LSP Task 2 Freecools Flag

Logic Systems And Processors (České Vysoké Učení Technické v Praze)

2nd Task for Computer Systems and Processors

Version 1.0 - October 25, 2018

Content

Project assignment 2: Promotional flag - upto 25 points

Reason for implementation:

Manager Freecoolin, see https://www.urbandictionary.com/define.php?term=freecoolin, has performed a long series of most in-depth insights into pie charts and excellent tables. After many nonsleeping nights, he had got an idea of adding panels to tourist info boards. On 640x480 pixel display, a flag is shown at the beginning in 320x240 pixel size and elegantly composed of 4 sub-parts to let tourist know what they needed to buy at the local info center. Storing its picture into memory would require at least 153 kilobits, and the cheap FPGA would not contain enough space for the information about local sights. Partial creation of the flag by logic reduces memory consumption.

Assignment: Design a flag displayed on 640x480 @ 60Hz industry standard VGA, which is supported by most LCD monitors, see https://en.wikipedia.org/wiki/Video_Graphics_Array and the 3rd lecture. Select your flag at https://dcenet.felk.cvut.com/fpga/.

Split the image into 4 parts according to its color areas. If you do not have four color areas in your flag, divide one of them equally. Each part of your flag must move independently across the screen.

Student Maxo Groucho's comment: I understand that the super retina HD OLED, found on my incredible iPhone, is not everywhere, but even the most shunt brands use the LCD. Who utilizes a VGA monitor today? A museum may be!

Answer: iPhone images are feed in the VGA style, much like as most LCDs. It would be ineffective to address the pixel per pixel and push them into the LCD controller mouth on the spoons. Thus, single streams of pixel data, organized in rows, transfer the whole picture analogously to the VGA. On LCD, only synchronization pulses are shortened or wholly omitted depending on a type used. Therefore, the solution debugged on VGA suits to LCDs as well.

Project scoring

You obtain points for a properly functioning project based according to a technical level of your solution.

1. For a static solution, you receive 10 points.
2. After dividing it into 4 parts, you obtain an additional 15 points. However, the points can be reduced due to its nonsensical decomposition into parts, for example in the style of squares cut off regardless of the color theme, and so on. You should consult the teacher before designing the splitting.
   WARNING - the main file you have to name according to your flag! You can use some parts of the sample code in DisplayLogicCurve4parts.vhdl or others, but you must rename your entity according to the selected flag, then also filename and instance in the testbench.
3. In 3rd task, you can add a position control and receive additional points, up to 55.

The teacher can reduce the points if you have wrong code style, for example, source code without comments. You should add them at least to significant operations. You should also define constants and use proper variable names.

The joy of the student Maxo Groucho: Whoopee, a simple solution for 25 points is enough, and I have a credit in my pocket!

Answer: Yes, not everyone tends to hardware. With a 10-point flag and its controlling by counters, you obtain 21 points + 25(beacon) + 10 (attendance) = 56 points, i.e., you receive 6 points transferred into your exam. Even with an excellently written test, you can hardly achieve E; you probably repeat the exam several times. If you choose a 15-point flag and improve moving parts, you have 40 points for the exam and a higher chance to obtain A or another mark. When you select a 25-point flag with good part control, you can fill the exam written test to a minimum, and you have A.

For two tasks, you get 14 (attendance) +15 (beacon) +25 (flag) = 54 points. Even with a very final exam, approaching a maximum of 60 points, you obtain D, but you will probably come next time.

If you make the third task, perhaps only for 25 points, you achieve 79 points, and a chance for B. With the maximum solution of all three tasks, you get 109 points and you need to write final exam 1 point above the minimum, and you have A certain.
The project is part of your exams. It depends on you where you prove your knowledge, either during practical exercise or in your exam.

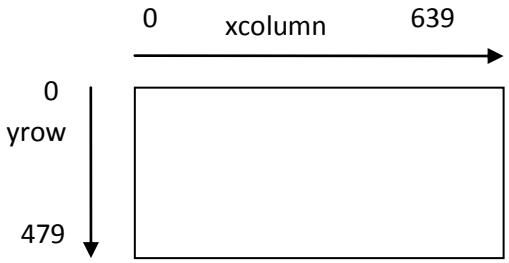
Maxo Groucho's muttering: Fine, I expect hard work before I hide under A-frame tent. Moreover, I learn just picture drawing. When do we use this? Probably never.

Answer: I agree twice. The task is challenging, though, its difficulty lies only in understanding how circuits are programmed. Moreover, such drawings are sometimes done also on the FPGA, but seldom.
You learn how to solving circuit tasks. The methods are always similar. You find out a way how to process data and then convert the algorithm into hardware level implementation by logic or by memories :-)

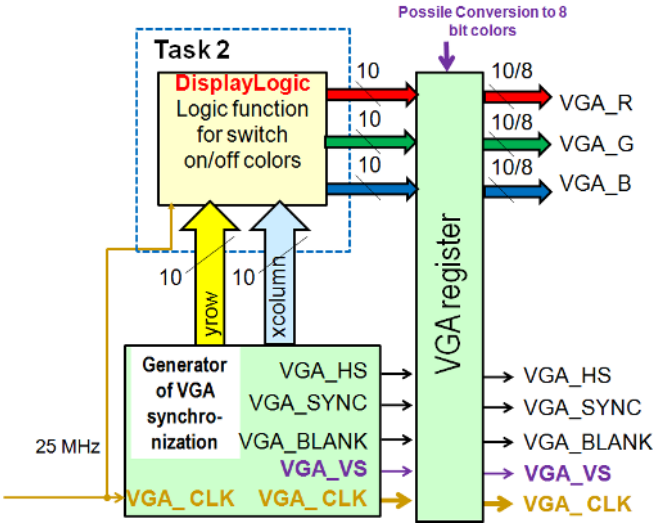
Circuits in design

DisplayLogic - designed by you

DisplayLogic is a combinational circuit, as in previous Morse project. Its inputs receive coordinates xcolumn and yrow from VGA sync generator. Orientation x and y axes here corresponds to the classical screen image. The output of DisplayLogic specifies a color value for a given pixel.



This time, you do not design logical equations from Karnaugh maps, they would be complicated, but you let Quartus development environment to design them from VHDL code.



Inputs of DisplayLogic

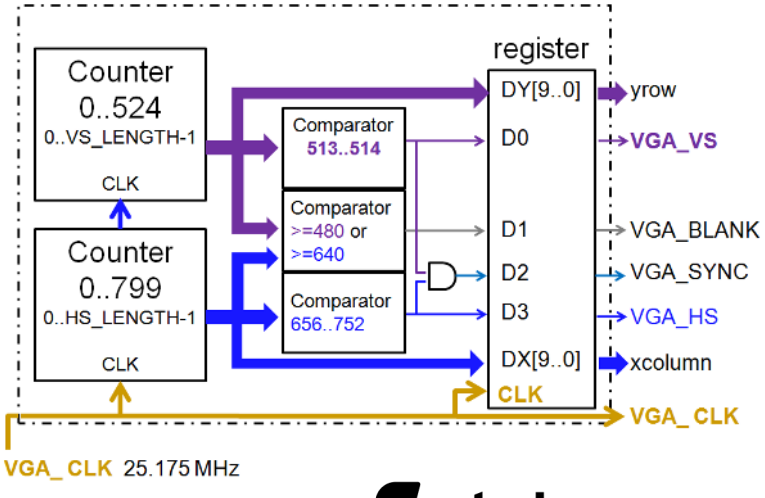
- **Size** - 2-bit signal indicative of the desired maximum size from 00 to 11 minimum.
- **xcolumn, yrow** : 10-bit signals indicate coordinates of the displayed pixels.

Outputs of DisplayLogic

- **VGA_R, VGA_G, VGA_B**: 10-bit color values. 10bit color is defined by a triad of 0x3FF, 0x3FF, 0x3FF for white and by 0,0,0 for black. It is a direct video signal that does not contain alpha channel with transparency information (opacity) - colors are always opaque.

VGASynchro - VGA Synchronization Generator

Generator for VGA 640x480@60Hz contains two counters connected in series and comparators. The column counter runs from 0 to 799, but columns from 0 to 639 only lay inside visible part of the image. The row counter counts from 0 to 524, but lines from 0 to 479 are only visible.



The circuit was discussed in the lectures, and you do not have to create it, just use it. It is a hardware analogy of two for-loops of C language:

```
unsigned short int xcolumn, yrow;  RGB color;
for (yrow = 0; yrow < 525; yrow++)
 {  for (xcolumn = 0; xcolumn <800; xcolumn++)
     color = DisplayLogic(xcolumn, yrow);
 }
```

Inputs of VGASynchro:

- CLK_25MHz175 – pixel frequency 25.175 MHz, but today's LCD monitors have reasonably good synchronization properties so that you can use 25 MHz as input. On DE2 board, you can create 25 MHz from CLOCK_50 by dividing 2. We explain a better way in later lectures.

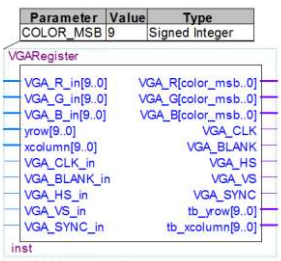
Outputs of VGASynchro

- VGA_CLK – a mere copy of the input frequency is used as output just not to forget their connection to DE2 pin - the monitor shows only a black screen without VGA_CLK connected;
- VGA_BLANK - is in '0' only during the horizontal sync pulse, i.e., only for a period of 3.81 microseconds.
- VGA_HS - it occurs every 31.77 microseconds, i.e., with a frequency of 31.47 kHz (time values assume that input CLK_25MHz175 a frequency 25.175 MHz);
- VGA_VS - is in '0' only during the vertical sync pulse, i.e., for a period of 63.6 µs. It occurs each 16.68 ms, i.e., with frequency 60 Hz;
- VGA_SYNC: is in '0' when horizontal or vertical synchronization pulses are active, VGA driver requires it. Thus, it is composed by their and-operation;
- yrow: **unsigned(9 downto 0)**; - 10-bit signal indicating a row number, i.e. a y coordinate, ranging from 0 to 524. Rowe numbering runs from top to bottom. The row at the top is numbered 0, the last visible bottom row has number 479. Rows 480 and higher are already in the vertical darkening pulse.
- xcolumn :**unsigned(9 downto 0)**; - 10-bit signal ranging from 0 to 799, which indicates the column number, i.e. the x-coordinate. The value 0 is left and the maximum value on the right. The visible image is from 0 to 639. The 640 and higher are already in the horizontal blackout pulse. Note: Although all LCD monitors support the VGA 640x480 @ 60Hz standard, it is not their native resolution. It may happen that an LCD monitor does not display last one or two columns.

VGA Register

The VGARegister samples the outputs of VGASynchro and DisplayLogic at rising edge of VGA_VS, the beginning of the image. Their values remain stable during display which prevents blur caused by various latencies in DisplayLogic.

The registry can also be used to reduce the 10-bit DE2 color 8-bit VGA card used by DE2-115 boards, with which some can work in later exercises.



VHDL requirements

If your project does not fulfill them, it is considered as inappropriate.

1. Amount of used resources are customary conditions in many projects. The following limits apply only to DisplayLogic related circuits; they do not including VGARegister, VGASynchro, and NIOS:

- You may use a maximum of **35 DSP** blocks that are 9-bit hardware multipliers on Cyclon II.
- Up to **8 M4K** memory blocks can be used, giving a total of 32 kilobits under optimal usage.
- You can maximally consume **2000 logical elements** (LE). Each is equivalent to 16 memory bits.

2. The circuit must not contain **LATCHes and combinational loops** - their presence always indicates design errors.

3. For arithmetic operations, you can use only the **ieee.numeric_std** library . Older, not recommended libraries ieee.std_logic_arith, ieee.std_logic_unsigned; ieee.std_logic_signed are forbidden because they are not portable. Bevare, many web demos utilize them.

4. The project may not use **shared variables**.

5. The integer numbers are only allowed for internal variables, in accord with the recommendations of designs.

6. Integer numbers must have **range** definitions. If they are missing, their suboptimal conversion can occur.

   Here's the advice:

   a) Leave all integer numbers temporarily unlimited during the debugging phase. Add ranges at the end.

   b) It is better to avoid calculation with unsigned types, thus, convert xcolumn and yrow to integer numbers. Unsigned numbers can give errors when comparing and subtracting.

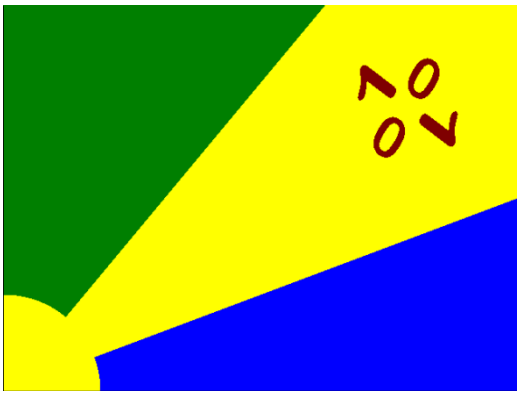
7. In a project, you can **not divide** the variables by numbers other than 2 or perform any modulo operation (the remainder after division), because both operations are very challenging in hardware. However, you can multiply, but only by whole numbers, because the circuit has 9-bit hardware multipliers. Division and modulo can, of course, be used with constants, as this is done already during translation.

- How to write a condition, for example, without dividing: **x / 3 <y / 5**? Simply: 5 * x <3 * y.
- **Divisions by powers of 2**, i.e .numbers 2, 4, 8, 16, 32..., can be inserted into the code because bit shifts easily implement them.
- Moreover, how to perform a remainder of the division by powers 2? Again easily, mask lower bits using the logical AND.

8. The VHDL code must not be programmed in an unclear style all in one file. You can store sub-elements into separate VHDLs and then you can insert them as components and create their instances (PORT MAP).

Solved example

Create a flag of LSP flag - ‘0’ and ‘1’ wondering in the yellow No man’s land and looking for a way to a safe area of green logic truth and blue untruths.



Step 1 - Static Flag

First, we measure flag geometry, taking into account that the y-axis runs from top to bottom, and therefore the slopes of straight lines have minus signs.

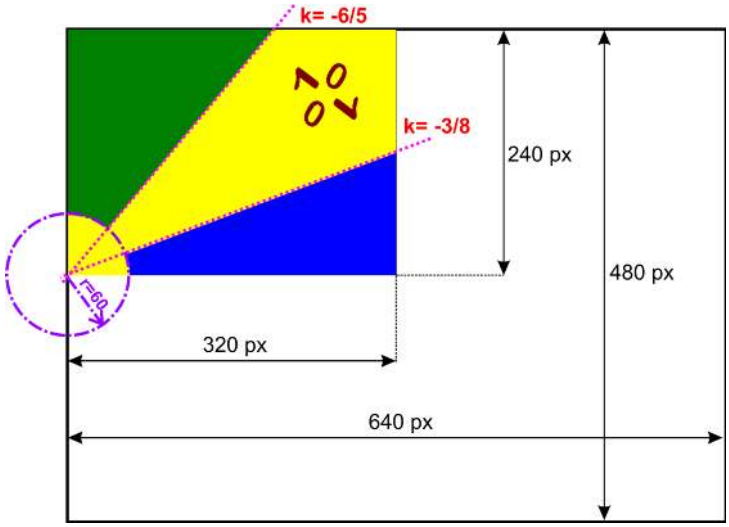


Figure 1 - Geometry of LSP flag

We find that the following equations roughly approximate yellow part as the circle:

$$x*x+(y-240)*(y-240)= 60*60 = 240*240/16$$

moreover, two straight lines:

$$y=240-6/5*x \quad a \quad y=240-3/8*x$$

We cannot divide by 5. We could divide by 8, the power of 2, but we would lose the accuracy. We rather rewrite the equation of the lines to:

$$\mathbf{5*y=5*240-6*x} \quad a \quad \mathbf{8*y=8*240-3*x}$$

We build the flag from the cascade of multiplexers using conditions of form IF-THEN ELSEIF...

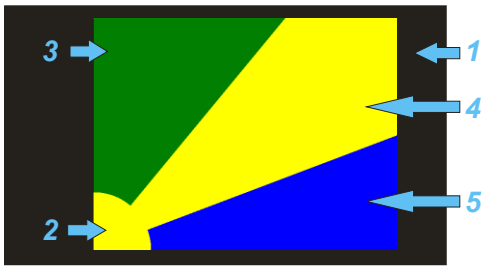


Figure 2 - Podmínky

Bleached numbers in the image added for orientation refer to the following conditions:

1/ The first condition with the highest priority cut out all outside of the flag. We declare constants:

```
constant YSIZE : integer := 240; constant XSIZE : integer := 320;
```

We introduce auxiliary variables:

```
variable x, y : integer;
```

and assign them inputs xcolumn and yrow converted to integers:

```
x:=to_integer(xcolumn); y:=to_integer(yrow);
```

Then, we easily write the first condition:

```
if(x<0) or (x>=XSIZE) or (y<0) or (y>=YSIZE) then  RGB:=BLACK;
```

2/ The second condition circumscribe the yellow circle by its equation. We place it into ELSIF statement to assign it a lower priority and keep the border clipping given by the first condition

```
elsif x*x+(y-YSIZE)*(y-YSIZE) < YSIZE*YSIZE/16  then  RGB:=YELLOW;
```

3/ The third condition limits the green part using the straight line equation. We add it again with ELSIF statement, so border and circle clipping specified by conditions 1 and 2 remain.

```
elsif 5*y < 5*YSIZE - 6*x then  RGB:=GREEN; -- line equation  y = 240-(6/5)*x
```

4/ The fourth condition, bound again by ELSIF, cuts off the rest of yellow part by straight line equation

```
elsif 8*y < 8*YSIZE- 3*x then   RGB:=YELLOW; -- line equation  y = 240-(3/8)*x
```

5/ The fifth condition assign blue color to the remaining picture. We write it by ELSE statement, i.e., all else set to blue, and terminate the IF statement:

```
else    RGB:=BLUE;
end if;
```

You find whole VHDL code in file DisplayLogic.vhd. The block of conditions looks like the following:

```
begin  -- architecture
   LSPflag : process(xcolumn, yrow) -- output of process depends on xcolumn and yrow
   variable RGB : RGB_type; -- output colors
   variable x, y : integer; -- renamed xcolumn and yrow
   begin
            x:=to_integer(xcolumn); y:=to_integer(yrow); -- convert to integer
            if(x<0) or (x>=XSIZE) or (y<0) or (y>=YSIZE) then  RGB:=BLACK; --black outside of visible frame
            elsif x*x+(y-YSIZE)*(y-YSIZE) < YSIZE*YSIZE/16 then  RGB:=YELLOW;
            elsif 5*y < 5*YSIZE - 6*x then  RGB:=GREEN; -- line equation  y = 240-(6/5)*x
            elsif 8*y < 8*YSIZE- 3*x then   RGB:=YELLOW; -- line equation  y = 240-(3/8)*x
            else    RGB:=BLUE;
            end if;
      -- Copy results in RGB to outputs of the entity
            VGA_R<=RGB.R; VGA_G<=RGB.G; VGA_B<=RGB.B;
-------------------------------------------------------------------------
end process;
```

After we create DisplayLogic symbol from DisplayLogic Quartus menu (*File->Create/Update -> Create Symbol Files from Current FileSymbol*), we can test the result in the following circuit:

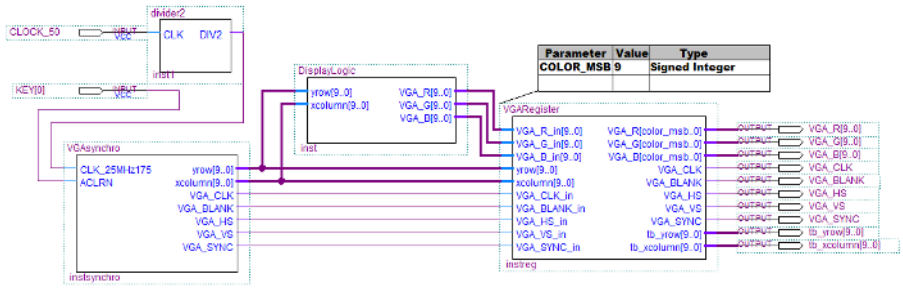


Figure 3 - Test circuit

The code can also be tested without a board using the testbench in ModelSim. For this purpose, VGARegister includes the outputs tb_yrow and tb_xcolumn outputs. ModelSim is the topic of LSP lectures and an exercise.

The result of our VHDL program is a circuit, no instructions, so we check in the RTL Viewer what we have created. We open it In the Project Navigator context menu on the Hierarchy tab.

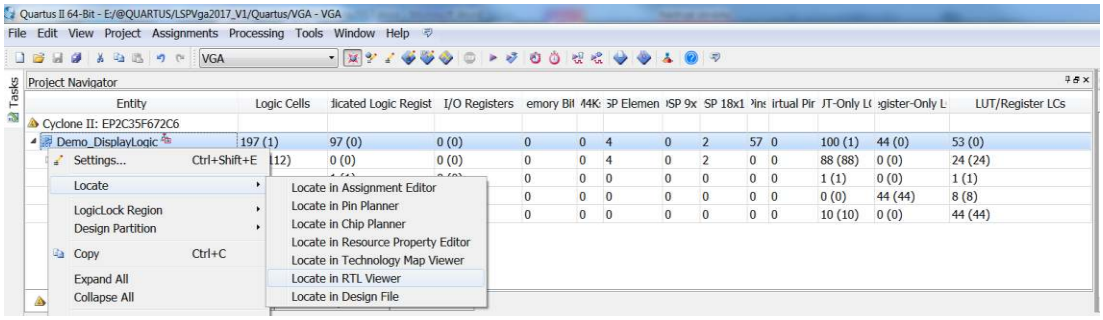


Figure 4 - RTL Viewer

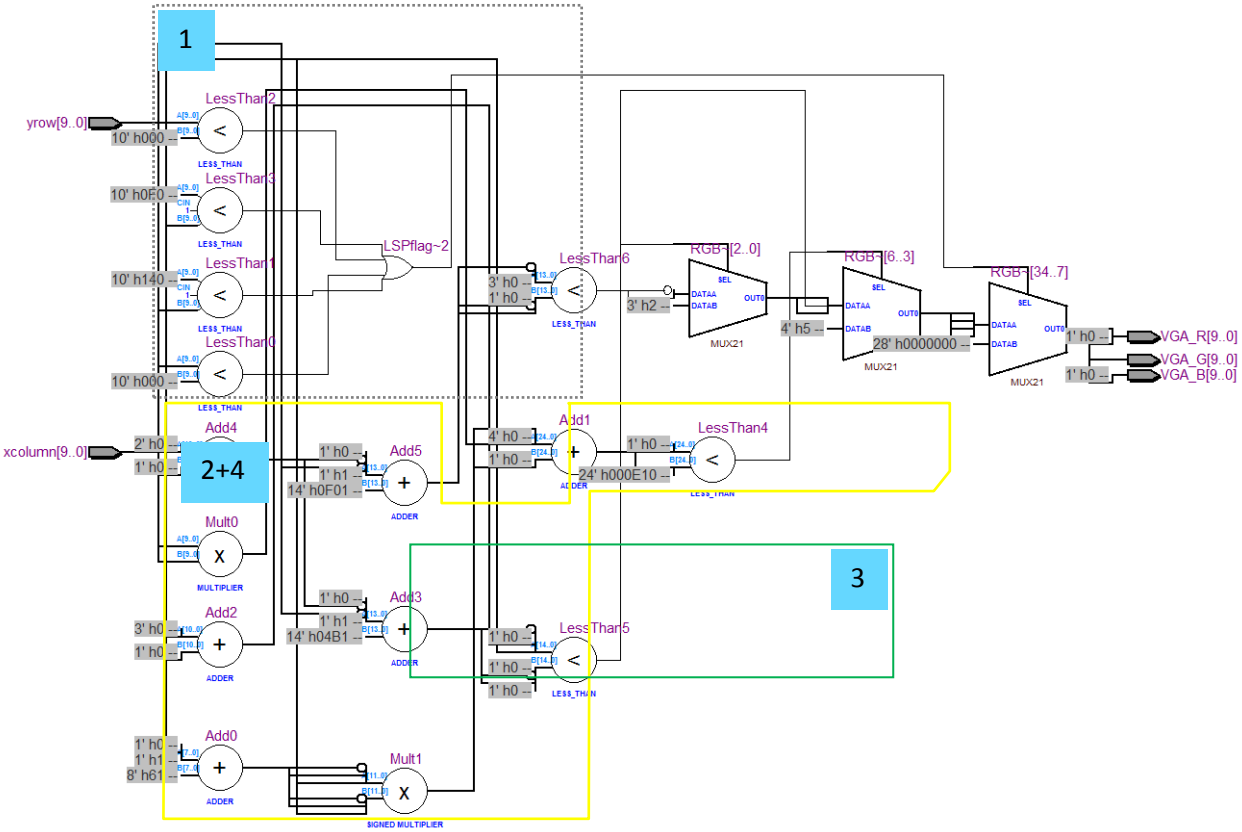
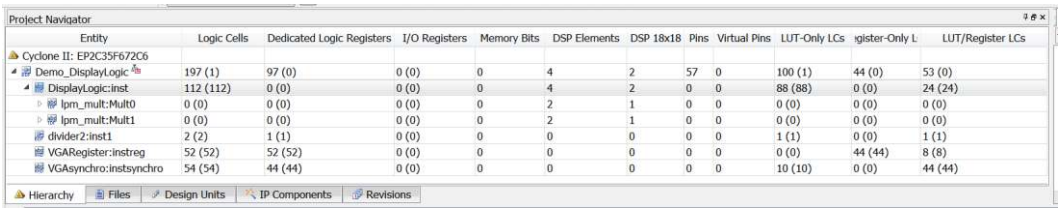


Figure 5 - Final circuit in RTL Viewer

On the Hierarchy tab of Project Navigator, we also see how many logical elements we have used for DisplayLogic itself. So far, we are economical; we only utilized 112 LE and 4 DSP blocks as two 18-bit multipliers, i.e., four 9-bit. Until now, however, the number increases after dividing the flag into four parts.



If we write LATCH or loop into message filter field, the list must remain empty

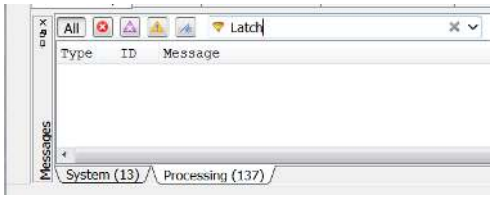
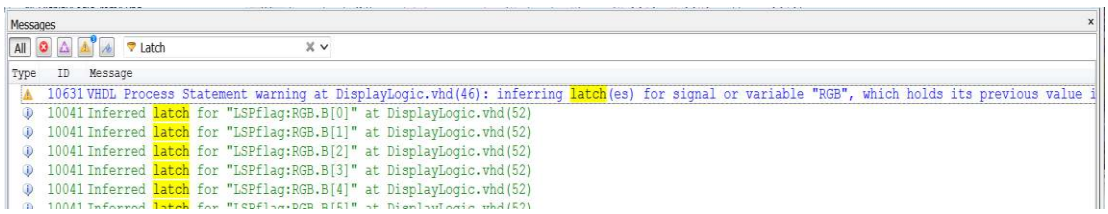


Figure 6 - LATCH check

The LATCH messages would appear after commenting out ELSE part of the previous VHDL code.

```
elsif 8*y < 8*YSIZE- 3*x then     -- line equation  y = 240-(3/8)*x
  RGB:=YELLOW;
-- else              --  RGB:=BLUE;
end if;
```

Missing final ELSE forces Quartus to maintain the previous RGB value. Latches are the only possible solution for such storages, but they are always susceptible to random operations on the FPGA, therefore, unstable on them.



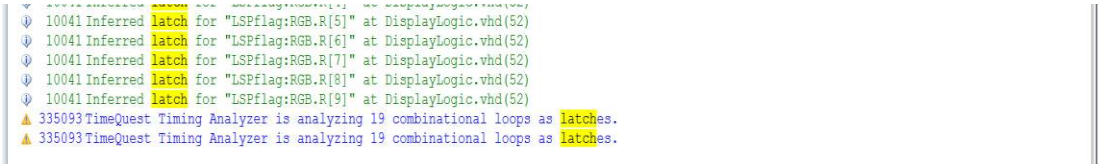


Figure 7 - Latch always announce errors in your design

Comment of student Maxo Groucho: We had worked with Latches at my technical school, and they function well!

Answer: Latches work if they were constructed directly from gates and you satisfy strictly defined conditions for their inputs, called fundamental mode, more in later lectures. However, FPGAs implement logic in a different way. Many years ago, one stubborn long-distance student tried to apply a solution he knew from his high school. He ignored my objections that he had Latches there, and he had been trying for the entire semester without succeeding. He wore a printed board with him, on which he had connected the circuit from TTL gates. It worked well here. However, FPGAs make logic by multiplexers with much faster responses than TTL, so they are much more sensitive to disturbing pulses :-)

Step 2 - Adding image

We demonstrate all in practical exercises, so wait until you hear the explanation. Here is an only brief guide.

First, we create our image of the appropriate size and resolution in a graphical editor. For our 0 and 1 picture in the flag, a black and white image, i.e., 1-bit color: '1' is red and '0' is not red. We saved it without "Dithering", sometimes called "Anti-Aliasing". We take into account that memory reading has a minimum delay of one pixel because the address is always internally stored in the registers, so we leave the leftmost column blank. We can easily edit stored B/W picture, for example, in the freeware Greenfish Icon Editor Pro (http://greenfishsoftware.org/), which allows higher zoom than Windows Paint, and shows real-size looks at the same time.

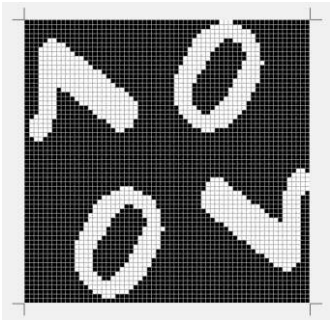


Figure 8 - Figure 0 a 1 uložený jako Black & White

We convert bitmap with the aid of utility in ZIP folder '\LSPVga2017_V1\WindowsUtility\Bitmap2Mif.exe' to MIF initialization file for memories.

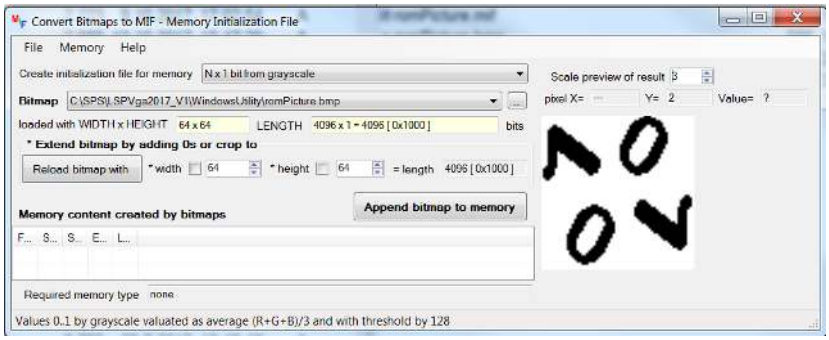


Figure 9 - Bitmap2Mif

Do not forget to correct image sizes to the appropriate values using [Reload bitmap with]. Here we chose 64x64, since multiplying by 64 is easily performed by logical shifts. Then, we add our bitmap to the list [Append bitmap to memory]. Bitmap2Mif allows embedding several bitmaps into one memory, see its HELP, to optimize the FPGA memory blocks.

Finally, we save the bitmap via the "Memory-> Save as memory Initialization File" menu to the Quartus project directory. Bitmap2Mif shows ROM parameters, so we write them down.

Memories in FPGA and their creation

In addition to logic, FPGAs also contain memory blocks, because they consume less energy and silicon, thus, they allow a higher density of integration than logical elements. So we have two ways how to store our picture data in FPGA. Each has its advantages and disadvantages:

- The memory block allows up to 2-port read and write access, see the lectures, so two independent parts can share the same data. In any case, our data always utilize the whole memory block, even if we do not fill it whole. Everything is how to design a memory content.
- Logical elements (LE) allow more optimal data storage. We utilize only as many as logic as our data requirements, and sometimes even less due to possible logical minimization. However, they are not shared, so each their usage means a new allocation of resources. Take also into account that LEs allow creating of a more extensive range of circuits than mere remembering values, so it is better to save them for more demanding operations unrealizable by memories.

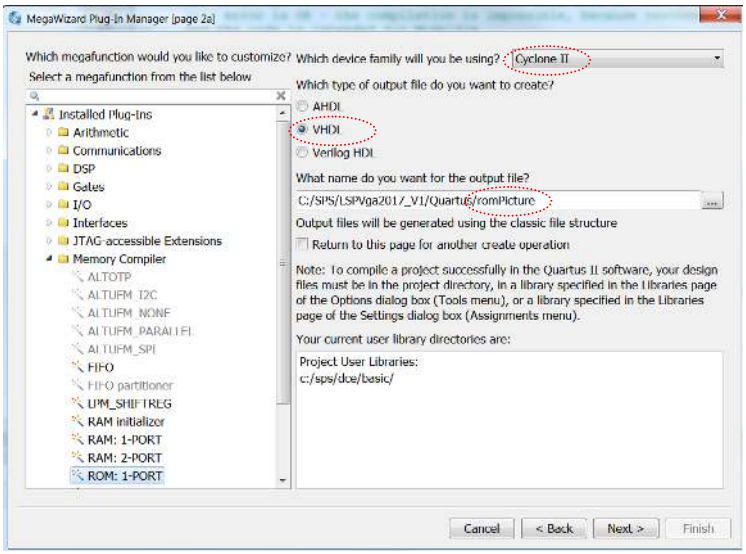
Cyclon II series EP2C35 contains 105 M4k blocks. Each one offers 4608 bits - part of them is sometimes used for parity. In theory, we have up to 483 840 bits, for more see Zip file \ LSPVga2017_V1 \ Documentation \ cyc2_cii51008_MemoryBlocks.pdf.

For each block, there are the following possible configurations: 4096 × 1, 2048 × 2, 1024 × 4, 512 × 8, 512 × 9, 256 × 16, 256 × 18, 128 × 32 and 128 × 36. The first number is the count of words and the second the width in bits of our memory words. For example, 1024 x 4 means that the memory contains 4-bit words addressed by a 10-bit address, 2^{10} = 1024, which is a total of 1024 words of 4-bit width, i.e., 4096 bits in total.

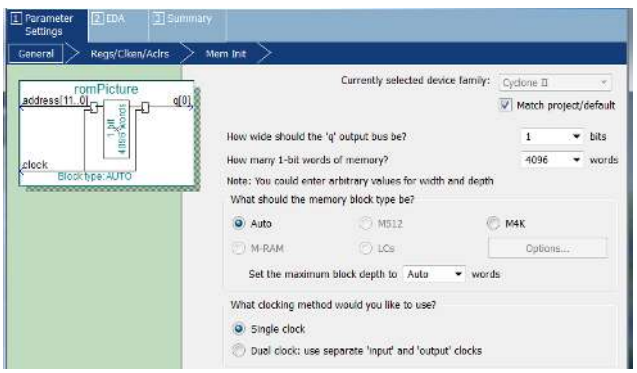
Creating memory

A detailed description is in ZIP folder \LSPVga2017_V1\Dokumentace\ug_lpm_rom_MegaWizard.pdf.

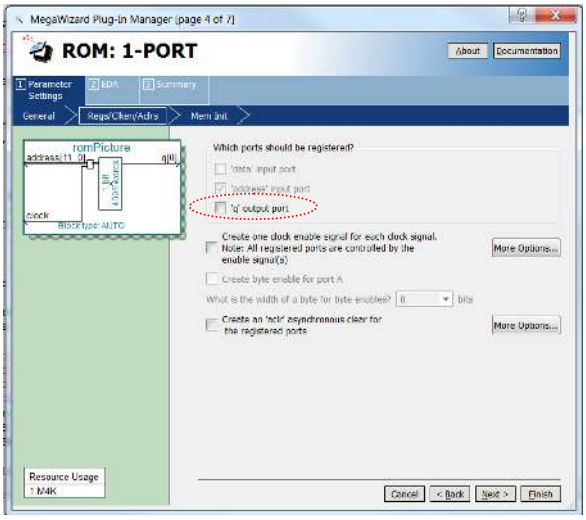
1/ We select in Quartus menu: Tools->**MegaWizard Plug-In Manager** a and we choose **ROM 1-Port** on initial screen. We enter a name of memory, for example, '**romPicture'**, VHDL code a certainly family Cyclon II.



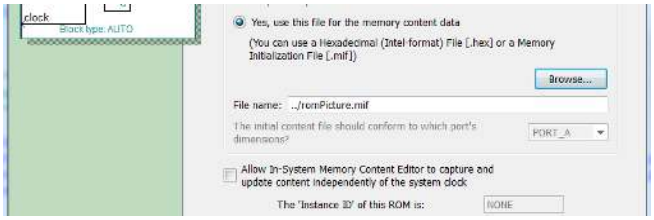
2/ On page 3, we select the required amount of memory according to the information Bitmap2Mif has provided. Note that the values are always powers of 2 because words are selected from the memory by the address sent as a binary number. We have entered 4096 words with 1-bit length:



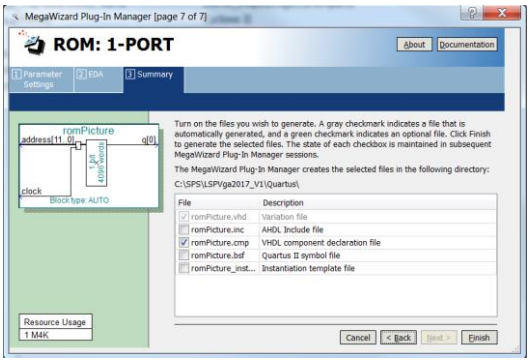
3/ On next page 4, **we do not forget to uncheck q output port** otherwise our memory sends data with a delay of 2 clocks instead of 1 caused by its address register.



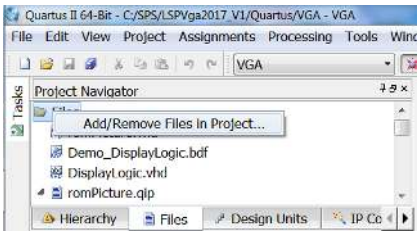
4/ On the next dialog, page 5, we enter the name of initialization file created in Bitmap2Mif. We find out it by button [Browse...].

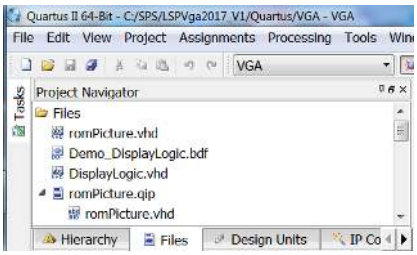


5/ We skip the following dialog, and we confirm the creation of memory on final page 7 by [Finish].



In the Quartus file list, we have a new element, **romPicture.qip**, and **romPicture.vhd** under it. If we do not see file romPicture.vhd, we add it from the context menu of Files (the right mouse button) and Add/Remove Files in Project.





We open romPicture.vhd a search for 'init_file =>' line with the name of our MIF file.

```
altsyncram_component : altsyncram
GENERIC MAP (
        clock_enable_input_a => "BYPASS",
        clock_enable_output_a => "BYPASS",
        init_file => "../romPicture.mif",
```

If you see something in front of the name of the file, it is the result of a wrong synchronization with Windows directory names, so we edit line to **init_file => "romPicture.mif"**, that is just the name of our file. Quartus sometimes finds out the file without editions, but the bug would appear later in ModelSim. Then, we will save the repaired file romPicture.vhd.

Adding memory into our project

We save our **DisplayLogic.vhd** under new name **DisplayLogicWithRom.vhd**. Indeed, we also rename its ‚entity' and ‚architecture' to the same name, i.e., DisplayLogicWithRom.

We insert signal VGA_CLK required for ROM into DisplayLogicWithRom entity ports:

```
entity DisplayLogicWithRom is
port(  yrow, xcolumn : in unsigned(9 downto 0); -- row and  column indexes of VGA video
       VGA_CLK : in std_logic;
      VGA_R, VGA_G, VGA_B: out std_logic_vector(9 downto 0)); --  color information
end;
```

Now, we open in Quartus, or in any text editor, the romPicture.cmp file created with our ROM memory. We select the component description and paste it into DisplayLogicWithRom.vhd, into the 'architecture' section in front of its ‚begin'. Then, we close romPicture.cmp, we will not needed it anymore. We define constants specifying parameters of ROM a signals: picture_address_s and picture_q_s for its connection with other code.

```
constant EMBORGX : integer := 220; -- positions of picture in the flag
constant EMBORGY : integer := 32;
constant MEMROWSIZE : integer := 64; -- memory organization
constant MEMROWCOUNT : integer := 64;
constant MEM_END_ADDRESS : integer := 4095;
COMPONENT romPicture is
  PORT (  clock        : in std_logic  := '1';
          address      : in std_logic_vector (11 downto 0);
          q            : out std_logic_vector (0 downto 0) );
END COMPONENT;
signal picture_address_s : STD_LOGIC_VECTOR(11 DOWNTO 0);
signal picture_q_s : std_logic;
```

In section begin of architecture, we insert mapping ports of our component to our signals as PORT MAP, i.e., a hardware analogy of calling constructor ‚new' in object programming. This statement creates a usable instance of the circuit.

```
begin -- architecture
rom_inst : romPicture PORT MAP( clock => VGA_CLK,          address => picture_address_s,      q(0) => picture_q_s);
```

Our LSPflag process changes only slightly. First, we must add picture_q_s into its 'sensitivity list'.

We also define isPicture variable, which contains the information whether x and y coordinates are in the part where the image lies. We use this variable in ELSIF conditions and for calculating the address sent to memory. We calculate it as a linear coordinate transformation: ((y-EMBORGY)*MEMROWSIZE + (x-EMBORGX).

```
 LSPflag : process(xcolumn, yrow, picture_q_s) -- output of process depends on xcolumn and yrow
   variable RGB : RGB_type; -- output colors
   variable x, y : integer; -- renamed xcolumn and yrow
   variable isPicture:boolean;
   begin
    x:=to_integer(xcolumn); y:=to_integer(yrow); -- convert to integer
    isPicture:= x>=EMBORGX and x<EMBORGX+MEMROWSIZE   and y>=EMBORGY and y<EMBORGY+MEMROWCOUNT;
    if(x<0) or (x>=XSIZE) or (y<0) or (y>=YSIZE) then RGB:=BLACK; --black outside of visible frame
    elsif  isPicture  and  picture_q_s = '1'  then RGB:=RED;
    elsif x*x+(y-YSIZE)*(y-YSIZE) < YSIZE*YSIZE/16 then RGB:=YELLOW; -- yellow circle
    elsif 5*y < 5*YSIZE - 6*x then  RGB:=GREEN;  -- line equation  y = 240-(6/5)*x
    elsif 8*y < 8*YSIZE- 3*x then     RGB:=YELLOW;   -- line equation  y = 240-(3/8)*x
    else  RGB:=BLUE;
    end if;
    if isPicture then picture_address_s <= std_logic_vector(
                to_unsigned((y-EMBORGY)*MEMROWSIZE + (x-EMBORGX), picture_address_s'LENGTH)
                                                          );
    else picture_address_s <=(others=>'0'); end if;
  -- Copy results in RGB to outputs of entity
    VGA_R<=RGB.R; VGA_G<=RGB.G; VGA_B<=RGB.B;
-------------------------------------------------------------------------
end process;
```

After creating symbol of **DisplayLogicWithRom.vhd** (File->Create/Update -> Create Symbol Files from Current File), we can test the result.

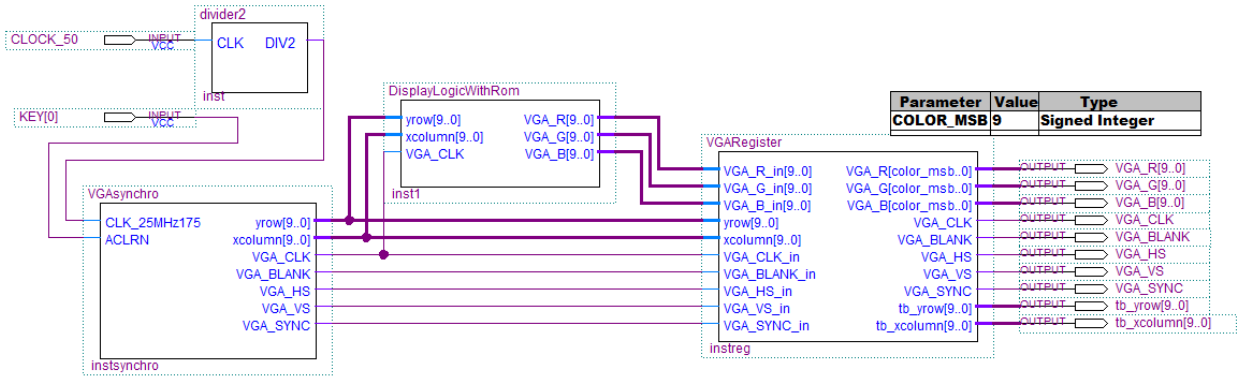


Figure 10 - Testing DisplayLogicWithRom

The test circuit is virtually the same as for DisplayLogic, and there is only added VGA_CLK signal input required by ROM memory.

Important warning:

1. You send the address memory, but the result of reading it, that is, the sent data, will appear next VGA_CLK. Thus, you are receiving a color information of a point with one cycle lag. The image bitmap should not have visible pixels in the zero column.

2. The images were deliberately selected to have sizes of a preferred width, MEMROWSIZE, can be easily multiplied during calculating of addresses based on the formula

address_sent_to_MEM: = y_MEM_row * MEMROWSIZE + x_MEM_column;

3. The appropriate values for the MEMROWSIZE constant are, of course, all 2 powers, bit shifts calculate them, or their simple sums, e.g, 96 = 64 + 32, and so forth. There is no MEMROWCOUNT in the formula above. So, you can create ROM with any number of its rows.

4. If you have more colors in your image, you can make their encoding. For example, 00 corresponds to nothing, 01 to red, 10 to yellow, and 11 to purple. As you define your mapping, colors are displayed. Bitmap2Mif allows direct encoding of RGB color channels, see the ComboBox option at the top of the Bitmap2Mif main window

Sneer of student Maxo Groucho: Great, but I do not clap to hail your wires, Mr. Teach. I would also slash the picture with only straight lines and a wheel. Well, as I look at DCENET site of our subject, the more pointed flags contain such strange zigzag lines that look like high order curves or goniometric rills. Have you noticed that at all, for hell on silicon?

Answer: I simplified my job, I confess. Why should I complicate example life? First, I have created the LSP flag in Quartus to obtain lovely picture :-) Then, I print screen the result. However, even curvatures can be remembered.

OK, if you insist, I open Corel Draw, I play with splines a little, and then I program wavy flag.

Step 3 - Curved flag

The flag was a little distorted. Using on-line regression, such as http://polynomialregression.drque.net/online.php, we find out the curve-point equations from the 320x240 pixel image by measuring points, for which you can use Windows painting that shows pixel coordinates.

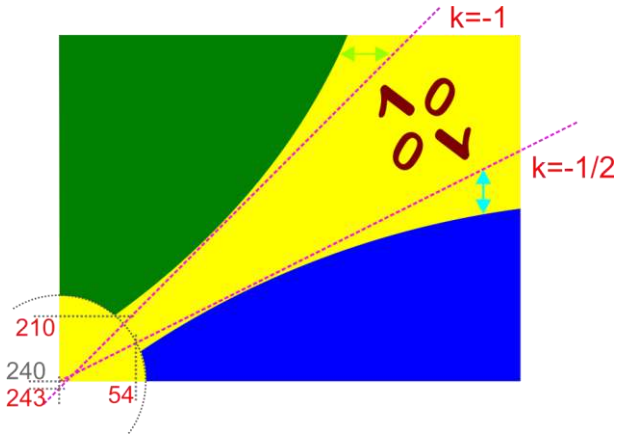


Figure 11 - Geometry of curved flag

For higher accuracy, we calculate x = f (y) for the green curve and y = g (x) for the blue part. We find out for green area curve the equation:

```
x = 200.07962123026428 - 0.45668091895971996 * y
    - 0.0010379498107116998 * y * y - 0.00000461262340701744 * y * y * y;
```

a and for blue:

```
y= 257.99876330742086 - 0.7704613646428253 * x
    + 0.001154052342444034 * x * x - 2.9903379875482e-7 * x * x * x;
```

Complex curves cannot be evaluated in the hardware. If we saved them directly as x or y coordinates, we should store too big numbers that use a lot of memory. If we do not consume more M4k blocks by this way it would not matter for ROMs, but it would be a very serious problem in logic.

Let us take a ruler and try to find out straight lines in the picture that can be well calculated. Then, we save only the differences between them and curves. We select horizontal or vertical straight lines preferably with easy calculated slopes: +/- 1, 1/2 or 2. We find out two suitable straight lines in our flag:

x = 243-y a y = 240- x/2 => **2*y = 2*240 - x**

We write a program to calculate the differences from the lines. The distances from the blue part are stored already multiplied by 2 to avoid rounding errors in straight line calculations where is multiplication by 2.

We store the blue part as a constant array and we create a ROM memory for the green curve because its address depends on y-axes and changes only at the beginning of each line, so the delay of one cycle for reading ROM has no influence.

In the zip example, you can look at C# program written for this purpose, in LSPVga2017_V1\WindowsUtility\ IntArray2Mif '. It can generate a field of constants and a MIF file. You can use its code as the basis for your program.

For a blue curve, we generate constants and the initialization file for the memory of the green curve . Of course, the blue curve is stored in a separate VHDL, for example into BlueCurve.vhd ', to satisfy conditions that we should split code into more files, if it is possible, for increasing its clarity.

```
use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity BlueCurve is
port( xcolumn : in unsigned(9 downto 0);
  q  : out unsigned(6 downto 0)
);
end BlueCurve;

architecture rtl of BlueCurve is
type MemBlueCurve_type is array(54 to 319) of integer range 3 to 78;
constant MemBlueCurve : MemBlueCurve_type := (
13, 13, 12, 12, 12, 11, 11, 11, 11, 10,  -- 63
10, 10, 10, 9, 9, 9, 9, 9, 8, 8,  -- 73
8, 8, 7, 7, 7, 7, 7, 6, 6,  -- 83
6, 6, 6, 6, 5, 5, 5, 5, 5,  -- 93
5, 4, 4, 4, 4, 4, 4, 4, 4,  -- 103
4, 3, 3, 3, 3, 3, 3, 3, 3,  -- 113
3, 3, 3, 3, 3, 3, 3, 3, 3,  -- 123
3, 3, 3, 3, 3, 3, 3, 3, 3,  -- 133
3, 3, 3, 3, 3, 3, 3, 3, 4, 4,  -- 143
4, 4, 4, 4, 4, 4, 4, 4, 5, 5,  -- 153
5, 5, 5, 5, 5, 5, 6, 6, 6, 6,  -- 163
6, 6, 7, 7, 7, 7, 7, 8, 8, 8,  -- 173
8, 8, 9, 9, 9, 9, 9, 10, 10, 10,  -- 183
10, 11, 11, 11, 11, 12, 12, 12, 12, 13,  -- 193
13, 13, 14, 14, 14, 15, 15, 15, 15, 16,  -- 203
16, 16, 17, 17, 17, 18, 18, 19, 19, 19,  -- 213
20, 20, 20, 21, 21, 21, 22, 22, 23, 23,  -- 223
23, 24, 24, 25, 25, 25, 26, 26, 27, 27,  -- 233
28, 28, 29, 29, 29, 30, 30, 31, 31, 32,  -- 243
32, 33, 33, 34, 34, 35, 35, 36, 36, 37,  -- 253
37, 38, 38, 39, 39, 40, 40, 41, 41, 42,  -- 263
43, 43, 44, 44, 45, 45, 46, 47, 47, 48,  -- 273
48, 49, 49, 50, 51, 51, 52, 52, 53, 54,  -- 283
54, 55, 56, 56, 57, 58, 58, 59, 59, 60,  -- 293
61, 61, 62, 63, 63, 64, 65, 65, 66, 67,  -- 303
68, 68, 69, 70, 70, 71, 72, 73, 73, 74,  -- 313
75, 75, 76, 77, 78, 78);  -- 319
begin
  readG : process(xcolumn)
          variable index : integer range 0 to 1023;
          begin
            index := to_integer(xcolumn);
            if(index>=MemBlueCurve'LEFT and index<=MemBlueCurve'RIGHT) then
              q <= to_unsigned(MemBlueCurve(index),q'LENGTH);
            else
              q<=(others=>'0');
            end if;
          end process;
end;
```

For the green curve, we create romGreenCurve as 1-port ROM using the same procedure as in the case of pictureROM and enter its initialization file. We again check romGreenCurve.vhd and correct the path of the initialization file.

We save DisplayLogicWithRom.vhd under the new name DisplayLogicCurve.vhd, renaming the entity and the architecture of the new file. We put the memory components into the architecture declaration section and create their instances after 'begin' statement of architecture:

```
COMPONENT BlueCurve is
  PORT(
    xcolumn : in unsigned(9 downto 0);
    q  : out unsigned(6 downto 0));
END COMPONENT;

COMPONENT romGreenCurve IS
  PORT (
    clock: in std_logic  := '1';
    address         : in std_logic_vector (7 downto 0);
    q       : out std_logic_vector (5 downto 0) );
END COMPONENT;

signal picture_address_s : std_logic_vector(11 downto 0);
signal picture_q_s : std_logic;

signal greenCurve_address_s : std_logic_vector(7 downto 0);
signal greenCurve_q_s : std_logic_vector (5 downto 0);

signal blueCurve_q_s : unsigned(6 downto 0);

begin   -- architecture

picture_rom : romPicture
PORT MAP( clock => VGA_CLK, address => picture_address_s, q(0) => picture_q_s);

greenC_rom : romGreenCurve
PORT MAP(clock => VGA_CLK,address => greenCurve_address_s,q => greenCurve_q_s);

blueC_inst : BlueCurve
PORT MAP(xcolumn => xcolumn,  q => blueCurve_q_s);
```

Finally, we slightly modify LSPflag process, not forgetting to extend its sensitivity list by adding memory outputs.

```
 LSPflag : process(xcolumn, yrow, greenCurve_q_s, blueCurve_q_s, picture_q_s)
    variable RGB : RGB_type; -- output colors
    variable x, y : integer; -- renamed xcolumn and yrow
    variable isPicture:boolean;
    constant GREEN_LINE_ORG:integer:=243;
    begin
     x:=to_integer(xcolumn); y:=to_integer(yrow); -- convert to integer
     isPicture:= x>=EMBORGX and x<EMBORGX+MEMROWSIZE
            and y>=EMBORGY and y<EMBORGY+MEMROWCOUNT;
     if(x<0) or (x>=XSIZE) or (y<0) or (y>=YSIZE) then
        RGB:=BLACK; --black outside of visible frame
     elsif isPicture and picture_q_s = '1'  then RGB:=RED;
     elsif x*x+(y-YSIZE)*(y-YSIZE) < YSIZE*YSIZE/16 then  RGB:=YELLOW;
     elsif x < GREEN_LINE_ORG-y-to_integer(unsigned(greenCurve_q_s)) then  RGB:=GREEN;
     elsif 2*y < 2*YSIZE - x + to_integer(blueCurve_q_s) then  RGB:=YELLOW;
     else    RGB:=BLUE;
     end if;
     if isPicture then picture_address_s <= std_logic_vector(
            to_unsigned((y-EMBORGY)*MEMROWSIZE + (x-EMBORGX),picture_address_s'LENGTH) );
     else picture_address_s <=(others=>'0');
     end if;
      greenCurve_address_s<=std_logic_vector(yrow(greenCurve_address_s'RANGE));
     VGA_R<=RGB.R; VGA_G<=RGB.G; VGA_B<=RGB.B;-- Copy results in RGB to outputs of entity
-------------------------------------------------------------------------
end process;
```

After compiling, we check Latches and loops in the message window and look at Project Navigator windows how much we used. We have only consumed 242 logic elements, 4 DSPs and 5632 bits of memory, but two M4Ks - one for the image and the other for the green curve. Thus, we utilized 8 kilobytes of memory. The green curve used only 1536 bits, while the blue wanted only 95 logical elements.

Project Navigator

Entity	Logic Cells	I/O Registers	Memory Bits	M4Ks	DSP Elements	DSP 9x9	DSP 18x18	Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Cyclone II: EP2C35F672C6											
Demo_DisplayLogicCurve	326 (1)	0 (0)	5632	2	4	0	2	57	228 (1)	43 (0)	55 (0)
divider2:inst	2 (2)	0 (0)	0	0	0	0	0	0	1 (1)	0 (0)	1 (1)
DisplayLogicCurve:inst1	242 (147)	0 (0)	5632	2	4	0	2	0	216 (123)	0 (0)	26 (24)
BlueCurve:blueC_inst	95 (95)	0 (0)	0	0	0	0	0	0	93 (93)	0 (0)	2 (2)
romGreenCurve:greenC_rom	0 (0)	0 (0)	1536	1	0	0	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult0	0 (0)	0 (0)	0	0	2	0	1	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult1	0 (0)	0 (0)	0	0	2	0	1	0	0 (0)	0 (0)	0 (0)
romPicture:picture_rom	0 (0)	0 (0)	4096	1	0	0	0	0	0 (0)	0 (0)	0 (0)
VGARegister:instreg	53 (53)	0 (0)	0	0	0	0	0	0	0 (0)	43 (43)	10 (10)
VGAsynchro:instsynchro	54 (54)	0 (0)	0	0	0	0	0	0	10 (10)	0 (0)	44 (44)

Hierarchy | Files | Design Units | IP Components | Revisions

Figure 12 - DisplayLogicCurve in FPGA

Comment of student Maxo Groucho: Why do we use memory? Logical elements are easier to write and even smaller!

*Answer: Only seemingly. It is just a sweet smile :-) Each logical element (LE) stores 16 bits and 16 * 95 = 1520. Both ways of implementation have roughly the same demands, except for the fact that memories always eat whole blocks.*

It could arise illusion that arrays of constants are suitable. However, we will have to calculate each curve twice after breaking flag into four parts. Unlike to arrays of constants, ROM does not need to be duplicated because it offers 2-port access. Additionally, LEs have more flexible usage. They implement a broader number of circuits, while the memories are useful only for storing values. If we have such memorable data, we prefer ROM and save logical elements.

Keep also in mind that the usage of FPGAs resources never reaches 100%. Real designs suggest that the average percentage lies somewhere between 60 and 70% of all available elements. Therefore, FPGAs are used only for development and small production series. Their circuit designs encounter severe limit -the number of available interconnection links between the parts, more in later lectures. Memories save these connections because they utilize efficient internal grid for reading and writing operations.

Finally, we look with the aid of RTL Viewer what we have created.

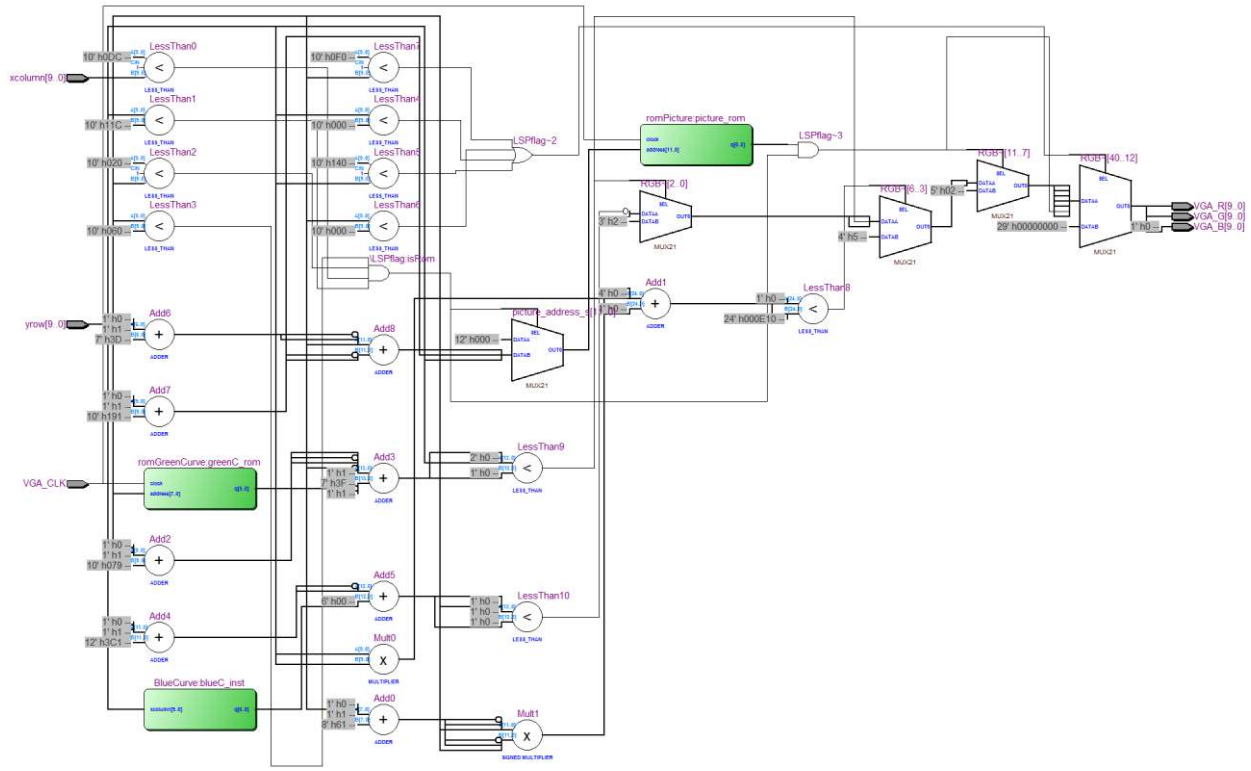


Figure 13 - DisplayLogicCurve implementation

In the schematic, we can see a cascade of multiplexers and memories as green rectangles, which is the result of a good file structure. If we have inserted all BlueCurve array code into DisplayLogicCurve.vhd, RTL Viewer shows a tangle of connections, so the teacher could easily discover and reject our task. :-(

Step 4 - Preparing for Splitting out Flag into 4 Parts - Definition of Functions.

If we first create a static flag and debug it, then, it is relatively easy to slip it into 4 parts. Parts may, however, have different positions on the screen and overlap, so we create a separate multiplex cascade for each ot them. We will have 4, thus, many conditions will be repeated. That's why we'll introduce functions and procedures to make our code clearer and easier to reuse.

Because a direct division into four parts would introduce too many new concepts, we first make an intermediate step, in which we rewrite DisplayLogicCurve.vhd by using functions and procedures, although they are unnecessary here because DisplayLogicCurve.vhd contains only one cascade of multiplexers, so each condition is calculated only once.

Remember that **the functions and procedures are never invoked in the VHDL**, but their bodies are all inserted into code by a style known as inline expansion or inlining. This means that **each call of a function or procedure is replaced by the insertion of its entire body**, in which only the formal parameters are replaced by those actually used.

In C programs, functions and procedures reduce a resultant EXE, but in the VHDL, a final circuits have the same size. But they make the code clearer, and any errors remain in one place, unlike 'copy-paste' writing techniques.

We save DisplayLogicCurve.vhd as **DisplayLogicCurveFunctions.vhd** a rename also entity and architecture in new VHDL. We add definitions of subtypu after constant YSIZE and XSIZE

```
constant YSIZE : integer := 240; -- height of flag
constant XSIZE : integer := 320; -- width of flag
constant XY_MAXSIZE : integer := XSIZE;
-- Note, VHDL2008 contains MAXIMUM function, but ModelSim requires VHDL1993
subtype xyintd is integer range 0 to XY_MAXSIZE; -- relative coordinates
```

and we define function for expressions that calculate surrounding of parts and addresses for memories.

```
function InCircle(x,y:xyintd) return boolean is
    begin
      return x*x + (y - YSIZE)*(y - YSIZE) < YSIZE*YSIZE/16;
    end function;

function InGreen(x,y:xyintd; xoffset:std_logic_vector) return boolean is
    begin
      return x < GREEN_LINE_ORG - y - to_integer(unsigned(xoffset));
    end function;

function InBlue(x,y:xyintd;yoffset:unsigned) return boolean is
    begin
      return 2*y >= 2*YSIZE - x + to_integer(yoffset);
    end function;

-- Return true if x and y are inside of picture
function InPicture(x,y:xyintd) return boolean is
    begin
      return x >= EMBORGX and x < EMBORGX + MEMROWSIZE
                  and y >= EMBORGY and y < EMBORGY + MEMROWCOUNT;
    end function;
```

```
function CalculatePictureAddress(x,y:xyintd) return std_logic_vector is
    variable address:std_logic_vector(11 DOWNTO 0);
    begin
      address := std_logic_vector(to_unsigned((y - EMBORGY)*MEMROWSIZE + (x - EMBORGX), address'LENGTH));
      return address;
    end function;
```

To convert to xyintd, we create a procedure that, unlike a function, can return more values. Remember that there are **no reference types in VHDL**. We compile a circuit and it must know the values of the signals, not a mere reference to them.

```
      -- convert xy unsigned to <0,limit>
procedure set_limit(
            xy: in unsigned(9 downto 0); --input value
            limit: in natural;  -- integer>=0
            z: out xyintd; --output
            outside:out boolean -- xy outside of <0,limit>
            )  is
    variable m:integer range 0 to 1023;
    begin
        m:=to_integer(xy);
        if m>=limit then z:=limit; outside:=true;
        else z:=xyintd(m); outside:=false;
        end if;
  end procedure set_limit;
```

We simplified the body of the main process

```
  begin
    set_limit(xcolumn, XSIZE, xd, x_outside);
    set_limit(yrow, YSIZE, yd, y_outside);

    isPicture:= InPicture(xd,yd);

    if x_outside or y_outside then RGB:=BLACK; --black outside of visible frame
    elsif isPicture and picture_q_s = '1' then RGB:=RED;
    elsif InCircle(xd,yd) then RGB:=YELLOW;
    elsif InGreen(xd,yd,greenCurve_q_s) then RGB:=GREEN;
    elsif not InBlue(xd,yd,blueCurve_q_s) then RGB:=YELLOW;
    else  RGB:=BLUE;
    end if;
    if isPicture then  picture_address_s <= CalculatePictureAddress(xd,yd);
    else picture_address_s <=(others=>'0');
    end if;

    greenCurve_address_s<=std_logic_vector(yrow(greenCurve_address_s'RANGE));
    -- Copy results in RGB to outputs of entity
    VGA_R<=RGB.R; VGA_G<=RGB.G; VGA_B<=RGB.B;
--------------------------------------------------------------------------------
    end process;
```

Step 5 - Splitting flag to 4 parts

We save DisplayLogicCurveFunctions.vhd as DisplayLogicCurve4parts.vhd and rename both the entity and the architecture in the new VHDL file.

Editing is complicated by the need to evaluate twice each green and blue curve, because the yellow part lays between them. We have the blue curve in the constant array. Thus, we must necessarily create another its instance and of course also declare signals for addresses.

```
blueCB_inst : BlueCurve
PORT MAP(xcolumn => blueCurve_addressB_s,
         q       => blueCurveB_q_s);
blueCY_inst : BlueCurve
PORT MAP(xcolumn => blueCurve_addressY_s,
         q       => blueCurveY_q_s);
```

We have two possible solutions for the green curve. Either we can create another instance of its 1-port ROM, or we can re-generate ROM memory as a 2-port, which is much more convenient.

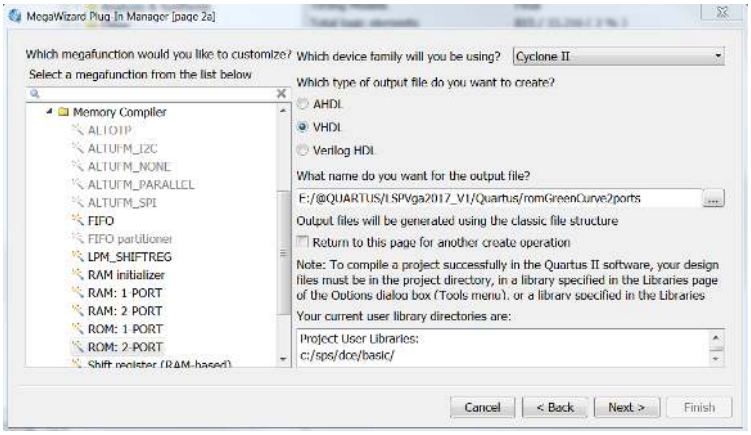


Figure 14 - Creation of 2-port ROM for green curve

On the other MegaWizard dialog pages, enter the same following values as for 1-port memory. Do not forget also to correct the initialization file name in created file romGreenCurve2ports .vhd. Then we put its component into architectural declarations:

```
COMPONENT romGreenCurve2ports IS
PORT
( clock      : IN std_logic  := '1';
  address_a       : IN std_logic_vector (7 DOWNTO 0);
  address_b       : IN std_logic_vector (7 DOWNTO 0);
  q_a        : OUT std_logic_vector (5 DOWNTO 0);
  q_b        : OUT std_logic_vector (5 DOWNTO 0)
);
END COMPONENT;
```

Finally, we create instance of 2-port ROM in the architecture implementation section. In the example code, you can fin out the both solutions, two instances of 1-port memory, or one from 2-port ROM.

```
-------------------------------------------------------------------
-- If you want to see the solution for 2 instances of 1-port ROM, only uncomment greenCG_rom and greenCY_rom port maps
-- and comment out green2_rom port map
-------------------------------------------------------------------
--   greenCG_rom : romGreenCurve
--       PORT MAP(clock   => VGA_CLK,  address => greenCurve_addressG_s,  q => greenCurveG_q_s);
--   greenCY_rom : romGreenCurve
--       PORT MAP(clock   => VGA_CLK,  address => greenCurve_addressY_s, q => greenCurveY_q_s);
  green2_rom : romGreenCurve2ports -- 2 port rom
    PORT MAP(clock   => VGA_CLK, address_a => greenCurve_addressG_s, address_b => greenCurve_addressY_s,
          q_a => greenCurveG_q_s,          q_b => greenCurveY_q_s);
-------------------------------------------------------------------
```

We have not increased memory consumption with 2-port Rom , there is just added double reading logic, see lectures, we see only one ROM memory so in RTL Viewer. See also the RTL Viewer Schema in Appendix, pg. 33.

We also create the 0-1 picture as 2-port ROM to cut out if from the yellow part.

We should specify the ranges of integers. To simplify ranging, we create a subtype xyinteger for x and y and subtype xyintd for the relative coordinates of flags parts.

```
subtype xyinteger is integer range -1024 to 1023;
subtype xyintd is integer range -1 to XSIZE-1; -- relative coordinates
type xyPosition_type is array (0 to 7) of xyinteger;
signal xyPosition : xyPosition_type;
```

In process LSPflag, we utilize our functions from previous 4th step.

```
LSPflag : process(xcolumn, yrow, xyPosition, blueCurveB_q_s, blueCurveY_q_s,
          greenCurveG_q_s, greenCurveY_q_s, picture_q_s)
  variable RGB : RGB_type; -- output colors
  variable x0d, y0d, x1d, y1d, x2d, y2d, x3d, y3d : xyintd;
  variable x0d_outside, y0d_outside, x1d_outside, y1d_outside: boolean;
  variable x2d_outside, y2d_outside: boolean; -- x3d_outside, y3d_outside are not necessary

  variable isPicture, isPictureY, isGreenInFlag, isYellowInFlag, isBlueInFlag : boolean;
  constant GREEN_LINE_ORG : integer := 243;
  function InCircle(x,y:xyintd) return boolean is
  begin
      return x*x + (y - YSIZE)*(y - YSIZE) < YSIZE*YSIZE/16;
  end function;
  function InGreen(x,y:xyintd; xoffset:std_logic_vector) return boolean is
  begin
      return x < GREEN_LINE_ORG - y - to_integer(unsigned(xoffset));
  end function;
  function InBlue(x,y:xyintd;yoffset:unsigned) return boolean is
  begin
      return 2*y >= 2*YSIZE - x + to_integer(yoffset);
  end function;
  -- Return true if x and y are inside of  1/0 picture in flag
  function InPicture(x,y:xyintd) return boolean is
  begin
    return x >= EMBORGX and x < EMBORGX + MEMROWSIZE
        and y >= EMBORGY and y < EMBORGY + MEMROWCOUNT;
  end function;
  function CalculatePictureAddress(x,y:xyintd) return std_logic_vector is
  variable address:std_logic_vector(11 DOWNTO 0);
  begin
    address := std_logic_vector(to_unsigned((y - EMBORGY)*MEMROWSIZE + (x - EMBORGX), address'LENGTH));
      return address;
  end function;
```

We replace the **set_limit** procedure with another **diff_limit**, where we do not only convert, but we also apply shifts of the parts using the offset set to the placement sticker in the center.

How do we get the flag into the center of the image? Easily, we subtract constants XSIZE / 2 and YSIZE / 2 from the xcolumn and yrow , i.e., we perform a linear translation of the coordinate using translation.

```
procedure diff_limit(
        xy: in unsigned(9 downto 0); --input value
        offset: in xyinteger; --offset of position
        limit: in natural;  -- integer>=0
        z: out xyintd; -- result
        outside:out boolean -- xy is outside of <0,limit>
        ) is
variable diff:integer;
begin

  diff:=to_integer(xy) - limit/2 - offset;

  if diff>=limit or diff<0 then z:=limit; outside:=true;
     else z:=xyintd(diff); outside:=false;
     end if;
end procedure diff_limit;
```

We overload „or" operator for RGB colors:

```
function "or"(color1, color2 : RGB_type) return RGB_type is
variable result : RGB_type;
begin
  if color2/=WHITE  then
     result.R := color1.R or color2.R;
     result.G := color1.G or color2.G;
     result.B := color1.B or color2.B;
    else -- white color is not in our flag, but this code can be inspiration for you
     result.R := color1.R xor color2.R;
     result.G := color1.G xor color2.G;
     result.B := color1.B xor color2.B;
    end if;
    return result;
end function;
```

The IF-ELSEIF conditions are relatively simple, see appendix. Before we calculate them, we just move the flag to the center of stceen and calculate relative coordinates of parts:

```
 begin
-- transformation of coordinates
    diff_limit(xcolumn, xyPosition(0), XSIZE, x0d, x0d_outside);diff_limit(xcolumn, xyPosition(1), XSIZE, x1d, x1d_outside);
    diff_limit(xcolumn, xyPosition(2), XSIZE, x2d, x2d_outside);
    diff_limit(xcolumn, xyPosition(3), XSIZE, x3d); -- out paramaters can be omitted

    diff_limit(yrow, xyPosition(4), YSIZE, y0d, y0d_outside);   diff_limit(yrow, xyPosition(5), YSIZE, y1d, y1d_outside);
    diff_limit(yrow, xyPosition(6), YSIZE, y2d, y2d_outside);
    diff_limit(yrow, xyPosition(7), YSIZE, y3d); -- out paramaters can be omitted here
```

```
isGreenInFlag  := not x0d_outside and  not y0d_outside;
isYellowInFlag := not x1d_outside and  not y1d_outside;
isBlueInFlag   := not x2d_outside and  not y2d_outside;
isPicture := InPicture(x3d,y3d); -- drawing picture
isPictureY:= InPicture(x1d,y1d); -- cut out of picture

RGB := BLACK;  -- Very important line, to prevent latches, we must initialize RGB color to the known value

-- Green part has x0d, y0d
if isGreenInFlag and not InCircle(x0d,y0d) and InGreen(x0d,y0d,greenCurveG_q_s)
then RGB := RGB or GREEN;
end if;

-- Yellow part has x1d, y1d
if isYellowInFlag and (not isPictureY or picture_qY_s='0') and
   (InCircle(x1d,y1d) or (not InGreen(x1d,y1d,greenCurveY_q_s) and not InBlue(x1d,y1d, blueCurveY_q_s)))
  then RGB := RGB or YELLOW;
  end if;

-- Blue part has x2d, y2d
if isBlueInFlag and not InCircle(x2d,y2d) and InBlue(x2d,y2d,blueCurveB_q_s)
then RGB := RGB or BLUE;
end if;

if isPicture and picture_q_s='1' then
   RGB := RGB or RED;
end if;

 -- Green curve memory utilizes 8 lower bits, thus, its address never overflows
 -- To prevent error messages in ModelSim, we converted to10 bits and select lower 8 bits
 if isGreenInFlag then
   greenCurve_addressG_s <=  std_logic_vector(to_unsigned(y0d, greenCurve_addressG_s'LENGTH));
else
   greenCurve_addressG_s <= (others=>'0');
end if;

if isYellowInFlag then
    blueCurve_addressY_s  <= to_unsigned(x1d, blueCurve_addressY_s'LENGTH);
    greenCurve_addressY_s <=  std_logic_vector(to_unsigned(y1d, greenCurve_addressY_s'LENGTH));
else
   blueCurve_addressY_s <= (others=>'0');
    greenCurve_addressY_s <= (others=>'0');
end if;

 if isBlueInFlag then    blueCurve_addressB_s  <= to_unsigned(x2d, blueCurve_addressB_s'LENGTH);
 else    blueCurve_addressB_s <= (others=>'0');
 end if;
```

```
    if isPicture then        picture_address_s <= CalculatePictureAddress(x3d,y3d);
    else
      picture_address_s <= (others => '0');
    end if;

    -- cut out picture from yellow part
    if isPictureY then       picture_addressY_s <= CalculatePictureAddress(x1d,y1d);
    else
      picture_addressY_s <= (others => '0');
    end if;

    -- Copy results in RGB to outputs of entity
    VGA_R <= RGB.R; VGA_G <= RGB.G; VGA_B <= RGB.B;
    ---------------------------------------------------------------------------
  end process;
```

To store the values of positions of parts, we add a short process inserted directly into DisplayLogicCurve4parts because it sets its internal memory.

```
  vs_copy : process(VGA_VS)
  variable dify, minusdify, difx, minusdifx : xyinteger;
  begin
    if rising_edge(VGA_VS) then -- Copy data to assure their stability in picture
        dify := to_integer(signed(xyoffset)); minusdify:=-dify;
        difx := dify+dify/4; minusdifx :=-difx;        -- x size is 5/4 of y
        xyPosition(0) <= minusdifx; xyPosition(4) <= minusdify;  -- move green part to the left upper corner
        xyPosition(1) <= minusdifx; xyPosition(5) <= dify;  -- move yellow part to the left bottom corner
        xyPosition(2) <= difx; xyPosition(6) <= dify;  -- move blue part to the right bottom corner
        xyPosition(3) <= difx; xyPosition(7) <= minusdify;  -- move 1/0 picture to the right upper corner
    end if;
end process;
```

The complete code of DisplayLogicCurve4parts.vhd is in appendix on pg. 27.

To test the result, we set the xyoffset of flag parts. ShowValue.vhd circuit takes switch data (SW[x]) as straight binary and displays them in decimal format. Its code is on pg. 34.

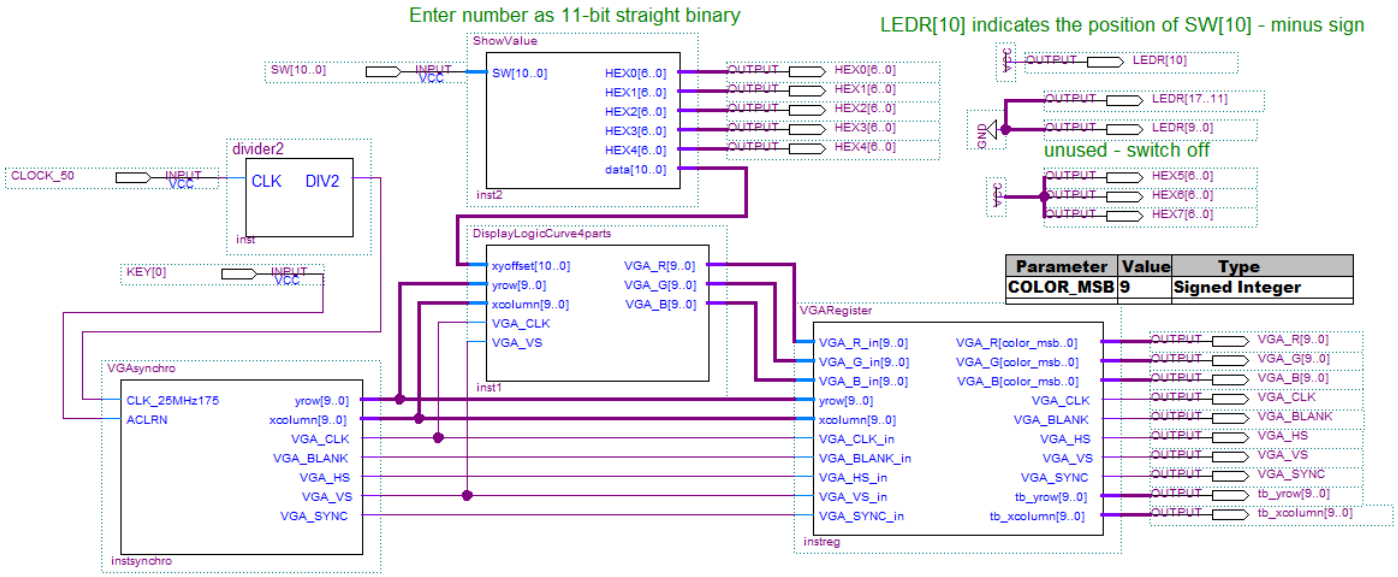


Figure 15 - Testing circuit

In Project Navigator, we check used elements. We are economical enough. The blue curve, inserted as a constant array, needed twice the logical elements, while the green and the picture took the same amount as before, which is the advantage of the usage of ROM memories.

Project Navigator

Entity	Logic Cells	I/O Registers	Memory Bits	M4Ks	DSP Elements	DSP 9x9	DSP 18x18	Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Cyclone II: EP2C35F672C6											
Demo_DisplayLogicCurve4parts	836 (1)	0 (0)	5632	2	4	4	0	142	696 (1)	43 (0)	97 (0)
divider2:inst	2 (2)	0 (0)	0	0	0	0	0	0	1 (1)	0 (0)	1 (1)
DisplayLogicCurve4parts:inst1	679 (470)	0 (0)	5632	2	4	4	0	0	611 (402)	0 (0)	68 (68)
BlueCurve:blueCB_inst	102 (102)	0 (0)	0	0	0	0	0	0	102 (102)	0 (0)	0 (0)
BlueCurve:blueCY_inst	107 (107)	0 (0)	0	0	0	0	0	0	107 (107)	0 (0)	0 (0)
romGreenCurve2ports:green2...	0 (0)	0 (0)	1536	1	0	0	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult1	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult2	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult4	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult5	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
romPicture2port:picture_rom	0 (0)	0 (0)	4096	1	0	0	0	0	0 (0)	0 (0)	0 (0)
ShowValue:inst2	83 (83)	0 (0)	0	0	0	0	0	0	73 (73)	0 (0)	10 (10)
VGARegister:instreg	53 (53)	0 (0)	0	0	0	0	0	0	0 (0)	43 (43)	10 (10)
VGAsynchro:instsynchro	54 (54)	0 (0)	0	0	0	0	0	0	10 (10)	0 (0)	44 (44)

Hierarchy | Files | Design Units | IP Components | Revisions

Figure 16 - Used resources in case of 2-port ROM and range limits after integers

For interest, try to comment out range specifications for ,subtype xyinteger a xyintd, for example, modify their definitions as:

```
subtype xyinteger is integer;    -- range -1024 to 1023;
subtype xyintd is integer;  -- range -1 to XSIZE-1; -- relative coordinates
```

Then, compile the project a look at changes:

Project Navigator

Entity	Logic Cells	I/O Registers	Memory Bits	M4Ks	DSP Elements	DSP 9x9	DSP 18x18	Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Cyclone II: EP2C35F672C6											
Demo_DisplayLogicCurve4parts	1419 (1)	0 (0)	5632	2	24	0	12	142	1271 (1)	43 (0)	105 (0)
divider2:inst	2 (2)	0 (0)	0	0	0	0	0	0	1 (1)	0 (0)	1 (1)
DisplayLogicCurve4parts:inst1	1262 (946)	0 (0)	5632	2	24	0	12	0	1186 (870)	0 (0)	76 (76)
BlueCurve:blueCB_inst	103 (103)	0 (0)	0	0	0	0	0	0	103 (103)	0 (0)	0 (0)
BlueCurve:blueCY_inst	101 (101)	0 (0)	0	0	0	0	0	0	101 (101)	0 (0)	0 (0)
romGreenCurve2ports:green2...	0 (0)	0 (0)	1536	1	0	0	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult1	28 (0)	0 (0)	0	0	6	0	3	0	28 (0)	0 (0)	0 (0)
lpm_mult:Mult2	28 (0)	0 (0)	0	0	6	0	3	0	28 (0)	0 (0)	0 (0)
lpm_mult:Mult4	28 (0)	0 (0)	0	0	6	0	3	0	28 (0)	0 (0)	0 (0)
lpm_mult:Mult5	28 (0)	0 (0)	0	0	6	0	3	0	28 (0)	0 (0)	0 (0)
romPicture2port:picture_rom	0 (0)	0 (0)	4096	1	0	0	0	0	0 (0)	0 (0)	0 (0)
ShowValue:inst2	83 (83)	0 (0)	0	0	0	0	0	0	73 (73)	0 (0)	10 (10)
VGARegister:instreg	53 (53)	0 (0)	0	0	0	0	0	0	0 (0)	43 (43)	10 (10)
VGAsynchro:instsynchro	54 (54)	0 (0)	0	0	0	0	0	0	10 (10)	0 (0)	44 (44)

Hierarchy | Files | Design Units | IP Components | Revisions

Figure 17 - Increase of resources wit 2-port ROMs, but without range after integer

The number of logical elements has now almost doubled, and multipliers have grown to three times, i.e, to 24, since Quartus's many could not estimate sizes of many integers and interpreted them as 32-bit numbers in the circuit. RTL Viewer also lists the number widths, so you can easily discover your non-range sins.

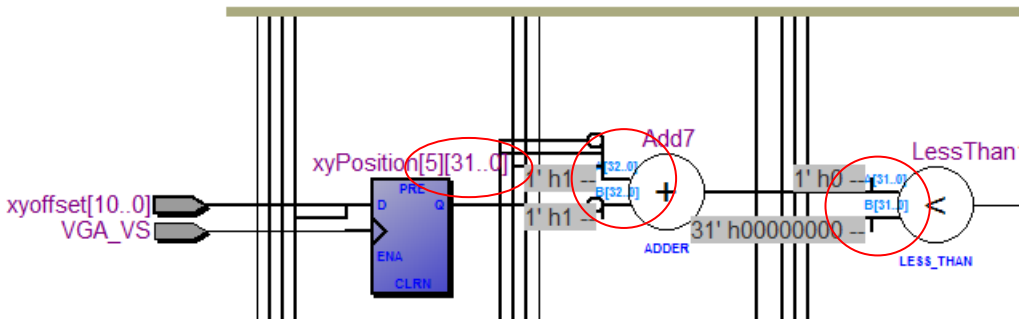


Figure 18 - Non-range sins

We perform another experiment by returning range specifications for both integer types and by creating two instances from 1-port image memories and green curves, making only changes to the comments at the beginning of the respective rows.

```
  picture_rom : romPicture
    PORT MAP(clock   => VGA_CLK,  address => picture_address_s,  q(0)    => picture_q_s);
  picture_romY : romPicture
    PORT MAP(clock   => VGA_CLK,  address => picture_addressY_s,   q(0)    => picture_qY_s);
--picture_rom : romPicture2port
--   PORT MAP(clock   => VGA_CLK,
--          address_a => picture_address_s,  -- picture
--          address_b => picture_addressY_s,  -- yellow - cut out from yellow part
--          q_a(0)    => picture_q_s, q_b(0)    => picture_qY_s);

  greenCG_rom : romGreenCurve
    PORT MAP(clock   => VGA_CLK,  address => greenCurve_addressG_s,    q   => greenCurveG_q_s);
  greenCY_rom : romGreenCurve
    PORT MAP(clock   => VGA_CLK,  address => greenCurve_addressY_s,    q      => greenCurveY_q_s);
--   green2_rom : romGreenCurve2ports -- 2 port rom
--   PORT MAP(clock   => VGA_CLK,
--          address_a => greenCurve_addressG_s,
--                   address_b => greenCurve_addressY_s,
--                   q_a => greenCurveG_q_s, q_b => greenCurveY_q_s);
```

Project Navigator

Entity	Logic Cells	I/O Registers	Memory Bits	M4Ks	DSP Elements	DSP 9x9	DSP 18x18	Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Cyclone II: EP2C35F672C6											
Demo_DisplayLogicCurve4parts	835 (1)	0 (0)	11264	4	4	4	0	142	695 (1)	43 (0)	97 (0)
divider2:inst	2 (2)	0 (0)	0	0	0	0	0	0	1 (1)	0 (0)	1 (1)
DisplayLogicCurve4parts:inst1	678 (469)	0 (0)	11264	4	4	4	0	0	610 (401)	0 (0)	68 (68)
BlueCurve:blueCB_inst	102 (102)	0 (0)	0	0	0	0	0	0	102 (102)	0 (0)	0 (0)
BlueCurve:blueCY_inst	107 (107)	0 (0)	0	0	0	0	0	0	107 (107)	0 (0)	0 (0)
romGreenCurve:greenCG_rom	0 (0)	0 (0)	1536	1	0	0	0	0	0 (0)	0 (0)	0 (0)
romGreenCurve:greenCY_rom	0 (0)	0 (0)	1536	1	0	0	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult1	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult2	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult4	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
lpm_mult:Mult5	0 (0)	0 (0)	0	0	1	1	0	0	0 (0)	0 (0)	0 (0)
romPicture:picture_rom	0 (0)	0 (0)	4096	1	0	0	0	0	0 (0)	0 (0)	0 (0)
romPicture:picture_romY	0 (0)	0 (0)	4096	1	0	0	0	0	0 (0)	0 (0)	0 (0)
ShowValue:inst2	83 (83)	0 (0)	0	0	0	0	0	0	73 (73)	0 (0)	10 (10)
VGARegister:instreg	53 (53)	0 (0)	0	0	0	0	0	0	0 (0)	43 (43)	10 (10)
VGAsynchro:instsynchro	54 (54)	0 (0)	0	0	0	0	0	0	10 (10)	0 (0)	44 (44)

Hierarchy | Files | Design Units | IP Components | Revisions

Figure 19 - Used resources with 1-port ROMs a range after integer

The number of logic elements remains nearly the same, compare with Figure 16, but we consumed 4 memory blocks.

The objection of student Maxo Groucho: We can always create 2-port ROM a ignore 1-ports.

Answer: Yes and no. Our ModelSim Started Edition does not understand much to 2-port ROM a and better communicate with two instances of 1-port. It is better to debug a program with them a then change ROMs to 2-port. Moreover, to create a memory, it's a minute to work when you know how to proceed. For the first time, it can be longer, but only for the first time.

Besides, I do not recommend you to skip. It is better to build a static flag with 1-port RAM, then, write functions and use them for the conversion it to 4 parts. Without previous VHDL experience, it is not a good idea to perform long frog jumps, or even to try editing a sample example to adjust it to your assignment. You lose your way in a vast maze of wires, and you do not find its beginning, neither its end :-)

That's everything, your second task ends here. You can cheerfully jump to the third, final project.

~ o ~

Appendix 1: DisplayLogicCurve4parts.vhd

```
----------------------------------------------------------
-- CTU-FFE Prague, Dept. of Control Eng. [Richard Susta]
-- Published under GNU General Public License
----------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;  -- integers, signed and unsigned types

entity DisplayLogicCurve4parts is
  port(
    xyoffset          : in  std_logic_vector(10 DOWNTO 0); -- position of parts
    yrow, xcolumn     : in  unsigned(9 downto 0); -- row and  column number of VGA video
    VGA_CLK           : in  std_logic; -- clock signal for memories
    VGA_VS            : in  std_logic; -- synchronization of position values
    VGA_R, VGA_G, VGA_B : out std_logic_vector(9 downto 0) --  color information
  );
end;

architecture behavioral of DisplayLogicCurve4parts is
  -- Intensity of 10bit color in percent
  constant C100 : std_logic_vector(9 downto 0) := (others => '1');
  constant C75  : std_logic_vector(9 downto 0) := (9 => '1', 8 => '0', others => '1');
  constant C50  : std_logic_vector(9 downto 0) := (9 => '0', others => '1');
  constant C25  : std_logic_vector(9 downto 0) := (9 => '0', 8 => '0', others => '1');
  constant C0   : std_logic_vector(9 downto 0) := (others => '0');

  -- records are VHDL equivalents of structures
  type RGB_type is record
    R : std_logic_vector(9 downto 0);
    G : std_logic_vector(9 downto 0);
    B : std_logic_vector(9 downto 0);
  end record;
-- Used colors - we defined them by the way that allows their OR merging
  constant BLUE   : RGB_type := (C0, C0, C50);
  constant GREEN  : RGB_type := (C0, C50, C0);
  constant RED    : RGB_type := (C50, C0, C0);
  constant YELLOW : RGB_type := (C75, C75, C0);
  constant BLACK  : RGB_type := (C0, C0, C0);
  constant WHITE  : RGB_type := (C100, C100, C100);

  ------------------------------------------------------------------------

  constant YSIZE         : integer := 240;  -- height of flag
  constant XSIZE         : integer := 320;  -- width of flag
  constant XY_MAXSIZE : integer := XSIZE;
-- Note, VHDL2008 contains MAXIMUM function, but ModelSim requires VHDL1993
  subtype xyintd is integer range 0 to XY_MAXSIZE; -- relative coordinates

  constant EMBORGX       : integer := 220;  -- position of picture in flag
  constant EMBORGY       : integer := 32;
  constant MEMROWSIZE    : integer := 64;   -- parameters of memory organization
  constant MEMROWCOUNT   : integer := 64;
  constant MEM_END_ADDRESS : integer := 4095;


  --defintion of array
  subtype xyinteger is integer range -1024 to 1023;
  type xyPosition_type is array (0 to 7) of xyinteger;
  signal xyPosition : xyPosition_type;
```

```
  COMPONENT romPicture IS
    PORT(clock   : in  std_logic := '1';
         address : in  std_logic_vector(11 DOWNTO 0);
         q       : out std_logic_vector(0 DOWNTO 0)
        );
  END COMPONENT;
  COMPONENT romPicture2port IS
    PORT
    (
         address_a            : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
         address_b            : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
         clock                : IN STD_LOGIC  := '1';
         q_a                  : OUT STD_LOGIC_VECTOR (0 DOWNTO 0);
         q_b                  : OUT STD_LOGIC_VECTOR (0 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT BlueCurve is
    port(
         xcolumn : in  unsigned(9 downto 0);
         q       : out unsigned(6 downto 0)
    );
  end COMPONENT;

  -- Not used here, but it remains as an option
  COMPONENT romGreenCurve IS
    PORT(
         clock   : in  std_logic := '1';
         address : in  std_logic_vector(7 DOWNTO 0);
         q       : out std_logic_vector(5 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT romGreenCurve2ports IS
  PORT
  (
    clock   : IN std_logic  := '1';
    address_a                : IN std_logic_vector (7 DOWNTO 0);
    address_b                : IN std_logic_vector (7 DOWNTO 0);
    q_a     : OUT std_logic_vector (5 DOWNTO 0);
    q_b     : OUT std_logic_vector (5 DOWNTO 0)
  );
  END COMPONENT;

-- signals of ROM with picture
  signal picture_address_s : std_logic_vector(11 DOWNTO 0);
  signal picture_q_s       : std_logic;
  signal picture_addressY_s : std_logic_vector(11 DOWNTO 0);
  signal picture_qY_s      : std_logic;

-- signals of ROM with green curve
  signal greenCurve_addressG_s, greenCurve_addressY_s : std_logic_vector(7 DOWNTO 0);
  signal greenCurveG_q_s, greenCurveY_q_s      : std_logic_vector(5 DOWNTO 0);

-- array of BlueCurve
  signal blueCurve_addressB_s, blueCurve_addressY_s : unsigned(9 downto 0);
  signal blueCurveB_q_s, blueCurveY_q_s      : unsigned(6 downto 0);
```

```
begin

-------------------------------------------------------------------
-- !!! 2 instances of 1-port ROM are more suitable for ModelSim !!!
-------------------------------------------------------------------

-- Changes between 1-port ROM and 2-port can be performed by commenting and uncommenting of 1-port / 2-port instances

--picture_rom : romPicture
--   PORT MAP(clock   => VGA_CLK,
--            address => picture_address_s,
--            q(0)    => picture_q_s);
--
--picture_romY : romPicture
--   PORT MAP(clock   => VGA_CLK,
--            address => picture_addressY_s,
--            q(0)    => picture_qY_s);

  picture_rom : romPicture2port
    PORT MAP(clock   => VGA_CLK,
             address_a => picture_address_s,  -- picture
             address_b => picture_addressY_s,  -- yellow - cut out from yellow part
             q_a(0)    => picture_q_s,
              q_b(0)    => picture_qY_s);

--greenCG_rom : romGreenCurve
--   PORT MAP(clock   => VGA_CLK,
--            address => greenCurve_addressG_s,
--            q       => greenCurveG_q_s);
--
--greenCY_rom : romGreenCurve
--   PORT MAP(clock   => VGA_CLK,
--            address => greenCurve_addressY_s,
--            q       => greenCurveY_q_s);

  green2_rom : romGreenCurve2ports -- 2 port rom
    PORT MAP(clock   => VGA_CLK,
             address_a => greenCurve_addressG_s,
              address_b => greenCurve_addressY_s,
              q_a => greenCurveG_q_s,
              q_b => greenCurveY_q_s);
-------------------------------------------------------------------
  -- blue curve is created as logic function, so we need 2 instances
  blueCB_inst : BlueCurve
    PORT MAP(xcolumn => blueCurve_addressB_s,
             q       => blueCurveB_q_s);

  blueCY_inst : BlueCurve
    PORT MAP(xcolumn => blueCurve_addressY_s,
             q       => blueCurveY_q_s);
-------------------------------------------------------------------

  LSPflag : process(xcolumn, yrow, xyPosition, blueCurveB_q_s, blueCurveY_q_s,
                    greenCurveG_q_s, greenCurveY_q_s, picture_q_s)
    variable RGB : RGB_type; -- output colors
    variable x0d, y0d, x1d, y1d, x2d, y2d, x3d, y3d : xyintd;
    variable x0d_outside, y0d_outside, x1d_outside, y1d_outside: boolean;
    variable x2d_outside, y2d_outside: boolean; -- x3d_outside, y3d_outside are not necessary

    variable isPicture, isPictureY, isGreenInFlag, isYellowInFlag, isBlueInFlag : boolean;
    constant GREEN_LINE_ORG : integer := 243;
-------------------------------------------------------------------------------------------------
```

```
-- Remember, that VHDL functions and procedures are never called, but their whole bodies are inserted!!!
-- Thus, each their usage means replacing the call by the body with invocation parameters.
-- Note, C compilers sometimes perform the similar operation, known as inline expansion, or inlining.
-----------------------------------------------------------------------------------------------
    function InCircle(x,y:xyintd) return boolean is
    begin
        return x*x + (y - YSIZE)*(y - YSIZE) < YSIZE*YSIZE/16;
    end function;

    function InGreen(x,y:xyintd; xoffset:std_logic_vector) return boolean is
    begin
        return x < GREEN_LINE_ORG - y - to_integer(unsigned(xoffset));
    end function;

    function InBlue(x,y:xyintd;yoffset:unsigned) return boolean is
    begin
        return 2*y >= 2*YSIZE - x + to_integer(yoffset);
    end function;

    -- Return true if x and y are inside of "1/0" picture in flag
    function InPicture(x,y:xyintd) return boolean is
    begin
      return x >= EMBORGX and x < EMBORGX + MEMROWSIZE and y >= EMBORGY and y < EMBORGY +
MEMROWCOUNT;
    end function;

    function CalculatePictureAddress(x,y:xyintd) return std_logic_vector is
    variable address:std_logic_vector(11 DOWNTO 0);
    begin
      address := std_logic_vector(to_unsigned((y - EMBORGY)*MEMROWSIZE + (x - EMBORGX), address'LENGTH));
        return address;
    end function;

    -- we convert xy unsigned input to integer, then we move it to center of screen and apply offset
    -- The result z is always limited to interval <0,limit>
    procedure diff_limit(
            xy: in unsigned(9 downto 0); --input value
            offset: in xyinteger; --offset of position
              limit: in natural;  -- integer>=0
              z: out xyintd; -- result
              outside:out boolean -- xy is outside of <0,limit>
              ) is
    variable diff:integer;
    begin
    -- we convert input to integer, move it to the center of screen (-limit/2),
    -- and then, we subtract position offset
          diff:=to_integer(xy)-limit/2-offset;
    -- Note, the subtraction of limit/2 will be repeated in each insertion of the procedure,
    -- but Quartus will optimize code and join -limit/2 subtraction to a shared operation, see, Locate in RTL viewer.
      if diff>=limit or diff<0 then z:=limit; outside:=true;
        else z:=xyintd(diff); outside:=false;
        end if;
    end procedure diff_limit;
```

```
  -- overloading of or operator
  function "or"(color1, color2 : RGB_type) return RGB_type is
      variable result : RGB_type;
  begin
    if color2/=WHITE then
        result.R := color1.R or color2.R;
        result.G := color1.G or color2.G;
        result.B := color1.B or color2.B;
      else --white color is not in our flag, but this code can be inspiration for you
        result.R := color1.R xor color2.R;
        result.G := color1.G xor color2.G;
        result.B := color1.B xor color2.B;
      end if;
      return result;
  end function;

begin

  diff_limit(xcolumn, xyPosition(0), XSIZE, x0d, x0d_outside);
  diff_limit(xcolumn, xyPosition(1), XSIZE, x1d, x1d_outside);
  diff_limit(xcolumn, xyPosition(2), XSIZE, x2d, x2d_outside);
  diff_limit(xcolumn, xyPosition(3), XSIZE, x3d);       -- out paramaters can be omitted

  diff_limit(yrow, xyPosition(4), YSIZE, y0d, y0d_outside);
  diff_limit(yrow, xyPosition(5), YSIZE, y1d, y1d_outside);
  diff_limit(yrow, xyPosition(6), YSIZE, y2d, y2d_outside);
  diff_limit(yrow, xyPosition(7), YSIZE, y3d); -- out paramaters can be omitted

  isGreenInFlag  := not x0d_outside and  not y0d_outside;
  isYellowInFlag := not x1d_outside and  not y1d_outside;
  isBlueInFlag   := not x2d_outside and  not y2d_outside;
  isPicture := InPicture(x3d,y3d); -- drawing picture
  isPictureY:= InPicture(x1d,y1d); -- cut out of picture

  RGB := BLACK;  -- Very important line, to prevent latches, we must initialize RGB color to the known value

  -- Green part has x0d, y0d
  if isGreenInFlag and not InCircle(x0d,y0d) and InGreen(x0d,y0d,greenCurveG_q_s)
  then RGB := RGB or GREEN;
  end if;

  -- Yellow part has x1d, y1d
  if isYellowInFlag and (not isPictureY or picture_qY_s='0') and
    (InCircle(x1d,y1d) or (not InGreen(x1d,y1d,greenCurveY_q_s) and not InBlue(x1d,y1d, blueCurveY_q_s)))
   then RGB := RGB or YELLOW;
   end if;

  -- Blue part has x2d, y2d
  if isBlueInFlag and not InCircle(x2d,y2d) and InBlue(x2d,y2d,blueCurveB_q_s)
  then RGB := RGB or BLUE;
  end if;

  if isPicture and picture_q_s='1' then
      RGB := RGB or RED;
  end if;
```

```
    -- Green curve memory utilizes 8 lower bits, thus, its address never overflows
    -- To prevent error messages in ModelSim, we converted to 10 bit and select lower 8 bits
    if isGreenInFlag then
      greenCurve_addressG_s <=  std_logic_vector(to_unsigned(y0d, greenCurve_addressG_s'LENGTH));
    else
     greenCurve_addressG_s <= (others=>'0');
    end if;

   if isYellowInFlag then
       blueCurve_addressY_s  <= to_unsigned(x1d, blueCurve_addressY_s'LENGTH);
       greenCurve_addressY_s <=  std_logic_vector(to_unsigned(y1d, greenCurve_addressY_s'LENGTH));
    else
       blueCurve_addressY_s <= (others=>'0');
       greenCurve_addressY_s <= (others=>'0');
    end if;


    if isBlueInFlag then
      blueCurve_addressB_s  <= to_unsigned(x2d, blueCurve_addressB_s'LENGTH);
    else
      blueCurve_addressB_s <= (others=>'0');
    end if;

    if isPicture then           picture_address_s <= CalculatePictureAddress(x3d,y3d);
    else
        picture_address_s <= (others => '0');
    end if;

    -- cut out picture from yellow part
    if isPictureY then          picture_addressY_s <= CalculatePictureAddress(x1d,y1d);
    else
        picture_addressY_s <= (others => '0');
    end if;

    -- Copy results in RGB to outputs of entity
    VGA_R <= RGB.R; VGA_G <= RGB.G; VGA_B <= RGB.B;
    ---------------------------------------------------------------------------
  end process;


  vs_copy : process(VGA_VS)
  variable dify, minusdify, difx, minusdifx : xyinteger;
  begin
    if rising_edge(VGA_VS) then -- Copy data to assure their stability in picture
       dify := to_integer(signed(xyoffset)); minusdify:=-dify;
       -- x size is 5/4 of y
         difx := dify+dify/4; minusdifx :=-difx;
         xyPosition(0) <= minusdifx; xyPosition(4) <= minusdify;  -- move green part to the left upper corner
         xyPosition(1) <= minusdifx; xyPosition(5) <= dify;  -- move yellow part to the left bottom corner
         xyPosition(2) <= difx; xyPosition(6) <= dify;  -- move blue part to the right bottom corner
         xyPosition(3) <= difx; xyPosition(7) <= minusdify;  -- move 1/0 picture to the right upper corner
    end if;
  end process;

end architecture behavioral;
```

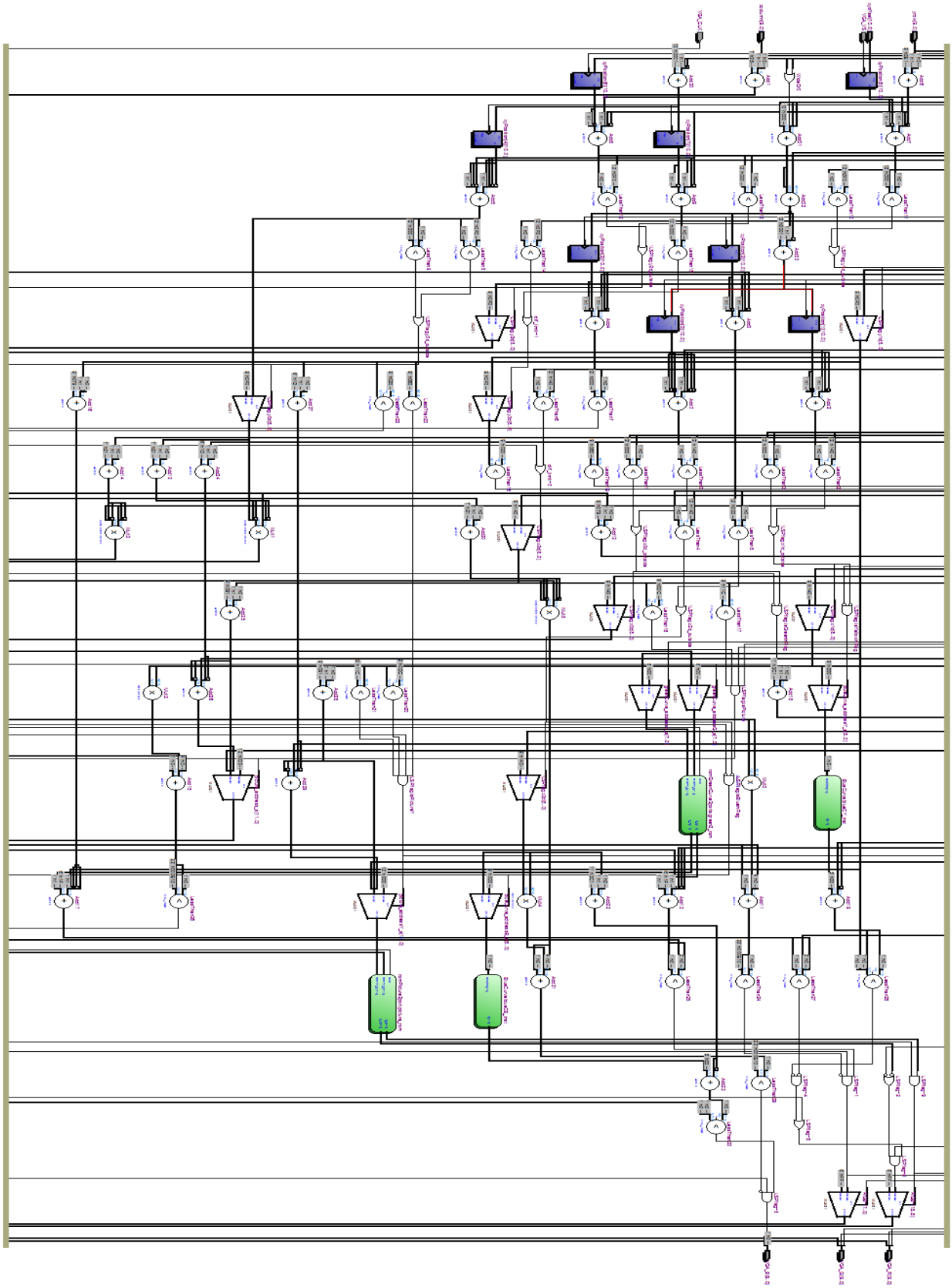


Figure 20 - Result of the copilation of DisplayLogicCurve4parts - 2-port ROM

Appendix 2: ShowValue.vhd

```
----------------------------------------------------------------
-- CTU-FFE Prague, Dept. of Control Eng. [Richard Susta]
-- Published under GNU General Public License
----------------------------------------------------------------
LIBRARY ieee; USE ieee.std_logic_1164.all; USE ieee.numeric_std.all;

-- For simplicity, the switches SW(9 downto 0) specify absolute value of a number,
-- SW(10) is its sign, i.e., the straight binary code is used.
-- This input value is converted to two's complement and sent to out data output.
ENTITY ShowValue IS
    PORT
    (
        SW : in std_logic_vector(10 downto 0);
        HEX0, HEX1, HEX2, HEX3, HEX4 : out std_logic_vector(6 downto 0);
        data : out std_logic_vector(10 downto 0)
    );
END ShowValue;

architecture rtl of ShowValue is
begin
 show : process(SW)
     type hexarray_t is array(0 to 15) of std_logic_vector(HEX0'range);
     constant hexarray : hexarray_t :=
                                 ("1000000","1111001","0100100","0110000",
                                  "0011001","0010010","0000010","1111000",
                                  "0000000","0010000","0001000","0000011",
                                  "1000110","0100001","0000110","0001110");
   -- convert to 7 segment display
     function to7segment(x:std_logic_vector) return std_logic_vector is
     variable number:integer;
     begin  number:=to_integer(unsigned(x));
             if number>=0 and number<=15 then return hexarray(number);
             else return "1111111";
             end if;
     end function;

    subtype bcd_digit_type is std_logic_vector(3 downto 0);
    function adjust(x:bcd_digit_type) return bcd_digit_type is
    begin
      if unsigned(x)>"0100" then return bcd_digit_type(unsigned(x)+3);
        else return x;
        end if;
   end function;
```

```
  -- convert integer from 0 to 9999 to 4 BCD digits
   function binary2bcd(number : integer) return std_logic_vector is
  variable bcd : std_logic_vector(15 downto 0) ;
  variable hex_src : std_logic_vector (13 downto 0) ;
  begin
     hex_src := std_logic_vector(to_unsigned(number,hex_src'LENGTH));
     bcd     := (others => '0') ;
     for i in 0 to hex_src'LENGTH-1 loop
        bcd(3 downto 0):=adjust(bcd(3 downto 0));      bcd(7 downto 4):=adjust(bcd(7 downto 4));
        bcd(11 downto 8):=adjust(bcd(11 downto 8));  bcd(15 downto 12):=adjust(bcd(15 downto 12));
        bcd := bcd(bcd'LEFT-1 downto 0) & hex_src(hex_src'LEFT) ; -- shift bcd + 1 new entry
        hex_src := hex_src(hex_src'LEFT-1 downto 0) & '0' ; -- shift src + pad with 0
     end loop ;
     return std_logic_vector(bcd);
  end function;

   -- we overload the previous function. The overloading must occur always after an existing function.
 function binary2bcd(x : std_logic_vector) return std_logic_vector is
   begin
      return binary2bcd(to_integer(unsigned(x)));
   end function;

variable bcd_number : std_logic_vector(15 downto 0);

 begin
  -- we used a straight binary number with 10 bit specifying its sign
 bcd_number:=binary2bcd(to_integer(unsigned(SW(9 downto 0)))); -- straight binary magnitude
   if(SW(10)='0') then -- sign bit
       data<=SW(10 downto 0);
       HEX4<=(others=>'1'); -- blank
    else
        data<= std_logic_vector(-signed('0' & SW(9 downto 0))); -- convert straight to two's complement
        HEX4<="0111111";  -- minus sign
   end if;

   HEX3<=to7segment(bcd_number(15 downto 12));
   HEX2<=to7segment(bcd_number(11 downto 8));
   HEX1<=to7segment(bcd_number(7 downto 4));
   HEX0<=to7segment(bcd_number(3 downto 0));

 end process;
end rtl;
```

Appendix 3: ModelSim for DisplayLogicCurve4parts

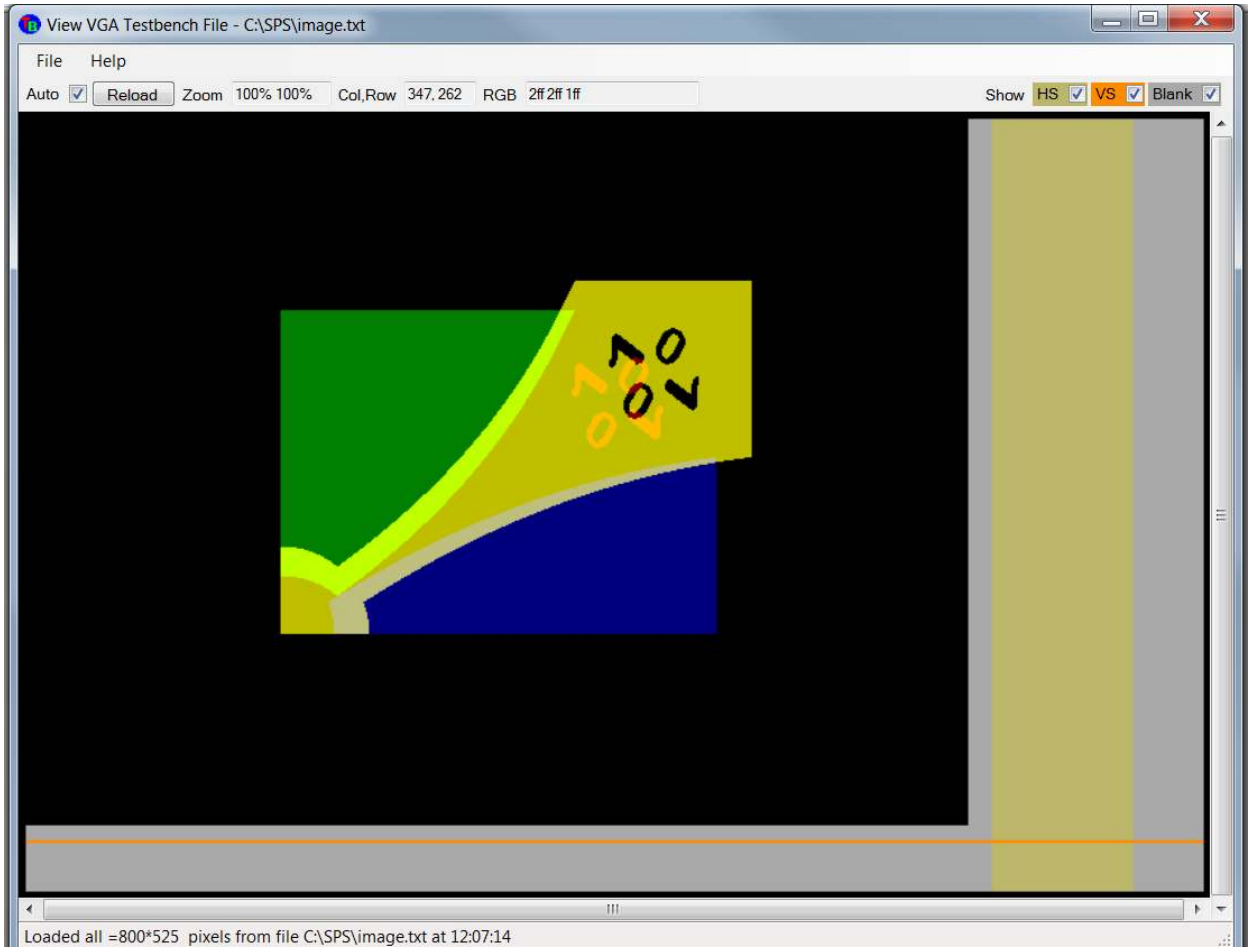


Figure 21 - Visualization of ModelSim for offsetí=-64