02/11/2021- 09/11/2021

Lab#3

– Digital Electronics –

On Chip Memory and IPs

Group members :

Wojciech Paluszkiewicz, Muhammad Arshad,

Martin Callegarin-Demangeat, Nicolas Masson

This third lab is about on-chip memories and IPs. An on-chip memory, conversely to off-chip memory, means that the processor chip contains a reserved storage for most-used data so that latency is decreased.

I. Design of a VGA Controller

To operate a communication between our code, the board and the VGA interface we need several clocks. Among them, there is the Pixel clock, defined by the capability of a graphic card to process pixels per second. By checking on the user manual, we have, for a 640x480 resolution and 60Hz VGA controller, a 25MHz pixel clock frequency.

In order to represent any figure such as 4 colored squares on a VGA display, we had to configure the inside of the FPGA so that it could work with it is integrated Digital-Analog Converter in order to send a couple of acceptable signals on the output of the VGA connector. These signals sent will be a mix of analog and digital as we'll see it.

First, let's explain the shape of the horizontal and vertical synchronization signals. Based on figure 1.

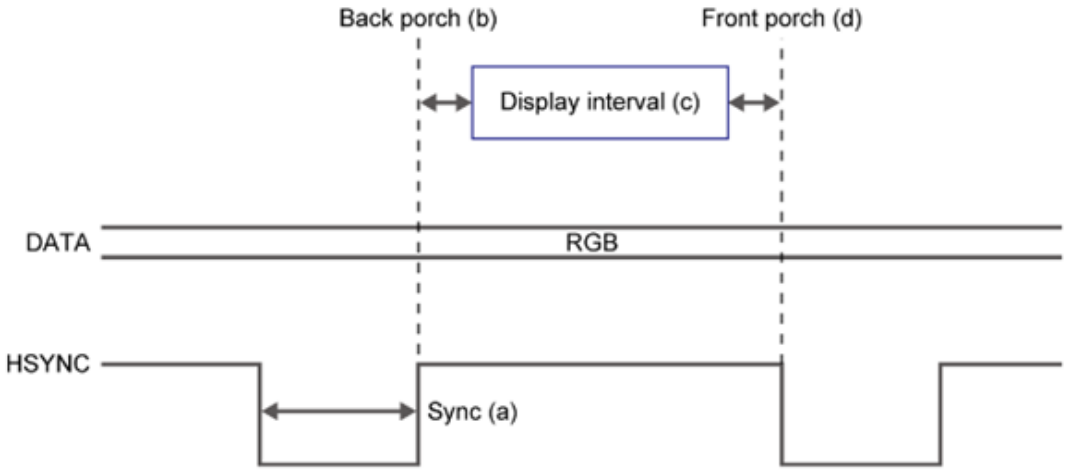


Figure 1 : Horizontal synchronization signal for a VGA display.

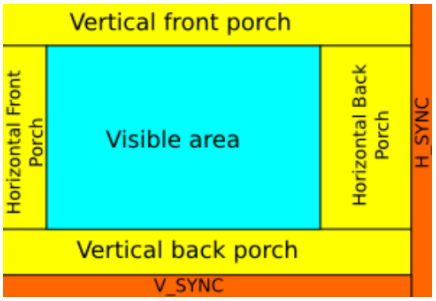


Figure 2 : Design of a classic monitor

Looking at the digital signal HSYNC, we can see it takes different values for different time moments. There are 4 lapses of time to describe. The front porch, the display interval, the back porch and the sync pulse. It's only during this last lapse of time that the HSYNC signal must be driven to 0. By checking on the device manual or on the internet we found approximation of the time values for each lapse.

Why ? Because in order to know when to display a colored set of RGB pixel at different places of the visible screen, the displayer needs to know when to pass from one line to another and from one frame to the next one. This is done by sending him a logical 0 on the HSYNC and VSYNC respectively to change line and frame. After this logical 0 of a very precise timing (3.8us for the hsync of our screen configuration) the displayer will keep on displaying the analog RGB signal he receive but on another line.

However, we must only send colored display during the display interval. During the sync, bp and fp, it is imperative to send a "logical" 0 to the analog signal. In order to not have any conflict of display during the changing of position of the displayer.

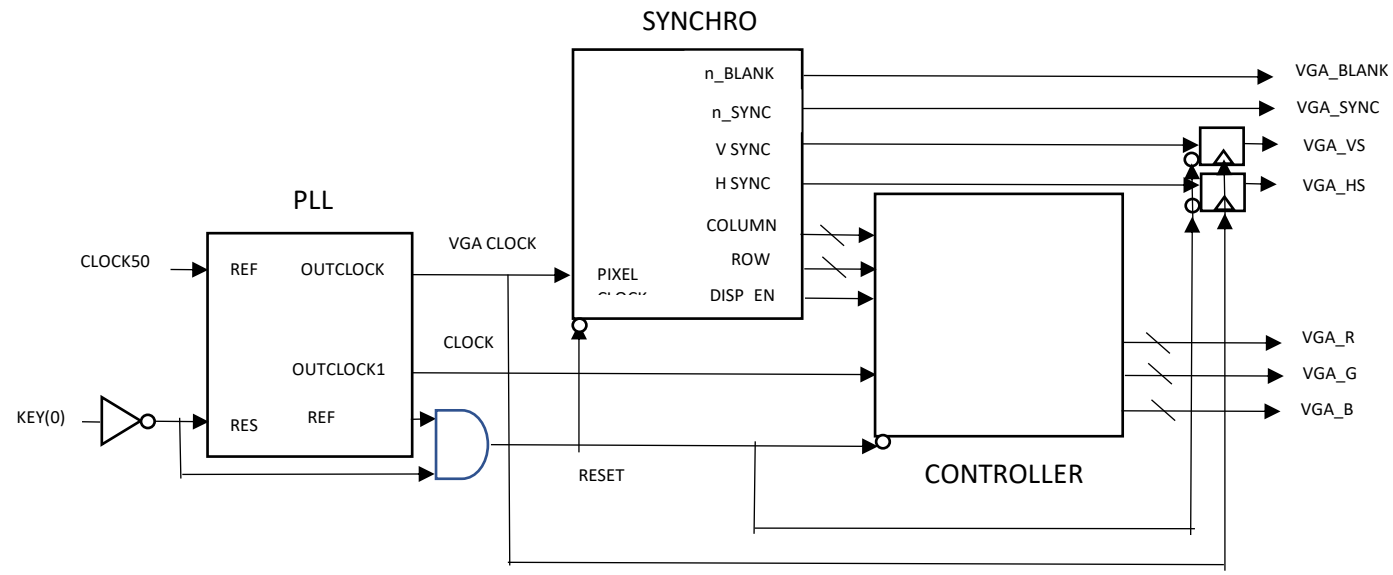
But before that, there is a FPGA to program:

In it we have for the first task, two main components. One is the synchronizer and the other is the controller. The synchronizer will take in input the pixel clock at 25 MHz and an asynchronous reset.

Managed by the input clock, the synchronizer big role will be expliciting to the electrical circuit where we are exactly on the screen. The visible one as well as the hidden front porches, back porches and synchronization pulse that are more like virtual places of the screen. he will therefore output the hsync, vsync, column and row placement of the displayer as an integer ranging from 0 to the maximum length of the horizontal screen for "column" and vertical screen for "row". Just like on a matrix, we will know the coordinate of the displayer. It is then easy to know when we are on the display area and when we aren't, resulting in an authorization to enable the colors or not. This is why we output a std_logic called disp_en in the code shown below on figure 2.1

To implement the code on the DE1-SoC Board we need to use a PLL. A PLL (Phase-Locked Loop) is a component already present board and generates an output signal whose phase is related to the phase of an input signal so that we don't have to use logic components to generate the pixel clock from the CLOCK_50. Another major interest of using PLL is that it generates a clock with a frequency higher or lower and with a specific phase regarding the reference clock. This last point is very practical concerning the sending of all the VGA outputs (VGA_xxxx in block scheme) at the same time although their previous signals come out from different components, using different clocks. We'll come back on this advantage later on during the 2nd task explanations

Now that we've described every components here is the block scheme to have a general view :




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

ENTITY synchro IS
  GENERIC(
    h_pulse   : INTEGER := 96;    --horiztonal sync pulse width in pixels
    h_bp      : INTEGER := 48;    --horiztonal back porch width in pixels
    h_pixels  : INTEGER := 640;   --horiztonal display width in pixels
    h_fp      : INTEGER := 16;    --horiztonal front porch width in pixels
    h_pol     : STD_LOGIC := '0';  --horizontal sync pulse polarity (1 = positive, 0 = negative)
    v_pulse   : INTEGER := 2;      --vertical sync pulse width in rows
    v_bp      : INTEGER := 33;     --vertical back porch width in rows
    v_pixels  : INTEGER := 480;   --vertical display width in rows
    v_fp      : INTEGER := 10;     --vertical front porch width in rows
    v_pol     : STD_LOGIC := '0'); --vertical sync pulse polarity (1 = positive, 0 = negative)//was at 1 by default
  PORT(
    pixel_clk : IN   STD_LOGIC;  --pixel clock at frequency of VGA mode being used is gonna be 25MHz
    reset_n   : IN   STD_LOGIC;  --active low asycnchronous reset
    h_sync    : OUT  STD_LOGIC;  --horiztonal sync pulse
    v_sync    : OUT  STD_LOGIC;  --vertical sync pulse
    disp_ena  : OUT  STD_LOGIC;  --display enable ('1' = display time, '0' = blanking time)
    column    : OUT  INTEGER;    --horizontal pixel coordinate// hcount and
    row       : OUT  INTEGER;    --vertical pixel coordinate//v count
    n_blank   : OUT  STD_LOGIC;  --direct blacking output to DAC//things that needs to be at 1 directly
    n_sync    : OUT  STD_LOGIC   --sync-on-green output to DAC
    );
END synchro;
```

Figure 2.1 : main ports of synchronizer component.

The basic functioning of the synchro component is having two counters, hcount and vcount. Hcount will increment until it reaches the end of the horizontal screen and when doing so, will increment vcount. If vcount is at the end of a frame it will reset both the counter to 0 and the process will start again. The rest of the code is based on the value of those two counters. Either their values are saying we are in the visible screen or not, either we are in the sync pulse or not will impact the output that we've talked about previously. (hsync,vsync,disp_en). The n_blank and n_sync signals are irrelevant but needed to be driven anyway for the integrated DAC. If n_blank is 0, all pixel inputs of R, G and B are ignored. We put them at 1 directly after the beginning of architecture.

```
ARCHITECTURE behavior OF synchro IS
  CONSTANT h_period : INTEGER := h_pulse + h_bp + h_pixels + h_fp; --total number of pixel in a row
  CONSTANT v_period : INTEGER := v_pulse + v_bp + v_pixels + v_fp; --total number of rows in a frame
BEGIN

  n_blank <= '1';  --don't know what they do. only know they must be to 1
  n_sync <= '1';

  PROCESS(pixel_clk, reset_n)
    VARIABLE h_count : INTEGER RANGE 0 TO h_period - 1 := 0;  --horizontal counter (counts the columns)
    VARIABLE v_count : INTEGER RANGE 0 TO v_period - 1 := 0;  --vertical counter (counts the rows)

  BEGIN

    IF(reset_n = '0') THEN    --reset asserted
      h_count := 0;            --reset horizontal counter
      v_count := 0;            --reset vertical counter
      h_sync <= NOT h_pol;     --deassert horizontal sync
      v_sync <= NOT v_pol;     --deassert vertical sync
      disp_ena <= '0';         --disable display
      column <= 0;             --reset column pixel coordinate
      row <= 0;                --reset row pixel coordinate

    ELSIF(pixel_clk'EVENT AND pixel_clk = '1') THEN

      --counters
      IF(h_count < h_period - 1) THEN    --horizontal counter (pixels)
        h_count := h_count + 1;
      ELSE
        h_count := 0;
        IF(v_count < v_period - 1) THEN  --veritcal counter (rows)
          v_count := v_count + 1;
        ELSE
          v_count := 0;
        END IF;
      END IF;

      --horizontal sync signal
      IF(h_count < h_pixels + h_fp OR h_count >= h_pixels + h_fp + h_pulse) THEN--we are out of the sync zone
        h_sync <= NOT h_pol;    --deassert horiztonal sync pulse// originaly at '0', h_pol will give 1 to h_sync. meaning we are not changing row
      ELSE
        h_sync <= h_pol;        --assert horiztonal sync pulse
      END IF;

      --vertical sync signal
      IF(v_count < v_pixels + v_fp OR v_count >= v_pixels + v_fp + v_pulse) THEN--same behaviour than hsync, weird that v_pol was at 1 by default
        v_sync <= NOT v_pol;    --deassert vertical sync pulse//mettre v_sync à 1
      ELSE
        v_sync <= v_pol;        --assert vertical sync pulse
      END IF;

      --set pixel coordinates
```

You can see the architecture of synchro figure 2.2

```
      --set pixel coordinates
      IF(h_count < h_pixels) THEN  --horiztonal display time
        column <= h_count;          --set horiztonal pixel coordinate
      END IF;
      IF(v_count < v_pixels) THEN  --vertical display time
        row <= v_count;             --set vertical pixel coordinate
      END IF;

      --set display enable output
      IF(h_count < h_pixels AND v_count < v_pixels) THEN  --display time
        disp_ena <= '1';                                   --enable display
      ELSE                                                --blanking time
        disp_ena <= '0';                                   --disable display
      END IF;

    END IF;
  END PROCESS;

END behavior;
```

figure 2.2 architecture of the synchronizer component

Disclaimer information. The main part of this code has been released the 05/10/2013 by a certain Scott Larson. After trying by ourselves to get a code running but after being stopped by non-understandable syntaxes errors, we took, modified, and then implemented this code as a component in our main code. As the theory behind it is understood and as we've tried writing it by ourselves in a first time, we don't really think this step is jeopardizing the project in any way. All the other line are self-written

The synchronizer component will manage the next one called controller. This one is in charge of sending the RGB signals in respect to where the displayer is on the screen. Let see on figure 3 the ports of this component.

```
 4
 5  entity control is
 6  port(
 7      enable, rst, clock          : in std_logic;
 8      column                      : in integer;--the x axis
 9      row                         : in integer;--the y axis
10      vgaR, vgaG, vgaB            : out std_logic_vector(7 downto 0));
11  end control;
12
13  architecture beh of control is
14
15  constant hsize : integer := 640;
16  constant vsize : integer := 480;
17
18  begin--architecture
19
20  --checking reset
21  process(rst, clock)
22  begin--process
23
24  if(rst = '0' or enable = '0') then--we either can't write because of reset
25      vgaR<=(others=>'0');-- or because the syncrhoniser tells us no
26      vgaG<=(others=>'0');
27      vgaB<=(others=>'0');
28      else
29          if(clock'event and clock='1') then--rising edge
30              if(column < hsize/2 and row < vsize/2) then --top left
31                  vgaR<=(others=>'1');
32                  vgaG<=(others=>'0');
33                  vgaB<=(others=>'0');
34              end if;
35              if(column > hsize/2 and row < vsize/2) then --top right
36                  vgaR<=(others=>'0');
37                  vgaG<=(others=>'1');
38                  vgaB<=(others=>'0');
39              end if;
40              if(column < hsize/2 and row > vsize/2) then --bottom left
41                  vgaR<=(others=>'0');
42                  vgaG<=(others=>'0');
43                  vgaB<=(others=>'1');
44              end if;
45              if(column > hsize/2 and row > vsize/2) then --bottom left
46                  vgaR<=(others=>'1');
47                  vgaG<=(others=>'1');
48                  vgaB<=(others=>'1');
49              end if;--colors
50          end if;--clock
51      end if;--reset
52  end process;
53  end beh;
```

Figure 3 code of the controller

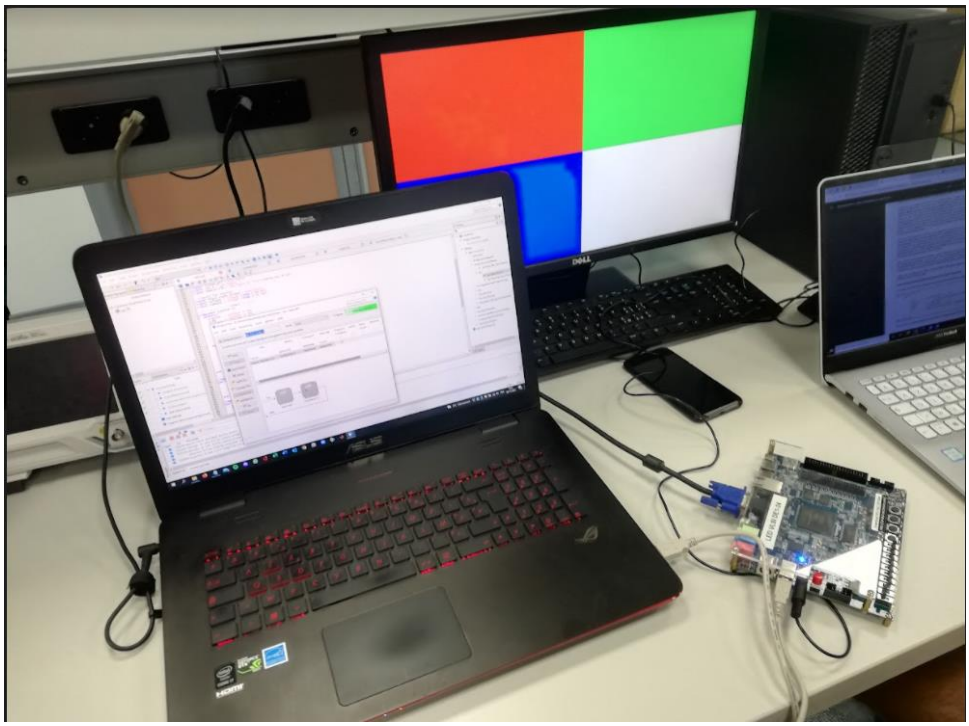
As shown, the input column and row are the same output from the synchronizer. The enable is the disp_en. The controller will output the RGB 7bits words used for the DAC and he will do so according to the position on the screen that the synchronizer informs him.

Knowing that if the disp_en is at 0 meaning that we aren't on the display interval, we have to send a logical 0 to every RGB signals. Otherwise, rest of the code is simply acknowledging where we are on the screen and sending either red, green, blue or white pixel to form the square. We could also easily send the French flag but we don't want to get in bad terms with our trans-alpine neighbors.

As it is explained in the lab subject, our FPGA will produce outputs on single bits (vsync,hsync) or on bits word like the R G B combination, it is the role of an included DAConverter on the DE1 chip to always compute the R G B signals in order to output the analog signal on the VGA port while the Hsync and Vsync can directly go on the VGA port. That is why a D flip flop is required in order to have a delay of 1 clock signal. This pipeline task is done figure 4 in the main code.

```
137     ---------D flip flop -----------------
138     process(vgaclk, rst_n)
139     begin--process
140        if(rst_n = '0') then
141           VGA_HS <= '0';
142           VGA_VS <= '0';
143        else
144           if(vgaclk'event and vgaclk ='1') then
145              VGA_HS <= hsync;--hsync and vsync being std_logic signals created for the main
146              VGA_VS <= vsync;--they link the synchro and the flipflop together.
147           end if;--they also send the VGA_HS/VS to the VGA port
148        end if;
149     end process;
150
```

Figure 4: delaying the Vsync and Hsync output of the FPGA before the VGA port



Sweet results.

II. Display a picture on the monitor

Now that we've understood the basics of VGA displaying, we can try and display picture from a memory file.

For this we will need to add multiple components and signals on the code.

First, the three memory that will provide the red, green, and blue combination of 8 bits to send to the controller. We create and fill them according to the instruction of the lab pdf. Then, we need to extract the data from all the occupied addresses. Going from the first and then jump to the next one

That is why we need a new counter ranging from 0 to 25343. This number being the total number of addresses used by a 176*144 pixels picture in a ROM. (-1 since we have the address 0)

We could have included this counter in the synchronizer component, but then we would have to add the clock_0 of the pll in the synchronizer and so in the port map. It would not have required a very long process, but we did it our way and created a new component. (As Mr. Martina says, there is a lot of way to resolve a problem.) Let's have a look at his ports. See figure 5

```
---------counter for memory adresses----
component counter is
port(
clock : in std_logic;
reset : in std_logic;
enable : in std_logic;
counter : out integer
);
end component counter;
```

Figure 5: the counter port.

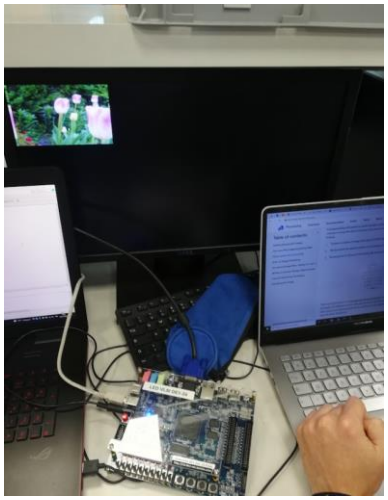
He receives the clock_0 of the pll as input. Why not the VGA_CLK you ask? Glad you asked, this is related to the convenience of the controller extraction of the RGB data presented in the PLL part of the lab. The clock of the pll being phase shifted by pi, the first one will change the RGB combination (counter with memory will do that and present the memory data to the input of controller) and then the second clock will arrive a bit later to make the controller extract the data given by the ROM.

The counter as a reset to go back at the first address. And an enable. This enable as input in governed by the synchronizer output "display_enable" Since we know thank to this bit if we are on the visible area of the screen. if we are not, it is imperative to stop counting and reading the ROM and wait for the synchronizer to tell us; "We are back in the display zone, send me ROM DATA"

This will be done in the controller component. In the figure 6 we have the simple lines of code responsible of sending these datas :

```
27  process(rst, clock)
28  begin--process
29
30  if(rst = '0' or enable = '0') then--we either can't write because of reset or because the syncrhoniser tells us no
31      vgaR<=(others=>'0');
32      vgaG<=(others=>'0');
33      vgaB<=(others=>'0');
34      else
35          if(clock'event and clock='1') then--rising edge
36              if(column < 176 and row < 144) then --in the visible screen
37                  vgaR <= memR;
38                  vgaG <= memG;
39                  vgaB <= memB;
40                  cntenabl <='1';
41              else--we are not in the right rectangle
42                  vgaR<=(others=>'0');
43                  vgaG<=(others=>'0');
44                  vgaB<=(others=>'0');
45                  cntenabl <='0';
46              end if;--colors
47          end if;--clock
48      end if;--reset
49  end process;
50  end beh;
```

Figure 6: controlling wether to output black pixels or the ROM datas



III. filtering the image

Even though we didn't find the time to achieve the final result awaited, we had a theory in mind to make it achievable. Let us present you the idea. And what a better and easier way to start than with the presentation of the new component's code. Let's have a look at it figure 7.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity offsetcount is
6  port(
7     clk, rst : in std_logic;
8     plus,minus : in std_logic;
9     offset : out integer);
10 end offsetcount;
11
12 architecture beh of offsetcount is
13
14 constant countrange : integer := 25344;
15
16 begin
17
18 process (rst,clk)
19 variable cat : std_logic_vector (1 downto 0);
20 variable off : integer range 0 to countrange-1 :=0;
21
22 begin
23 cat := plus & minus;
24 if(rst ='0') then
25    off := 0;
26 else
27    if(clk'event and clk='1') then
28       case cat is
29          when "01" => off := off + 1;
30          when "10" => off := off - 1;
31          when others => off := off;
32       end case;
33       offset <= off;
34    end if;
35 end if;
36 end process;
37
38 end beh;
```

Figure 7: the offset creator.

For a start, it has the traditional clock and asynchronous reset. Yes but which clock? We started with the clock_0 of the pll but we'll come back on this choice, explain why it's a bad one and present you the idea of a better solution.

The plus and minus output would have been respectively key(1) and key(2) of the DE1 board.

And of course, it would output an integer used as our offset afterward. The code is pretty simple. As we cannot offset the image more than the total number of pixel it has, the offset should not exceed the memory usage of the picture. Otherwise we would have to program a modulo calculator but the standar range of the integer signal will do that for us. Line 23 we choose to concatenate the two input to make our live easier with a XOR gate. It is either adding to the offset or staying the same if both the keys are pushed or aren't.

Now a small problem in theory is that by using a 25MHz clock from the pll, by even sligthly pushing the buttons, the offset would increases itself way too fast. We can simply create another pll with a

third clock at a more human scale level of comprehension, making the offset visible. 10Hz would have been nice.

The afterpart was about sending this offset integer signal to the counter. Ordering it to start counting from the memory not from 0 but from 0 + offset. In theory the image would be shifted diagonaly. However, we couldn’t manage to have showable results.

Conclusion

With the two first exercises we manage to operate different types of clocks and phasing them. We faced several problems such as overflow, timing problems or even understanding how monitors are configurated with the different parts (backporch, frontporch, hsync...).

We managed to create a dialogue between ROMs and a sequential circuit in order to display more complex shapes than craftable squares.