Training Task: LCD background

Logic System and Processors

Richard Šusta, Department of Control Engineering, CTU-FEL Prague

Version 2.0 of 17. September 2025

Table of Contents

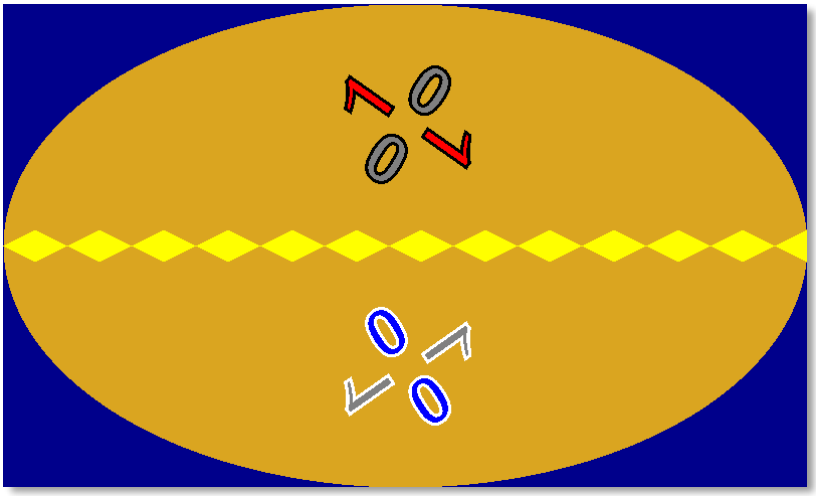

Figure 1 - Example of LCD control panel background

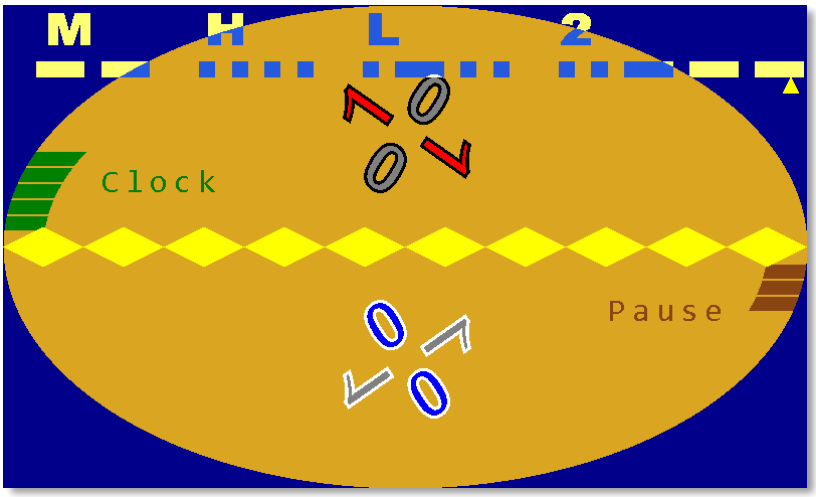

Figure 2 - Using the background in the Morse Beacon control panel

Assignment

Reason for implementation:

Manager Freecoolin, see https://www.urbandictionary.com/define.php?term=freecoolin, has performed a long series of in-depth insights into pie charts and excellent tables. After many sleepless nights, he concluded that city delivery drones could not use unreliable GPS with easily jammed weak signals. He ordered navigation beacons similar to the VOR that air traffic still uses today.

VOR identifications are sent as Morse code transmitted repeatedly. The transmitter control panels need backgrounds, but storing images in FPGAs requires reducing memory consumption. With logic, we can achieve higher intelligent compression than JPEG or PNG.

Task: Create a combinational logic circuit generating an ecological background for Freecoolin's beacons. Choose the one you like at the https://dcenet.felk.cvut.cz/fpga/ website.

Important notes:

1. All LCD backgrounds contain curves, which are always parts of **circles or ellipses**.
2. Each background includes two complex images, but they are identical. They can only be rotated, inverted, or colored differently. Ignore their pixel differences in shape; these are inaccuracies caused during the *rasterization* process that converts the vector graphics designs to bitmaps**.**
3. Create repeated shapes by division and **modulo** (by the remainder after division) of a power of 2.

Student Maxo Groucho's venomous remark*: I know industrial LCD panels do not have Super Retina XDR OLED displays like my divine iPhone. But even the shittier stuff can load a PNG image. Why do we have to pat Freecoolin's scribbles with logic? Are we supplying a museum?!*
*A**: Iphoney's display is indeed in a different price range. You can also buy 800x480 LCD panels for 33 EUR (Neven price from Sept 5, 2025), i.e., for 5% of the cost of IPhone's replacement LCD miracle. Even on a small processor, the image is created more optimally by drawing than unzipping from PNG ot JPEG-*
***S**tudent Maxo Groucho: Well, hello, crawling under the A tent now, it's gonna take honest hard work.*
A: I only half agree. Your actual VHDL code will be short. The only thing that might be more challenging is understanding how circuits are built. But that's what our LSP course is all about. We find a suitable way to describe them as implementation at the hardware level, either working with logic or using memory:-)

Requirements for VHDL code

Industrial designs are always bound by a number of limitations and resources used as a common condition of contracting authorities. The following constraints apply only to the circuits that make up the figure.

1. Only the **ieee.numeric_std** library is allowed for arithmetic operations. The older, now generally not recommended, libraries ieee.std_logic_arith, ieee.std_logic_unsigned; ieee.std_logic_signed must not be used. These can be found in a number of older examples on the web. They define conversions differently and are not portable.
2. The project must not use shared **variables** - leave them for simulations.
3. Each background contains two complex images generated by ROM, but are completely identical except for the color shades and possibly inverted or rotated by N*90 degrees. Any minor pixel variations you may find are inaccuracies caused by converting drawings from vector graphics to bitmaps, called *rasterization*. So consider the images identical. And they don't overlap anywhere, so to save resources, **both** must be **loaded from the same ROM**.
4. From memory blocks, you can create only these images, nothing else :-)
5. The circuit must not contain **LATCH** and **LOOP** loops - their presence always indicates design errors. Always check your translated project with the **Quartus Checker** tool from FPGA-LCD Utils.

6. Beware: Only the VHDL types **signed** and **unsigned** are the direct equivalent of the C **int** types**.** The VHDL **integer** data type has an automatic adjustment of its bit length with a limit to a maximum of 32 bits, but this sometimes either detects more bit length than is physically necessary or overflows. It adds to this by limiting the length of the range.
7. Check the bitwidths of your RTL Viewer Quartus variable implementations to see if any unnecessary stretching would make all the downstream logic, such as adders, comparators, and so on, grow. Teachers will then reject your solution as ineffective, even if it works.
   Advice: a) When developing, leave your integer definitions unconstrained. When your display starts to work, add integer ranges to the definitions.
   b) In your calculations, don't use unsigned types, but prefer variables of type **integer**. Unsigned numbers can give comparison errors and do not allow subtraction.
8. VHDL automatic **integer** types do not have a fixed bit width. You must always add range constraints if integers are used in inputs or outputs definitions in VHDL entities. More professional designs prefers only std_logic_types in entities..
9. Be careful with integer multiplication. The FPGA includes hardware multipliers, but may overflow for automatic **integer** types. They work reliably up to a maximum of about 28 bits. If you want to multiply by very large constants, do it better with VHDL types signed or unsigned
10. If possible, choose constants equal to **powers of 2** for multiplication, which is implemented by bit shifts. Alternatively, prefer multiplication **by the sum of two powers of 2** (done by summing two bit shifts). In a circuit, it is easier to multiply by, say, 64, but also 65 (64+1), or 66 (64+2), or 68 (64+4), etc., without needing a multiplier. Hardware multipliers are inside the FPGAs; the Cyclone IV FPGA in Veek-MT2 has 266 of them, 18x18 bits wide, each reconfigurable to two 9x9 bits, but they lie in fixed positions, which forces placing logic blocks near them. The adders can be located anywhere. So give the compiler more freedom.

Requirements for division and modulo operations

In logic, general divisors or modulo operations (remainders after divisions) can be created only by emulating the manual division procedures. It is a slow, clumsy way because it leads to long cascades of subtractors and comparators that take a lot of logic and time. And in the designs, we try to avoid operations with inefficient implementations, so we must omit general divisions and modulos. Check your compiled project occasionally with the Quartus Checker tool from FPGA-LCD Utils, which detects similar transgressions. It will save you time. Your teacher will use it to check your task and reject it if the checker announces something wrong. Y

- Some divisions can be replaced. For example, we write the condition x/3 < y/5 as 5*x<3*y.
- You can anywhere divide by powers of 2 at will, i.e., 2, 4, 8, 16, 32, etc. - such operations are implemented by bit shifting.
- Even the remainder after division by powers of 2 is converted to selecting lower bits.
- **What about other divisions and modules**? Division can always be approximated by multiplication; also, see chapters 6.3.3 to 6.3.5 of Logic Circuits on FPGAs, pages 108 to 112. Dividing successively increasing x and y coordinates of pixels can also be done by counters. It will be in the lectures.

Note: Even fast computing algorithms avoid divisions. They are not efficient even on large processors.

The above restrictions do not apply to evaluations done at compile time! So, initializing constant values can use anything, including operations with real numbers. Testbenches are also not synthesized but emulated. You use any operations in parts intended only for simulation.

Design for hardware!

The background picture is created by logical conditions in which equations you know from linear algebra are written. **You should always choose patterns that are suitable for hardware!**

Use LCD Geometry Rulers in FPGA-LCD Utils. The rulers are similar to the well-known Geodebra tool, but adapted to the LCD coordinate system, and give results in integers - just what we need in circuits.

Adjust the equations of curves and lines. The steering wheel icon runs automatic optimization, which searches near your line or ellipse for more favorable implementations.

For example, the slope of the 297/800 line is not implementation-friendly because both numbers cannot be truncated. The slope 300/800 = 3/8 is easier to work out; this line differs from 297/800 by only 0.22 angular degrees. It can be truncated by the greatest common divisor (gdc). After that, we multiply by smaller numbers in the equation, which requires significantly fewer logic elements.

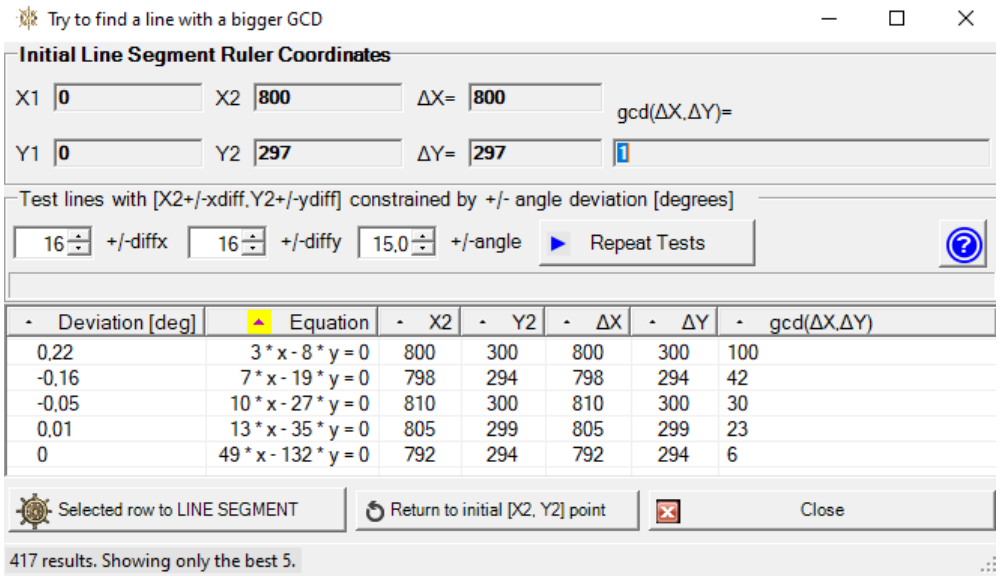

Figure 3 - Line optimization dialog in LCD Geometry Rulers from FPGA-LCD Utils

By adjusting the geometric coefficients, we perform the world-famous "measurement error compensation", refining the results obtained by ingeniously multiplying them by the cheat-constants miraculously found by our heuristic experts. Welcome to the world of science! ☺

You can also consider the collection of LCD control panel backgrounds at the LSP website to be mere ideas. After all, *Freecoolin* is an ordinary manager, and the art muses didn't invite him to their talent lottery.

Adjust colours or shapes as you wish. Feel free to design your own motif :-) Your creativity is limited only by the image's complexity. In your background, it should be no less than those in the origins, with two images differently oriented. Ask the teachers if you are not sure. Indeed, backgrounds that are too simple, e.g., a few circles on a blue screen, and so on, will be denied.

Student Maxo Groucho's dismay*: Yeah, but Frikulin can then tear our point invoice to shreds!!!*

A: If Freecoolin had objections, count for him. Your implementation will certainly reduce power consumption. Even if it drops by a mere milliwatt-hour, it will save 9 Wh per year and spare the power of a giant power plant per eon. Manager Freecoolin will cheer and cash points in! After all, in the next board meeting, he can proudly boast that he has secured a giant future savings with "his" background design!

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 its usage is more complex. Even if Questa sometimes detects more errors, working with GHDL is much faster and simpler. ☺

Note: We can load the debugged VHDL in Quartus into the board if we compile it.
But you ignore any compiler messages that warn 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...*

It is a tax for simpler and clearer **top-level entities** in our BDF scheme:-)

END OF ASSIGNMENT

---

*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: The images are too individual in various backgrounds. In my code, their insertion requires only 24 lines. Copying them is useless for other designs. You could only be confused by all pasting, rewriting…

Writing your VHDL image code in LCDlogic0 will be more apparent. Use this assignment for fun and move on to the final task, for which you will receive a simple assignment.

And the tutorial, "LCD Backgrounds," presents templates of geometric shapes and the source code of a selected example is in LCDbackgrounds_VeekMT2.zip as stepping stone for your own work.

That's all…

Appendix: Table of Figures