# LCD Backgrounds:

## Tvorba obrazu pomocí logiky FPGA

*Verze 2.3 přeložená do češtiny 13. září 2025*

### Studijní text předmětu

### Logické systémy a procesory


## Richard Šusta


### Katedra řídicí techniky

### ČVUT-FEL v Praze


**Domovská stránka** dokumentu a zdrojový kód: https://dcenet.fel.cvut.cz/edu/fpga/navody.aspx

**GHDL** návod instalace: https://dcenet.fel.cvut.cz/edu/fpga/install.aspx

**FPGA-LCD Tools** zde zmíněné: https://github.com/cvut/FPGA-LCD_Utils



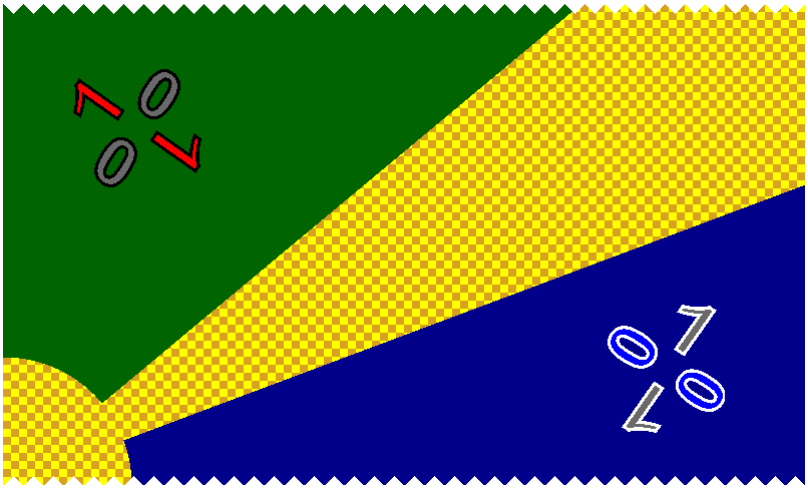

| | |
|---|---|
| Autor: | Richard Šusta,<br>susta@fel.cvut.cz, richard@susta.cz, https://susta.cz/ |
| Ilustrace: | Richard Šusta |
| Publisher: | Katedra řídicí techniky ČVUT-FEL v Praze,<br>Karlovo nám. 13<br>121 35 Praha 2<br>https://control.fel.cvut.cz/ |
| Datum vydání: | Srpen 2025 |

**Obsah**

# Úvod

Konfigurovatelné logické elementy, hlavní prvky FPGA obvodů, mohou některé obrázky vytvořit efektivněji, než v případě jejich načtení z BMP, JPEG či PNG souborů. Ty naopak dovedou úsporněji uložit snímky s řadou různorodých detailů. Pokud však kreslíme pozadí LCD ovládacího panelu, pak závisí jedině na nás, jak si ho navrhneme. Můžeme ho tedy složit z tvarů, v nichž exceluje logika.

Na obrázku dole vidíte zkušební pozadí 800x480 pixelů, které by se uložilo do 33817 bytového JPEG v 80% kvalitě či v bezeztrátové kompresi do **7384**-bytového PNG (Portable Network Graphics) souboru, jehož metody jsou více vyladěné pro grafiku s opakujícími se motivy.



**Obrázek 1— Příklad pozadí výhodněji tvořeného logikou**

Když se realizovalo logikou, stačilo mu 339 LE (Logických Elementů). Jeden LE uloží 2 byty, takže zabralo ekvivalent cca 680 bytů. Dále se na něj využilo 4096 bytů v ROM paměti, pomocí níž se tvořila písmena. Dohromady se v FPGA obsadil ekvivalent cca **4800** bytů, tedy zhruba 2/3 velikosti PNG souboru.

Úspora třetiny není zde ale rozhodujícím faktorem. PNG či JPEG obrázky nejsou totiž zakódované jako souvislá pole pixelů, ale po blocích a jejich dekomprese má několik kroků, při nichž vyplňuje a přepisuje různé části bitmapy, která se kvůli tomu musí umístit celá do paměti.

Pozadí nahoře by potřebovalo dalších 240 kilobytů paměti FPGA i při jeho úsporné konverzi na čtyřbitové indexy do tabulky barev, i tak by zabralo 40krát více, než potřebuje logika. A rozpakování obrázku bude zdržovat procesor, na němž se jeho složitý algoritmus musí realizovat.

Logické řešení navíc posílá pixely jako proud bitů (*stream*), tedy přesně tak, jak LCD panel pracuje. Můžeme je předávat přímo jemu bez nutnosti vyrovnávací paměti.

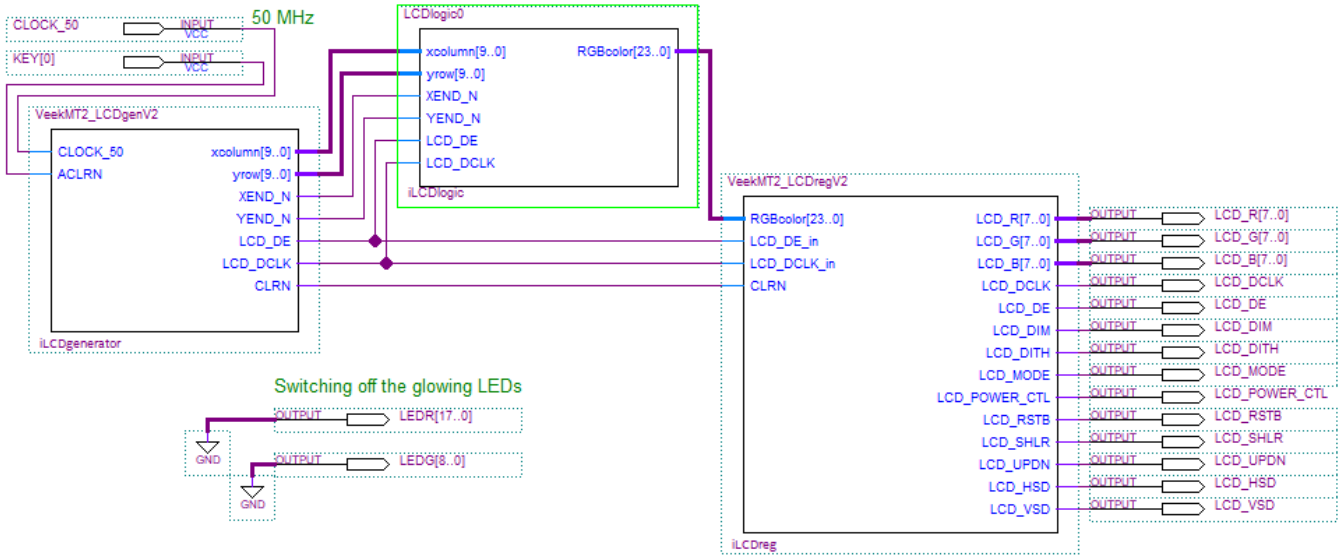
Přípustnou kompresí je samostatná RLE (Run-Length Encoding) metoda, která je rovněž dílčím krokem JPEG komprese. Její výsledek se snadno rozpakuje konečným automatem na proud bitů, jak LCD potřebuje. RLE komprese obrázku nahoře využije i při své optimální konfiguraci skoro 32 kilobytů paměti. Můžeme ji samozřejmě aplikovat na dílčí části, máme-li dost volného místa v FPGA a nebrání tomu podmínky zadání. Nástroj Bitmap2Vhdl z FPGA Utils má volbu možnosti uložení v RLE kompresi.

RLE komprese postrádá však jakoukoli možnost proměny. Konečný automat dokáže jen zobrazit obrázek. Je-li vytvořený logikou, pak ho lze dynamicky modifikovat dle vstupních dat.

Vytvořili jsme malý vzorník grafických motivů spolu s VHDL kódy, které je vykreslily, jako inspiraci demonstrující možnosti logiky. Všechny se zkoušely na vývojové desce Veek-MT2 od firmy Terasic, ale lze je upravit i pro jiná prostředí.

## Obvody LCD verze 2

Základní koncepce tvorby pozadí se sestavila ve vývojovém prostředí Quartus Lite, viz obrázek dole. Pouze prostřední entita LCDlogic0 přiřazuje barvy pixelům a lze ji pokládat za analogii kreslení. Generátor vlevo jí jen posílá synchronizační signály a registr vpravo uloží její výsledek, který si od něj převezme LCD panel.




Obrázek 2 — Základní zapojení tvorby obrázků

**Poznámka 1:** Součástí tohoto dokumentu je i přiložený ukázkový kód, který obsahuje všechny tři obvody ve VHDL, a to VeekMT2_LCDgenV2, VeekMT2_LCDregV2 a výchozí LCDlogic0.

**Poznámka 2:** Navržené uspořádání funguje obdobně jako procesorové pipelines (proudové zpracování). Generátor synchronizace vyšle hodnoty souřadnic okamžitého x a y pixelu v jednom hodinovém cyklu. LCDlogic0 mu v dalším taktu přiřadí barvu, která se načte do registru. Z něho se odešle do panelu LCD. Mezitím se již provedly dvě předchozí fáze, v nichž se počítaly následující pixely.

| Hodiny / Clocks | 1: VeekMT2_LCDgenV2 | 2: LCDlogic0 | 3: VeekMT2_LCDregV2 |
|---|---|---|---|
| ***Zapnuto*** | - | - | - |
| ***Hodiny 1*** | Souřadnice pixelu [x,y]=**[0,0]** | - | - |
| ***Hodiny 2*** | Souřadnice pixelu [x,y]=**[1,0]** | přiřadí barvu **[0,0]** | - |
| ***Hodiny 3*** | Souřadnice pixelu [x,y]=**[2,0]** | přiřadí barvu **[1,0]** | Barva **[0,0]**→ LCD |
| ***Hodiny 4*** | Souřadnice pixelu [x,y]=**[3,0]** | přiřadí barvu **[2,0]** | Barva **[1,0]**→ LCD |

## LCDpackV2.vhd – knihovna definic

**LCDpackV2.vhd** je balíček VHDL. Tento dokument předpokládá jeho verzi 2.1 nebo vyšší doplněnou o další definicemi. Ta je uvedena v záhlaví a V2.1 je shora kompatibilní s V2.0. Balíček definuje konstanty a funkce pro geometrii LCD panelu a převody barev. Je odkazován ve všech následujících kódech VHDL.

Jeho hlavní definice:

```
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
```

*Pozn. Balíčky jsou vysvětlené v Úvod do návrhu obvodů v jazyce VHDL I., kapitola 7, na str. 58 až 62.*

## VeekMT2_LCDgenV2 — generátor synchronizace

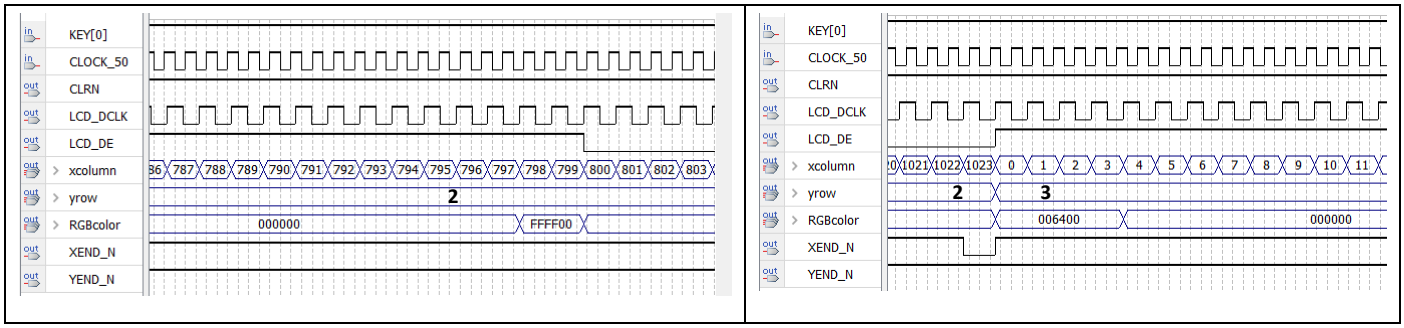
Generátor VeekMT2_LCDgenV2 obsahuje dvojici čítačů a komparátory jejich hodnot. Má jednoduché zapojení, které se hlavně dodržuje časování dle hardwarových specifikací v katalogu výrobce LCD panelu.

Analogii generátoru bychom v jazyce C napsali pomocí dvou cyklů:

```
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);
    }    // V hardwaru je funkce LCDlogic0 vytvořena logikou a dostává navíc hodiny LCD_DCLK.
}
```

**Vstupy**

- CLOCK_50 – vstup frekvence 50 MHz ze stejnojmenného pinu vývojové desky. Generátor se musí připojit přímo na něj bez jakékoli vložené logiky, jak to vyžaduje PLL (Phase-locked loop, cz. fázový závěs), který je v něm zahloubený a mění frekvenci z 50 Hz na 33 MHz. Každá elektronika, která pracuje na vyšších kmitočtech, zpravidla obsahuje nějaký typ PLL. Úpravami jeho parametrů se například provádí i přetaktování procesorů nebo grafických karet.
- ACLRN — je inicializace po zapnutí napájení. Na desce VEEK-MT2 se připojuje na KEY[0].



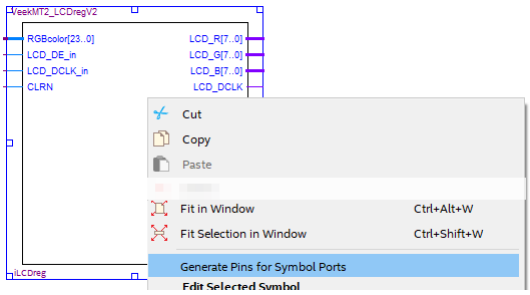

Obrázek 3 - Simulace výstupu generátoru

## VeekMT2_LCDregV2 — poslání barvy na LCD

Registr při náběžné hraně LCD_DCLK uloží barvu, kterou mu byla poslaná LCDlogic0 obvodem. Provádí i oříznutí obrázku tak, aby jeho výstupy LCD_R, LCD_G a LCD_B byly v 0 (černá barva), když je LCD_DE='0', jak požaduje LCD panel.

Výstupy registru se připojují k velkému zadnímu LCD displeji desky Veek-MT2.

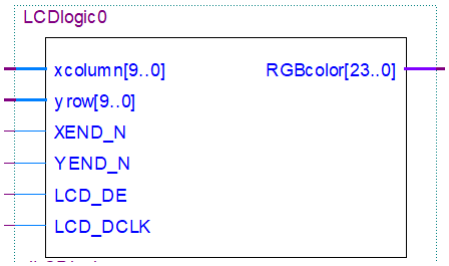
*Poznámka: Výstupní špičky VeekMT2_LCDregister se ve schématu *.bdf vygenerují automaticky přes kontextové menu jeho symbolu volbou* Generate Pins for Symbol Ports.



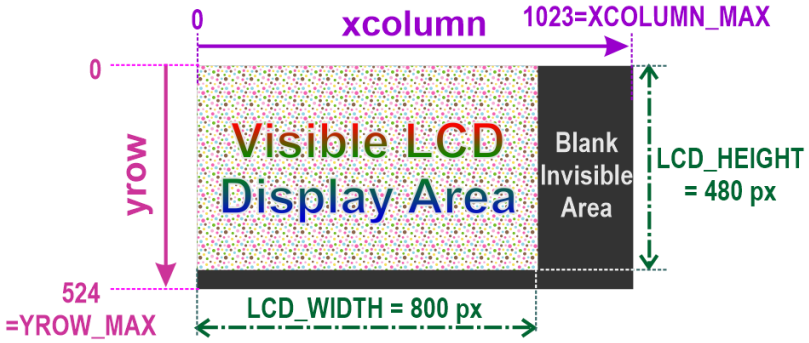

Obrázek 4 - Generate Pins for Symbol Ports

## LCDlogic0 — obvod kreslení obrázku

LCDlogic dostává souřadnice **xcolumn** a **yrow** z generátoru LCD synchronizace, přičemž orientace x a y os odpovídá Windows grafice. Obsahuje kombinační logiku, která okamžitému x, y pixelu přiřadí barvu RGBcolor. Jeho prototyp LCDlogic0 najdete v souboru ZIP spolu s generátorem a registrem.




Obrázek 5 — Kombinační obvod LCDlogic




Obrázek 6 — Rozměry dotykového LCD

**Vstupy LCDlogic0**

**xcolumn, yrow** - 10bitové signály souřadnic pixelů mají unsigned typ xy_t definovaný v balíčku LCDpackV2.vhd, v němž se nacházejí i další níže uvedené konstanty.
Sloupec **xcolumn** se mění od 0 do **1023=XCOLUMN_MAX**, ale viditelný obraz leží jen v rozsahu od 0 do **799=LCD_WIDTH-1**.
Řádek yrow se mění od 0 do **524=YROW_MAX**, avšak viditelná část bude jen v rozsahu od 0 do **479=LCD_HEIGHT-1**.

**XEND_N** bude v logické '0' při xcolumn=1023, jinak v '1'**.** Signalizuje poslední sloupec.
Má frekvenci **32.2 kHz** = 33 MHz/1024 = 33 MHz/(XCOLUMN_MAX+1)

**YEND_N** je v logické '0' při yrow=524, jinak v '1'**.** Signalizuje poslední řádek ze snímku (frame).
...Má frekvenci **61.4 Hz**=33 MHz/(1024*525)=33 MHz/**(**(XCOLUMN_MAX+1)*(YROW_MAX+1)**)**

**LCD_DE** je LCD Data Enable synchronizační signál. Při LCD_DE= '1' se posílají pixely patřící do viditelné oblasti. Ve sloupcích 800 až 1023 a v řádkách 480 až 524, tj. mimo viditelnou část, je signál **LCD_DE**='0'. LCD potřebuje neviditelné části, aby zapsalo načtený řádek a připravilo na příjem následujícího řádku, respektive snímku. Manuál výrobce vymezuje úseky signálu, v nichž musí být LCD_DE='0' a barva černá. Potřebné oříznutí obstarává VeekMT2_LCDregister.

**LCD_DCLK** — LCD Data Clock má frekvenci 33 MHz přesně, s pracovním cyklem 50 %.

**Výstupy**

**RGBcolor** - 24bitový std_logic_vector s 8bitovými hodnotami RGB barev. R barva je v horní osmici bitů, B v dolní. *Poznámka: Není zde alfa kanál nesoucí informací o neprůsvitnosti (opacity), protože případné iluze prosvítání se vytvářejí během předchozího zpracování. Monitory neznají alfa kanál barvy.*

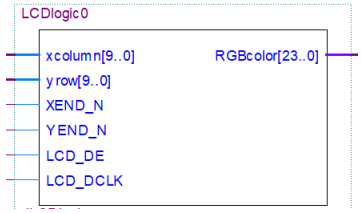
## Soubor testbenchV2_LCDlogic.vhd

Simuluje kreslení a ukládá barvy pixelů v komprimované formě do textového souboru, který lze načíst z Testbench Viewer z FPGA-LCD Utils a zobrazit vytvořený LCD obrázek.

Testbench obsahuje vlastní generátor synchronizace v optimalizované formě pro simulaci a rovněž svůj registr. Vkládá se do něho jen LCDlogic*

## Prototyp kódu

Začneme specifikací vstupů a výstupů s využitím typů z LCDpackV2.




Použijeme následující definice z balíčku LCDpackV2:

```
subtype  xy_t is unsigned(9 downto 0); --pro data xcolumn a 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
```

Napíšeme si entitu a architekturu:

```
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;
```

**Důležité poznámky:**

1. Dodržujte pojmenovací trojici, jinak se kód stane nekompilovatelným. Soubor **LCDlogic0.vhd** obsahuje entitu **LCDlogic0** s architekturou behavioral pro **LCDlogic0**. Identifikátor behavioral je lokální, platný pouze v rámci částí patřících entitě. V jiné entitě ho lze opět použít.
2. Entita obsahuje i vstupy, na něž se zatím neodkazuje, aby se zvýšila její univerzálnost. Překladač vynechá vše nepoužité při minimalizaci. Budeme-li však v budoucnu potřebovat nějaký další vstup, máme ho k dispozici, nemusíme ho doplnit do entity a znovu generovat její schematickou značku.
3. Inicializace hodnot signálů a proměnných v definicích se uvádějí hlavně kvůli simulaci. V syntéze se vykonají pokaždé jedině u definic konstant nebo u lokálních proměnných ve funkcích a procedurách. V sekci **process** se proměnné musí **vždy inicializovat přiřazovacím příkazem v jeho hlavním kódu**.
4. Klíčové slovo **process** otevírá sekvenční doménu VHDL. LSPimage je u něho nepovinné návěští. V syntéze nelze na něj odkazovat, ale v simulaci ho uvidíme jako orientační odkaz. Seznamy v závorkách za klíčovým slovem **process** nejsou parametry, ale "sensitivity list", který vyjmenovává signály, jejichž změny mohou způsobit změnu jeho výstupů.

## Soubor runlcd.bat

Dávka **runlcd.bat** obsahuje příkazy skriptovacího shell jazyka a podobá se runmorse.bat popsanému v příručce na webu DCENET: Instalace a používání jazyka GHDL, strana 7. Nevytváří výstup pro GtkWave, a tak v "**ghdl.exe -r"** chybí paramter --vcd. Výsledek se zobrazí Testbench Viewer z FPGA-LCD Utils.

```
@ECHO OFF
rem SETLOCAL — následující definice budou zrušené po skončení dávky. DŮLEŽITÉ, nikdy nevynechejte!
SETLOCAL
rem Název souboru testbench musí být bez přípony, protože jeho jméno se využívá i pro další komponenty
set TBNAME=testbenchV2_LCDlogic
rem Soubory mají přípony a relativní cesty do nadřízeného adresáře. Uveďte je ve správném pořadí jejich kompilace!
set FILES=../LCDpackV2.vhd ../LCDlogic0.vhd
rem Doba běhu simulace v jejím čase.
nastavit SIMTIME=20ms
rem Přesuneme mingw64 na vrchol PATH, což bude jen dočasné díky SETLOCAL
set PATH=C:\msys64\mingw64\bin\;%PATH%
rem GHDL se kompiluje pro 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
```

Jeho příkazy podrobněji:

`ECHO OFF` – značí, že při zpracování *.bat se nevypisují příkazy, které byly provedené.
Při ON se vypisuje příkazy, které nezačínají znakem @

`SETLOCAL` - spustí lokalizaci proměnných prostředí. Změny budou platné jen do konce dávkového souboru nebo do dosažení odpovídajícího příkazu ENDLOCAL. **Bez SETLOCAL by byly trvalé!**

`rem` - následující text je komentářem až do konce řádky.

`set TBNAME` – entita testbench, a to pouze její název, bez přípony nebo cesty.
Na parametr nastavený `set` se odkazuje %jeho_nazev%, např. %TBNAME%.

`set FILES=` – soubor(y), z nichž se sestavuje obvod. Oddělují se mezerami a musí se uvést v pořadí jejich kompilace! Nepřidávejte generátor synchronizace a registr — testbench je má již v sobě.

`set SIMTIME` – pojistka doby simulačního běhu, který trvá 16,6 ms simulovaného času, pak se sám zastaví. Kdyby ne, někde se stala chyba, a tak ho pojistka stopne po 20 milisekundách.

`set PATH` – pouze dočasně přesuneme cestu k mingw64 na začátek díky předchozímu `SETLOCAL` .

`set GHDL_FLAGS` — zapnutí podpory VHDL 2008. GHDL ho umí skoro celý.

`ghdl.exe -a` – analyzujeme VHDL soubory. Parametr FILES musí předcházet *souboru testbench*.

`IF ERRORLEVEL 1 GOTO BAT-END` - skočíme na konec, pokud předchozí příkaz skončil chybou.

`ghdl.exe -e` – vytvoří se simulace obvodu v souboru TBNAME.exe

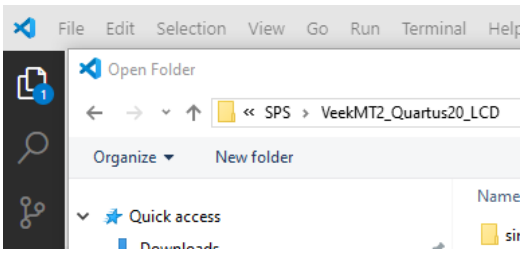
`ghdl.exe -r` – spustí se TBNAME.exe

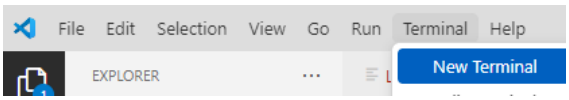
*Poznámka: Pokud chceme simulovat jiný VHDL soubor, změníme příkazy*
**set FILES=** a případně `set` TBNAME používáme-li jiný testbench

## Spuštění simulace GHDL

GHDL umožňuje rychlejší ladění. V projektu Demo se nachází **runlcd.bat**, úmyslně umístěný do podadresáře simulation, aby tam zůstaly i všechny dočasné soubory vytvořené GHDL.

Nejprve otevřeme složku projektu v bezplatné aplikaci VSC, jak se běžně zktracuje Visual Studio Code. Poté si vytvoříme nový terminál.




V terminálu zadáme dva příkazy Windows PowerShellu:

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

Lze napsat jen **cd s** a po stisku tabelátoru si VSC příkaz samo doplní. Stejně tak po napsání **./r** lze tabelátorem doplnit zbytek. Simulace vypíše provedené příkazy a úspěšně je ukončí oznámením:

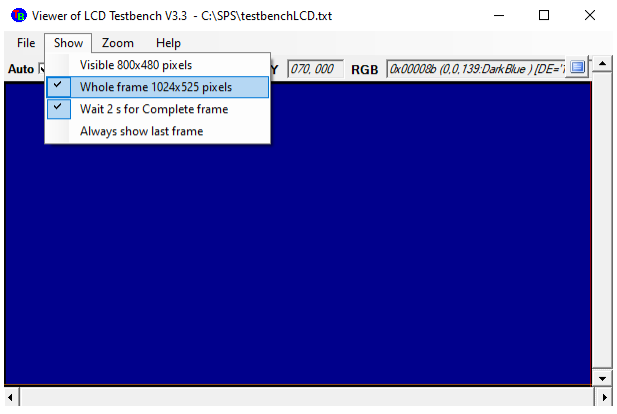
**:-) OK end of SINGLE frame simulation.**

```
.\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>
```

Ignorujeme "error" zprávy po **:-) OK end**. Ohlášení chyby, že vše OK:-), je ve VHDL paradoxně způsob k zastavení simulace. Její výsledek lze zobrazit pomocí programu Testbench Viewer.

Všem pixelům z viditelné i neviditelné oblasti jsme v protypu kódu (str. 7) přiřadili tmavě modrou barvu, což může být matoucí, je-li prohlížeč Testbench Viewer přepnut do celoobrazovkového režimu, jako na obrázcích dole. Pro lepší orientaci si můžete přidat oříznutí (clipping) příkazem **if**. Registr sice ořezává, takže není nutné ho vložit do LCDlogic, ale můžeme pro naši orientaci. Další kódy budou bez něho.

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




Obrázek 7 - Vše tmavomodré


Obrázek 8 - Oříznutí (crop)
+ zdůrazněné pozice XEND_N a YEND_N

## Tabulka barev

Šikovná tabulka barev uspořádaných podle jejich odstínu se nachází na Reddit. Webové hexadecimální kódy se ve VHDL píší s X. Například barvu maroon #800000 napíšeme jako X"800000" či konverzní funkcí ToRGB(128, 0, 0). *Pozn. Formátů hex- zápisů je hodně, viz přehled* Wiki: Distinguishing from decimal

### HEXADECIMAL COLOR CODES

| Color | Hex Code #RRGGBB |
|---|---|
| maroon | #800000 |
| dark red | #8B0000 |
| brown | #A52A2A |
| firebrick | #B22222 |
| crimson | #DC143C |
| red | #FF0000 |
| tomato | #FF6347 |
| coral | #FF7F50 |
| indian red | #CD5C5C |
| light coral | #F08080 |
| dark salmon | #E9967A |
| salmon | #FA8072 |
| light salmon | #FFA07A |
| orange red | #FF4500 |
| dark orange | #FF8C00 |
| orange | #FFA500 |
| gold | #FFD700 |
| dark golden rod | #B8860B |
| golden rod | #DAA520 |
| pale golden rod | #EEE8AA |
| dark khaki | #BDB76B |
| khaki | #F0E68C |
| olive | #808000 |
| yellow | #FFFF00 |
| yellow green | #9ACD32 |
| dark olive green | #556B2F |
| olive drab | #6B8E23 |
| lawn green | #7CFC00 |
| chart reuse | #7FFF00 |
| green yellow | #ADFF2F |
| dark green | #006400 |
| green | #008000 |
| forest green | #228B22 |
| lime | #00FF00 |
| lime green | #32CD32 |
| light green | #90EE90 |
| pale green | #98FB98 |
| dark sea green | #8FBC8F |
| medium spring green | #00FA9A |
| spring green | #00FF7F |
| sea green | #2E8B57 |
| medium aqua marine | #66CDAA |
| medium sea green | #3CB371 |
| light sea green | #20B2AA |
| dark slate gray | #2F4F4F |
| teal | #008080 |
| dark cyan | #008B8B |

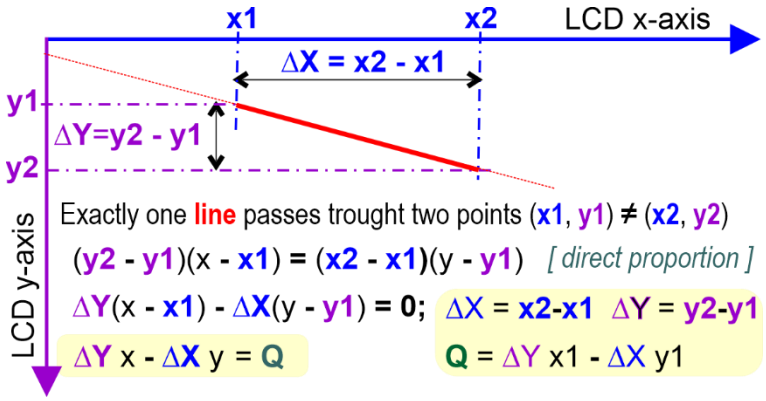
| Color | Hex Code #RRGGBB |
|---|---|
| aqua | #00FFFF |
| cyan | #00FFFF |
| light cyan | #E0FFFF |
| dark turquoise | #00CED1 |
| turquoise | #40E0D0 |
| medium turquoise | #48D1CC |
| pale turquoise | #AFEEEE |
| aqua marine | #7FFFD4 |
| powder blue | #B0E0E6 |
| cadet blue | #5F9EA0 |
| steel blue | #4682B4 |
| corn flower blue | #6495ED |
| deep sky blue | #00BFFF |
| dodger blue | #1E90FF |
| light blue | #ADD8E6 |
| sky blue | #87CEEB |
| light sky blue | #87CEFA |
| midnight blue | #191970 |
| navy | #000080 |
| dark blue | #00008B |
| medium blue | #0000CD |
| blue | #0000FF |
| royal blue | #4169E1 |
| blue violet | #8A2BE2 |
| indigo | #4B0082 |
| dark slate blue | #483D8B |
| slate blue | #6A5ACD |
| medium slate blue | #7B68EE |
| medium purple | #9370DB |
| dark magenta | #8B008B |
| dark violet | #9400D3 |
| dark orchid | #9932CC |
| medium orchid | #BA55D3 |
| purple | #800080 |
| thistle | #D8BFD8 |
| plum | #DDA0DD |
| violet | #EE82EE |
| magenta / fuchsia | #FF00FF |
| orchid | #DA70D6 |
| medium violet red | #C71585 |
| pale violet red | #DB7093 |
| deep pink | #FF1493 |
| hot pink | #FF69B4 |
| light pink | #FFB6C1 |
| pink | #FFC0CB |
| antique white | #FAEBD7 |

| Color | Hex Code #RRGGBB |
|---|---|
| beige | #F5F5DC |
| bisque | #FFE4C4 |
| blanched almond | #FFEBCD |
| wheat | #F5DEB3 |
| corn silk | #FFF8DC |
| lemon chiffon | #FFFACD |
| light golden rod yellow | #FAFAD2 |
| light yellow | #FFFFE0 |
| saddle brown | #8B4513 |
| sienna | #A0522D |
| chocolate | #D2691E |
| peru | #CD853F |
| sandy brown | #F4A460 |
| burly wood | #DEB887 |
| tan | #D2B48C |
| rosy brown | #BC8F8F |
| moccasin | #FFE4B5 |
| navajo white | #FFDEAD |
| peach puff | #FFDAB9 |
| misty rose | #FFE4E1 |
| lavender blush | #FFF0F5 |
| linen | #FAF0E6 |
| old lace | #FDF5E6 |
| papaya whip | #FFEFD5 |
| sea shell | #FFF5EE |
| mint cream | #F5FFFA |
| slate gray | #708090 |
| light slate gray | #778899 |
| light steel blue | #B0C4DE |
| lavender | #E6E6FA |
| floral white | #FFFAF0 |
| alice blue | #F0F8FF |
| ghost white | #F8F8FF |
| honeydew | #F0FFF0 |
| ivory | #FFFFF0 |
| azure | #F0FFFF |
| snow | #FFFAFA |
| black | #000000 |
| dim gray / dim grey | #696969 |
| gray / grey | #808080 |
| dark gray / dark grey | #A9A9A9 |
| silver | #C0C0C0 |
| light gray / light grey | #D3D3D3 |
| gainsboro | #DCDCDC |
| white smoke | #F5F5F5 |
| white | #FFFFFF |

**Obrázek 9 - Tabulka pojmenovaných barev převzatá z Reddit**

## Vzorník tvarů s přímkami

Rovnici přímky, která prochází dvojicí různých bodů, lze odvodit přímou úměrou. Hardwarová realizace přímky si však žádá celočíselné koeficienty a vede na úspornější obvod, mají-li i největší společný dělitel gdc větší než 1 a vykrátí se na menší čísla.



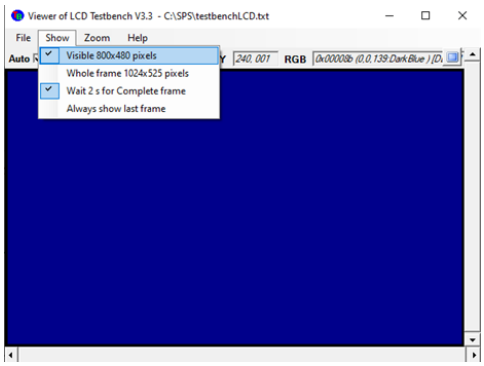

Obrázek 10 - Odvození rovnice přímky

Rovnice přímky (i elips) lze najít například pomocí již zmíněných LCD Geometry Rulers v FPGA-LCD Utils, které se v mnohém podobají známému nástroji Geodebra , avšak adaptovanému na celočíselné výsledky a souřadnicový systém LCD displejů, v nichž osa y běží z historických důvodů od shora dolů.

```
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;
```

**Výsledek**

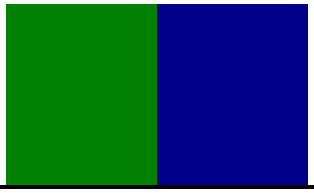
vsc



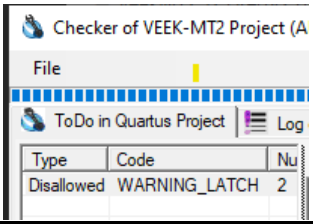
V dalších kódech se mění jen řádky v části "*our image* "

*Upřednostníme samostatným příkazům* **if-then**. *Příkazy* **if** *s vyšší prioritou se řadí za* **if** *s nižší prioritou, což pokládáme za vhodnější pro lepší orientaci v kódu než dlouhé kaskády příkazů* **if-elsif-elsif**... *Quartus implementuje samostatné příkazy* **if-then** *stejně efektivně jako kaskádu* **if-elsif-elseif**. *Vyzkoušeno.*
*V méně přehledné kaskádě* **if-elsif-elseif**... *se častěji udělá chyba. Též ověřeno mnoha našimi studenty*☺

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

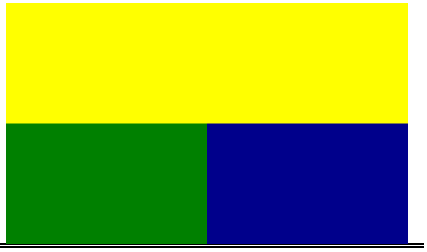


```
-- INCORRECT code without assigning the default value for RBG
---------- our image ----------------------
if x<LCD_WIDTH/2 then RGB:=GREEN; end if;
---------- LATCH in our code, RGB is not always assigned in the process code
--Compiler:Info (10041): Inferred latch for LSPimage:RGB
```

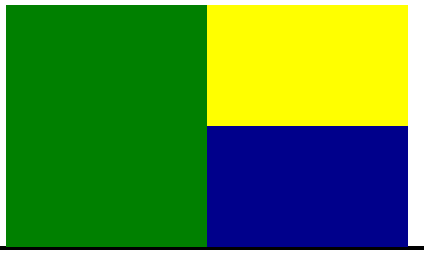


```
---------- our image ------------------------
RGB :=DARKBLUE;
if x<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
-----------------------------------------------
```

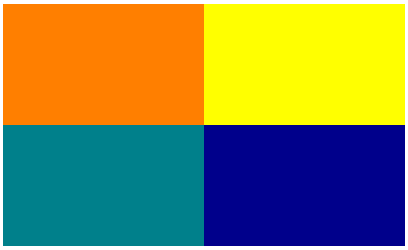
```
---------- our image ------------------------
RGB :=DARKBLUE;
if x<LCD_WIDTH/2 then RGB:=GREEN; end if;
if y<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
-----------------------------------------------
```



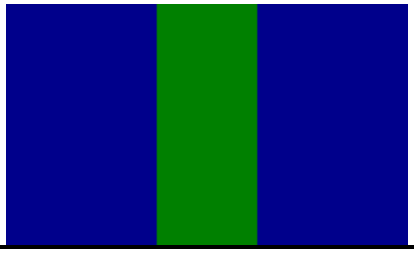
```
---------- our image ------------------------
RGB :=DARKBLUE;
if y<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
if x<LCD_WIDTH/2 then RGB:=GREEN; end if;
-----------------------------------------------
```



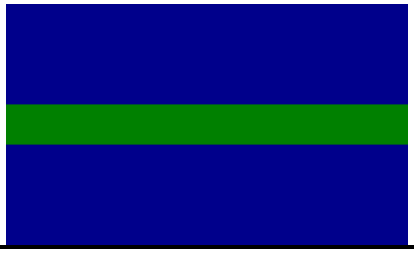
```
---------- our image ------------------------
RGB :=DARKBLUE;
if y<LCD_HEIGHT/2 then RGB:=YELLOW; end if;
if x<LCD_WIDTH/2 then RGB:=RGB xor GREEN; end if;
-----------------------------------------------
```



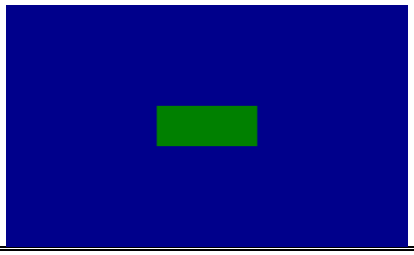
```
---------- our image ------------------------
RGB :=DARKBLUE;
if (x>=300) and (x<500) then RGB:= GREEN; end if;
-----------------------------------------------
```



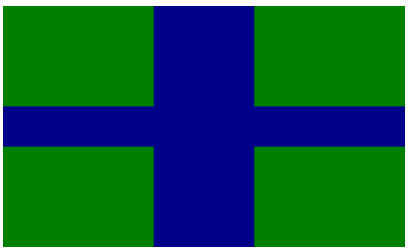
```
---------- our image ------------------------
RGB :=DARKBLUE;
if (y>=200) and (y<280) then RGB:= GREEN; end if;
-----------------------------------------------
```



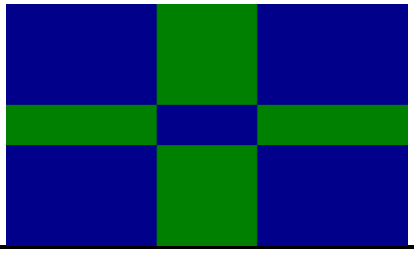
```
---------- our image ------------------------
RGB :=DARKBLUE;
if (x>=300) and (x<500) and (y>=200) and (y<280)
   then RGB:= GREEN; end if;
-----------------------------------------------
```



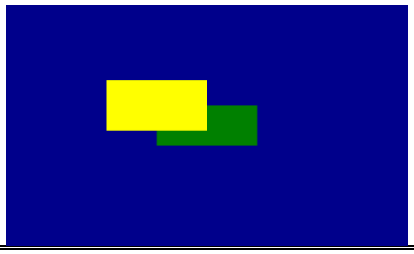
```
---------- our image ------------------------
RGB :=DARKBLUE;
if ((x<300) or (x>=500)) and ((y<200) or (y>=280))
   then RGB:= GREEN; end if;
-----------------------------------------------
```



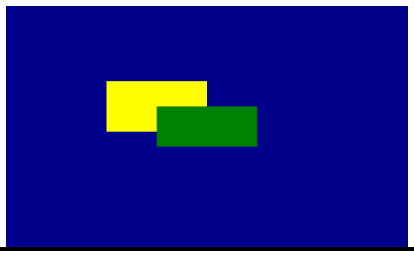
```
---------- our image ------------------------
RGB :=DARKBLUE;
if ((x<300) or (x>=500)) xor ((y<200) or (y>=280))
   then RGB:= GREEN; end if;
-----------------------------------------------
```



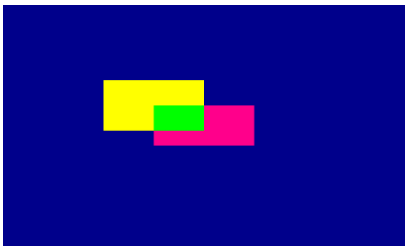
```
---------- our image ------------------------
RGB :=DARKBLUE;
if (x>=300) and (x<500) and (y>=200) and (y<280) then RGB:= GREEN; end if;
if (x>=200) and (x<400) and (y>=150) and (y<250) then RGB:= YELLOW; end if;
-----------------------------------------------
```



```
---------- 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;
-----------------------------------------------
```

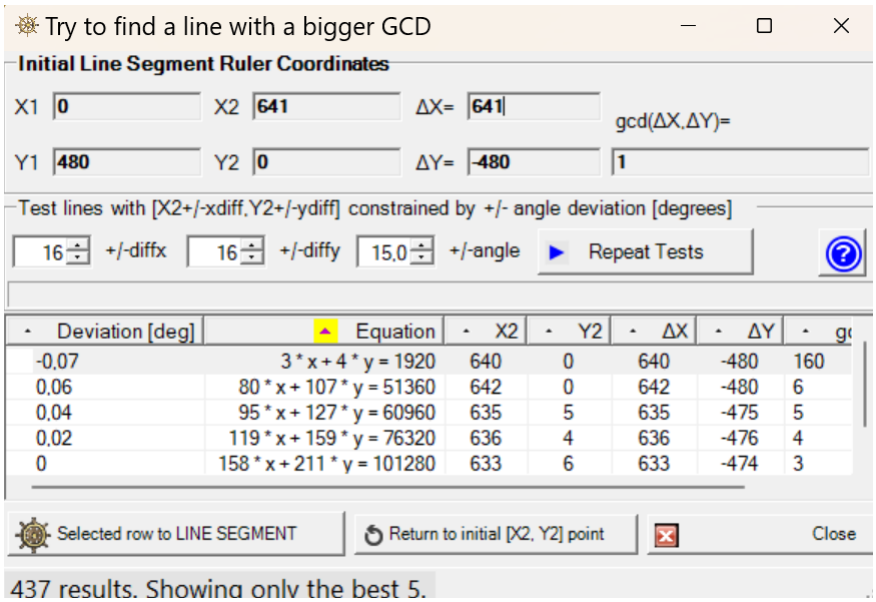


```
---------- our image ------------------------
RGB :=DARKBLUE;
if (x>=200) and (x<400) and (y>=150) and (y<250) then RGB:= YELLOW; end if;
if (x>=300) and (x<500) and (y>=200) and (y<280) then RGB:=RGB xor RED; end if;
-----------------------------------------------
```



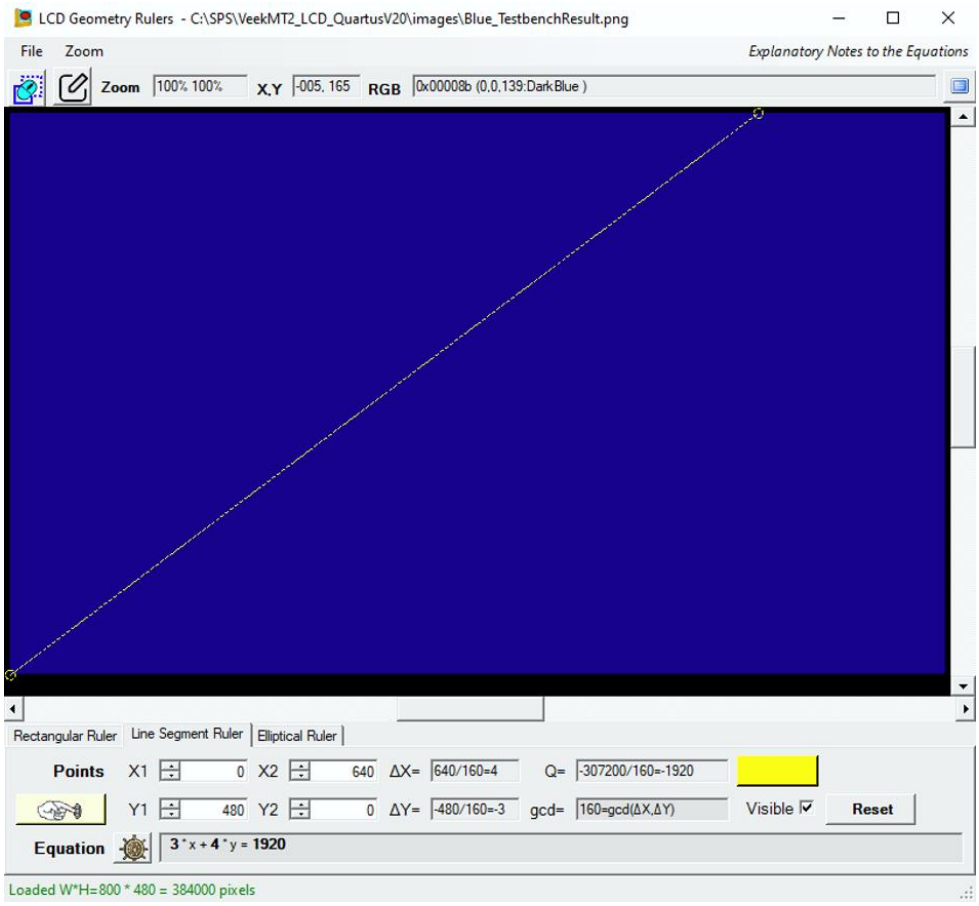
LCD Geometry Rulers z FPGA-LCD Utils naleznou i koeficienty šikmých úseček. Otevřeme si v nich obrázek o rozměru LCD 800x480 pixelů, například uložený obrázek z Testbench Viewer. Vložíme přímku a můžeme optimalizovat její pozici (ikona kormidelního kola), kdy se variacemi polohy koncového bodu X2,Y2 hledají přímky a elipsy s výhodnější obvodovou implementací.

Například sklon přímky 641/480 je méně vhodný. Snáze vyjde sklon 640/480 = 4/3, tahle přímka se liší jen o 0,07 úhlového stupně. Zkrátíme její zlomek největším společným dělitelem 160 a násobíme menšími čísly, což redukuje počet logických prvků.



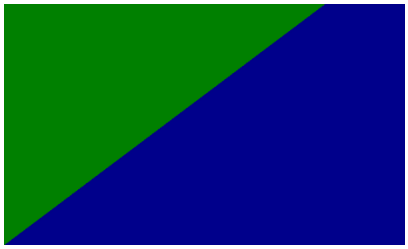

Obrázek 11 - Dialog optimalizace přímky v LCD Geometry Rulers z FPGA-LCD Utils

Pokud v rovnici přímky nahradíme = vhodnou nerovností, lze ji využít jako podmínku k přiřazení barvy celé oblasti LCD. Kombinací podmínek vytvoříme jakékoli útvary ohraničené přímkami, viz následující obrázky. Všechny se uložily z Testbench Viewer výstupů.



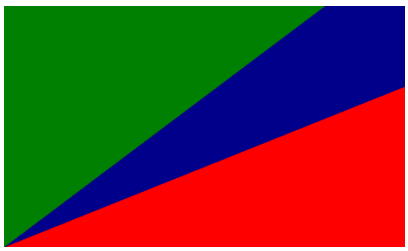

Obrázek 12 - LCD Geometry Rulers — Optimální rovnice přímky

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



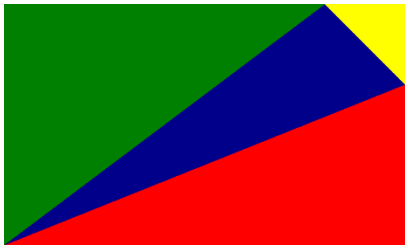
---

```
--------- 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;
-----------------------------------------------
```



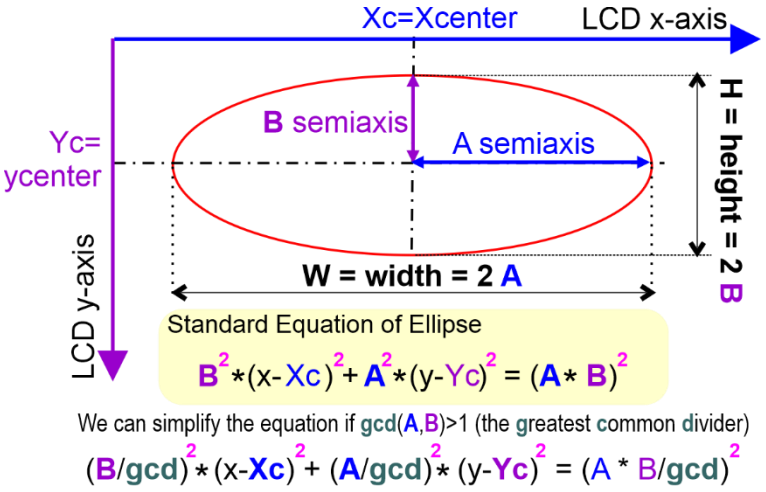
---

```
--------- 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;
-------------------------------------------------------
```



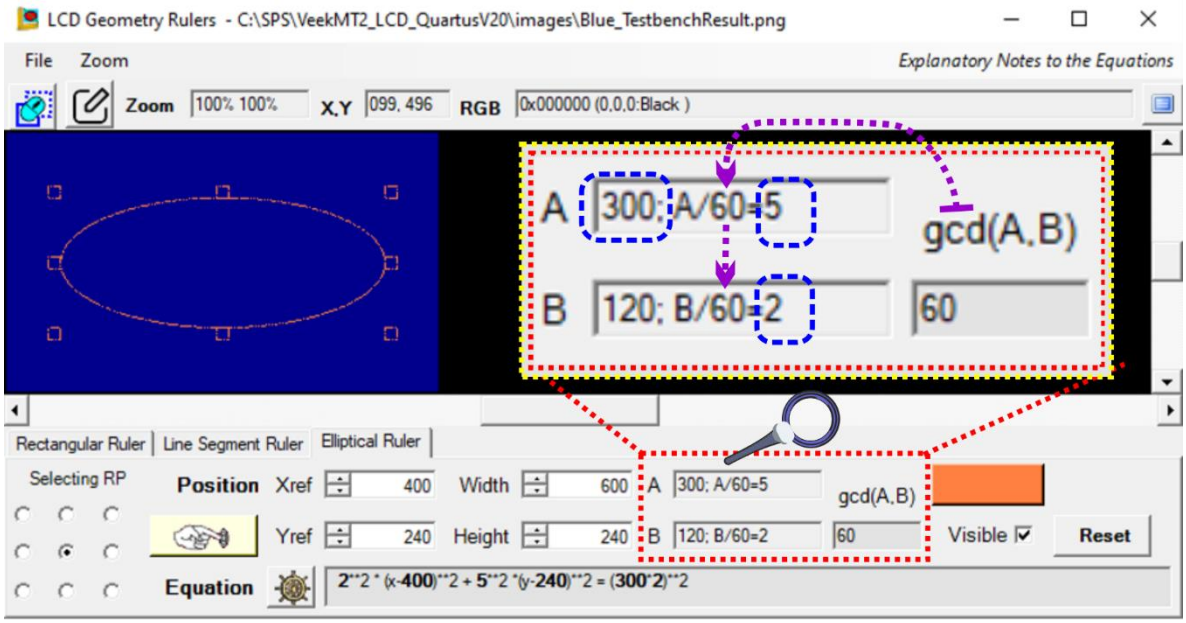
## Vzorník elips

Má-li elipsa vodorovné a kolmé osy, je v kanonickém tvaru. Její hardwarová implementace opět vyjde jednodušeji, když se koeficienty její rovnice dají vykrátit jejich největším společným dělitelem (gdc).



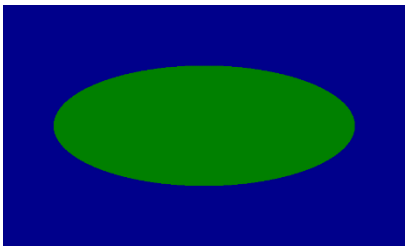

Obrázek 13 - Rovnice elipsy v kanonickém tvaru

Využijeme třeba opět LCD Geometric Rulers, kde vložený návrh optimalizujeme hledáním blízkých elips s výhodnější hardwarovou implementací (ikona kormidelního kola).




Obrázek 14 - Nalezení rovnice elipsy

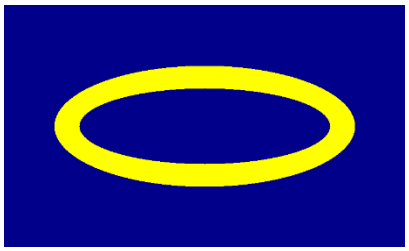
```
---------- our image ----------------------
RGB:=DARKBLUE;
-- (2=B/gdc)**2       (5=A/gdc)**2        (A*(B/gdc))**2
 if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2 then RGB:=GREEN; end if;
-------------------------------------------------------
```



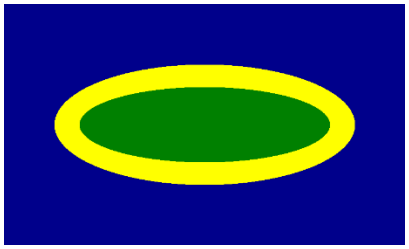
Obecná elipsa má osy otočené o úhel θ a její kvadratickou rovnici můžeme odvodit z kanonického tvaru, když se na něj aplikuje euklidovská rotace souřadnic. Potřebné vzorce, do nichž se zadávají kanonické A a B koeficienty a úhel θ, naleznete například na anglické Wiki stránce, odstavec General Ellipse, či na WolframCloud, sekce Details and Options.

Pokud sestavujeme LCD pozadí, bude výhodnější skládat ho z fragmentů elips v kanonickém tvaru.

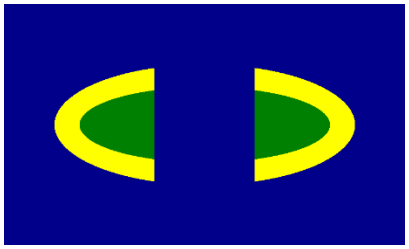
```
---------- our image ----------------------
RGB:=DARKBLUE;
-- (B/gdc)**2       (A/gdc)**2          (A*B/gdc)**2
if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2
  and 3**2 *(x-400)**2 + 10**2 *(y-240)**2 > (250*3)**2
  then RGB:=YELLOW; end if;
-----------------------------------------------------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



```
---------- 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;
-----------------------------------------------------
```

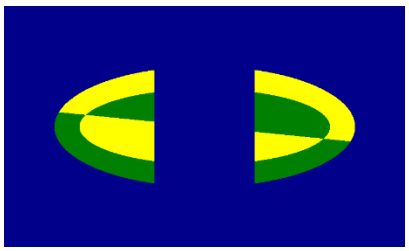


```
RGB:=DARKBLUE;
if (x<300) or (x>=500) then
    if 2**2 *(x-400)**2 + 5**2 *(y-240)**2 < (300*2)**2
        then RGB:=YELLOW; end if;
   if 3**2 *(x-400)**2 + 10**2 *(y-240)**2 < (250*3)**2
        then RGB:=GREEN; end if;
end if;
-----------------------------------------------------
```



Další obrázek má složitější kód, a tak vložíme jeho celou architekturu, a ne pouhou "*our image*" část

```
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;
```



FPGA rodiny Cyclone IV, ve vývojové desce Veek-MT2, obsahuje 115 tisíc logických elementů (LE). Obrázek nahoře potřeboval jen 177 LE, což je zhruba 360 bytů- K tomu se použilo deset hardwarových 9-bitových násobiček, což znamená jen 2 % ze všech v FPGA.

Obrázek uložený jako PNG by zabral cca 6,6 KB a JPEG v 80% kvalitě dokonce 15 KB.

## Otázka: Proč jsme podmíněné přiřazení nenapsali jako when - else?

VHDL-2008 dovoluje pomocí when-else zkrátit podmíněné přiřazení, pro které v jazyce C pro něj existuje operátor **? :**. VHDL kód by vypadal takto:

```
--------- 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
-------------------------------------------------------------------------------------------------------------
```

Simulátor GHDL umí skoro celé VHDL-2008 a v něm můžeme kratší when-else použít. Žel bezplatná verze Quartus Lite dovolí jen fragmenty z VHDL 2008 a tuhle šikovnou operaci, jako na potvoru, nepodporuje :-( Akceptuje ji jedině ve své placené verzi. A chceme-li výsledek nahrát do desky, musíme ho přeložit v Quartusu, a tak jsme se vynechali konstrukci, kterou by bezplatná verze odmítla .

Přiřazení lze nahradit šikovnou funkcí

```
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;
```

Pro časté své aplikace se přidala do verze 2.1 balíčku LCDpackV2.

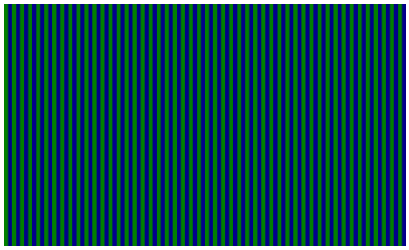
```
architecture behavioral of LCDlogic0 is
 constant DARKBLUE: RGB_t := ToRGB(0, 0, 139); -- the background
begin -- architecture
LSPimage : process( xcolumn, yrow, LCD_DE )
  variable RGB :RGB_t :=BLACK; -- the color of current pixel
  variable x : integer  range 0 to XCOLUMN_MAX:=0;
  variable y : 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;
```

Funkce assignIf převádí jen zadaný typ RGB_t, což je nevýhodou oproti univerzálnějšímu when-else. Lze ji však přetížit, podobně jako v C, a definovat si další stejného názvu. Balíček obsahuje i definice pro integer.

## Vzorník opakovaných tvarů pomocí dělení mocninou 2

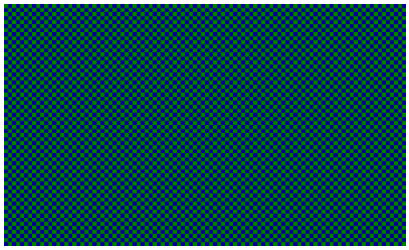
Logická tvorba obrázku umí velmi efektivně uložit tvary, které se opakují. Využívá toho, že každý snímek se celý generuje jako proud pixelů. Pokud vybranému elementu vhodně změníme souřadnice, které se mu posílají, na periodické, zopakuje se. Na ukázku budeme měnit barvu dle sudého výsledku celočíselného dělení x číslem $8=2^3$. V hardwaru se výraz ((x / 8) mod 2)=0 implementuje testem 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;
-------------------------------------
```



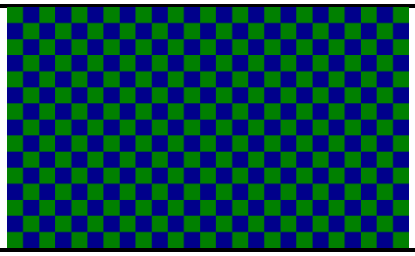
---

```
---------- 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;
-------------------------------------------------------
```



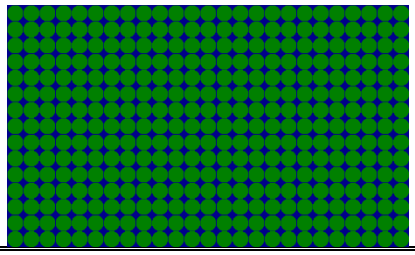
---

Opakovat můžeme i složitější tvary, když je duplikujeme na celou plochu. Začneme od jednoho výskytu:

```
---------- 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;
-------------------------------------------------------
```

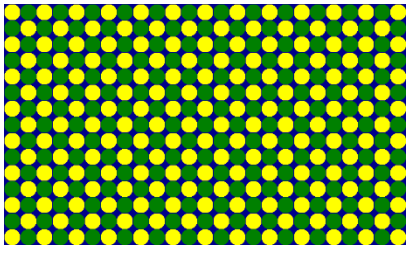
Nyní místo x a y použijeme jejich zbytky po jejich dělení 32.

```
---------- 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;
-------------------------------------------------------
```

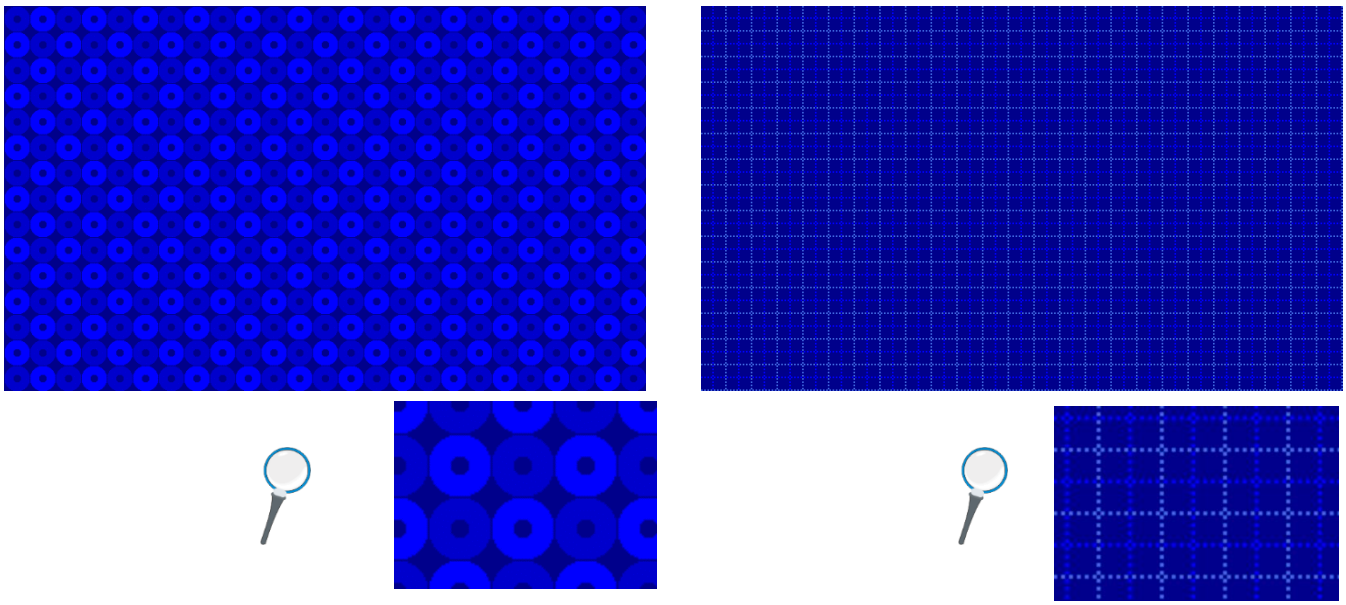


---

Složitost implementace posledního obrázku v LCDlogic0 je pouhých 9 logických elementů a dvě 9-bitové násobičky. Celé kreslení včetně generátoru a registru se vytvoří 77 logickými elementy a již zmíněnými dvěma 9-bitovými násobičkami. PNG by uložilo motiv s kruhy do 41 KB a JPEG dokonce do 141 KB.

Málokteré pozadí by se budovalo příliš výraznými kruhy, kromě demonstrace možností logiky:-) Když zeslabíme jejich barevné rozdíly, třeba podle tabulky na str. 10, v níž jsou barvy uložené podle jejich odstínů. Výsledný jemnější ozdobný motiv můžeme využít ke zkrášlení našeho návrhu:

```
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;
```

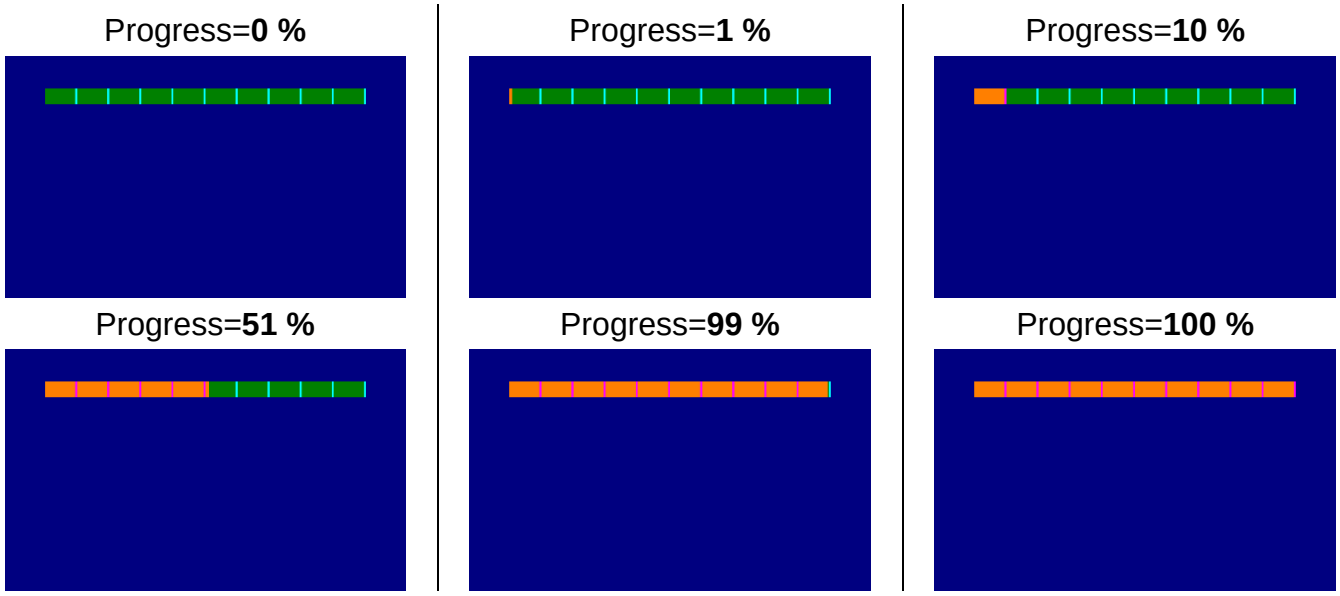


Obrázek 15 - Technické motivy: vlevo vytvoření kód nahoře, vpravo dole

Technické panely se ozdobí třeba analogiemi mřížek, nejlépe s přerušovanými linkami. Podmínky pro svislé a vodorovné čáry jsou oddělené — vždyť jejich kreslení je navzájem nezávislé! Přerušování se pak vytvořilo vložení podmínky na proměnnou běžící v ose čáry, např. čáry vymezené y je to x.

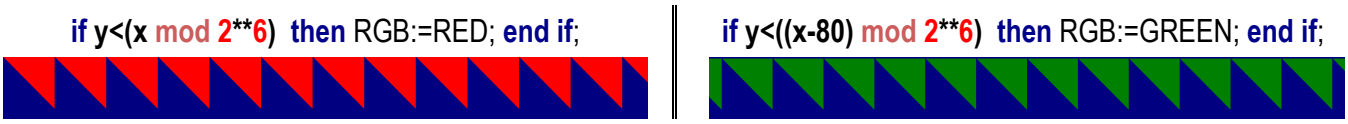
```
-------- our image ----------------------------------------------------------------------------
  RGB:=DARKBLUE;
  if (y mod 16>=14) and (x mod 4)<2 then
     if ((y/16) mod 2) = 0 then RGB:=X"0000FF"; else RGB:=X"4169E1"; end if;
  end if;
  if (x mod 16>=14) and (y mod 4)<2 then
     if ((x/16)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

Ukazatele postupu nějaké operace bývají častou součástí panelů, viz obrázek dole.




Obrázek 16 - Lineární ukazatel

K jeho realizaci využijeme zbytek po dělení číslem 2**6=64. Příkaz if vlevo ukazuje nesymetrický výsledek, protože 64 nedělí šířku LCD_WIDTH=800 bezezbytku. Příkaz if vpravo, v němž se pro odlišení použila jiná barva, vycentroval motiv posunem o 80 = (800-10*64)/2




Zavedeme-li si konstanty P0 pro počátek v ose x a krok ST (step) = 64, pak architektura bude

```
architecture indicator of LCDlogic0 is
signal progress:integer range 0 to 100:=51; -- 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;
```

## Opakované tvary generované čítačem

V předchozím kódu se hodnota uložená v **progress** přepočítávala na délku v ose x složitým vzorcem: **(progress***205+16)/32, v němž přičtení 16 emulovalo zaokrouhlení. Vztah, který roztáhne hodnotu progress, která běží od 0 do 100 % na interval 0 až 640 pixelů, lze přepsat jako

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

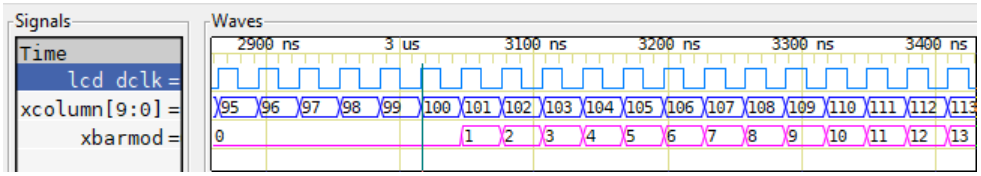
Výhodnější přepočet by nám vyšel, kdyby krok ST (step) měl hodnotu 60, pak by se progress násobil 6, ale obvod počítající x mod 60 by v hardwaru potřeboval mnoho logických elementů. Nahradíme ho čítačem Souřadnice pixelů **xcolumn a yrow** se mění na náběžnou hranu LCD_DCLK, a tak ho necháme běžet na sestupnou hranu LCD_DCLK, kdy jsou stabilní a dají se testovat bez rizika metastability. Výsledek přiřadíme signálu **xbarmod** na náběžnou hranu LCD_DCLK, tedy shodně se změnami souřadnic pixelů.
Využijeme zde i přetíženou **assignIf** pro čísla integer uvedenou v balíčku.

```
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;
```

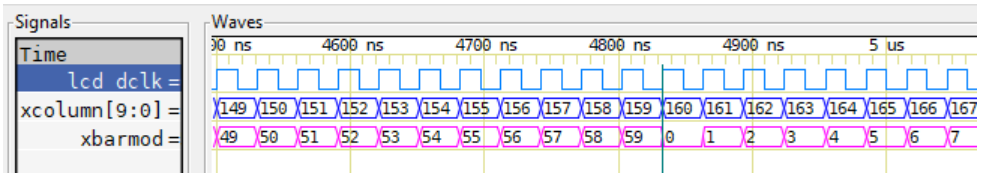
Chceme-li vidět výstupy procesu iModulo, napíšeme si testbench, do něhož vložíme jeho kód společně s nezbytnými definicemi:

```
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;
```

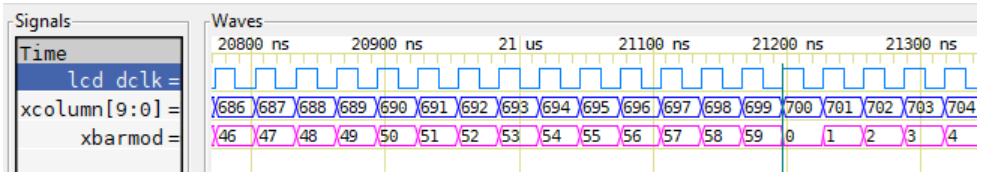
Simulace v GHDL ukáže následující průběhy v GTKView. (*Kvůli tisku se invertovaly barvy v černém poli signálů*). Hodnota xbarmod se začne počítat až při xcolumn=100, dříve ji ani nepotřebujeme:



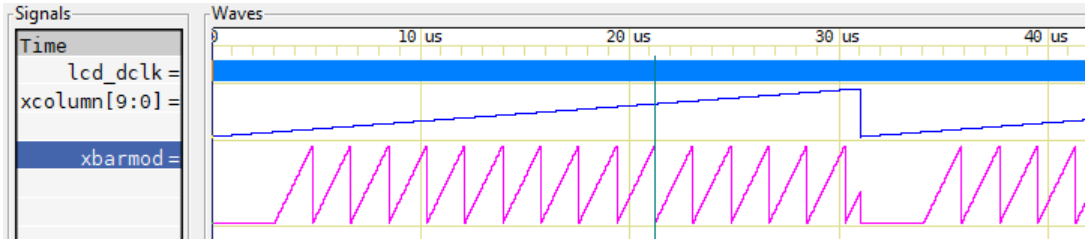

Roste až ke xcolumn = 159, poté začne od 0:




Přesně při xcolumn=699, kde na LCD končí náš ukazatel, je xbarmod = 59




V GTKWave si můžeme ještě navolit interpretaci xbarmod a xcolumn jako analogových signálů. Vybereme jeden signál a pravou myší zobrazíme jeho kontextové menu, kde volíme: Data Format →Analog→Step. Poté za každý signál vložíme Insert Analog Height Extension. Nyní dobře vidíme průběhy v LCD řádku. Svislá značka je zde na stejné pozici jako v předchozím obrázku, a to za xcolumn=699.




**Obrázek 17 - GTKView interpretující xcolumn a xbarmod hodnoty analogových signálů**

## Vložení obrázku z FPGA ROM paměti

Některé náročnější části se vyplatí konvertovat na paměť a číst je při vykreslování obrázku. Uvnitř FPGA máme na výběr dvě možnosti jejich uložení:

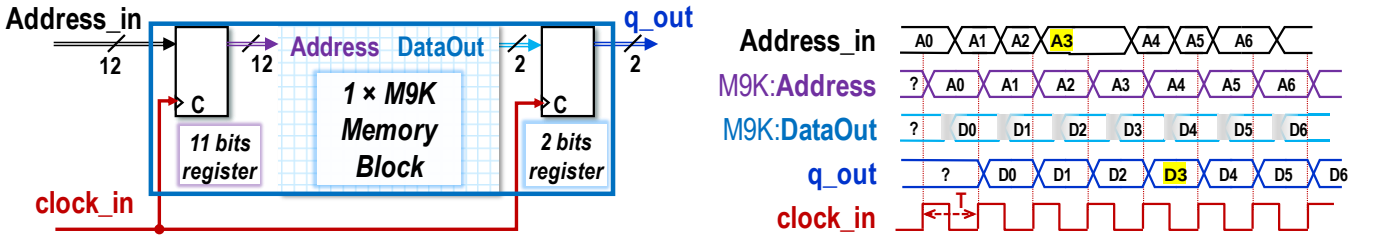
- **Logické elementy** (LE) umožňují nejrychlejší přístup k datům. Mají však mnohem flexibilnější využití než pouhé zapamatování hodnot, takže se šetří náročnější operace, které nelze realizovat pamětí.
- **Paměťové bloky** se v FPGA prioritně využívají pro velké objemy dat. I vývojové prostředí Quartus někdy konvertuje logické části na čtení z pamětí. Ty u dat dosahují vyšší hustoty informace, neboť se na jejich výrobu spotřebuje méně křemíku. Mohou mít i více portové přístupy dovolující nezávisle manipulovat s daty na různých adresách. Každý paměťový blok se ale použije celý, i když v něm obsadíme jenom jediný bit. Vše je tedy o vhodném návrhu obsahu paměti.

Třída obvodů Cyclone IV obsahuje paměťové bloky M9K konfigurovatelné na různé šířky výstupních dat. Možné varianty jednoho M9K bloku, uvedené jako počet slovo × bity v jednom slově, jsou:

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

Například 1024 x 8 značí konfiguraci paměti, kdy se z ní 10-bitovou adresou ($2^{10}$=1024) vybírají 8-bitová slova. Má tedy 1024 slov o šířce 8-bitů, tedy celkem 8192 bitů. M9K paměť lze nastavit i na 9-bitový výstup (tedy s možnou paritou), kdy využijeme všech 9216 bitů. Blíže manuál: Cyclone4_memoryM9Kblocks.pdf

Čtení z paměti je vždy synchronní – tak to vyžaduje princip její výběrové matice. Na jednu hranu hodin zapíšeme adresu a po časovém zpoždění se na výstupu paměti objeví data. Ta se zpravidla uloží do registru, viz obrázek dole, což přidá zpožděním celé periody hodin. Data sice dostaneme až s následující náběžnou hranou hodin, ale během jejich period mají vždy konstantní hodnoty, což je implementačně vhodnější.

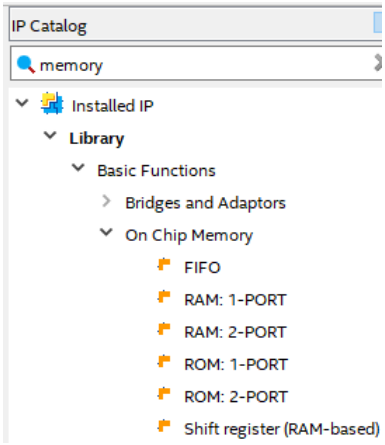



Obrázek 18 - Cyclone IV paměťový blok M9k v konfiguraci 4096x2

Větší paměti se sestavují z více M9K bloků a vyplatí se hlídat si jejich spotřebu velikost, protože i nepatrné zvýšení objemu data může přidat mnoho M9K bloků, neboť se použijí vždy celé.

Paměťové bloky musíme ale nějak inicializovat. V prostředí Quartus existují dvě možnosti:

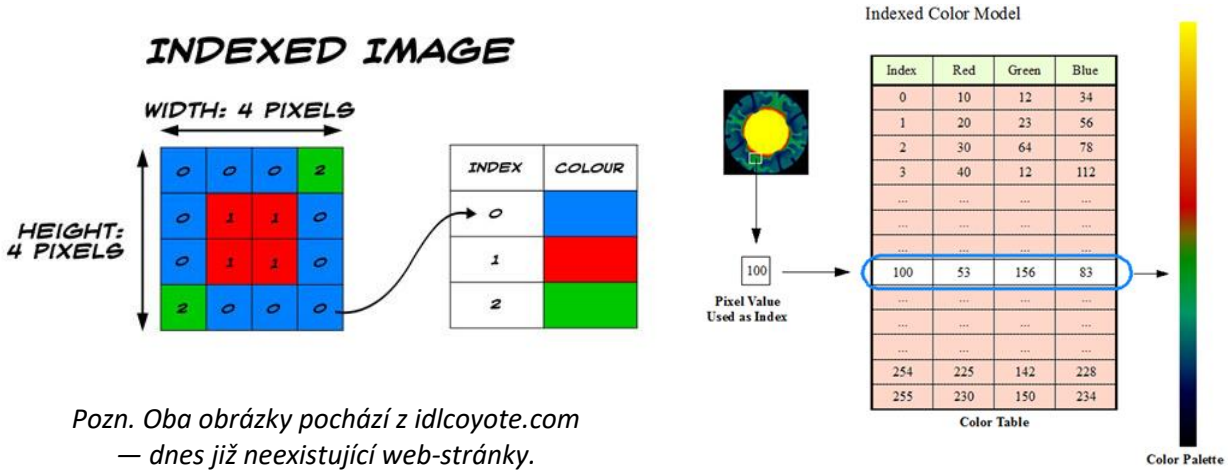
1) Vybereme si typ paměti z IP katalogu výrobce, viz obrázek vlevo. V Quartusu se tím spustí nástroj MegaWizard Plug-In Manager, v němž zadáme potřebné parametry paměti a její inicializační soubor typu *.MIF, Memory Initialization File. Postup je pracnější, ale lze si volit z více variant. Zkomplikuje se však simulace v GHDL, musí se komplikovaně vložit knihovny Quartusu.
2) Pokud nám stačí paměť ROM: 1-Port, lze vygenerovat VHDL soubor, který Quartus během překladu konvertuje na paměť. Jde o snazší postup, který dovoluje i GHDL simulaci, a tak si ho ukážeme. Předchozí najdete v již zmíněném manuálu M9K pamětí




Bitmap2VHDL z FPGA-LCD Utils umí z obrázku vytvořit jak inicializační *.MIF, tak i *.vhd soubor Quartus konvertovatelný na ROM: 1-Port.

## Konverze bitmapy

Je-li barev málo, snižuje se objem dat tím, že se jim přiřadí indexy a uloží se jen ty. Směřují do tabulky barev, což dovoluje i snadné změny. Podrobněji viz https://en.wikipedia.org/wiki/Indexed_color .




*Pozn. Oba obrázky pochází z idlcoyote.com — dnes již neexistující web-stránky.*

Obrázek 19 - indexované barvy

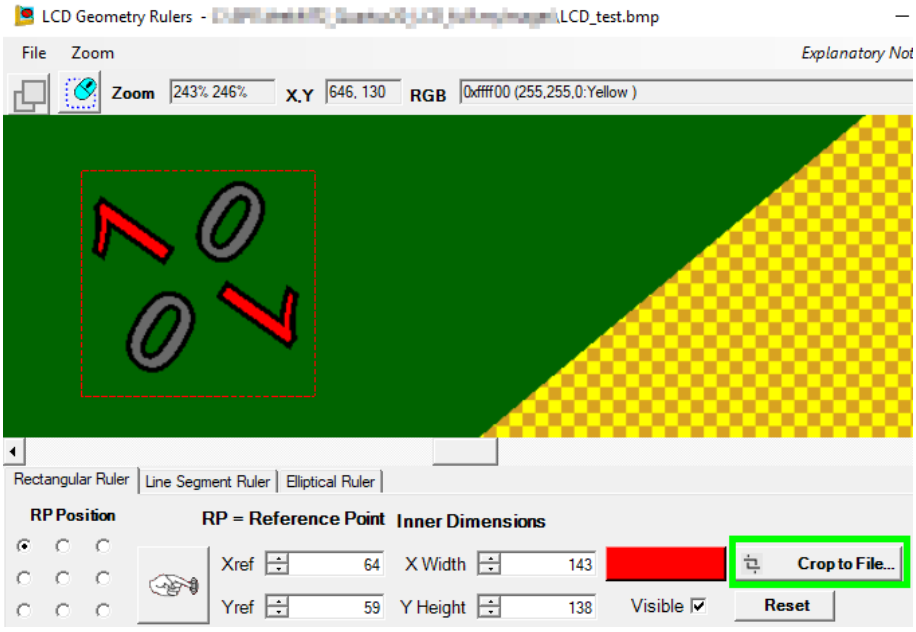
K indexování se nejlépe hodí obrázky uložené bez rasterizace, "spatial anti-aliasing", viz dole, která zvyšuje počet barev. Pokud vybraný obrázek má rasterizaci, hodí se v něm redukovat počet barev, což umí mnohý grafický nástroj, například freeeware FastStone Image Viewer.

Grafické nástroje dokážou rasterizací opticky vyhladit okraje přidáním přechodových odstínů barev. Protikladem je "dithering", cca rozptyl, kdy se vytvářejí polotóny k substituci chybějících barev, což též vyhladí obrázek. Hardwarově se relativně snadno implementuje konvoluční maticí Gaussian blur 3 × 3 nebo 5 × 5. Umí ji i LCD panel desky Veek-MT2.

VeekMT2_LCDregV2 vypíná jeho dithering, aby bylo přesně vidět, co se na něm vytvořilo. Lze ho zapnout nastavením jeho výstupu LCD_DITH na '0' .

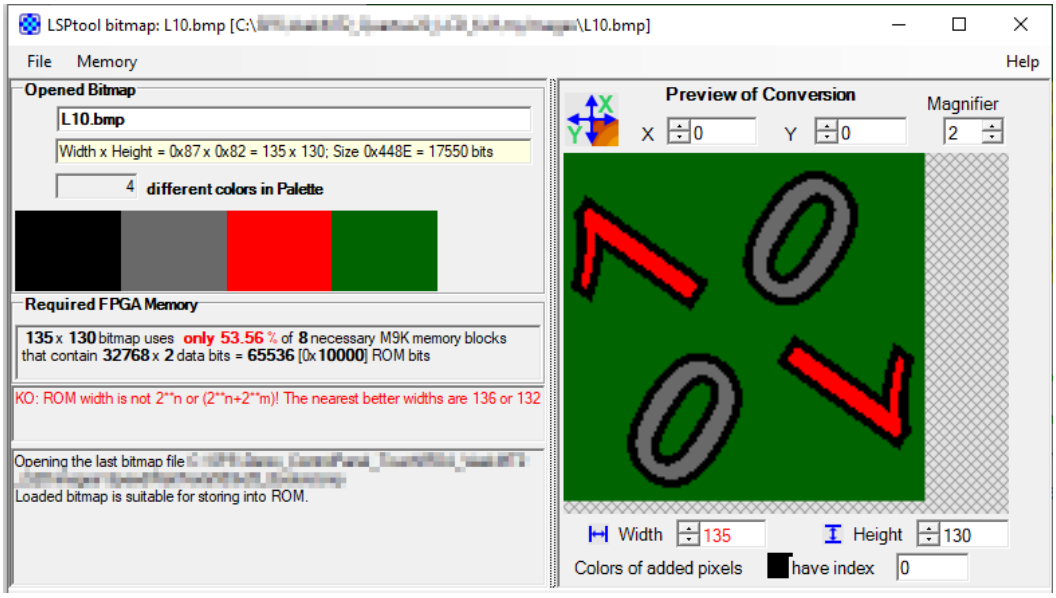

Obrázek 20 - Anti-aliasing

Vybranou část obrázku si vyřízneme nějakým grafickým nástrojem a uložíme ho jako bitmapu.




Obrázek 21 - Oříznutí do souboru z nástrojem LSP Geometry Rulers

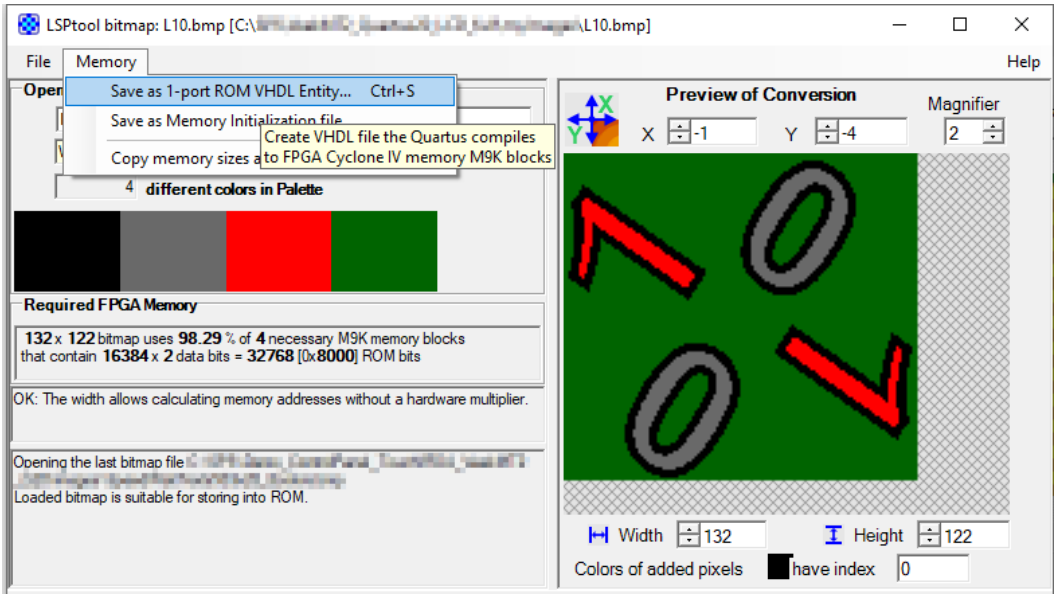
Poté si můžeme spustit konvertor BMP z FPGA-LCD Utils : a načteme bitmapu.



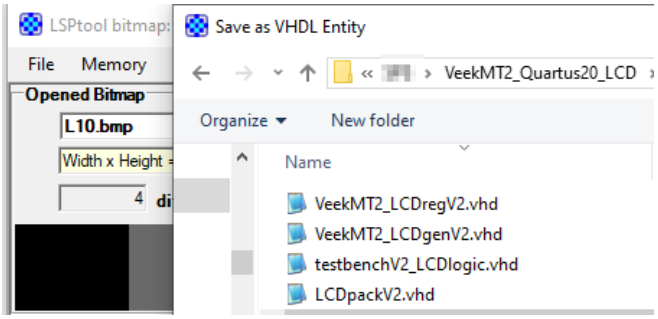

Obrázek 22 FPGA-LCD nástroj konverze obrázků

Obrázek používá 4 barvy, které zakódujeme do dvou datových bitů. Výsledná paměť by však spotřebovala 8 paměťových bloků M9K, které by využila tak z poloviny. Ořízneme obrázek, přičemž nezapomeneme, že **šířkou budeme násobit**, a tak zvolíme mocninu 2 či součet dvou mocnin dvou, obé vede na snazší hardwarovou implementaci, viz i část 6.3.2 Logické obvody na FPGA .

Upravíme rozměry číselnými ovládacími prvky nahoru a dolů. Necháme však krajní levý volný, protože budeme data číst z paměti se zpožděním 1 taktu.




Upravenou bitovou mapu uložíme přes menu "Memory->Save as 1-port ROM VHDL Entity", například pod názvem **L10rom.vhd** do hlavního adresáře projektu Quartus!



Vygenerovaný VHDL **L10rom.vhd** nijak neupravujeme, abychom neporušili jeho přesnou strukturu, kterou specifikuje dokumentace k vývojovému prostředí Quartusu jako vhodnou k implementaci pomocí bloků M9K. Můžeme ale číst jeho hlavičku s informací o velikosti paměti a barevné paletě.

```
--  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 an 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;
```

Moudře jsme nechali volný první sloupec konvertovaného obrázku, a tak zeleně zvýrazněná hodnota na začátku inicializace pole **constant** arr, odpovídá indexu barvy pozadí obrázku, které necháme průhledné.

Kód procesu skýtá otázku, proč jsme zaváděli novou proměnnou ix a nepoužili jsme složený příkaz:

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

Doporučený styl kódování, viz manuál Quartus Inferring ROM Functions from HDL Code, žádá multiplexor přepínající podle vstupu address typu std_logic_vector. Místo něho jsme zvolili úspornější kód s proměnnou ix, jelikož VHDL specifikace multiplexoru dovoluje zadat rozsahy, jimiž se zkrátí kód, jedině u integer. Kvůli změně typu jsme překladači předali přesnou informaci o velikosti indexu, aby ho správně interpretoval jako std_logic_vector hned v úvodním kroku překladu.

*Pozn. V popisu pole, to se překládá multiplexorem, je možné v Quartus překladači zadávat rozsahy i u typů odvozených od std_logic_vector, což norma VHDL nezná. Použitím nestandartní kontrukce bychom vytvořili nepřenositelný kód závislý na překladači, "compiler-dependent code", kvůli tomu se raději využilo integer.*

## Jak se čte obrázek z paměti?

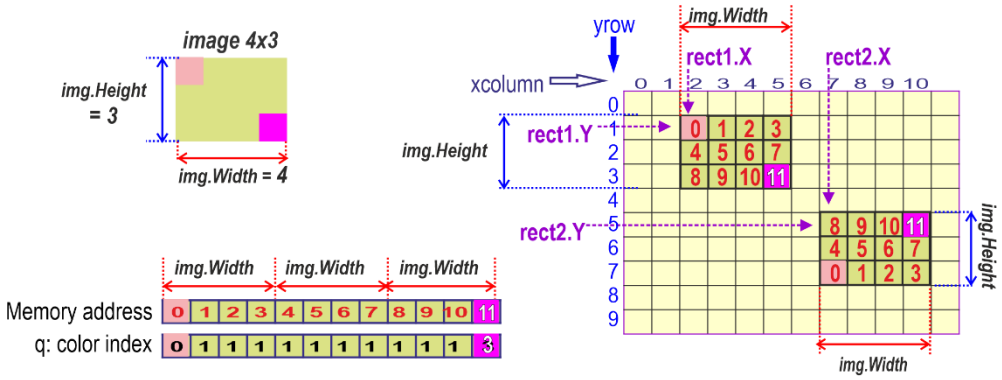
Obrázek dole ukazuje situaci, kdy se konvertovala bitmapa 4x3 pixely. Uložila se do paměti jako jednorozměrný vektor po řádkách, tedy právě tak, jak se tvoří obraz i na LCD. Jde o metodu uložení vícerozměrného pole zvanou "row-major order", kterou používá i jazyk C.

V paměti se nacházejí pouhé dvoubitové indexy barev, tedy hodnoty 0 až 3. Pošleme paměti adresu, kterou načteme data. Vztah pro adresu mapuje uložený obrázek na výsledné zobrazení na LCD. Vpravo máme dvě varianty z mnoha možných. Horní obrázek se umístil s levým horním rohem na LCD yrow=1, xcolumn=2. Paměť se sekvenčně přečetla. Zavedeme-li, že x=xcolumn a y=yrow, pak napřed x a y převedeme na relativní souřadnice vůči rohu obrázku, z nichž vypočteme adresu v ROM paměti

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

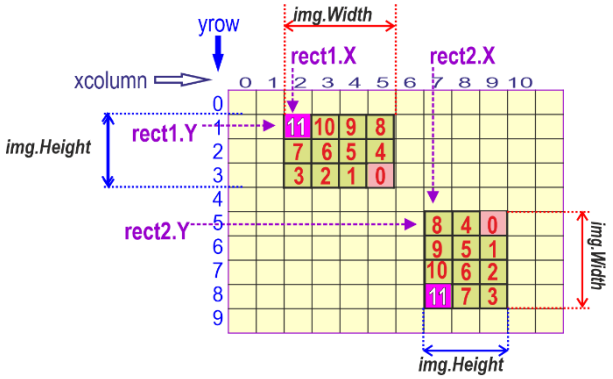
Šířkou obrázku násobíme, zde jsme s výhodou zvolili součet dvou mocnin dvou, což se v obvodu snáze implementuje. Druhý obrázek je převrácený, jeho relativní y osa běží opačně:

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




Obrázek 23 - Zobrazení 1: normální 2:převrácené

Možností polohování je hodně, všechny znamenají jen změny přepočtu adresy, což platí i pro rotace v modulu 90 stupňů. U 180 stupňů se čte z obou os pozpátku. U 90 stupňů se otočí jen jedna, ale vzájemně se prohodí. Musíme upravit i test na obdélník, v němž se kreslí.

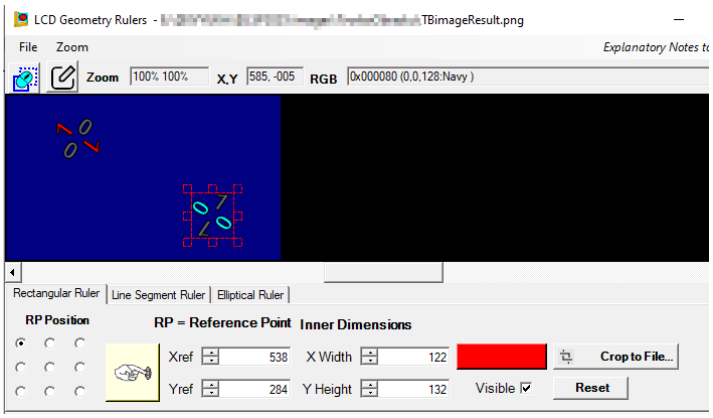



Obrázek 24 - Rotace 1: o 180 stupňů 2: o 90 stupňů

Rotace o 90 stupňů bude v kódu následující kapitoly.

## VHDL kód s vloženými obrázky z paměti

Pokud máme grafickou předlohu, stanovíme z ní souřadnice obrázku. Lze pochopitelně jen odhadnout jejich pozice a korigovat je dle testbench výsledku třeba pomocí LCD Geometry Rulers, až se nám pozice líbí:




Při psaní kódu dbáme na výstižné pojmenování proměnných. S výhodou využijeme VHDL typy record, přímé analogie C struktur. Entita se nezměnila, a tak začneme architekturou:

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

Konstanta typu sizes_t obsahuje šířku a výšku, aby obě hodnoty byly pohromadě.

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

Definujeme si velikosti obdélníků rect_t pro obě polohy. V konstantě **L10r2** jsou prohozené velikosti paměti, protože se vztahuje k poloze obrázku, který bude otočený o 90 stupňů.
Typ rect_t je v balíčku LCDpackV2 verze V2.1 spolu s funkcí InRect, která otestuje, zda okamžité souřadnice leží v obdélníku.

```
 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");
```

První základní paletu jsme vytvořili podle údajů ve VHDL souboru převedené bitmapy. Ve druhé paletě jsme přebarvili některé položky.

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

Budeme paměti posílat adresu a brát si z ní data. Velikost obou signálů musíme přesně vytvořit podle vstupů a výstupů paměťového souboru L10rom.vhd.

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

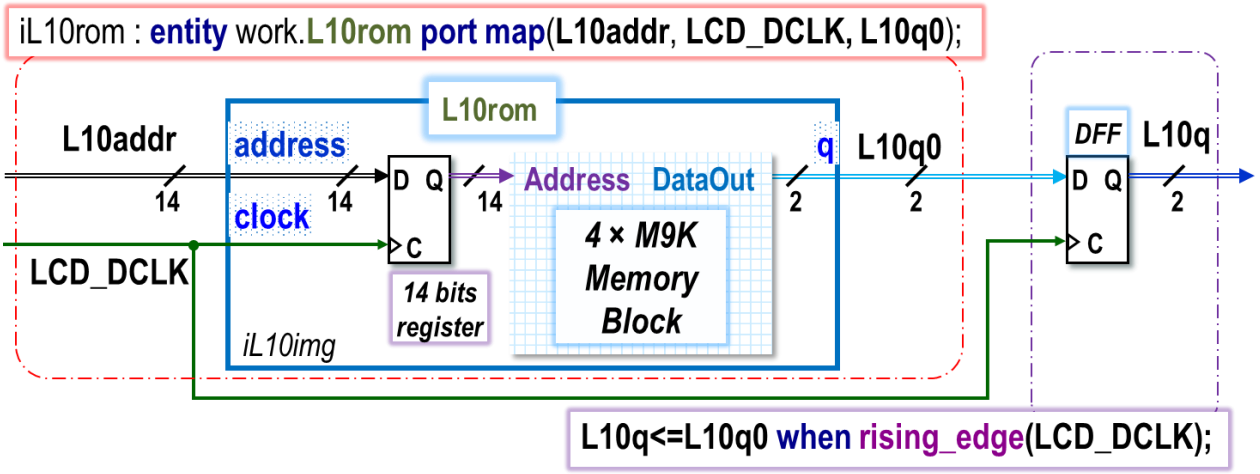
Budeme adresu počítat čísly integer, ale paměť ji má jako vstup typu std_logic_vector , tak jsme si definovali konverzní funkci, aby hlavní kód vyšel jednodušeji.

```
begin -- architecture
```

V kódu architekturu vytvoříme instanci entity paměti a za ní dáme registr výstupní hodnoty. Vložíme i DFF obvod, na jehož D-vstupu bude L10q0 signál a na výstupu Q signál L10q,

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

Obvod vytvořený dvojící příkazů nahoře ukazuje obrázek dole:




```
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
```

Do sensitivity list procesu jsme přidali L10q, tedy hodnotu načtenou z paměti, na níž též závisí výstup procesu RGBcolor s barvou pixelu. Poté jsme vložili definice nám již známých x a y proměnných.

```
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
```

První proměnná L10idRect list bude identifikátorem, že souřadnice x,y pixelu se nachází v některém obdélníku obrázku, přičemž 0 znamená mimo ně. Druhá proměnná je index barvy převedený na 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;
```

Stanovili jsme identifikátor obdélníku L10idRect postupnými testy polohy uvnitř některého z nich.

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

Hodnotu načtenou z paměti jsme převedli na integer, jímž budeme indexovat palety.

```
---------- our image ------------------------

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

Pokud se [x, y] pixel nachází v nějakém obrázkovém obdélníku (L10idRect>0 ) a současně byla barva načtená z paměti obrázku odlišná od 3, tedy od barevného indexu pozadí, které chceme průhledné (L10ixColor/=3 ), přepíšeme hodnotu RGB z odpovídající palety.

```
  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;
```

Adresu paměti v prvním obdélníku stanovíme přímým přepočtem indexu dvourozměrného pole na vektor. Druhý obdélník je otočený o 90 stupňů, má tedy prohozené své relativní osy x a y, přičemž osa-x se tady čte pozpátku po řádcích od L10img.Height-1 k 0, zatímco osa y běží ve směru řádků v obrázku.

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

**Rada od -a parametru z GHDL společenství**: Soubor paměti musíte samozřejmě také přidat do seznamu souborů v dávce runLCD.bat, a to před LCDlogic*, jinak ho nepřeložím.

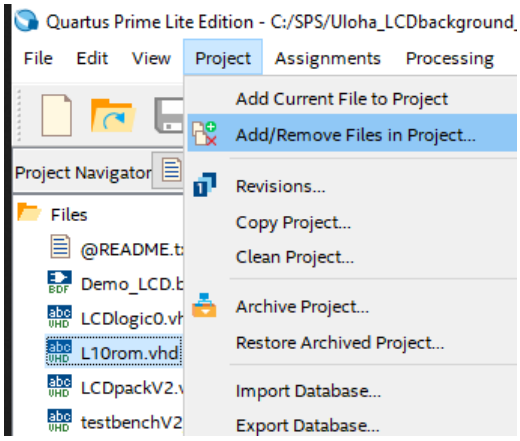
```
set FILES=../LCDpackV2.vhd ../L10rom.vhd ../LCDlogic0.vhd
```

Nemusí ale v seznamu zůstat trvale. Mým para-kolegům **-e** a **-r** stačí ke zdárnému exe-běhu jen soubory *.o (object files) mnou přeložené. Evokujte můj plný překlad pouze jednou, poté modifikujte runLCD.bat, či si vytvořte nový *.bat s upravenou řádkou FILES, v níž zůstane jen `../LCDlogic0.vhd`

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

Výsledek uvidíte rychleji. U předchozího příkladu se ušetřily celé 4 sekundy mojí vzácné práce!
Plný překlad s mým zapojením bude opět nutný až po změně L10rom.vhd a LCDpackV2.vhd.

**Dodatek od Quartus Lite:** Podobné triky na mě neplatí — nejsem žádný volnomyšlenkářský GHDL kumpán! Trvám na tom, že paměť bude vždy v seznamu souborů na mé záložce Files. Pokud v něm není, tak ji tam rychle přidejte. Beru buď pravou myš na Files, či přes moje menu:




Obrázek 25 - Přidání paměti do seznamu souborů

Překlad a simulace kódu VHDL trvá v GHDL přibližně 7 sekund a provádíme pouze jeden krok – spustíme dávkový soubor z terminálu Visual Studio Code.

Quartus také nabízí instalaci simulátoru Intel Questa, ale získání jeho bezplatné licence je stejně složité jako jeho používání. Questa nabízí menší výhody, protože někdy detekuje více chyb v časování, avšak její bezplatná verze jich zachytí jen o málo víc. Práce s GHDL je mnohem rychlejší a jednodušší. ☺

**Poznámky:**

1/ Odladěné VHDL můžeme v Quartusu přeložit a nahrát do desky.

2/ Je však možné, že kód VHDL nebude fungovat, i když vidíte jeho bezchybnou simulaci, jelikož jakákoli simulace je pouze simulací. Hardware rozhoduje o tom, zda jste dodrželi všechny časové požadavky.

3/ Pokud uvidíte zprávy kompilátoru Quartus, které varují před neúplnými definicemi časování či hodin:

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...*

*nebo*

Critical Warning (332049): Ignored create_generated_clock at VeekMT2_LCD.sdc(50): Argument <targets> is an empty collection

pak nejspíš máte špatné jméno instance v top-level **entitě** v schématu BDF, viz obrázek Obrázek 2, na str. 4

- VeekMT2_LCDgenV2 musí mít instanci **iLCDgenerator**

Pouze pro ni existují definice pro TimeQuest Analyzator v VeekMT2_LCD.sdc. Změníte-li název instance, musíte znovu vygenerovat soubor *.sdc... Ovej, oprava jména instance je mnohem jednodušší a rychlejší:-)

---

KONEC ZADÁNÍ

---

## Závěrečná debata pro LSP

*Protest Maxe Šťourala: Konec v tom nejnapínavějším okamžiku jako v mizerném hororu?! – to snad nemyslíte šustivě vážně. Vždyť nás v tom **V**elmi **H**odně **DL**ouho vykoupete!*

*Odpověď: Kdež dlouho! Simulace pomocí v GHDL je expresní, můžete volně experimentovat a využít vzorník k zábavné tvorbě.*

*Zděšení Maxe Šťourala: Nikde na webu nevidím VHDL kódy jednotlivých pozadí ze vzorníku, která by se dala fofrem ‚vokopčit'!*

*Odpověď: Nejsou tam a nebudou, a to pro úsporu Vašeho času. Pokud nepočítáme komentáře, celá architektura vkládání obrázku nemá ani 2 tisíce znaků (a to včetně mezer). Za pár minut ji vytvoříte, jelikož hodně klíčových slov za Vás dopíše automatické doplňování editoru. A lépe pochopíte její princip.*

*Vytvořte si vlastní kód s hodnotami Vašeho řešení! Vždyť budete mít jiné obrázky s odlišnými velikostmi a možná i rozsahy adres a dat. Kopčením originálu by se také ‚vokopčila' původní čísla a poté by se dlouze bádalo, kde se stala chyba. Též ověřeno :-(.*

---

Vše bylo popsáno...

~ Konec ~

---

~ o ~