but we will see it as a reference in simulation.
5. The lists in parentheses after the keyword **process** are not parameters, but a "sensitivity list" that lists signals whose changes can cause changes in its outputs. It is required for simulations!

The runlcd.bat file

The runlcd.bat batch file contains scripting shell language commands and is similar to runmorse.bat described in the DCENET manual: Installing and using the GHDL language, page 7. But it does not generate output for GtkWave, so the --vcd parameter is missing in "**ghdl.exe -r**". The result is displayed in the Testbench Viewer from FPGA-LCD Utils.

```
@ECHO OFF
rem SETLOCAL — the following definitions will be canceled after the batch ends. IMPORTANT, never omit!
SETLOCAL
rem The testbench file name must be without extension, because its name is also used for other components
set TBNAME=testbenchV2_LCDlogic
rem Files have extensions and relative paths to the parent directory. List them in the correct order of compilation!
set FILES=../LCDpackV2.vhd ../LCDlogic0.vhd
rem Simulation runtime in its time.
set SIMTIME=20ms
rem Move mingw64 to the top of PATH, which will only be temporary thanks to SETLOCAL

rem GHDL is compiled for VHDL-2008
set GHDL_FLAGS=-fsynopsys --std=08
@ECHO ON
ghdl.exe -a %GHDL_FLAGS% %FILES% ../%TBNAME%.vhd
@IF ERRORLEVEL 1 GOTO BAT-END
ghdl.exe -e %GHDL_FLAGS% %TBNAME%
@IF ERRORLEVEL 1 GOTO BAT-END
ghdl.exe -r %GHDL_FLAGS% %TBNAME% --stop-time=%SIMTIME%
:BAT-END
```

Its commands are explained in more detail:

`ECHO OFF` – executed commands are not displayed during processing of this *.bat file. When `ON`, commands that do not start with the @ sign are displayed.

`SETLOCAL` - runs environment variable localization. Changes are valid until the batch file's end or the corresponding ENDLOCAL command. **Without SETLOCAL, they would be permanent!**

`rem` - the following text is a comment until the end of the line.

`set TBNAME` – testbench entity, **only its name**, without extension or path. The parameter `set` is referenced as %its_name%, e.g., %TBNAME%.

`set FILES=` – file(s) from which the circuit is assembled. Spaces separate them and must be listed in the order of their compilation! Do not add a synchronization generator and register — the testbench has them.

`set SIMTIME` – the simulation should run for 16.6 ms, its internal time, then stop automatically. If it does not, there is an error somewhere, and the stop will be forced after 20 milliseconds.

`set PATH` – we only temporarily move the path to mingw64 to the beginning due to previous `SETLOCAL`

`set GHDL_FLAGS` — enable VHDL 2008 support. GHDL can handle almost all of it.

`ghdl.exe -a` – analyzes VHDL files. In its command line, FILES must precede *the TBNAME*.

`IF ERRORLEVEL 1 GOTO BAT-END` - jumps to the end if the previous command ended with an error.

`ghdl.exe -e` – creates a circuit simulation in the `TBNAME.exe` file

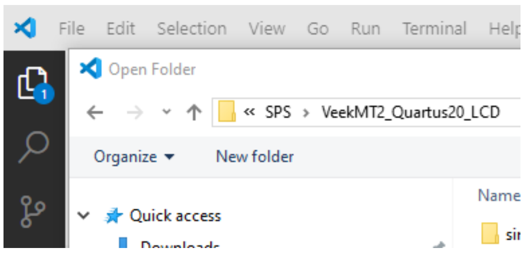
`ghdl.exe -r` – runs `TBNAME.exe`

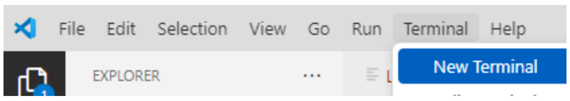
Note: If we want to simulate a different VHDL file, we change the commands **set FILES=** and, if necessary, `set TBNAME` if we are using a different testbench

Running the GHDL simulation

GHDL allows faster debugging. The Demo project contains **runlcd.bat**, which was intentionally placed in the simulation subdirectory so that all temporary files created by GHDL remain there.

First, open the project folder in the free VSC application, commonly known as Visual Studio Code. Then create a new terminal.


File Edit Selection View Go Run Terminal Help
Open Folder
« SPS › VeekMT2_Quartus20_LCD
Organize ▾ New folder
Quick access
Name



In the terminal, enter two Windows PowerShell commands:

```
PS C:\SPS\VeekMT2_Quartus20_LCD> cd .\simulation\
PS C:\SPS\VeekMT2_Quartus20_LCD\simulation> .\runlcd.bat
```

You can type cd and press the tab key, and VSC will complete the command. Similarly, after typing **./r**, you can use the tab key to complete the rest. The simulation will list the commands executed and successfully terminate with the message:

```
                        :-) OK end of SINGLE frame simulation.
.\testbenchv2_lcdlogic.exe:error: assertion failed in process .testbenchv2_lcdlogic(testbench).stimuls
.\testbenchv2_lcdlogic.exe:error: simulation failed
PS C:\SPS\VeekMT2_Quartus20_LCD\simulation>
```

We ignore the "error" messages after **:-) OK end**. Paradoxically, reporting an fatal error that everything is OK:-), is the way to stop the simulation in VHDL.

The testbench result can be viewed using the Testbench Viewer for FPGA LCD Utils.

In the prototype code (page 7), we have assigned a dark blue color to all pixels in visible and invisible areas, which can be sometimes confusing if we switch the Testbench Viewer to full-screen mode, as shown in the images below. For better orientation, you can add clipping with the **if** command.

The register does the clipping, so inserting it into LCDlogic is unnecessary, but we can do so for our own orientation. The following codes will be without it.

```
------------------------ our image ----------------------------------
RGB color<= DARKBLUE;
-----------------------------------------------------------------------
```

```
------------------------ our image ---------------------------------------------
RGB color<= DARKBLUE;
if LCD_DE = '0' then   RGBcolor <=  BLACK; end if;
-------------------------------------------------------------------------------
```

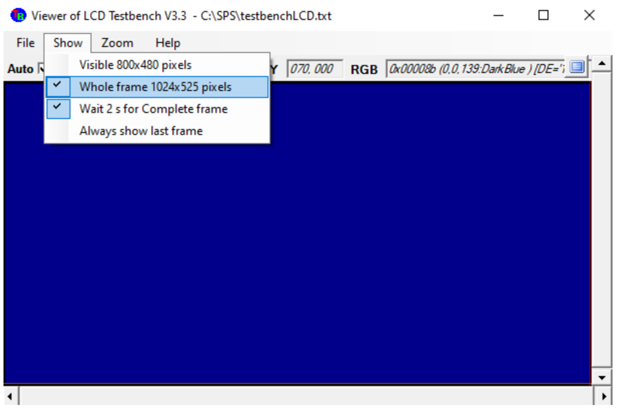

Viewer of LCD Testbench V3.3 - C:\SPS\testbenchLCD.txt
File Show Zoom Help
Auto
Visible 800x480 pixels
Whole frame 1024x525 pixels
Wait 2 s for Complete frame
Always show last frame
Y 070, 000
RGB 0x00008b (0,0,139:DarkBlue) [DE=

Figure 7- All pixels are dark blue

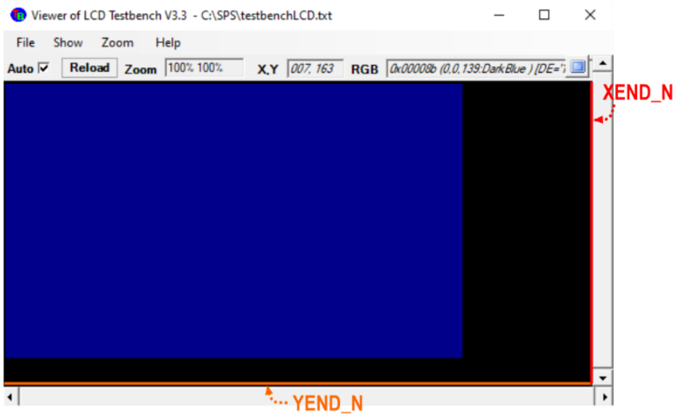
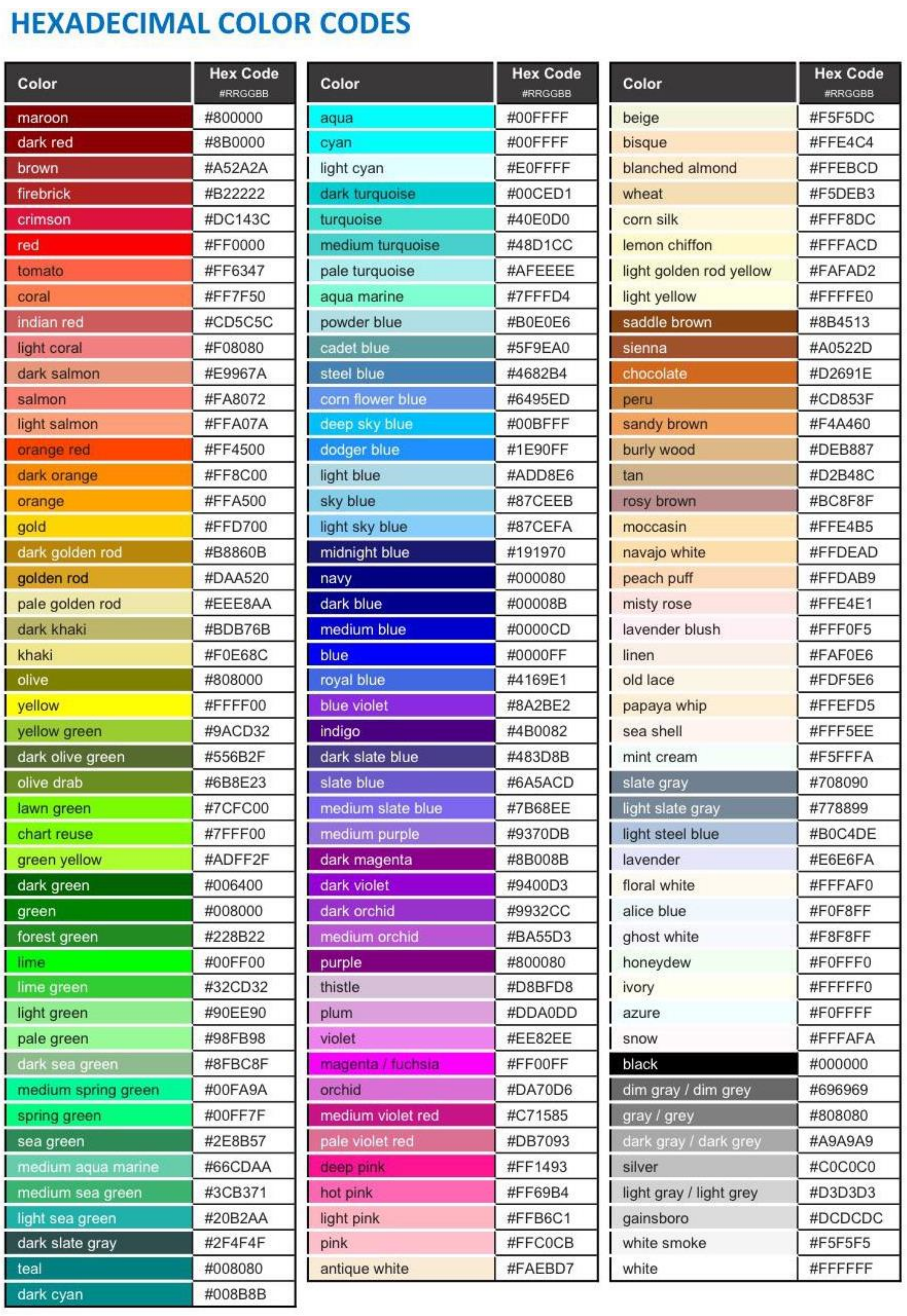

Viewer of LCD Testbench V3.3 - C:\SPS\testbenchLCD.txt
File Show Zoom Help
Auto Reload Zoom 100% 100%
X,Y 007, 163
RGB 0x00008b (0,0,139:DarkBlue) [DE=
XEND_N
YEND_N

Figure 8- Cropped
+ highlighted positions XEND_N and YEND_N

Color table

A handy table of colors organized by shade can be found on Reddit. Web hexadecimal codes are written with X in VHDL. For example, maroon color #800000 is written in VHDL as X"800000" or using the conversion function ToRGB(128, 0, 0).

Note: There are many hexadecimal formats, see the Wiki *overview*: Distinguishing from decimal



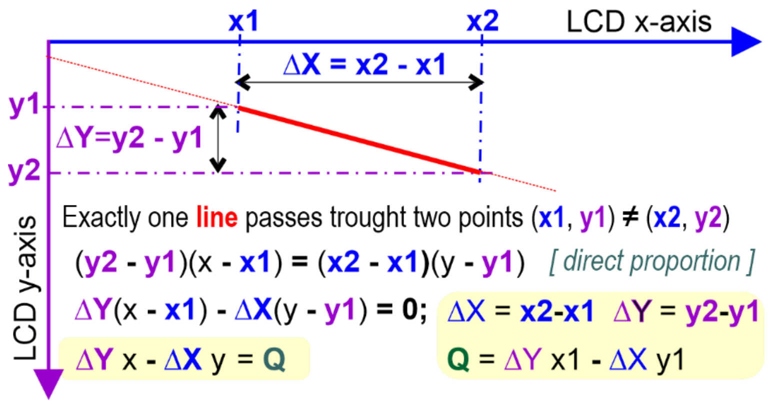
HEXADECIMAL COLOR CODES

Color	Hex Code #RRGGBB	Color	Hex Code #RRGGBB	Color	Hex Code #RRGGBB
maroon	#800000	aqua	#00FFFF	beige	#F5F5DC
dark red	#8B0000	cyan	#00FFFF	bisque	#FFE4C4
brown	#A52A2A	light cyan	#E0FFFF	blanched almond	#FFEBCD
firebrick	#B22222	dark turquoise	#00CED1	wheat	#F5DEB3
crimson	#DC143C	turquoise	#40E0D0	corn silk	#FFF8DC
red	#FF0000	medium turquoise	#48D1CC	lemon chiffon	#FFFACD
tomato	#FF6347	pale turquoise	#AFEEEE	light golden rod yellow	#FAFAD2
coral	#FF7F50	aqua marine	#7FFFD4	light yellow	#FFFFE0
indian red	#CD5C5C	powder blue	#B0E0E6	saddle brown	#8B4513
light coral	#F08080	cadet blue	#5F9EA0	sienna	#A0522D
dark salmon	#E9967A	steel blue	#4682B4	chocolate	#D2691E
salmon	#FA8072	corn flower blue	#6495ED	peru	#CD853F
light salmon	#FFA07A	deep sky blue	#00BFFF	sandy brown	#F4A460
orange red	#FF4500	dodger blue	#1E90FF	burly wood	#DEB887
dark orange	#FF8C00	light blue	#ADD8E6	tan	#D2B48C
orange	#FFA500	sky blue	#87CEEB	rosy brown	#BC8F8F
gold	#FFD700	light sky blue	#87CEFA	moccasin	#FFE4B5
dark golden rod	#B8860B	midnight blue	#191970	navajo white	#FFDEAD
golden rod	#DAA520	navy	#000080	peach puff	#FFDAB9
pale golden rod	#EEE8AA	dark blue	#00008B	misty rose	#FFE4E1
dark khaki	#BDB76B	medium blue	#0000CD	lavender blush	#FFF0F5
khaki	#F0E68C	blue	#0000FF	linen	#FAF0E6
olive	#808000	royal blue	#4169E1	old lace	#FDF5E6
yellow	#FFFF00	blue violet	#8A2BE2	papaya whip	#FFEFD5
yellow green	#9ACD32	indigo	#4B0082	sea shell	#FFF5EE
dark olive green	#556B2F	dark slate blue	#483D8B	mint cream	#F5FFFA
olive drab	#6B8E23	slate blue	#6A5ACD	slate gray	#708090
lawn green	#7CFC00	medium slate blue	#7B68EE	light slate gray	#778899
chart reuse	#7FFF00	medium purple	#9370DB	light steel blue	#B0C4DE
green yellow	#ADFF2F	dark magenta	#8B008B	lavender	#E6E6FA
dark green	#006400	dark violet	#9400D3	floral white	#FFFAF0
green	#008000	dark orchid	#9932CC	alice blue	#F0F8FF
forest green	#228B22	medium orchid	#BA55D3	ghost white	#F8F8FF
lime	#00FF00	purple	#800080	honeydew	#F0FFF0
lime green	#32CD32	thistle	#D8BFD8	ivory	#FFFFF0
light green	#90EE90	plum	#DDA0DD	azure	#F0FFFF
pale green	#98FB98	violet	#EE82EE	snow	#FFFAFA
dark sea green	#8FBC8F	magenta / fuchsia	#FF00FF	black	#000000
medium spring green	#00FA9A	orchid	#DA70D6	dim gray / dim grey	#696969
spring green	#00FF7F	medium violet red	#C71585	gray / grey	#808080
sea green	#2E8B57	pale violet red	#DB7093	dark gray / dark grey	#A9A9A9
medium aqua marine	#66CDAA	deep pink	#FF1493	silver	#C0C0C0
medium sea green	#3CB371	hot pink	#FF69B4	light gray / light grey	#D3D3D3
light sea green	#20B2AA	light pink	#FFB6C1	gainsboro	#DCDCDC
dark slate gray	#2F4F4F	pink	#FFC0CB	white smoke	#F5F5F5
teal	#008080	antique white	#FAEBD7	white	#FFFFFF
dark cyan	#008B8B				

Figure 9 - Table of named colors taken from Reddit

Templates with straight lines

The equation of a straight line passing through two different points can be derived by direct proportion. However, the hardware implementation of a straight line requires integer coefficients. It leads to a simpler circuit if the greatest common divisor gdc exceeds 1 and reduces the slope fraction to smaller numbers.



Exactly one **line** passes trought two points (x1, y1) ≠ (x2, y2)

(y2 - y1)(x - x1) = (x2 - x1)(y - y1) *[direct proportion]*

ΔY(x - x1) - ΔX(y - y1) = 0; ΔX = x2-x1 ΔY = y2-y1

ΔY x - ΔX y = Q   Q = ΔY x1 - ΔX y1

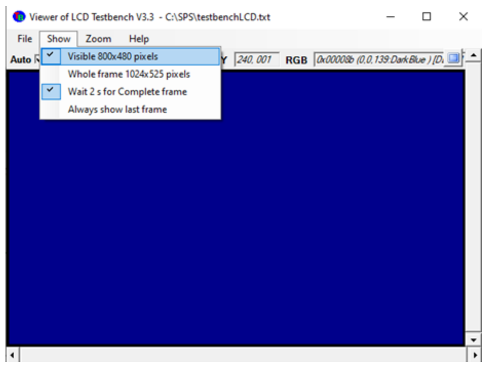
Figure 10 - Derivation of the equation of a straight line

The equations of straight lines (and also ellipses) can be found, for example, using the LCD Geometry Rulers in FPGA-LCD Utils, which are very similar to the well-known Geodebra tool, but adapted to integer results and the coordinate system of LCDs, in which the y-axis runs from top to bottom for historical reasons.

```
architecture behavioral of LCDlogic0 is
 constant DARKBLUE: RGB_t := ToRGB(0, 0, 139);
 begin -- architecture
LSPimage : process( xcolumn, yrow, LCD_DE )
  variable RGB :RGB_t :=BLACK; -- the color of xy-pixel
  variable x : integer  range 0 to XCOLUMN_MAX:=0;
  variable y : integer  range 0 to YROW_MAX:=0;
  begin   x := to_integer(xcolumn); y := to_integer(yrow);
    ---------- our image ------------------------
    RGB :=DARKBLUE;
   ---------------------------------------------------------
    RGBcolor <= RGB; -- assigning the output signal
  end process;
end architecture;
```

Result

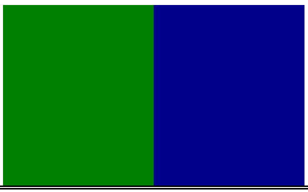
VSC


Viewer of LCD Testbench V3.3 - C:\SPS\testbenchLCD.txt
File
Show
Zoom
Help
Auto
Visible 800x480 pixels
Whole frame 1024x525 pixels
Wait 2 s for Complete frame
Always show last frame
Y
240, 001
RGB
0x00008b (0,0,139:DarkBlue)

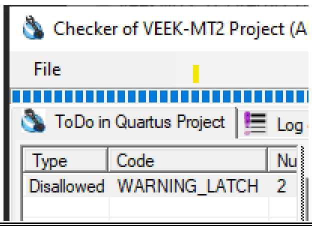
In the following codes, the lines in the "*our image*" section change only.

We will also prefer several separate **if-then** *statements. The* **if***s of higher priority assignments are placed* ***after*** *those with lower priority. We consider this style more comprehensible than long cascades of* **if-elsif-elsif***... statements. Quartus implements separate* **if-then** *just as effectively as a cascade* **of if-elsif-elsif***. Tested.*
And code typos are more frequent in cascades of if-elsif-elsif than in multiple separated if-then. Also verified by many of our students ☺

```
---------- our image ------------------------
RGB:= DARKBLUE; -- default value !!! REQUIRED !!!
if x< LCD_WIDTH/2 then RGB:=GREEN; end if;
---------------------------------------------------
```

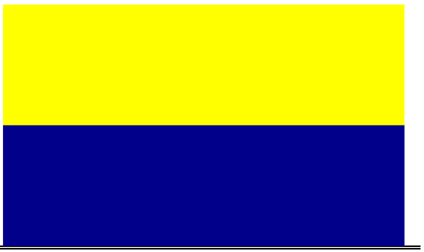


```
-- INCORRECT code without assigning the default value for RBG
---------- our image ------------------------
if x< LCD_WIDTH/2 then RGB:=GREEN; end if;
---------- LATCH in our code, RGB is not always assigned in the process code
--Compiler:Info (10041): Inferred latch for LSPimage:RGB
```

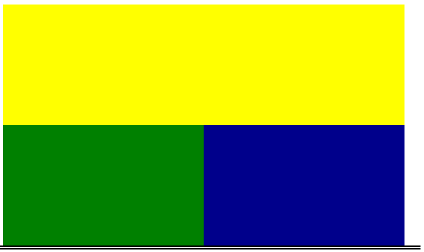


Type	Code	Nu
Disallowed	WARNING_LATCH	2

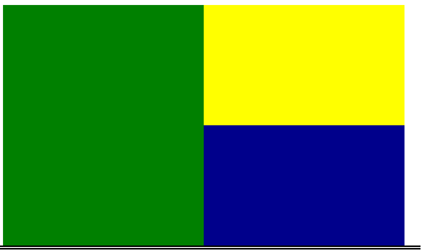
```
---------- our image -------------------------
RGB :=DARKBLUE;
if x<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
--------------------------------------------------
```



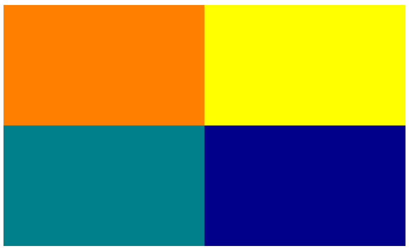
```
---------- our image -------------------------
RGB :=DARKBLUE;
if x<LCD_WIDTH/2 then RGB:=GREEN; end if;
if y<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
--------------------------------------------------
```



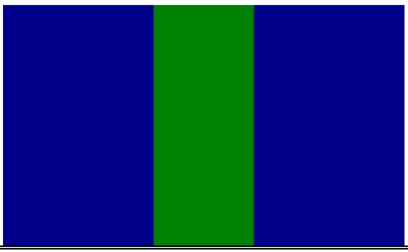
```
---------- our image -------------------------
RGB :=DARKBLUE;
if y<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
if x<LCD_WIDTH/2 then RGB:=GREEN; end if;
--------------------------------------------------
```



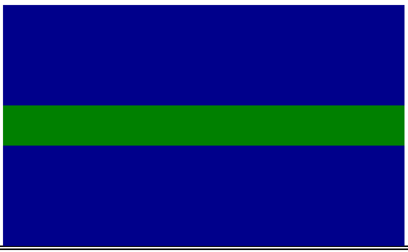
```
---------- our image -------------------------
RGB :=DARKBLUE;
if y<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
if x<LCD_WIDTH/2 then RGB:=RGB xor GREEN; end if;
--------------------------------------------------
```



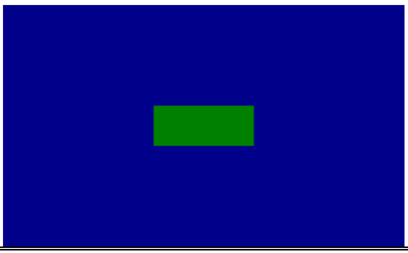
```
---------- our image -------------------------
RGB :=DARKBLUE;
if (x>=300) and (x<500) then RGB:= GREEN; end if;
--------------------------------------------------
```



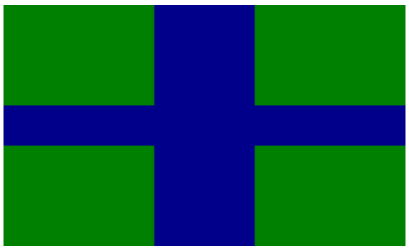
```
---------- our image -------------------------
RGB :=DARKBLUE;
if (y>=200) and (y<280) then RGB:= GREEN; end if;
--------------------------------------------------
```



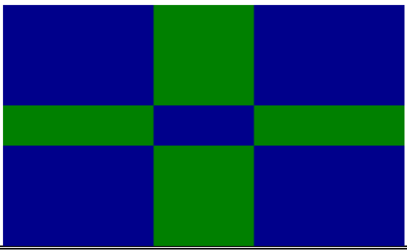
```
---------- our image -------------------------
RGB :=DARKBLUE;
if (x>=300) and (x<500) and (y>=200) and (y<280)
   then RGB:= GREEN; end if;
--------------------------------------------------
```



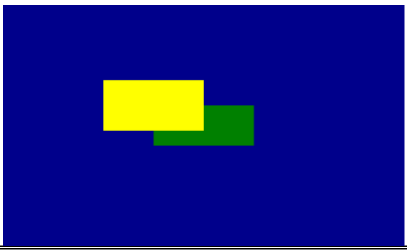
```
---------- our image -------------------------
RGB :=DARKBLUE;
if ((x<300) or (x>=500)) and ((y<200) or (y>=280))
   then RGB:= GREEN; end if;
--------------------------------------------------
```



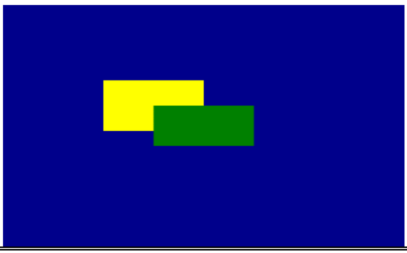
```
---------- our image ------------------------
RGB :=DARKBLUE;
if ((x<300) or (x>=500)) xor ((y<200) or (y>=280))
   then RGB:= GREEN; end if;
----------------------------------------------------
```



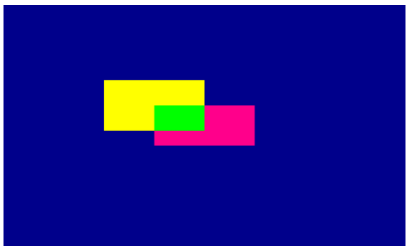
```
---------- our image ------------------------
RGB :=DARKBLUE;
if (x>=300) and (x<500) and (y>=200) and (y<280) then RGB:= GREEN; end if;
if (x>=200) and (x<400) and (y>=150) and (y<250) then RGB:= YELLOW; end if;
----------------------------------------------------
```



```
---------- our image ------------------------
RGB :=DARKBLUE;
if (x>=200) and (x<400) and (y>=150) and (y<250) then RGB:= YELLOW; end if;
if (x>=300) and (x<500) and (y>=200) and (y<280) then RGB:= GREEN; end if;
----------------------------------------------------
```

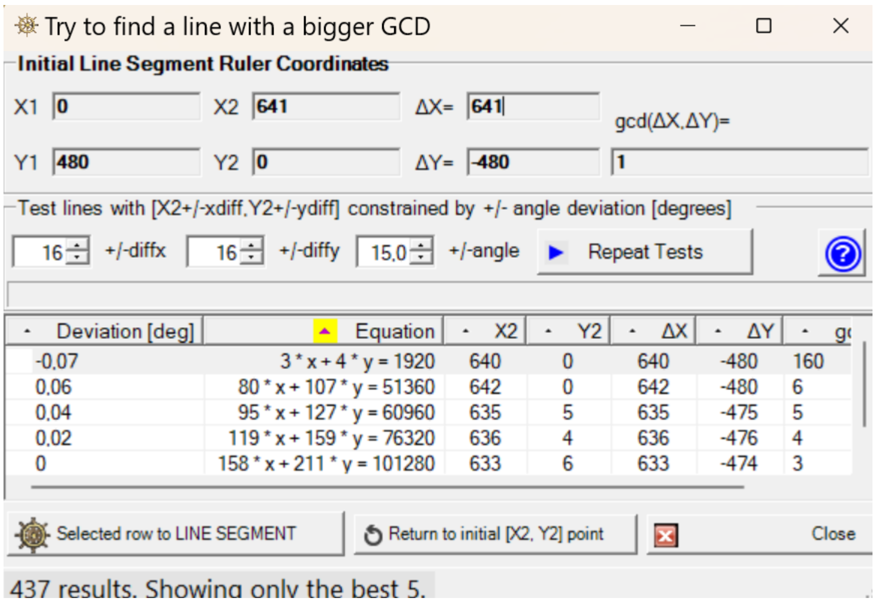


```
---------- our image ------------------------
RGB :=DARKBLUE;
if (x>=200) and (x<400) and (y>=150) and (y<250) then RGB:= YELLOW; end if;
if (x>=300) and (x<500) and (y>=200) and (y<280) then RGB:=RGB xor RED; end if;
----------------------------------------------------
```



LCD Geometry Rulers from FPGA-LCD Utils also find the coefficients of slanted line segments. Open an image with LCD dimensions of 800x480 pixels, for example, saved from Testbench Viewer. Insert our line and optimize its position (steering wheel icon). The optimizer will vary the line's end point X2, Y2 to find a greater gcd.

For example, the 641/480 slope line has gdc 1 (the greatest common divisor). The line with slope 640/480 is better and differs by only 0.07 degrees. We shorten its slope by 160 to 4/3. The implementation will multiply by smaller numbers, reducing the required logic elements.



Try to find a line with a bigger GCD

Initial Line Segment Ruler Coordinates

X1 0 X2 641 ΔX= 641 gcd(ΔX,ΔY)=

Y1 480 Y2 0 ΔY= -480 1

Test lines with [X2+/-xdiff,Y2+/-ydiff] constrained by +/- angle deviation [degrees]

16 +/-diffx 16 +/-diffy 15,0 +/-angle Repeat Tests

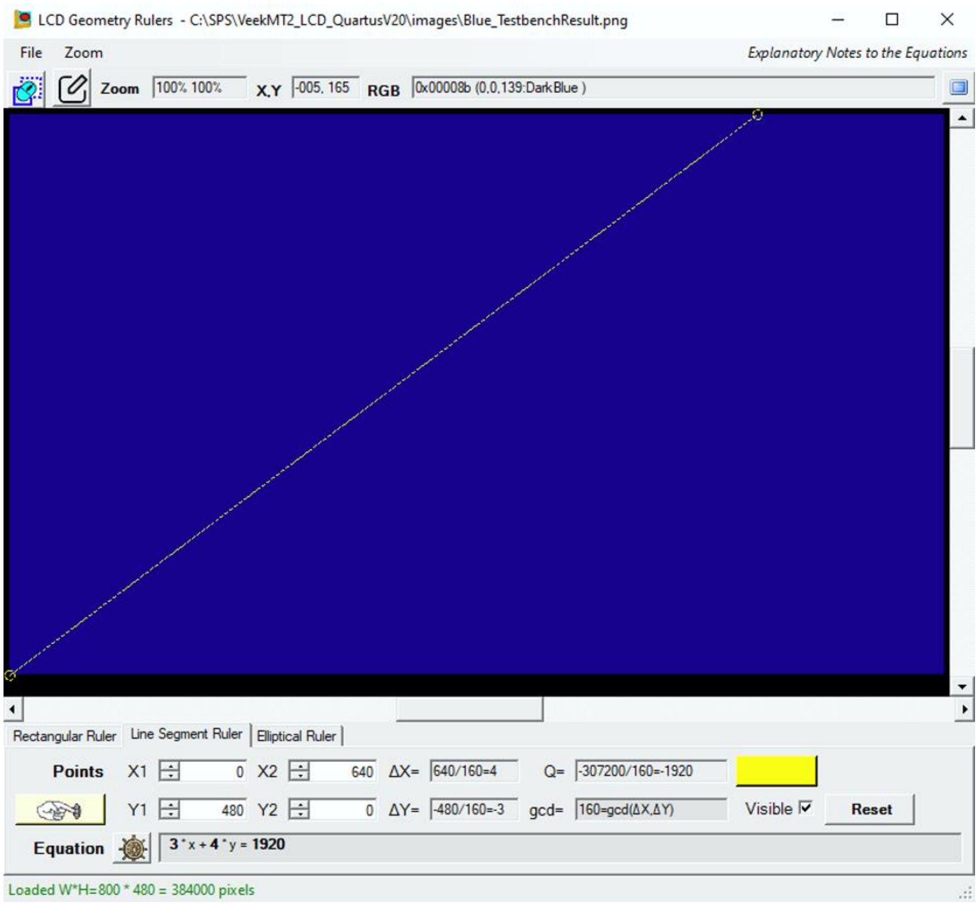
Deviation [deg]	Equation	X2	Y2	ΔX	ΔY	gc
-0,07	3 * x + 4 * y = 1920	640	0	640	-480	160
0,06	80 * x + 107 * y = 51360	642	0	642	-480	6
0,04	95 * x + 127 * y = 60960	635	5	635	-475	5
0,02	119 * x + 159 * y = 76320	636	4	636	-476	4
0	158 * x + 211 * y = 101280	633	6	633	-474	3

Selected row to LINE SEGMENT | Return to initial [X2, Y2] point | Close

437 results. Showing only the best 5.

Figure 11 - Line optimization dialog in LCD Geometry Rulers from FPGA-LCD Utils

If we replace the = equality with a suitable inequality in the line equation, it can be used as a condition for assigning a color to the entire LCD area. Combining conditions allows us to create shapes bounded by lines, as shown in the following figures, which were saved from the Testbench Viewer outputs.


LCD Geometry Rulers - C:\SPS\VeekMT2_LCD_QuartusV20\images\Blue_TestbenchResult.png

File Zoom — Explanatory Notes to the Equations

Zoom: 100% 100% | X,Y: -005, 165 | RGB: 0x00008b (0,0,139:DarkBlue)

Rectangular Ruler | Line Segment Ruler | Elliptical Ruler

Points: X1 = 0, X2 = 640, ΔX= 640/160=4, Q= -307200/160=-1920

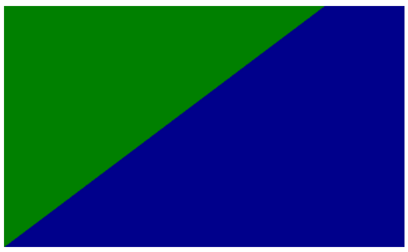
Y1 = 480, Y2 = 0, ΔY= -480/160=-3, gcd= 160=gcd(ΔX,ΔY), Visible ☑, Reset

Equation: 3 * x + 4 * y = 1920

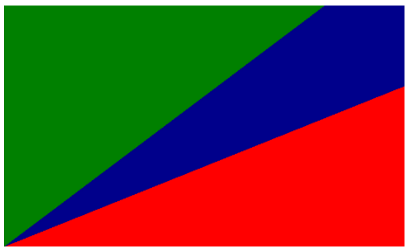
Loaded W*H=800 * 480 = 384000 pixels

Figure 12 - LCD Geometry Rulers — Optimal line equation

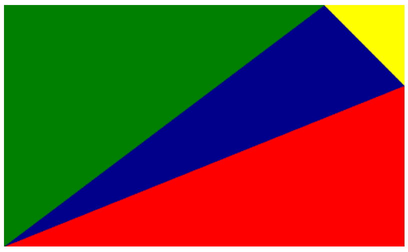
```
---------- our image -------------------------
RGB:=DARKBLUE;
if  3 * x + 4 * y < 1920  then RGB:=GREEN; end if;
-------------------------------------------------------------
```



```
---------- our image -----------------------------
RGB:=DARKBLUE;
if  3 * x + 4 * y < 1920  then RGB:=GREEN; end if;
if  2 * x + 5 * y > 2400  then RGB:=RED; end if;
--------------------------------------------------
```

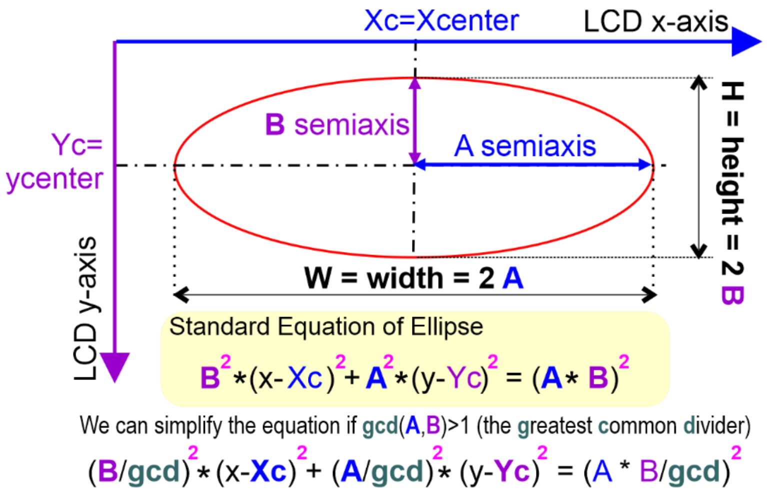


```
---------- our image -------------------------
RGB:=DARKBLUE;
if  x - y >= 640 then RGB:=YELLOW; end if;
if  3 * x + 4 * y < 1920  then RGB:=GREEN; end if;
if  2 * x + 5 * y > 2400  then RGB:=RED; end if;
-------------------------------------------------------------
```



Ellipse templates

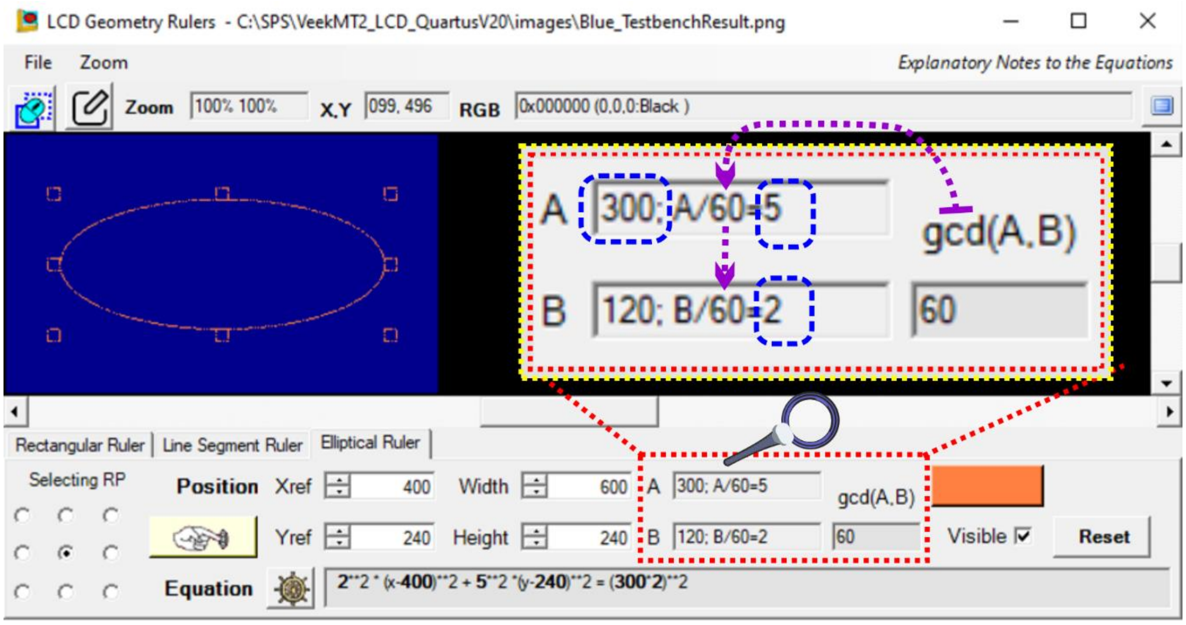
If an ellipse has horizontal and perpendicular axes, it is in canonical form. Its hardware implementation is again better if the coefficients of its equation can be divided by their greatest common divisor (gcd).



Xc=Xcenter
LCD x-axis
Yc= ycenter
LCD y-axis
B semiaxis
A semiaxis
H = height = 2 B
W = width = 2 A
Standard Equation of Ellipse
$B^2*(x-Xc)^2+A^2*(y-Yc)^2 = (A*B)^2$
We can simplify the equation if gcd(A,B)>1 (the greatest common divider)
$(B/gcd)^2*(x-Xc)^2+(A/gcd)^2*(y-Yc)^2 = (A*B/gcd)^2$

Figure 13 - Ellipse equation in canonical form

We can again use LCD Geometric Rulers to optimize searching for nearby ellipses with higher gdc and more advantageous hardware implementation (steering wheel icon).



LCD Geometry Rulers - C:\SPS\VeekMT2_LCD_QuartusV20\images\Blue_TestbenchResult.png
File
Zoom
Explanatory Notes to the Equations
Rectangular Ruler
Line Segment Ruler
Elliptical Ruler
Selecting RP
Reset

Zoom: 100% 100% X,Y: 099, 496 RGB: 0x000000 (0,0,0:Black)

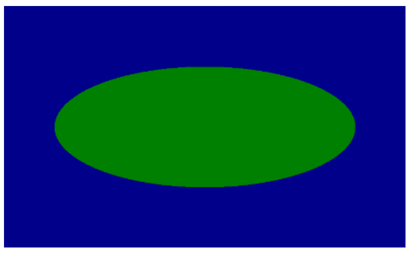
Position Xref: 400 Width: 600 A: 300; A/60=5 gcd(A,B)

Yref: 240 Height: 240 B: 120; B/60=2 60 Visible ☑

Equation: 2**2 * (x-400)**2 + 5**2 *(y-240)**2 = (300*2)**2

Figure 14 - Finding the equation of the ellipse

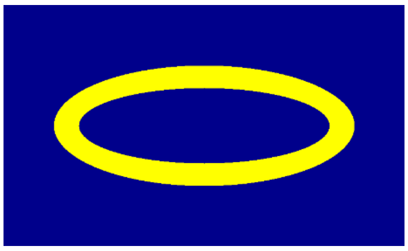
```
---------- our image ------------------------
RGB:=DARKBLUE;
-- (2=B/gdc)**2       (5=A/gdc)**2          (A*(B/gdc))**2
 if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2 then RGB:=GREEN; end if;
-------------------------------------------------------
```



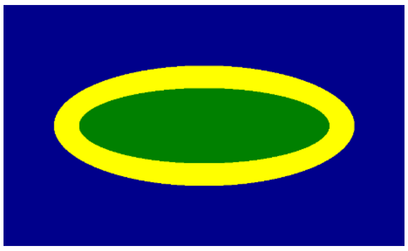
A general ellipse has axes rotated by an angle θ, and its quadratic equation can be derived from the canonical form when a Euclidean rotation of coordinates is applied to it. You can find the necessary formulas, into which the canonical A and B coefficients and the angle θ are entered, for example, on the English Wiki page, in the General Ellipse section, or on WolframCloud, in the Details and Options section.

However, if we design our LCD background, we should compose it from fragments of ellipses in canonical form, which will be more convenient.

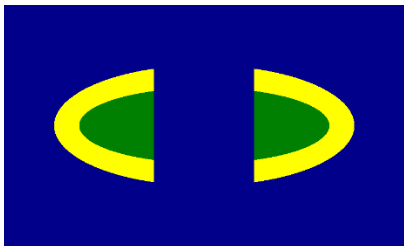
```
---------- our image ------------------------
RGB:=DARKBLUE;
-- (B/gdc)**2        (A/gdc)**2          (A*B/gdc)**2
if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2
   and 3**2 *(x-400)**2 + 10**2 *(y-240)**2 > (250*3)**2
   then RGB:=YELLOW; end if;
---------------------------------------------------------
```



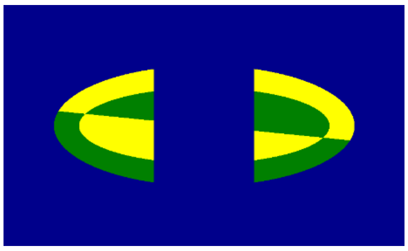
```
---------- our image ------------------------
RGB:=DARKBLUE;
if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2
    then RGB:=YELLOW; end if;
if 3**2 *(x-400)**2 + 10**2 *(y-240)**2 < (250*3)**2
   then RGB:=GREEN; end if;
---------------------------------------------------------
```



```
RGB:=DARKBLUE;
if (x<300) or (x>=500) then
     if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2
          then RGB:=YELLOW; end if;
    if 3**2 *(x-400)**2 + 10**2 *(y-240)**2 < (250*3)**2
           then RGB:=GREEN; end if;
end if;
---------------------------------------------------------
```



The following image has a complex code, so we insert its entire architecture, not just its "*our image*" part



```
architecture behavioral of LCDlogic0 is
  constant DARKBLUE: RGB_t := ToRGB(0, 0, 139); -- the background
  begin
LSPimage : process( xcolumn, yrow, LCD_DE )
   variable RGB :RGB_t :=BLACK; -- the color of current pixel
   variable x : integer  range 0 to XCOLUMN_MAX:=0;
   variable y : integer  range 0 to YROW_MAX:=0;
   variable isAboveLine: boolean:=false; -- Above our straight line
   begin -- process
     x := to_integer(xcolumn); y := to_integer(yrow); -- converting unsigned inputs to integers
---------- our image ---------------------------------------------------------------------------------
    RGB:=DARKBLUE;  isAboveLine:=( x - 10*y >= -2000 );
    if (x<300) or (x>=500) then
          if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2    then
                if isAboveLine then RGB:=YELLOW; else RGB:=GREEN; end if;
          end if;
         if 3**2 *(x-400)**2 + 10**2 *(y-240)**2 < (250*3)**2  then
               if isAboveLine then RGB:=GREEN; else RGB:=YELLOW; end if;
      end if;
     end if;  -- if ((x<300) or (x>=500)) then
-------------------------------------------------------------------------------------------------------------
      RGBcolor <= RGB; -- assigning the output signal
   end process;

end architecture;
```

The Veek-MT2 development board has the Cyclone IV FPGA, which contains 115,000 logic elements (LE). The image above only needed 177 LE, which is roughly 360 bytes. Ten hardware 9-bit multipliers were used for this, which is only 2% of all those in the FPGA.

The image saved as a PNG file would take up about 6.6 KB, and a JPEG file with 80% quality would take up as much as 15 KB.

Question: Why didn't we use the conditional assignment as when - else?

VHDL-2008 allows when-else conditional assignments, the equivalent of an **? :** operator in the C language. The VHDL code could look like this:

```
---------- our image -----------------------------------------------------------------------------------------
  RGB:=DARKBLUE; isAboveLine:=( x - 10 * y >= -2000);
   if (x<300) or (x>=500) then
         if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2    then
 --              if isAboveLine then RGB:=YELLOW; else RGB:=GREEN; end if;
🟥             RGB:=YELLOW when isAboveLine else GREEN;
         end if;
        if 3**2 *(x-400)**2 + 10**2 *(y-240)**2 < (250*3)**2  then
 --             if isAboveLine then RGB:=GREEN; else RGB:=YELLOW; end if;
🟥            RGB:=GREEN when isAboveLine else YELLOW;
      end if;
    end if;  -- if ((x<300) or (x>=500)) then
---------------------------------------------------------------------------------------------------------------
```

The GHDL simulator supports almost all of VHDL-2008, and we can use shorter when-else statements. Unfortunately, the free version of Quartus Lite only allows fragments from VHDL 2008 and does not support this handy operation :-(It is only present in its paid version. And if we want to upload the result to the board, we must compile it in Quartus, so we omitted the construction that the free version would reject.

But we can replace when-else with a handy function:

```
 function assignIf(cond:boolean; colorTrue, colorFalse:RGB_t) return RGB_t is
 begin
    if cond then return colorTrue; else return colorFalse; end if;
 end function;
```

It is included in LcdPackV2 version V2.1 and higher.

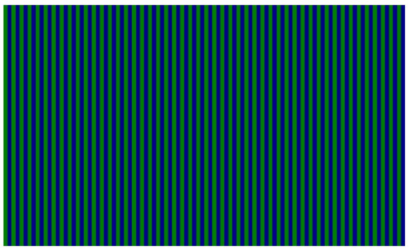
```
architecture behavioral of LCDlogic0 is
  constant DARKBLUE: RGB_t := ToRGB(0, 0, 139); -- the background
  begin -- architecture
LSPimage : process( xcolumn, yrow, LCD_DE )
   variable RGB :RGB_t :=BLACK; -- the color of current pixel
   variable x : integer  range 0 to XCOLUMN_MAX:=0;
   variable y : integer  range 0 to YROW_MAX:=0;
   variable isAboveLine: boolean:=false; -- Above straight line
   begin -- process
     x := to_integer(xcolumn); y := to_integer(yrow); -- converting unsigned inputs to integers
---------- our image -----------------------------------------------------------------------------------------
    RGB:=DARKBLUE;  isAboveLine:=( x - 10*y >= -2000);
   if (x<300) or (x>=500) then
         if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2  then
                 RGB:=  assignIf(isAboveLine, YELLOW, GREEN);
         end if;
        if 3**2 *(x-400)**2 + 10**2 *(y-240)**2 < (250*3)**2  then
               RGB:= assignIf(isAboveLine, GREEN, YELLOW);
         end if;
    end if;  -- if ((x<300) or (x>=500)) then
---------------------------------------------------------------------------------------------------------------
      RGBcolor <= RGB; -- assigning the output signal
   end process;
end architecture;
```

The assignIf must be defined for each type, which is its disadvantage compared to the more universal when-else, but it can be overloaded, similarly to C. The package contains assignIf for integers.

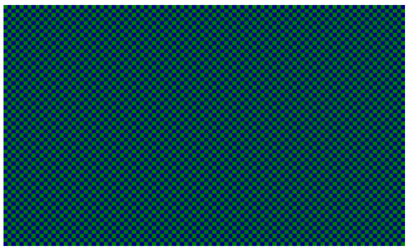
Pattern generator using division by powers of 2

Logic equations effectively create shapes that repeat themselves. It uses the fact that each LCD frame is generated as a stream of pixels. If we change the coordinates sent to the selected element to periodic ones, it will repeat itself. For example, we change the color according to the even result of integer division x by $8=2^3$. In hardware, the expression ((x / 8) mod 2)=0 is implemented by testing bit 3, xcolumn(3)='0'.

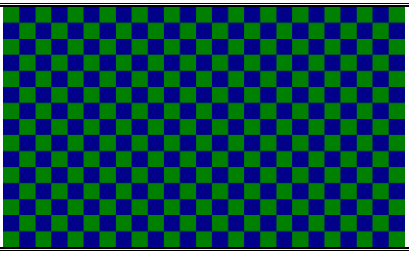
```
---------- our image ---------------------
    RGB:=DARKBLUE;
    if ((x / 8) mod 2)=0 then RGB:=GREEN; end if;
    if LCD_DE= '0' then RGB:=BLACK; end if;
------------------------------------------
```



```
---------- our image -------------------------
RGB:=DARKBLUE;
if ( xcolumn(3) xor yrow(3) )='0' then RGB:=GREEN; end if;
 if LCD_DE= '0' then RGB:=BLACK; end if;
-------------------------------------------------------------
```



```
---------- our image -------------------------
 RGB:=DARKBLUE;
if ( xcolumn(5) xor yrow(5) )='0' then RGB:=GREEN; end if;
if LCD_DE= '0' then RGB:=BLACK; end if;
-------------------------------------------------------------
```

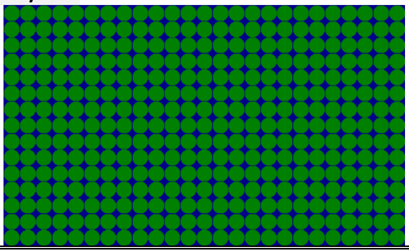


We can also repeat more complex shapes by duplicating them across the entire area. Let's start with a single occurrence:

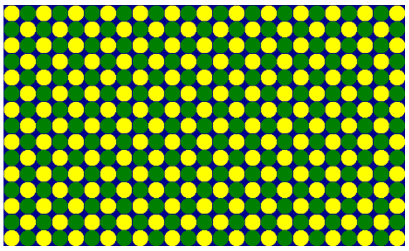
```
---------- our image -------------------------
RGB:=DARKBLUE;
if (x-16)**2+(y-16)**2< 16**2 then RGB:=GREEN; end if;
if LCD_DE= '0' then RGB:=BLACK; end if;
-------------------------------------------------------------
```

Now, instead of x and y, we will use their remainders after dividing them by 32.

```
---------- our image -------------------------
RGB:=DARKBLUE;
if (x mod 32-16)**2+(y mod 32 -16)**2< 16**2 then RGB:=GREEN; end if;
    if LCD_DE= '0' then RGB:=BLACK; end if;
-------------------------------------------------------------
```



```
---------- our image -------------------------
RGB:=DARKBLUE;
if (x mod 32 -16)**2+(y mod 32-16)**2< 16**2 then
   if ((x/32) mod 2= 0) xor ((y/32) mod 2= 0) then
         RGB:=GREEN; else RGB:=YELLOW; end if;
end if;
 if LCD_DE= '0' then RGB:=BLACK; end if;
-------------------------------------------------------------
```



The complexity of implementing the last image in LCDlogic0 is only nine logic elements and two 9-bit multipliers. The entire drawing, including the generator and register, is created with 77 logic elements and the two 9-bit multipliers mentioned above. PNG would store the motif with circles in 41 KB and JPEG in as much as 141 KB.

Such distinctive circles are probably suitable only for demonstrating the capabilities of logic :-) To make them more usable, we can reduce color differences. We choose new colors, for example, from Figure 9, page 10, with colors ordered by their hues. The resulting background has a softer decorative motif:

```
LSPimage : process( xcolumn, yrow, LCD_DE )
  variable RGB :RGB_t :=BLACK; -- the color of current pixel
  variable x, y : integer  range 0 to XCOLUMN_MAX:=0;
  variable eqcicle : integer range 0 to 2*(16**2):=0;
  begin -- process
    x := to_integer(xcolumn); y := to_integer(yrow); -- converting unsigned inputs to integers
---------- our image ------------------------------------------------------------------------------------------
    RGB:=DARKBLUE;
  eqcicle := (x mod 32 -16)**2+(y mod 32-16)**2;
  if eqcicle<16**2 and eqcicle>=12*2 then
    if ((x/32) mod 2=0) xor ((y/32) mod 2=0) then RGB:=X"0000FF"; else RGB:=X"0000CD"; end if;
  end if;
 if LCD_DE='0' then RGB:=BLACK; end if;
------------------------------------------------------------------------------------------------------------------------
    RGBcolor <= RGB; -- assigning the output signal
  end process;
end architecture;
```

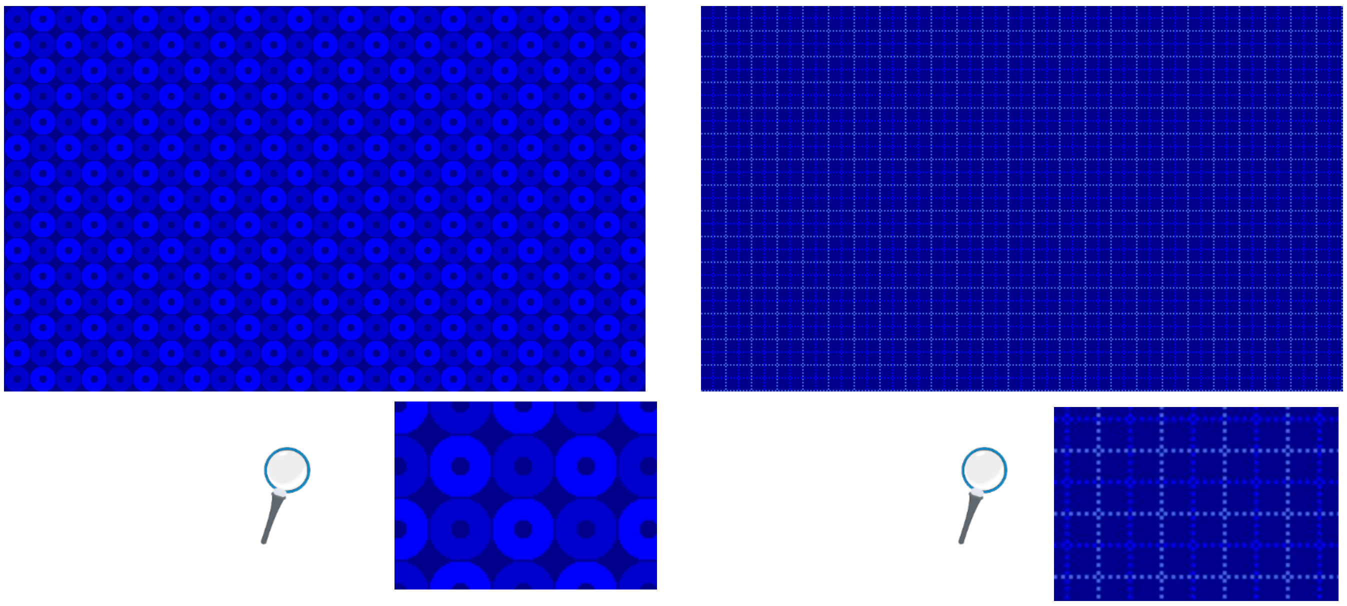


Figure 15 - Technical motifs: code creation at the top left, bottom right

We can decorate technical control panels with such dot-dashed grid analogies. The conditions for vertical and horizontal lines are separate — after all, their drawing is independent of each other! The dot-dashed are created by inserting a condition on the variable running along the line axis, e.g., y or x.

```
--------- our image ------------------------------------------------------------------------------------
  RGB:=DARKBLUE;
  if (y mod 16>=14) and (x mod 4)<2 then
     if ((y/16) mod 2) = 0 then RGB:=X"0000FF"; else RGB:=X"4169E1"; end if;
  end if;
  if (x mod 16>=14) and (y mod 4)<2 then
     if ((x/16) mod 2) = 0 then RGB:=X"0000FF"; else RGB:=X"4169E1"; end if;
  end if;
  if LCD_DE='0' then RGB:=BLACK; end if;
----------------------------------------------------------------------------------------------------
```

Progress bars are a common feature of panels, see the image below.

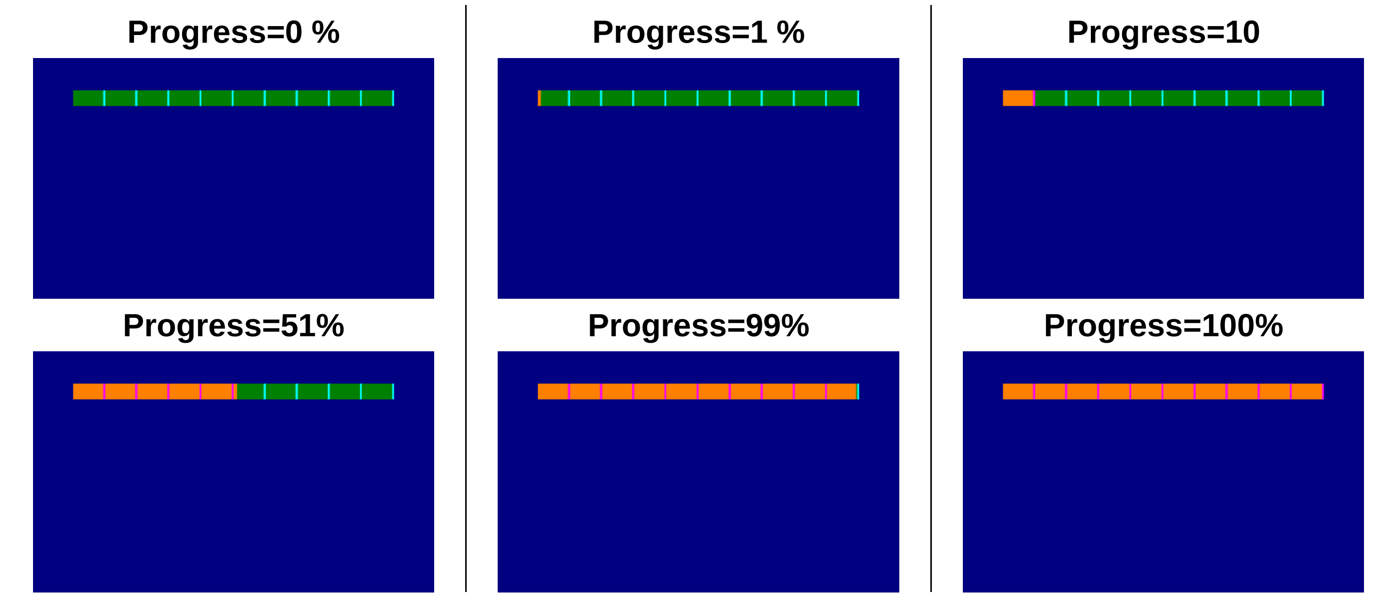
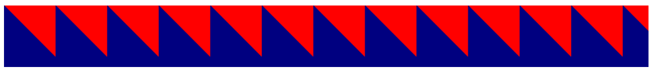


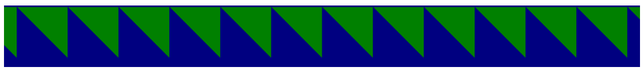
Figure 16 - Linear indicator

To implement this, we will use the remainder after dividing by 2**6=64. The result is asymmetrical because 64 does not divide the width LCD_WIDTH=800 without a remainder. The if statement on the right, which uses a different color for differentiation, centered the motif by shifting it by 80 = (800-10*64)/2

```
if y<(x mod 2**6)  then RGB:=RED; end if;
```



```
if y<((x-80) mod 2**6)  then RGB:=GREEN; end if;
```



If we introduce constants P0 for the origin on the x-axis and step ST = 64, then the architecture will be:

```
architecture indicator of LCDlogic0 is
signal progress:integer range 0 to 100:=51; -- the value is created from another process
begin -- architecture
LSPimage : process( xcolumn, yrow, progress)
 variable RGB :RGB_t :=BLACK; -- the color of pixel
 variable x : integer  range 0 to 1023:=0;
variable y : integer  range 0 to 524:=0; -- YROW_MAX-1
 constant P0: integer  := 80;   constant ST: integer  := 2**6; --P-origin, Step
 begin x := to_integer(xcolumn); y := to_integer(yrow); RGB :=  NAVY;
   ---------- progress bar -------------------------
   if  y>=ST and y<ST+ST/2  and x>=P0 and x<P0+10*ST  then-- height, in   <64,96) width, in <80,720 )
        RGB:=assignIf( ((x-P0) mod ST)<ST - 4, GREEN, AQUA);  --gaps
        if x<((progress*205+16)/32 + P0) then RGB:=RGB xor YELLOW; end if;
   end if;
   ----------------------------------------------------------
 RGBcolor <= RGB;
 end process;
iProgress : process(YEND_N) -- the dynamic simulation of a progress signal
   constant MD:integer:=2**5; variable cntr : integer range 0 to MD*100:=0;
     begin if falling_edge(YEND_N) then
                     if cntr< MD*100 then cntr:=cntr+1; else cntr:=0; end if;
               end if;
               progress<=cntr/MD;
    end process;
end architecture;
```

Repeated shapes generated by a counter

In the previous code, the value stored in **progress** was converted to the length in the x-axis using a complex formula: **(**progress*205+16)/32, where adding 16 emulated rounding. The relationship that stretches the progress value, which runs from 0 to 100%, to an interval of 0 to 640 pixels can be rewritten as

round(progress*205.0/2**5)≈ progress*205.0/32= progress*6.40625≈ progress*6.4.

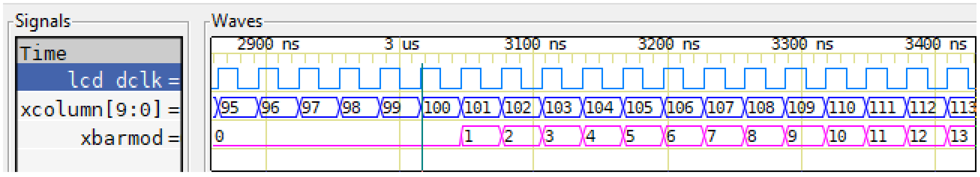
A more advantageous conversion would be if the ST (step) value was 60, then progress would be multiplied by 6, but the circuit calculating (x mod 60) would require many logic elements in the hardware. We will replace modulo with a counter. The coordinates of pixels **xcolumn and yrow** change to the rising edge of LCD_DCLK, so we let the counter run to the falling edge of LCD_DCLK, when they are stable and can be tested without the risk of metastability. We assign the result to the **xbarmod** signal at the rising edge of LCD_DCLK, i.e., in line with the changes in pixel coordinates.

```
architecture bar60 of LCDlogic0 is
 signal progress:integer range 0 to 100:=1; -- from another process
 constant P0: integer  := 100;  constant ST:integer:=60;
 signal xbarmod : integer range 0 to ST-1:=0;
begin -- architecture

  iModulo : process(LCD_DCLK)
  variable cntr : integer range 0 to ST-1:=0;
   begin if falling_edge(LCD_DCLK) then cntr:=assignIf(cntr>=ST-1 or xcolumn<P0, 0, cntr+1); end if;
          if rising_edge(LCD_DCLK) then xbarmod<=cntr; end if;
  end process;

 LSPimage : process( xcolumn, yrow, progress, xbarmod)
  variable RGB :RGB_t :=BLACK; -- the color of pixel
  variable x : integer  range 0 to 1023:=0; -- XCOLUMN_MAX-1
  variable y : integer  range 0 to 524:=0; -- YROW_MAX-1
  begin  x := to_integer(xcolumn); y := to_integer(yrow); RGB :=  NAVY;
   ---------- our image -------------------------
   if   y>=ST and y<ST+ST/2 and x>=P0 and x<P0+10*ST  then  -- height + width
          RGB:=assignIf(xbarmod<ST-4, GREEN, AQUA);--gaps
          if(x<(6*progress + P0)) then RGB:=RGB xor YELLOW; end if;
   end if;
  ---------------------------------------------------------
  RGBcolor <= RGB;
  end process;

  iProgress : process(YEND_N) -- the dynamic simulation of a progress signal
  constant MD:integer:=2**5;
   variable cntr : integer range 0 to MD*100:=0;
   begin  if falling_edge(YEND_N) then cntr:=assignIf(cntr< MD*100,cntr+1,0); end if;
          progress<=cntr/MD;
   end process;

 end architecture;
```

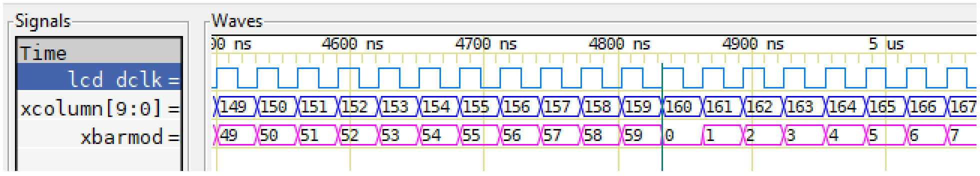
If we want to see the outputs of the iModulo process, we write a testbench, into which we insert its code together with the necessary definitions:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all; library work;
entity testbench_Modulo is end entity;
architecture rtl of testbench_Modulo is
  signal xcolumn: unsigned(9 downto 0):=(others=> '0'); -- the simulation of LCDgen output
  signal LCD_DCLK : std_logic:='0';
  constant P0: integer  :=  100;  constant ST:integer:=60;
  signal xbarmod : integer range 0 to ST-1:=0;
 begin -- architecture
   iModulo : process(LCD_DCLK) -- the copy of tested code
  variable cntr : integer range 0 to ST-1:=0;
   begin if falling_edge(LCD_DCLK) then cntr:=assignIf(cntr>= ST-1 or xcolumn< P0, 0, cntr+1); end if;
           if rising_edge(LCD_DCLK) then xbarmod<= cntr; end if;
  end process;
   LCD_DCLK<= not LCD_DCLK after (1 sec)/(2*33000000); -- the period/2 of 33 MHz signal
   xcolumn<= xcolumn + 1 when rising_edge(LCD_DCLK);
end architecture;
```

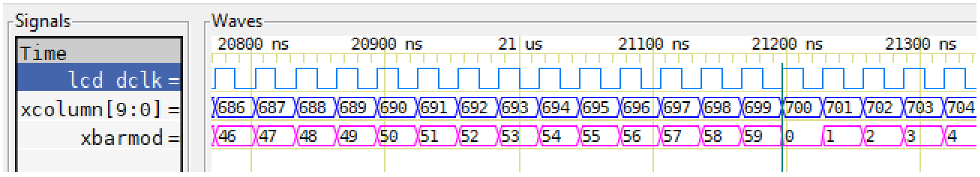
Simulation in GHDL shows the following curves in GTKView. (*The colors have been partially inverted for printing*.) The value of xbarmod is not calculated until xcolumn=100; we do not need it before that:



It grows to xcolumn = 159, then starts from 0:



Exactly at xcolumn=699, where our pointer ends on the LCD, xbarmod = 59



In GTKWave, we can also select the interpretation of xbarmod and xcolumn as analog signals. Select one signal and right-click to display its context menu, where you select: Data Format → Analog→Step. Then add "Insert Analog Height Extension for each signal. Now you can clearly see the waveforms in the LCD line. The vertical mark is in the same position as in the previous figure, at xcolumn=699.

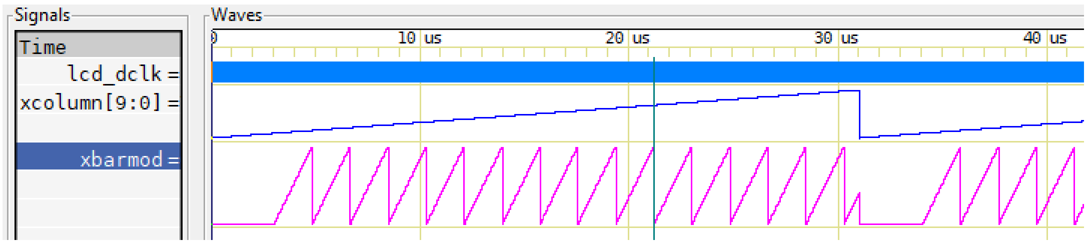


Figure 17 - - GTKView interpreting xcolumn and xbarmod values of analog signals

Inserting an image from FPGA ROM memory

Some more demanding parts are worth converting to memory and reading when rendering the image. Inside the FPGA, we have two options for storing them:

- **Logic elements** (LE) allow the fastest access to data. However, they have a much more flexible use than simply storing values, thus saving more demanding operations that memory cannot perform.
- **Memory blocks** are primarily used in FPGAs for large amounts of data. Even the Quartus development environment sometimes converts logic parts to memory reads. These achieve higher information density for data because less silicon is used to produce them. They can also have multiple port accesses, allowing independent data manipulation at different addresses. However, each memory block is used entirely, even if it is only occupied by a bit. It's all about designing the memory content.

The Cyclone IV circuit includes M9K memory blocks configurable for different output data widths. The possible variants of a single M9K block, listed as the number of words × bits in a word, are:

8192 × 1, 4096 × 2, 2048 × 4, 1024 × 8, 1024 × 9, 512 × 16, 512 × 18, 256 × 32, 256 × 36

For example, 1024 x 8 indicates a memory configuration where 8-bit words are selected using a 10-bit address (2^{10}= 1024). Thus, it has 1024 words with a width of 8 bits, i.e., 8192 bits. M9K memory can also be set to 9-bit output (i.e., with possible parity), using all 9216 bits, see Cyclone4_memoryM9Kblocks.pdf.

Reading from memory is always synchronous – the selection matrix requires this. We write the address to one clock edge, and the data appears at the memory output after a delay. They are stored in a register in the figure below, which delays them by one clock period, but they will always have constant values during clock periods, which is more suitable for implementation.

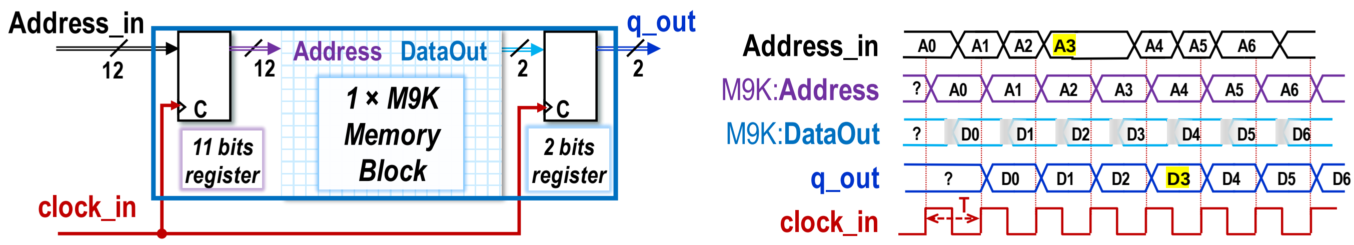
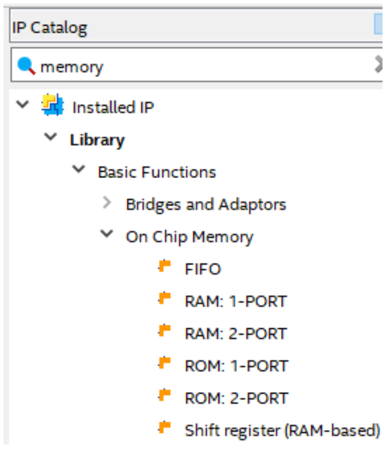


Figure 18 - - Cyclone IV memory block M9k in 4096x2 configuration

Larger memories are assembled from multiple M9K blocks, and it is worth monitoring their consumption size, because even a slight increase in data volume can add many M9K blocks, as they are always used in their entirety. However, we need to initialize the memory blocks somehow. There are two options in the Quartus environment:

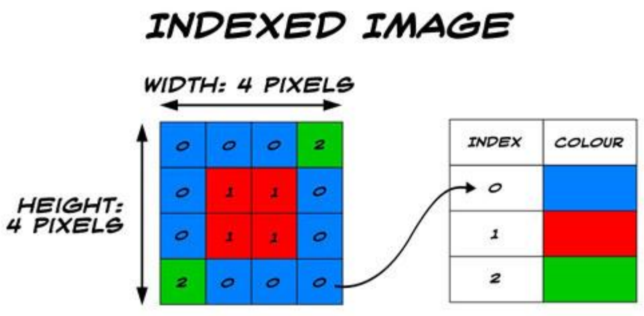

IP Catalog
memory
Installed IP
Library
Basic Functions
Bridges and Adaptors
On Chip Memory
FIFO
RAM: 1-PORT
RAM: 2-PORT
ROM: 1-PORT
ROM: 2-PORT
Shift register (RAM-based)

1) Select the memory type from the manufacturer's IP catalog, see the image on the left. The catalog launches the MegaWizard Plug-In Manager tool in Quartus, where we enter the necessary memory parameters and the initialization file of type *.MIF, Memory Initialization File. The procedure is more laborious, but you can choose from several options. However, this complicates simulation in GHDL, as Quartus' internal libraries must be inserted, which is not simple.
2) If 1-Port ROM memory is sufficient, a VHDL file can be generated. Quartus converts it to memory. This way also allows easier GHDL simulation, which we will demonstrate here. The previous steps can be found in the aforementioned M9K memory manual.

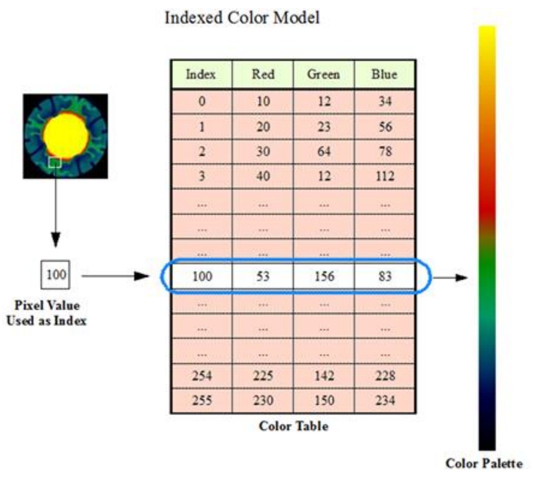
Bitmap2VHDL from FPGA-LCD Utils can create both an initialization *.MIF file and a *.vhd file that Quartus can convert to ROM: 1-Port.

Bitmap conversion

If there are few colors, the data volume is reduced by assigning indices to them and storing only those. They are directed to a color table, allowing easy changes. For more details, see https://en.wikipedia.org/wiki/Indexed_color.



INDEX	COLOUR
0	🟦
1	🟥
2	🟩



Index	Red	Green	Blue
0	10	12	34
1	20	23	56
2	30	64	78
3	40	12	112
...	...	...	...
...	...	...	...
...	...	...	...
100	53	156	83
...	...	...	...
...	...	...	...
...	...	...	...
254	225	142	228
255	230	150	234

Note: Both images come from idlcoyote.com
— a website that no longer exists.

Figure 19 - Indexed colors

Images stored without rasterization, "spatial anti-aliasing" (see below), which increases the number of colors, are best suited for indexing. If the selected image has rasterization, it is advisable to reduce the number of colors, which can be done with many graphics tools, such as the free FastStone Image Viewer.

Graphic tools can visually smooth edges by adding transitional color shades through rasterization. The opposite of this is "dithering," which is a kind of dispersion that creates halftones to substitute missing colors and smooths the image. In terms of hardware, it is relatively easy to implement 3 × 3 or 5 × 5 Gaussian blur convolution matrix . The Veek-MT2 LCD panel can also do this.

VeekMT2_LCDregV2 turns off its dithering so that you can see exactly what has been created on it. It can be turned on by setting its LCD_DITH output to '0'.

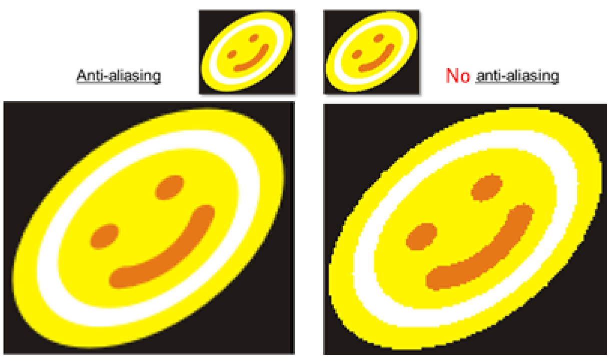
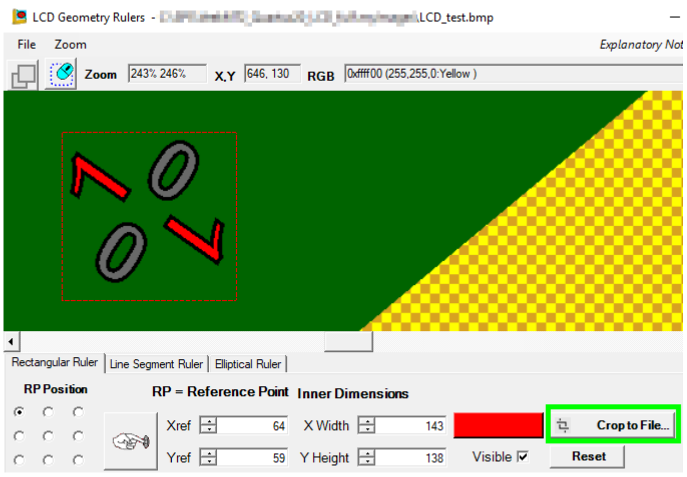


Figure 20- Anti-aliasing

Cut out the selected part of the image using a graphics tool and save it as a bitmap.



LCD Geometry Rulers - [redacted]LCD_test.bmp

Zoom 243% 246% X,Y 646, 130 RGB 0xffff00 (255,255,0:Yellow)

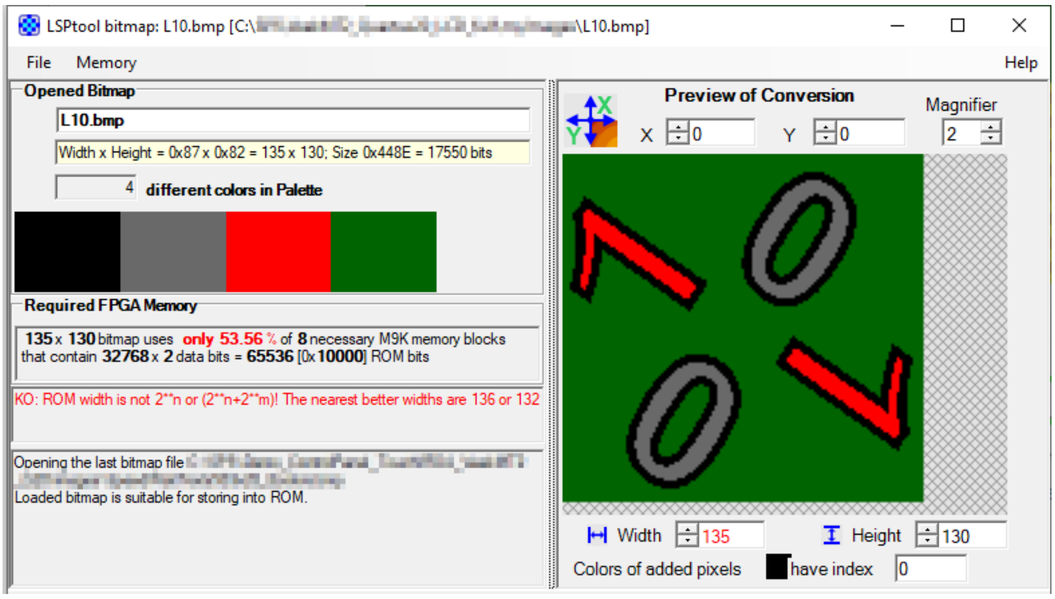
RP Position RP = Reference Point Inner Dimensions

Xref 64 X Width 143 Crop to File...

Yref 59 Y Height 138 Visible ☑ Reset

Figure 21- Cropping to a file using the LSP Geometry Rulers tool

Then we can run the BMP converter from FPGA-LCD Utils: and load the bitmap.


LSPtool bitmap: L10.bmp [C:\...\L10.bmp]

Opened Bitmap

L10.bmp

Width x Height = 0x87 x 0x82 = 135 x 130; Size 0x448E = 17550 bits

4 different colors in Palette

Required FPGA Memory

135 x 130 bitmap uses only 53.56 % of 8 necessary M9K memory blocks that contain 32768 x 2 data bits = 65536 [0x10000] ROM bits

KO: ROM width is not 2**n or (2**n+2**m)! The nearest better widths are 136 or 132

Opening the last bitmap file ...

Loaded bitmap is suitable for storing into ROM.

Preview of Conversion: X 0, Y 0, Magnifier 2

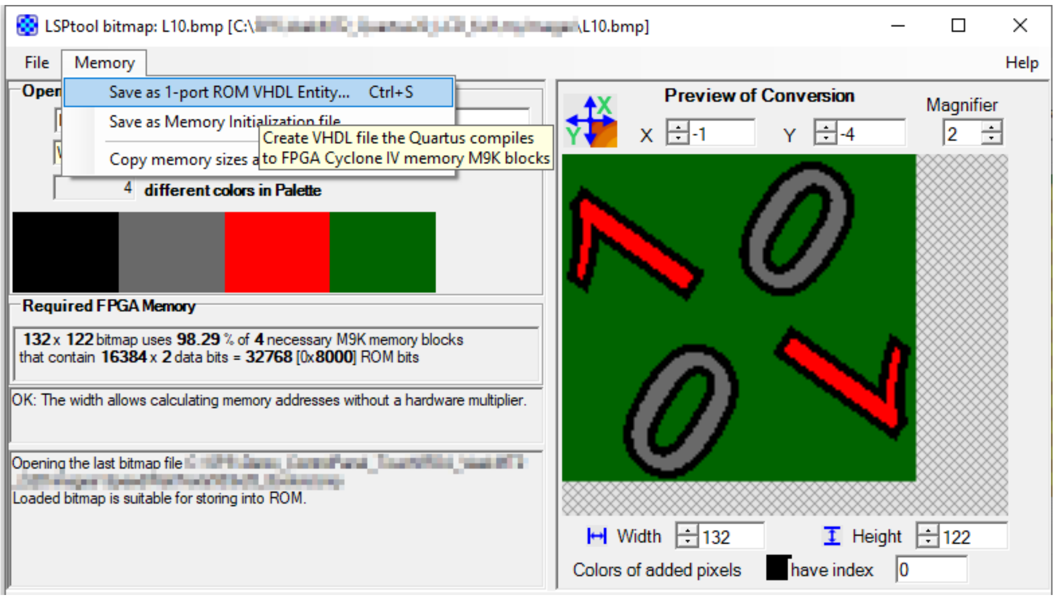
Width 135, Height 130

Colors of added pixels have index 0

Figure 22 - FPGA-LCD tool Bitmap to VHDL

The image uses four colors, which we encode into two data bits. However, the resulting memory would consume 8 M9K memory blocks, which would be used only half of their capacity. We crop the image, remembering that **we will multiply by its width,** so we choose a power of 2 or the sum of two powers of 2, leading to more straightforward hardware implementation, see also section 6.3.2 Logic circuits on FPGA.

Adjust the dimensions using the up and down controls. Leave the leftmost one free, because we will read the data from memory with a delay of 1 clock cycle.


LSPtool bitmap: L10.bmp [C:\...\L10.bmp]

Memory menu:
- Save as 1-port ROM VHDL Entity... Ctrl+S
- Save as Memory Initialization file
- Copy memory sizes

Tooltip: Create VHDL file the Quartus compiles to FPGA Cyclone IV memory M9K blocks

4 different colors in Palette

Required FPGA Memory

132 x 122 bitmap uses 98.29 % of 4 necessary M9K memory blocks that contain 16384 x 2 data bits = 32768 [0x8000] ROM bits

OK: The width allows calculating memory addresses without a hardware multiplier.

Opening the last bitmap file ...

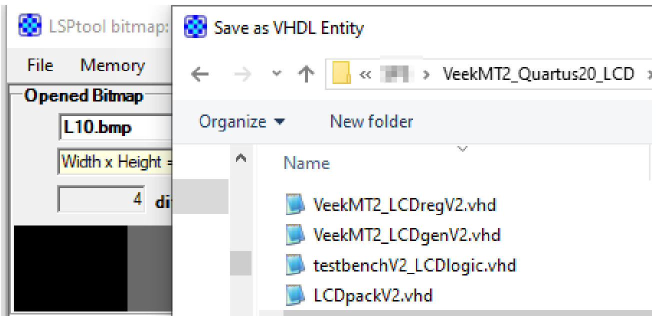
Loaded bitmap is suitable for storing into ROM.

Preview of Conversion: X -1, Y -4, Magnifier 2

Width 132, Height 122

Colors of added pixels have index 0

Save the modified bitmap via the menu "Memory->Save as 1-port ROM VHDL Entity", for example, under the name **L10rom.vhd** in the main directory of the Quartus project!


Save as VHDL Entity

« ... › VeekMT2_Quartus20_LCD

Name
- VeekMT2_LCDregV2.vhd
- VeekMT2_LCDgenV2.vhd
- testbenchV2_LCDlogic.vhd
- LCDpackV2.vhd

Never modify the generated VHDL **L10rom.vhd** in any way to not disrupt its precise structure specified in the Quartus development environment documentation as suitable for implementation using M9K blocks. However, we can read its header, which contains information about the memory size and color palette.

```vhdl
--  FPGA-LCD Utils generated file  from bitmap L10.bmp
--  adjusted to the sizes: Width x Height= 132x122=16104 [0x3EE8] pixels.
-- 32768 [0x8000] bit memory is arranged for a 14-bit address bus reading a 2-bit data output.
-- The color palette in the index order as std_logic_vector(23 downto 0) items:
-- X"000000",  X"696969",  X"FF0000",  X"006400"  -- 0 to 3
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity L10rom is
port ( address: in std_logic_vector(13 downto 0):=(others=> '0');
        clock: in std_logic:= '1';
        q: out std_logic_vector(1 downto 0):=(others=> '0'));
end entity;
architecture rtl of L10rom is
  type arr_t is array(0 to 2**address'LENGTH-1) of std_logic_vector(q'RANGE);
  constant arr :arr_t:=(   0 to 484=> "11", 485 to 492=> "00", 493 to 614=> "11", 615 to 626=> "00",
-----------------------
-- another rows with the definitions of memory content
                                    15741 to 15753=> "00", 15754 to 15875=> "11", 15876 to 15882=> "00",
                                    others=> "11");
-----------------------------------------------------
begin
  process(clock)
  variable ix:integer range 0 to 2**address'LENGTH-1:=0;
  begin
     if rising_edge(clock) then
        ix := to_integer(unsigned(address));
        q<=  arr(ix);
     end if;
  end process;
end architecture;
```

We wisely left the first column of the converted image free, so that the value highlighted in green at the beginning of the initialization of the **constant** arr array corresponds to the index of the image's background color, which we leave transparent.

The process code raises the question of why we introduced a new variable ix, and did not use a compound command:

q<= arr(to_integer(unsigned(address))); ?

The recommended coding style, see the Quartus Inferring ROM Functions from HDL Code manual, requires the address input of the std_logic_vector type. Instead, we wrote a shorter code using the ix variable of integer type, for which the to ranges in addresses are defined. Then, the ix variable is also correctly compiled by Quartus because we add the subtype that contains precise information about its value range.

Note: Quartus also allows writing the to ranges also for the std_logic_vector types. However, the VHDL standard does not include them. Using something like this would create non-portable code dependent on the compiler, known as "compiler-dependent code."

How is an image read from memory?

The figure below shows the situation where a 4x3 pixel bitmap was converted. It was stored in memory as a one-dimensional vector by rows, precisely how images are written on an LCD. This method of storing a multidimensional array, called " row-major order" is also used by the C language.

The memory contains only two-bit color indices, i.e., values from 0 to 3. We send the memory the address from which to read the data. The address relationship maps the stored image to the resulting display on the LCD. On the right, we have two of many possible variants. The upper image is positioned with its upper left corner at yrow=1, xcolumn=2 on the LCD. The memory is read sequentially. If we set x=xcolumn and y=yrow, then we first convert x and y to relative coordinates relative to the corner of the image, from which we calculate the address in the ROM memory:

Memory address ≡ (**y-rect1.Y**)***img.Width** + (**x-rect1.X**)

We multiply by the width of the image; here, we have advantageously chosen the sum of two powers of two, which is easier to implement in the circuit. The second image is inverted; its relative y-axis runs in the opposite direction:

Memory address ≡ (**img.Height-1-(y-rect2.Y)**)*img.Width+ (x-rect2.X)

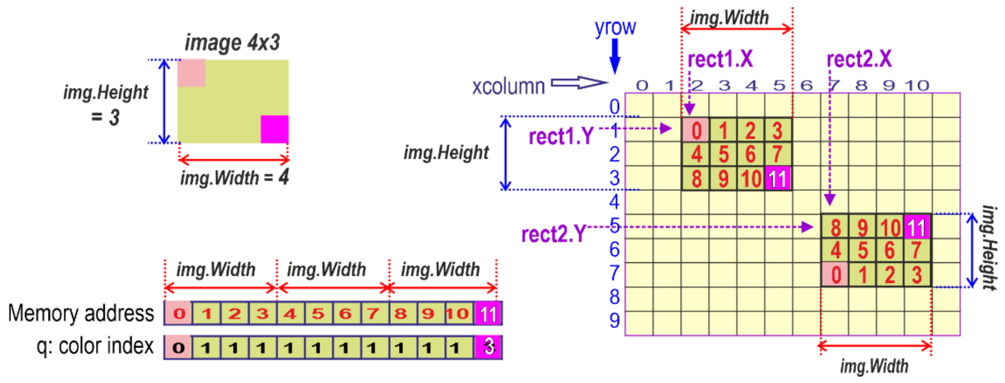


Figure 23 - Image 1: normal 2: inverted

There are many positioning options, all of which only involve changes to the address calculation, which also applies to rotations in the 90-degree module. For 180 degrees, both axes are read backwards. For 90 degrees, only one axis is rotated, but they are swapped. We also need to modify the test for the rectangle in which it is drawn.

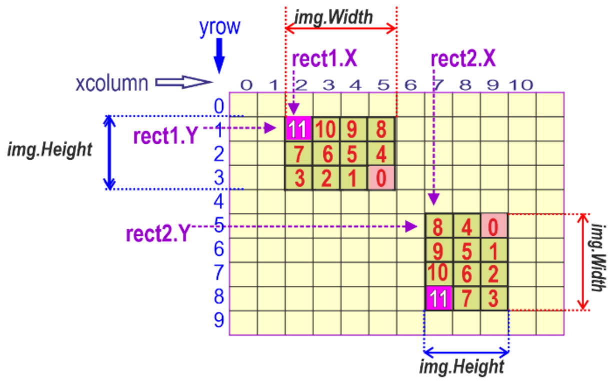
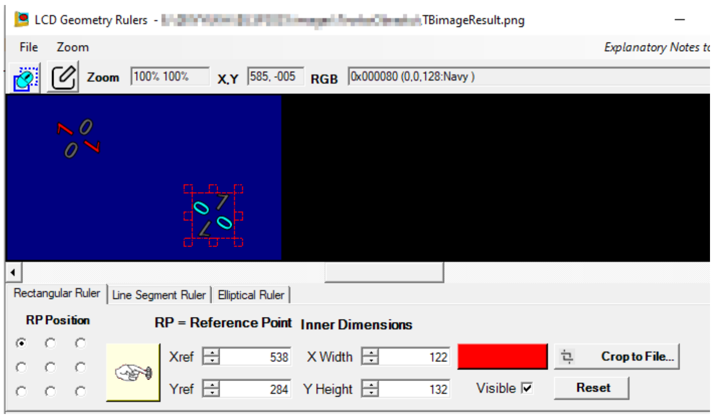


Figure 24 - Rotation 1: 180 degrees 2: 90 degrees

Rotation by 90 degrees will be in the code in the next chapter.

VHDL code with images inserted from memory

If we have a graphic template, we determine the coordinates of the image from it. Of course, we can only estimate their positions and correct them according to the testbench result, for example, using LCD Geometry Rulers, until we are satisfied with the positions:



Zoom 100% 100% X,Y 585, -005 RGB 0x000080 (0,0,128:Navy)

RP Position

RP = Reference Point Inner Dimensions

Xref 538 X Width 122

Yref 284 Y Height 132 Visible ✓

When writing code, we pay attention to variable names. We use VHDL record types, which are direct analogies of C structures. The entity has not changed, so we will start with the architecture:

```
architecture img of LCDlogic0 is
 type sizes_t is record  Width, Height: integer; end record;
 constant L10img : sizes_t :=(132,122);
```

The constant of type sizes_t contains the width and height so that both values are together.

```
 constant L10r1 : rect_t :=(140, 64, L10img.Width, L10img.Height);
 constant L10r2 : rect_t :=(538, 284, L10img.Height, L10img.Width);
```

We define the sizes of rect_t rectangles for both positions. In the constant **L10r2**, the memory sizes are swapped because it refers to the position of the image that will be rotated by 90 degrees.

-- type rect_t and function inRect that uses it, are in LcdPackV2 version V2.1 and higher

```
 function inRect(r:rect_t; x,y:integer) return boolean is
 begin return x>=r.X and x<r.X+r.W and y>=r.Y and y<r.Y+r.H;
 end function;
```

Let's simplify the main code by defining a function that tests whether the current coordinates lie within the rectangle.

```
 type palette4_t is array (0 to 3) of RGB_t;
 constant L10p1:palette4_t:=(BLACK,  X"696969",  X"FF0000",  X"006400");
 constant L10p2:palette4_t:=(BLACK,  AQUA,  X"696969",  X"006400");
```

We created the first basic palette based on the data in the VHDL file of the converted bitmap. In the second palette, we recolored some items.

```
 signal L10addr: std_logic_vector(13 downto 0):=(others=> '0');
 signal L10q, L10q0: std_logic_vector(1 downto 0):=(others=> '0');
```

We will send the address to the memory and retrieve data from it. Both signals' size must be created according to the inputs and outputs of the memory file L10rom.vhd.

```
 function toSlv(n:integer; slvWidth:positive) return std_logic_vector is
 begin return std_logic_vector(to_unsigned(n,slvWidth));
 end function;
```

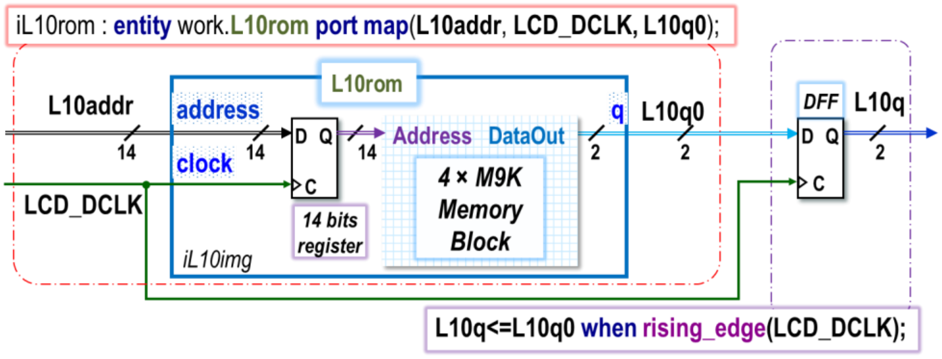
We will calculate the address using integers, but the memory has it as an input of type std_logic_vector, so we have defined a conversion function to simplify the main code.

```
begin -- architecture
```

In the code, we will create an instance of the memory entity and place the output value register behind it. We will also insert a DFF circuit, whose D input will be the L10q0 signal and output the L10q signal.

```
iL10rom : entity work.L10rom port map(L10addr, LCD_DCLK, L10q0);
          L10q<=L10q0 when rising_edge(LCD_DCLK);
```

The circuit created by the pair of commands above is shown in the figure below:



```
LSPimage : process( xcolumn, yrow, L10q)
 variable RGB :RGB_t :=BLACK; -- the color of pixel
 variable x : integer  range 0 to 1023:=0; -- to XCOLUMN_MAX-1
 variable y : integer  range 0 to 524:=0; -- to YROW_MAX-1
```

We added L10q, i.e., the value read from memory, to the sensitivity list of the process, on which the output of the RGBcolor process with the pixel color also depends. Then, we inserted the definitions of the x and y variables that were already known.

```
 variable L10idRect : integer range 0 to 2:=0;   -- the flag that the x,y pixel is inside a rectangle, 0 - no
 variable L10ixColor : integer range L10p1'RANGE:=0; -- the index of a color read from memory
```

The first variable L10idRect list will be an identifier that the x,y coordinates of a pixel are located in a rectangle of the image, where 0 means outside the image. The second variable is the color index converted to an integer.

```
 begin    -- process
   x :=  to_integer(xcolumn); y := to_integer(yrow);
   L10idRect:=0;   -- not inside a rectangle
   if InRect(L10r1, x, y) then L10idRect:=1; elsif InRect(L10r2,x,y) then L10idRect:=2; end if;
```

We have assign the rectangle identifier L10idRect by successive tests of the position inside one of them.

```
  L10ixColor :=  to_integer(unsigned(L10q));   -- index into palette
```

We converted the value read from memory to an integer, which we will use to index the palettes.

```
 ---------- our image ------------------------
 RGB :=  NAVY;
 if L10idRect> 0 and L10ixColor/=3 then   -- Is the current pixel in any rectangle and a color-index is an opacity color?
     if L10idRect= 1 then RGB:=L10p1(L10ixColor); else RGB:=L10p2(L10ixColor); end if;
 end if;
```

If [x, y] pixel is located in any image rectangle (L10idRect> 0) and at the same time the color read from the image memory is different from 3, i.e., from the color index of the background we want to make transparent (L10ixColor/=3), we overwrite the RGB value from the corresponding palette.

```
  case L10idRect is
    when 1=> L10addr<=toSlv( (y-L10r1.Y)*L10img.Width+(x-L10r1.X), L10addr'LENGTH );
    when 2=> L10addr<=toSlv( (L10img.Height-1-(x-L10r2.X))*L10img.Width+(y-L10r2.Y), L10addr'LENGTH );
    when others> L10addr<=(others=> '0');
 end case;
```

We calculate the memory address in the first rectangle by directly converting the index of the two-dimensional array to a vector. The second rectangle is rotated by 90 degrees, so its relative x and y axes are swapped, with the x-axis read backwards by lines from L10img.Height-1 to 0, while the y-axis runs in the direction of the lines in the image.

```
  ---------------------------------------------------------
  RGBcolor <= RGB;
  end process;
end architecture;
```

Advice from the GHDL buddies about -a parameter: You must also add the memory file to the list of files in the runLCD.bat batch file before LCDlogic*, otherwise it will not be compiled.

```
set FILES=../LCDpackV2.vhd ../L10rom.vhd ../LCDlogic0.vhd
```

However, the memory does not have to remain in the list permanently. My parameter colleagues -e and -r only need the *.o (object files), which are the results of my compilations. From them, they create the exe successfully. Run my full compilation only once, then modify runLCD.bat or create a new *.bat file with a modified FILES line, leaving only `../LCDlogic0.vhd`

```
rem set FILES=../LCDpackV2.vhd ../L10rom.vhd ../LCDlogic0.vhd
set FILES= ../LCDlogic0.vhd
```

You will see the result faster. The previous example saved 4 seconds of my precious time!
A complete compilation of all files will be necessary only after changing L10rom.vhd and LCDpackV2.vhd.

Addendum from Quartus Lite: Similar tricks don't work for me — I'm no free-thinking GHDL buddy!
I insist on the exactness! The memory (here `../L10rom.vhd`) will always be in the file list on my Files tab.

If it is not there, add it quickly. Either right-click on Files to open their associated context menu or use my main menu:

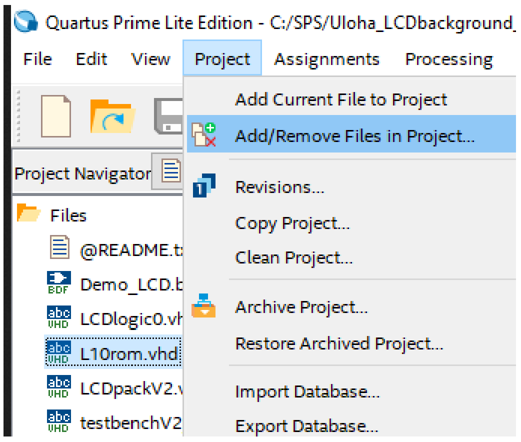


Figure 25 - Adding memory to the file list

Translation and simulation of VHDL code takes about 7 seconds in GHDL, and we perform only one step — we run the batch file from the Visual Studio Code terminal. The Quartus also offers installing the Intel Questa simulator, but obtaining its free license is as complex as its usage. Questa offers minor advantages, as it sometimes detects more timing errors, but its free version only catches a few more. However, working with GHDL is much faster and simpler. ☺

Note: 1/ We can load the debugged VHDL in Quartus into the board if we compile it.
2/ However, it is possible that the FPGA board is not working, even if the VHDL code has a beautiful simulation, because any simulation is only a simulation. The hardware is the final judge of whether you have met all the timing requirements.
3/ If you see Quartus compiler messages warning about incomplete timing definitions:
Critical Warning (332168): *The following clock transfers have no clock uncertainty assignment. For more accurate results, apply clock uncertainty assignments or use the derive_clock_uncertainty command...*
or
Critical Warning (332049): Ignored create_generated_clock at VeekMT2_LCD.sdc(50): Argument <targets> is an empty collection
Then, you have an incorrect instance in the top-level **entity** in the BDF schema, see Figure 2 on page 4:-)

- VeekMT2_LCDgenV2 must have an instance name **iLCDgenerator.**

Only this instance name has definitions for TimeQuest Analyzer in the file VeekMT2_LCD.sdc. If you change the instance name, you must regenerate sdc-file ... Oh, changing the instance is much simpler and faster.

END OF TEXTBOOK

Protest of *student Maxo Groucho: The ending is in the most suspenseful moment, like in a jumpy horror series?! You can't be serious. You're gonna bathe us in it for a* ***V****ery* ***H****ard,* ***D****epressing,* ***L****ong time! Somewhere, there will be ready-made solutions for individual backgrounds to be 'copied' quickly, right?*

Answer: Not at all! GHDL simulation is quick; you can experiment freely and use the template for fun creations.

Maxo Groucho's horror: I can't find the VHDL codes for the individual backgrounds from the template anywhere on the website that I could quickly copy!

Answer: They are not there and will not be there, to save you time. If we don't count comments, the entire background, including image insertion, will be less than 2,000 characters (including spaces). You can create it in a few minutes, as the editor's auto-complete feature will fill in many of the keywords for you. And you will better understand how it works.

Create your own code! After all, you surely have different images with different sizes and possibly different addresses and data ranges. Copying the original code would also copy the original numbers, and it would take a long time to find out where the error occurred!

That's all...

~ *End* ~