Educational demonstrator on Basys3

Project Goal

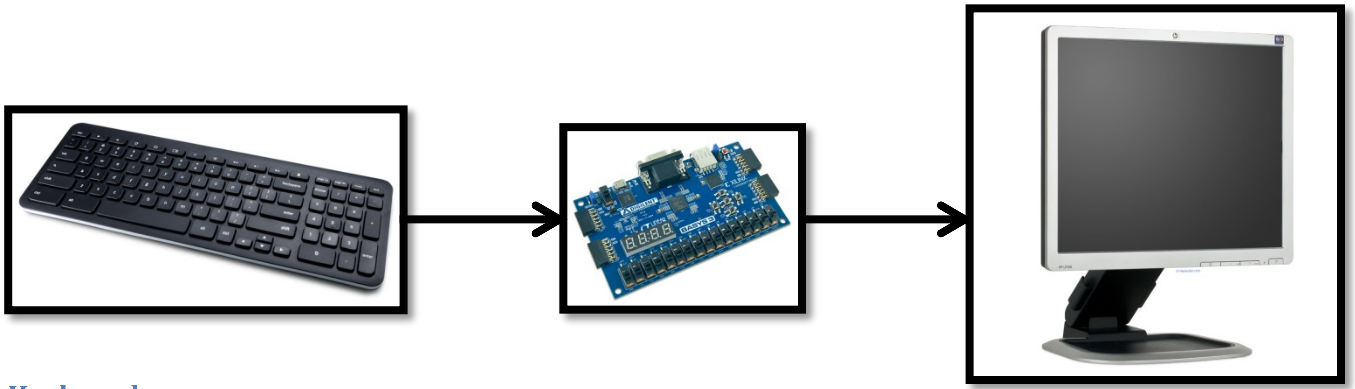
The aim of the project is to have an educational demonstrator, which highlights the different options of the Basys3 board and to address internal dedicated hardware of the Xilinx Artix 7 FPGA, with the use of VHDL and the IP-generator of the Vivado software from Xilinx.

The demonstrator uses the Basys3 board, the VGA output of the board and a USB keyboard with the USB connector of the board. The goal is to create a background with the VGA and have two separate figures, so-called sprites, on the screen, which can be moved separately with the use of a USB keyboard keys.

The project uses a PS2 interface between the FPGA and the PIC microcontroller, which converts the USB-keyboard output to PS2 output. It uses the Multimedia Clock Manager (MMCM) to build a steady 25MHz clock, for the VGA synchronization, it uses Block RAM to store the two sprites in and it uses VGA to address the VGA-screen.

External Block Diagram

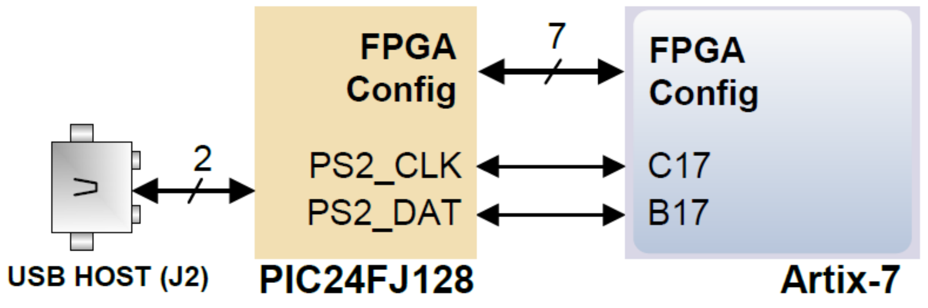
Externally we need a VGA screen, a standard USB keyboard and the Basys3 board.



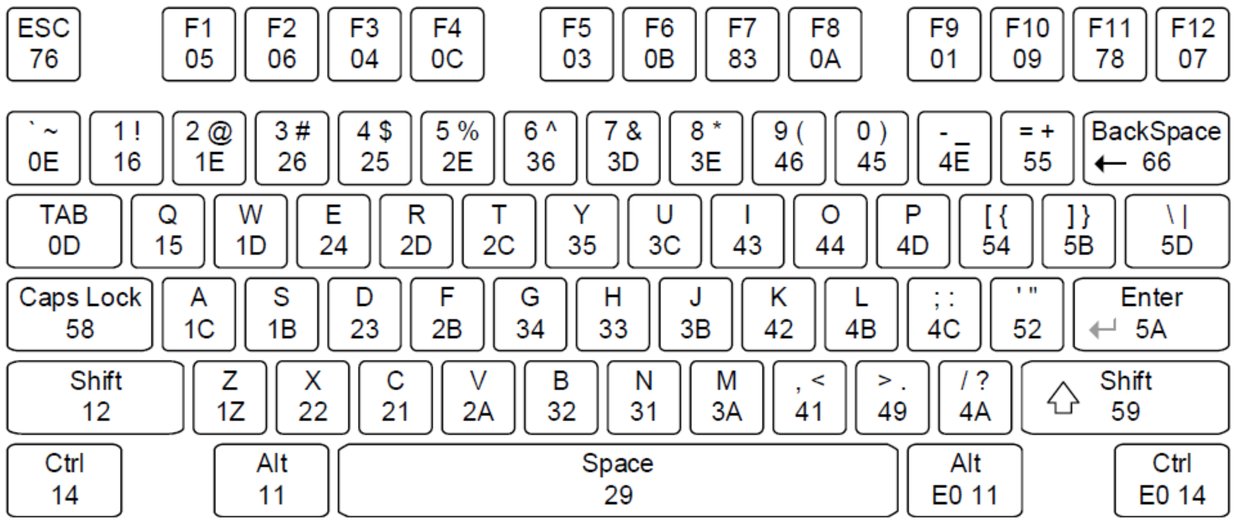
Keyboard

- The Auxiliary Function microcontroller on the Basys3 board (Microchip PIC24FJ128) provides the Basys3 with USB HID host capability

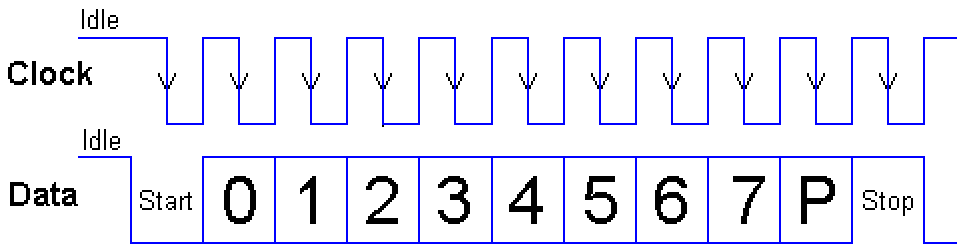
- The PIC24 drives two signals which are used to implement a standard PS/2 interface for communication with a mouse or keyboard.



- The keyboard sends 8-bit scan codes out, least significant bit first, with an extra start bit (‘0’), parity bit and a stop bit (‘1’).

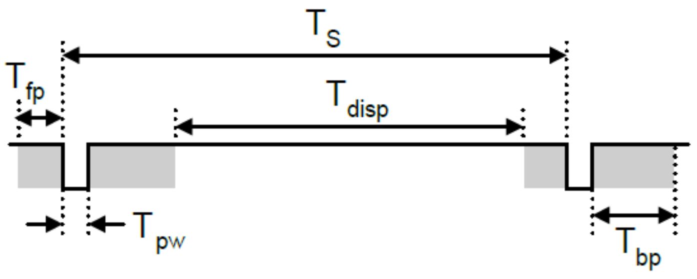


- The data PS2 data (PS2_DAT) from the PIC microcontroller is read on the falling edge of the PS2 clock (PS2_CLK).



VGA

- We work in 640 by 480 resolution mode. This is a total of 800 horizontal pixels by 521 vertical lines, of which 640 by 480 are visible.
- For synchronization a horizontal and vertical pulse is needed.
- To allow the VGA screen to start a new line or frame a front porch and back porch is needed.
- The 60Hz refresh rate is 60Hz => 16.7ms period => 16.7ms/521 = 32us line time
- With a 25MHz clock: 40ns period => 800 pulses of this 25MHz clock

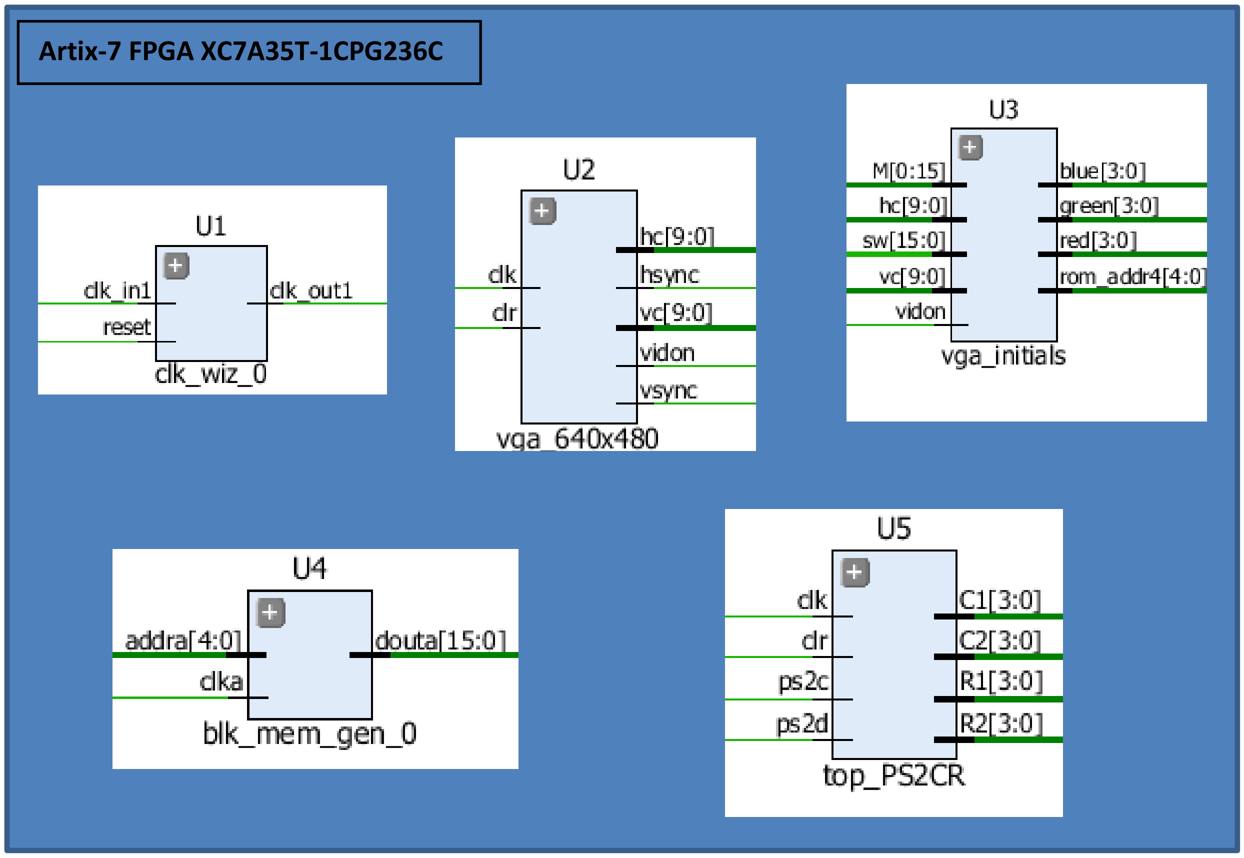


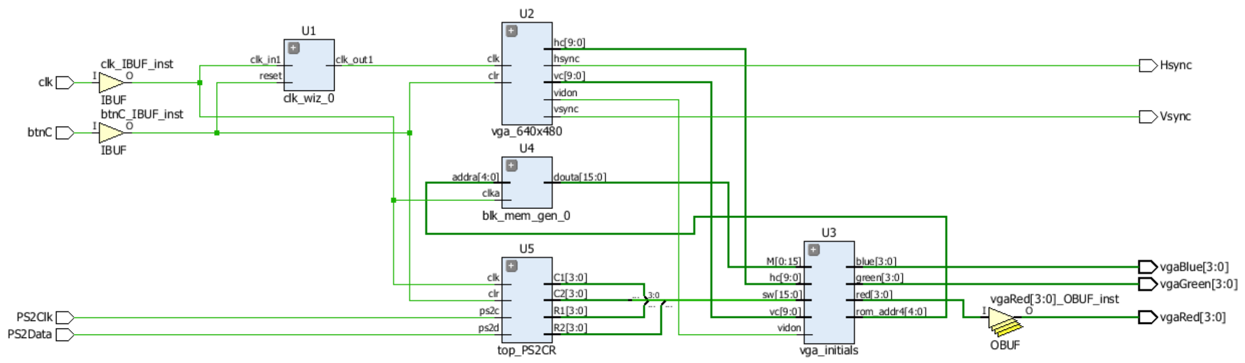
Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

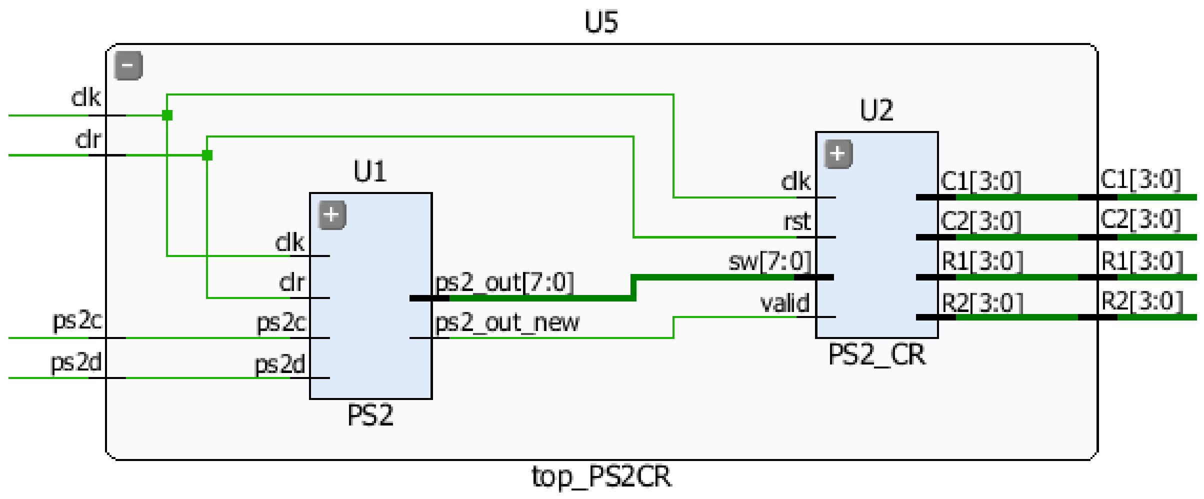
Internal Block Diagram

Internally we need a component for:

- Build a 25MHz clock signal
- VGA Synchronization
- vgaRed / vgaGreen / vgaBlue output
- Memory
- PS2 communication


Artix-7 FPGA XC7A35T-1CPG236C
U1
clk_in1
reset
clk_out1
clk_wiz_0
U2
clk
clr
hc[9:0]
hsync
vc[9:0]
vidon
vsync
vga_640x480
U3
M[0:15]
hc[9:0]
sw[15:0]
vc[9:0]
vidon
blue[3:0]
green[3:0]
red[3:0]
rom_addr4[4:0]
vga_initials
U4
addra[4:0]
clka
douta[15:0]
blk_mem_gen_0
U5
clk
clr
ps2c
ps2d
C1[3:0]
C2[3:0]
R1[3:0]
R2[3:0]
top_PS2CR





Build a 25MHz clock signal

This block converts the 100MHz system clock of the FPGA into a 25MHz, 40ns period, clock signal. This is needed to count pixels at the edge of this clock, 800 for each line.

This can be tested by applying a 100MHz clock in a test bench and check output timing.

Synchronization

The block builds the right synchronization signals for VGA, both line synchronization (hsync) and frame synchronization (vsync). It also puts out the pixel counter (hc) and the line counter (vc), which can be used as coordinates to draw a figure on the VGA screen. Finally it puts out a signal when the counters are in the visible area of the VGA screen (vidon). The block receives the 25MHz signal and an additional clear (clr) control signal, which puts all counter internally to zero.

This block can be tested by applying a 25MHz signal and check the timings of the synchronization pulses.

Vidon should also be tested. If the counters have the proper value, more than 144 and less than 784 for hc and more than 31 and less than 511 for vc vidon should be ‘1’.

vgaRed / vgaGreen / vgaBlue output

This block uses the content of a memory, combined with the pixel and line counters to display the sprites on the screen, on the right location, depending on the input of SW. It changes the ROM address depending on the values of the pixel and line counters.

This block can be tested to give an actual value to the input M from the ROM and change the pixel and line counters.

Memory

This block delivers 16-bit content M, depending on the received address.

This block can be tested by running through all address locations.

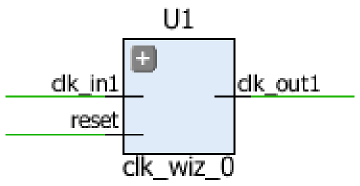
PS2 communication

This block is used to give values for the location of the two sprites, column C1, row R1 for sprite 1 and column C2, row R2 for sprite 2. It gets input for left-right, up-down for both sprites from keys on the USB keyboard.

To test this, the right clock and data signals should be built in a test bench, with the right scan codes for the data signals to check if C1, R1, C2 and R2 change.

Code & Documentation

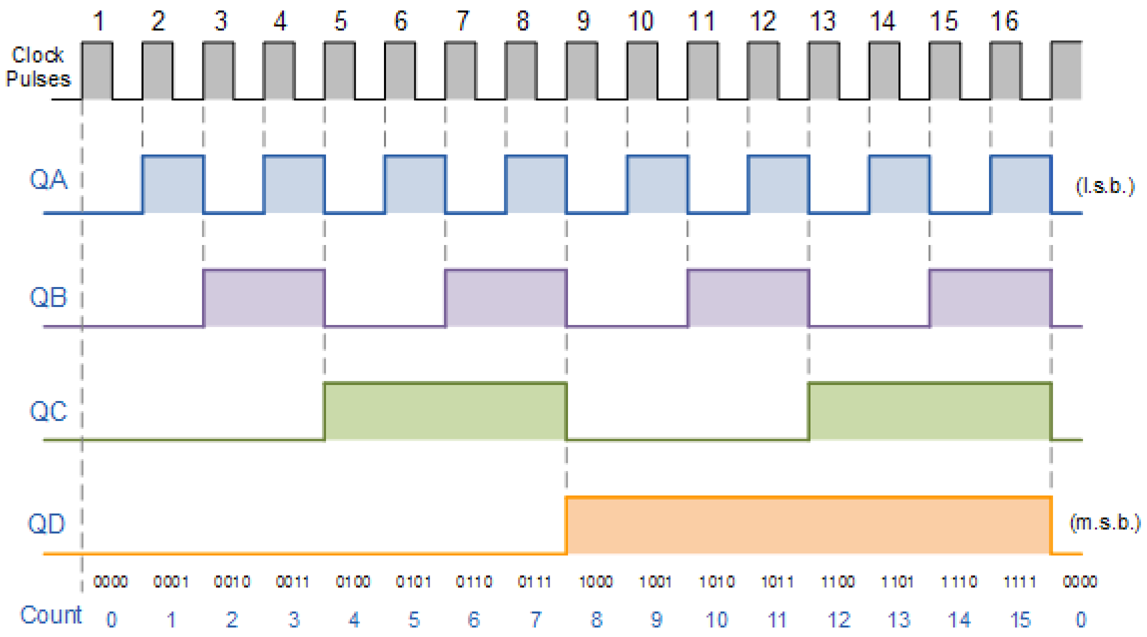
U1: 25MHz clock signal



There are two ways to tackle the generation of the 25 MHz clock, with a clock divider or to use the clocking infrastructure within the Artix 7 FPGA.

Clock Divider

First option is to use a counter to count clock pulses. Depending on which of the counted signals we use take we have a divided clock of the original clock.

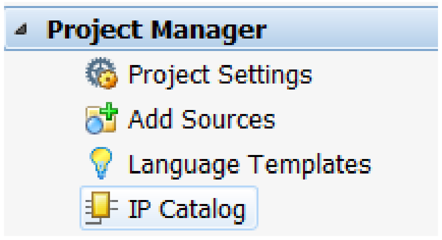


```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;
entity clkdiv is
        port(
                clk : in STD_LOGIC;
                clr : in STD_LOGIC;
                clk25 : out STD_LOGIC
          );
end clkdiv;

architecture div of clkdiv is
signal q:STD_LOGIC_VECTOR(3 downto 0);
begin

  process(clk, clr)
  begin
    if clr = '1' then
          q <= "0000";
    elsif rising_edge (clk) then
          q <= q + 1;
    end if;
  end process;

  clk25 <= q(1);   -- 25 MHz

end div;
```

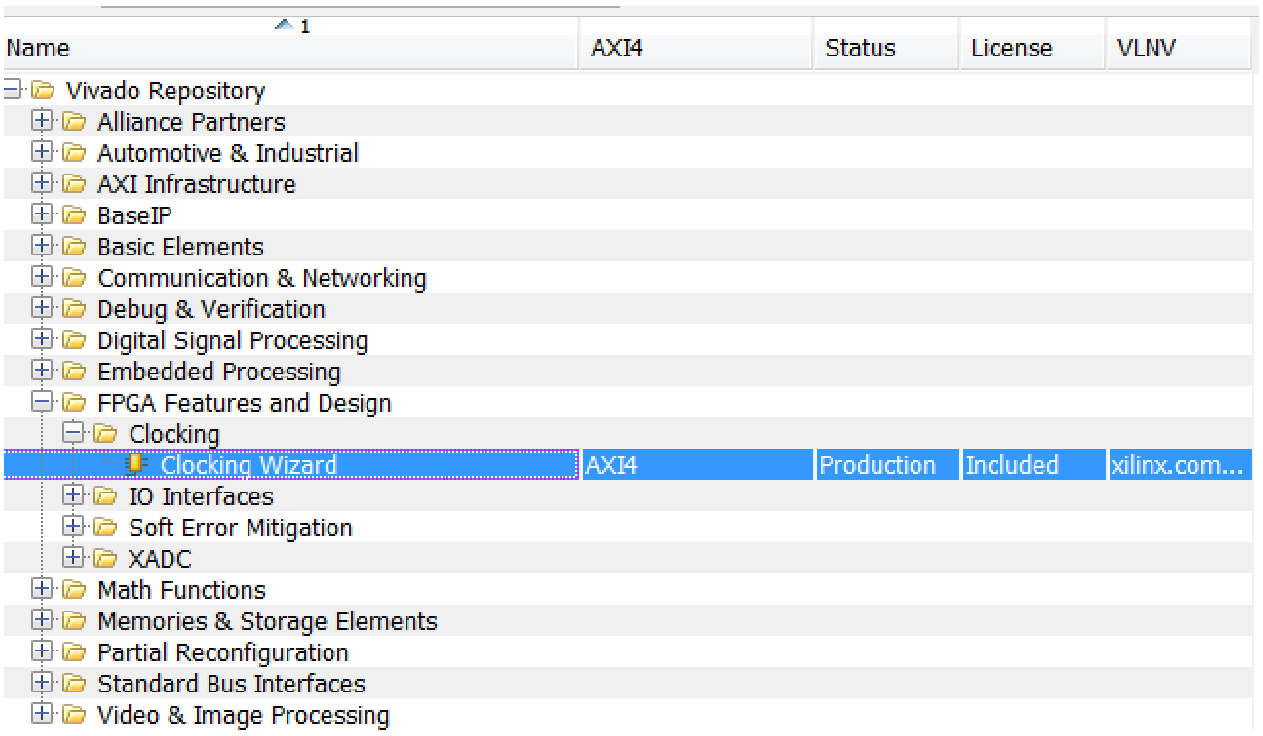
- clk50   <= q(0);   -- 50 MHz
- clk25   <= q(1);   -- 25 MHz
- Clk12.5 <= q(2);   -- 12.5 MHz

Clock IP

IP can be used through the IP catalog.

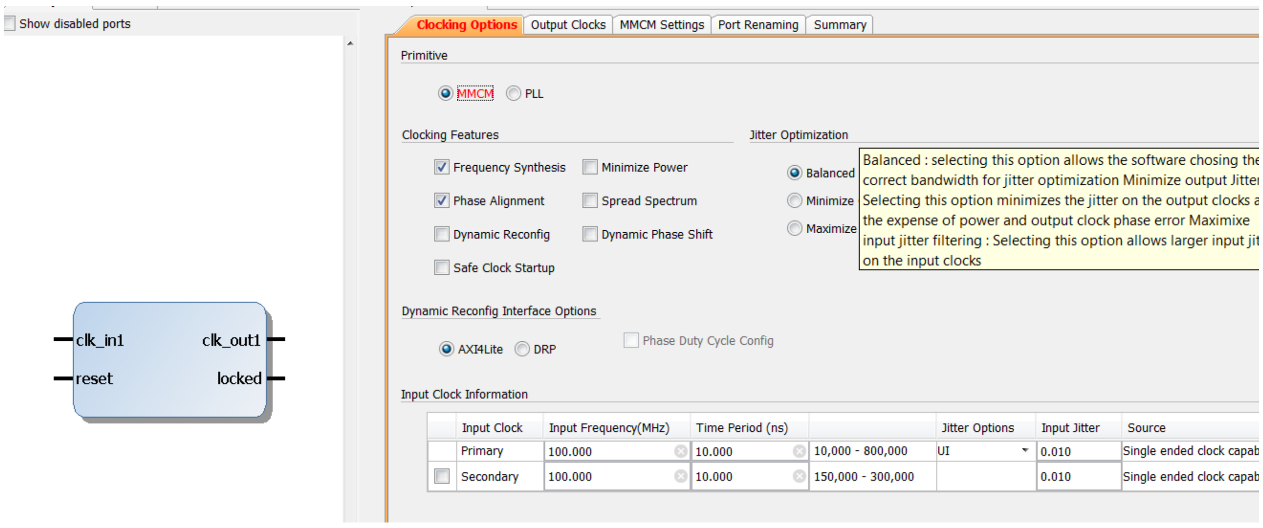

Project Manager

- Project Settings
- Add Sources
- Language Templates
- IP Catalog



Name	AXI4	Status	License	VLNV
Vivado Repository				
Alliance Partners				
Automotive & Industrial				
AXI Infrastructure				
BaseIP				
Basic Elements				
Communication & Networking				
Debug & Verification				
Digital Signal Processing				
Embedded Processing				
FPGA Features and Design				
Clocking				
Clocking Wizard	AXI4	Production	Included	xilinx.com...
IO Interfaces				
Soft Error Mitigation				
XADC				
Math Functions				
Memories & Storage Elements				
Partial Reconfiguration				
Standard Bus Interfaces				
Video & Image Processing				

First step is to define the input clock.



Primitive

[✓ MMCM] PLL

Clocking Features

[✓ Frequency Synthesis] Minimize Power
[✓ Phase Alignment] Spread Spectrum
Dynamic Reconfig Dynamic Phase Shift
Safe Clock Startup

Jitter Optimization

[✓ Balanced] Minimize Maximize

Balanced : selecting this option allows the software chosing the correct bandwidth for jitter optimization Minimize output Jitter Selecting this option minimizes the jitter on the output clocks a the expense of power and output clock phase error Maximize input jitter filtering : Selecting this option allows larger input jit on the input clocks

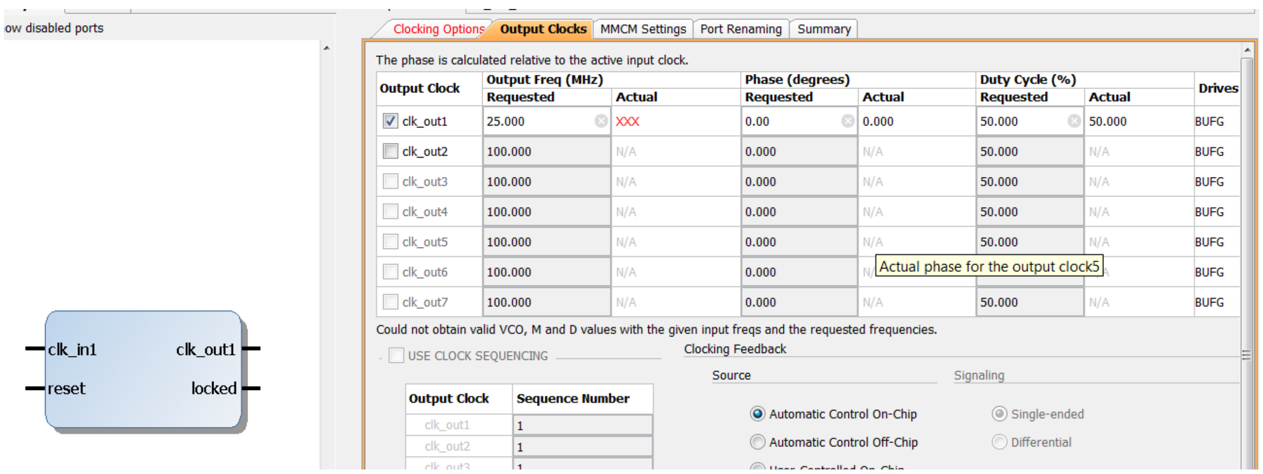
Dynamic Reconfig Interface Options

[✓ AXI4Lite] DRP Phase Duty Cycle Config

Input Clock Information

Input Clock	Input Frequency(MHz)	Time Period (ns)		Jitter Options	Input Jitter	Source
Primary	100.000	10.000	10.000 - 800,000	UI	0.010	Single ended clock capab
Secondary	100.000	10.000	150,000 - 300,000		0.010	Single ended clock capab

Next step is to define the desired output clock.



The phase is calculated relative to the active input clock.

Output Clock	Output Freq (MHz) Requested	Actual	Phase (degrees) Requested	Actual	Duty Cycle (%) Requested	Actual	Drives
✓ clk_out1	25.000	XXX	0.00	0.000	50.000	50.000	BUFG
clk_out2	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
clk_out3	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
clk_out4	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
clk_out5	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
clk_out6	100.000	N/A	0.000	N/A			BUFG
clk_out7	100.000	N/A	0.000	N/A	50.000	N/A	BUFG

Actual phase for the output clock5

Could not obtain valid VCO, M and D values with the given input freqs and the requested frequencies.

USE CLOCK SEQUENCING

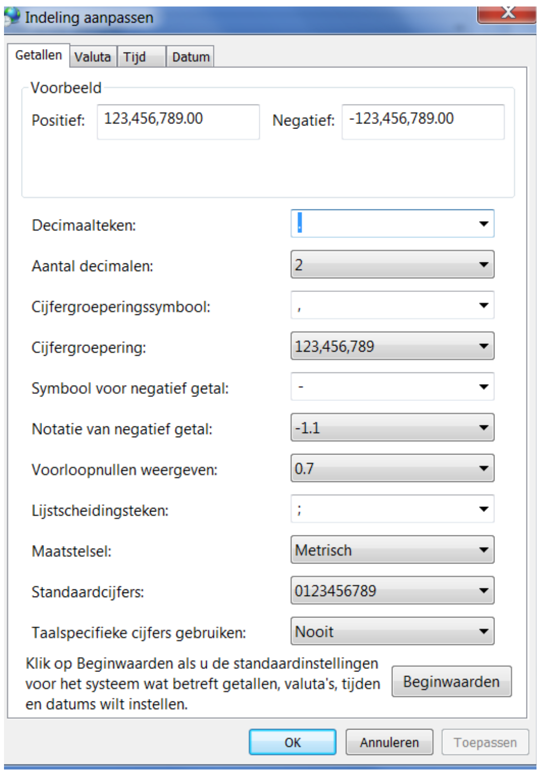
Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1

Clocking Feedback

Source: [✓ Automatic Control On-Chip] Automatic Control Off-Chip

Signaling: [✓ Single-ended] Differential

An unexpected erroneous operation might occur because of the difference in country settings between the software from the US and your system from the EU. You can overcome this problem by changing your settings => Configuration: country and language.



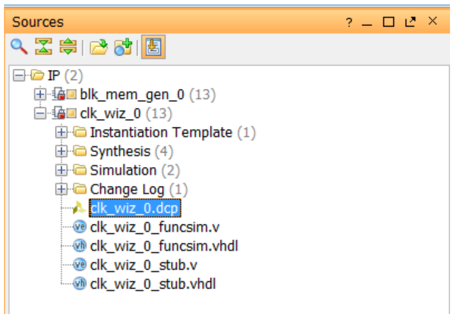
Voorbeeld

Positief: 123,456,789.00 Negatief: -123,456,789.00

Decimaalteken:	.
Aantal decimalen:	2
Cijfergroeperingssymbool:	,
Cijfergroepering:	123,456,789
Symbool voor negatief getal:	-
Notatie van negatief getal:	-1.1
Voorloopnullen weergeven:	0.7
Lijstscheidingsteken:	;
Maatstelsel:	Metrisch
Standaardcijfers:	0123456789
Taalspecifieke cijfers gebruiken:	Nooit

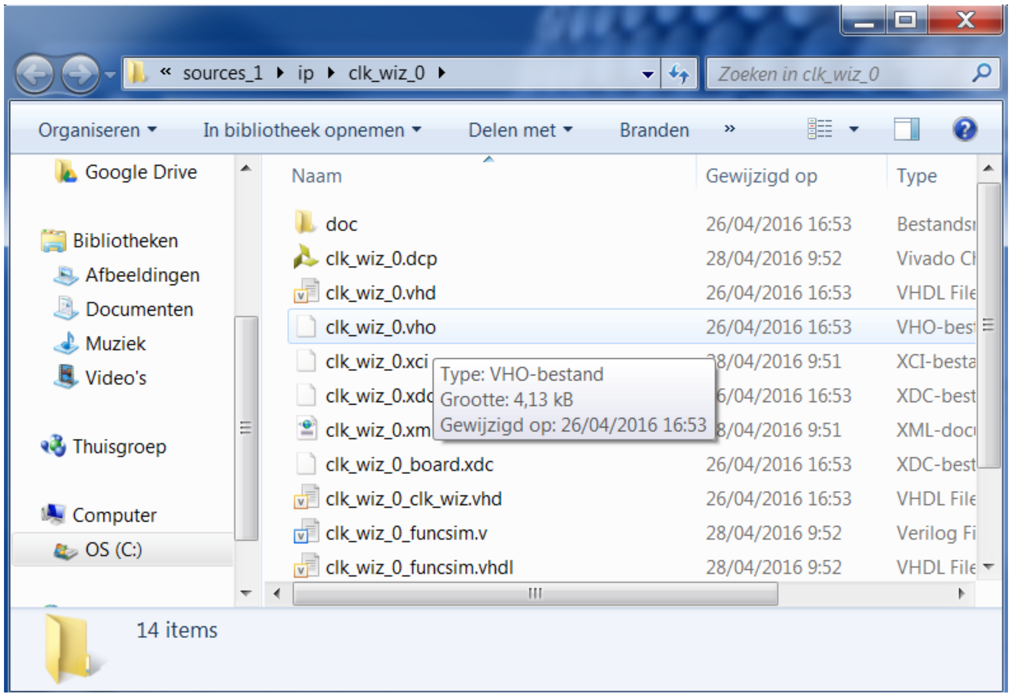
Klik op Beginwaarden als u de standaardinstellingen voor het systeem wat betreft getallen, valuta's, tijden en datums wilt instellen.

Next step is to add the component and the component instantiation in your code under IP sources: click clk_wiz_0.dcp, open directory and open clk_wiz_0.vho to find the component and the instantiation template. These are used to copy-paste between architecture & begin (component) and after begin (instantiation)



Sources

```
IP (2)
  blk_mem_gen_0 (13)
  clk_wiz_0 (13)
    Instantiation Template (1)
    Synthesis (4)
    Simulation (2)
    Change Log (1)
    clk_wiz_0.dcp
    clk_wiz_0_funcsim.v
    clk_wiz_0_funcsim.vhdl
    clk_wiz_0_stub.v
    clk_wiz_0_stub.vhdl
```



« sources_1 ▸ ip ▸ clk_wiz_0 ▸

Zoeken in clk_wiz_0

Organiseren ▾ In bibliotheek opnemen ▾ Delen met ▾ Branden »

Google Drive

Bibliotheken
Afbeeldingen
Documenten
Muziek
Video's

Thuisgroep

Computer
OS (C:)

Naam	Gewijzigd op	Type
doc	26/04/2016 16:53	Bestandsı
clk_wiz_0.dcp	28/04/2016 9:52	Vivado Cl
clk_wiz_0.vhd	26/04/2016 16:53	VHDL File
clk_wiz_0.vho	26/04/2016 16:53	VHO-best
clk_wiz_0.xci	28/04/2016 9:51	XCI-besta
clk_wiz_0.xdc	26/04/2016 16:53	XDC-best
clk_wiz_0.xml	28/04/2016 9:51	XML-docu
clk_wiz_0_board.xdc	26/04/2016 16:53	XDC-best
clk_wiz_0_clk_wiz.vhd	26/04/2016 16:53	VHDL File
clk_wiz_0_funcsim.v	28/04/2016 9:52	Verilog Fi
clk_wiz_0_funcsim.vhdl	28/04/2016 9:52	VHDL File

Type: VHO-bestand
Grootte: 4,13 kB
Gewijzigd op: 26/04/2016 16:53

14 items

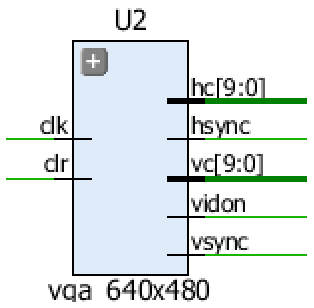
```
component clk_wiz_0
port
(-- Clock in ports
clk_in1          : in     std_logic;
-- Clock out ports
clk_out1         : out    std_logic;
-- Status and control signals
reset            : in     std_logic;
locked           : out    std_logic );
end component;
ATTRIBUTE SYN_BLACK_BOX : BOOLEAN;
ATTRIBUTE SYN_BLACK_BOX OF clk_wiz_0 : COMPONENT IS TRUE;
ATTRIBUTE BLACK_BOX_PAD_PIN : STRING;
ATTRIBUTE BLACK_BOX_PAD_PIN OF clk_wiz_0 : COMPONENT IS "clk_in1,clk_out1,reset,locked";
```

```
your_instance_name : clk_wiz_0
port map (
-- Clock in ports
clk_in1 => clk_in1,
-- Clock out ports
clk_out1 => clk_out1,
-- Status and control signals
reset => reset,
locked => locked
);
```

Change name and signals to your specific situation.

```
U1 : clk_wiz_0
  port map (     clk_in1 => clk,
                 clk_out1 => clk25,
                 reset => clr,
                 locked => open
 );
```

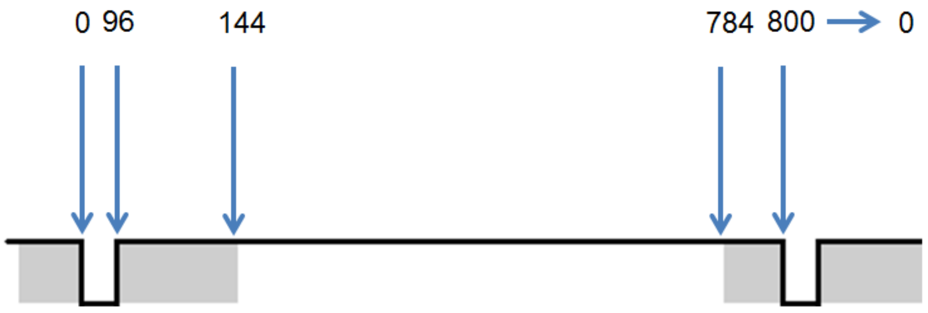
U2: Synchronization



Synchronization is done by simply counting clock pulses of the 25MHz clock for the pixels of the VGA screen and counting lines.

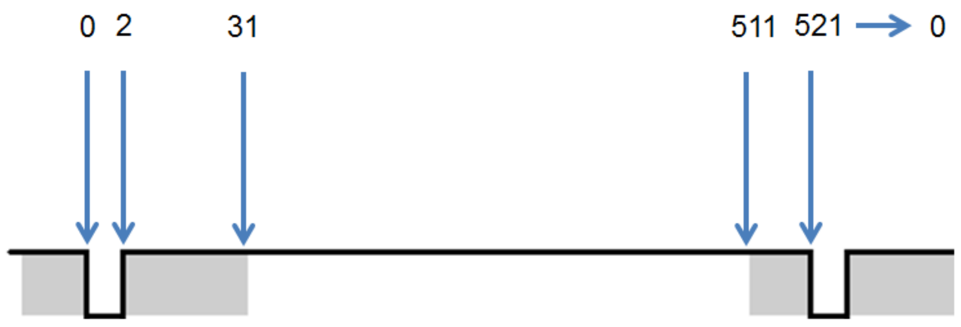
The sequence of events during the counting of the pixels is as follows:

- Start counting at 0 with hsync at '0'
- When the horizontal counter hc reaches 96, hsync becomes '1'
- When hc is 144 we are in the visible area of the line
- When hc is 784 we go outside the visible area of the line
- When hc is 800 all pixels are counted, we add one line to the vertical counter vc, the horizontal counter is put back to 0 and hsync to '0'.



The sequence of events during the counting of the lines is as follows:

- Start counting at 0 with vsync at '0'
- When the vertical counter vc reaches 2, vsync becomes '1'
- When vc is 31 we are in the visible area of the frame
- When vc is 511 we go outside the visible area of the frame
- When vc is 521 all lines are counted, the vertical counter vc, is put back to 0 and vsync to '0'.



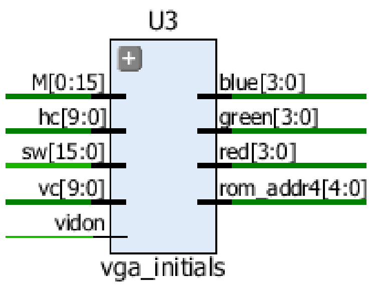
Vidon is 1 for hc is between 144 and 784 and vc is between 31 and 511.

The vertical counter vc, the horizontal counter hc are given as output because they are used by the other blocks in the design.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vga_640x480 is
   port ( clk, clr : in std_logic;
          hsync : out std_logic;
          vsync : out std_logic;
          hc : out std_logic_vector(9 downto 0);
          vc : out std_logic_vector(9 downto 0);
          vidon : out std_logic
            );
end vga_640x480;

architecture vga_640x480 of vga_640x480 is
constant hpixels: std_logic_vector(9 downto 0) := "1100100000";
            --Value of pixels in a horizontal line = 800
constant vlines: std_logic_vector(9 downto 0) := "1000001001";
            --Number of horizontal lines in the display = 521
constant hbp: std_logic_vector(9 downto 0) := "0010010000";
            --Horizontal back porch = 144 (128+16)
constant hfp: std_logic_vector(9 downto 0) := "1100010000";
            --Horizontal front porch = 784 (128+16+640)
constant vbp: std_logic_vector(9 downto 0) := "0000011111";
            --Vertical back porch = 31 (2+29)
constant vfp: std_logic_vector(9 downto 0) := "0111111111";
            --Vertical front porch = 511 (2+29+480)
signal hcs, vcs: std_logic_vector(9 downto 0);
    --These are the Horizontal and Vertical counters
signal vsenable: std_logic;
            --Enable for the Vertical counter
```

```vhdl
begin
        --Counter for the horizontal sync signal
        process(clk, clr)
        begin
          if clr = '1' then
                hcs <= "0000000000";
          elsif(clk'event and clk = '1') then
                if hcs = hpixels - 1 then           --The counter has reached the end of pixel count
                        hcs <= "0000000000";        --reset the counter
                        vsenable <= '1';            --Enable the vertical counter
                else
                        hcs <= hcs + 1;             --Increment the horizontal counter
                        vsenable <= '0';            --Leave the vsenable off
                end if;
          end if;
        end process;

        hsync <= '0' when hcs < 96 else '1';        --Horizontal Sync Pulse is low when hc is 0 - 96

        --Counter for the vertical sync signal
        process(clk, clr, vsenable)
        begin
          if clr = '1' then
                vcs <= "0000000000";
          elsif(clk'event and clk = '1' and vsenable='1') then
                                                    --Increment when enabled
                if vcs = vlines - 1 then            --Reset when the number of lines is reached
                        vcs <= "0000000000";
                else
                        vcs <= vcs + 1;             --Increment vertical counter
                end if;
          end if;
        end process;

        vsync <= '0' when vcs < 2 else '1';         --Vertical Sync Pulse is low when vc is 0 or 1

        --Enable video out when within the porches
        vidon <= '1' when (((hcs < hfp) and (hcs >= hbp))
           and ((vcs < vfp) and (vcs >= vbp))) else '0';

        -- output horizontal and vertical counters
        hc <= hcs;
        vc <= vcs;

end vga_640x480;
```

U3: VGA output



It uses horizontal counter hc and vertical counter vc for for addressing the right coordinates on the VGA screen.

Vidon is used to determine whether the counters are in the visible display area.

Input SW uses 16 bits to determine the XY-coordinates of the two sprites. These will get a value from the PS2 keyboard.

The component gives an address to a ROM, in which the sprites are stored, which gives the proper value back through M, a 16 bit value.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity vga_initials is
   port ( vidon: in std_logic;
         hc : in std_logic_vector(9 downto 0);
         vc : in std_logic_vector(9 downto 0);
         M: in std_logic_vector(0 to 15);
         sw: in std_logic_vector(15 downto 0);
         rom_addr4: out std_logic_vector(4 downto 0);
         red : out std_logic_vector(3 downto 0);
         green : out std_logic_vector(3 downto 0);
         blue : out std_logic_vector(3 downto 0)
            );
end vga_initials;
```

```
architecture vga_initials of vga_initials is
constant hbp: std_logic_vector(9 downto 0) := "0010010000";       --horizontal back porch
constant vbp: std_logic_vector(9 downto 0) := "0000011111";       --vertical back porch
constant w: integer := 16;                                     --width of the sprites
constant h: integer := 16;                                     -- height of the sprites
signal C1, R1, C2, R2: std_logic_vector(10 downto 0);          --columns and rows of sprite1 and sprite2
signal rom_addr, rom_pix: std_logic_vector(10 downto 0);--internal line address and pixel address
signal spriteon1, spriteon2, R, G, B: std_logic;       --signals to determine where the sprites are drawn
```

```
begin
--set C1, C2, R1 and R2 using switches
C1 <= "00" & SW(3 downto 0) & "00001";
R1 <= "00" & SW(7 downto 4) & "00001";
C2 <= "00" & SW(11 downto 8) & "00001";
R2 <= "00" & SW(15 downto 12) & "00001";
--Enable sprite1 video out when within the sprite region, depending on C1&R1
 spriteon1 <= '1' when (((hc >= C1 + hbp) and (hc < C1 + hbp + w)) and ((vc >= R1 + vbp) and (vc < R1 +
vbp + h))) else '0';
--Enable sprite2 video out when within the sprite region, depending on C2&R2
 spriteon2 <= '1' when (((hc >= C2 + hbp) and (hc < C2 + hbp + w)) and ((vc >= R2 + vbp) and (vc < R2 +
vbp + h))) else '0';
process(spriteon1, spriteon2, vidon, rom_pix, M, vc, hc, R1, C1, R2, C2,rom_addr, B)
  variable j: integer;
 begin
        red <= "0000";        --default values for red green & blue
        green <= "0000";
        blue <= "0000";
        if spriteon1 = '1' and vidon = '1' then   --draw the sprite in the right spot
--determine the address of the ROM, address increases when vc increases
        rom_addr <= vc - vbp - R1;
--determine the address of the pixel in the binary word M, address increases when hc increases
        rom_pix <= hc - hbp - C1;
        rom_addr4 <= rom_addr(4 downto 0);   --only the lowest bits are needed
                --convert binary value of rom_pix to integer value for index in M
                j := conv_integer(rom_pix);
                if M(j) = '1' then            --add  the value of M to  R,G en B if '1'
                        R <= M(j);
                        G <= M(j);
                        B <= M(j);
                        blue <= B & B & B & B;   --we only draw a blue sprite in this case
                        elsif hc(5) = '1' then       --make a dark and light green background
                                green <= "1111";
                         else
                                green <= "1011";
                        end if;
```

```
            elsif spriteon2 = '1' and vidon = '1' then
            rom_addr <= vc - vbp - R2 + 16;            -- next sprite 16 memory places down the ROM
            rom_pix <= hc - hbp - C2;
            rom_addr4 <= rom_addr(4 downto 0);
                  j := conv_integer(rom_pix);
                  if M(j) = '1' then
                  R <= M(j);
                  G <= M(j);
                  B <= M(j);
                  blue <= B & B & B & B;                --we only draw a blue sprite in this case
                  elsif hc(5) = '1' then                --make a dark and light green background
                     green <= "1111";
                   else
                     green <= "1011";
                   end if;
         elsif vidon = '1' then                         --make a dark and light green background
             if hc(5) = '1' then
                green <= "1111";
              else
                green <= "1011";
              end if;
        end if;
            end process;

end vga_initials;
```

U4: Block ROM

There are two ways to draw a sprite. One is using the standard resources in the FPGA to build a PROM, by making it in VHDL. Another solution is using the Block RAM available in the FPGA, through a wizard.

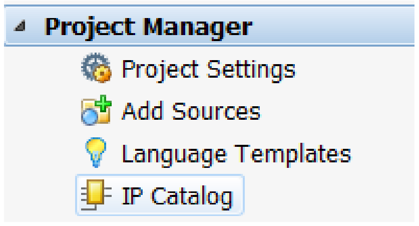
PROM

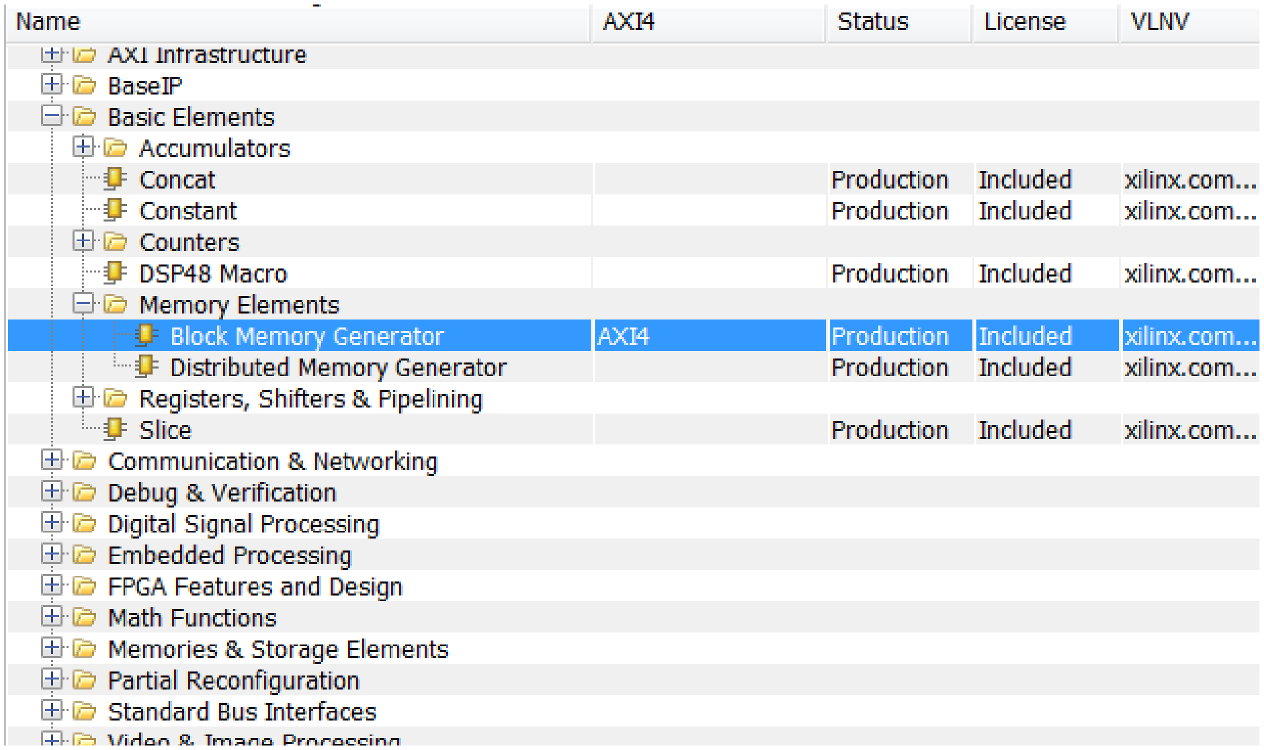
This uses an array in which you can build your sprite.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity prom_house is
   port (
      addr: in STD_LOGIC_VECTOR (3 downto 0);
      M: out STD_LOGIC_VECTOR (0 to 15)
   );
end prom_house;

architecture prom of prom_house is
type rom_array is array (NATURAL range <>)
of STD_LOGIC_VECTOR (0 to 15);
constant rom: rom_array := (
"0000000110000000",
"0100001111000010",
"0000011111100000",
"0000111111110000",
"0001111111111000",
"0011111111111100",
"0111111111111110",
"1111111111111111",
"1100000000000011",
"1100000001110011",
"1100000001110011",
"1100000001110011",
"1100111000000011",
"1100111000000011",
"1100111000000011",
"1100111000000011"
              );
begin
  process(addr)
  variable j: integer;
  begin
    j := conv_integer(addr);         --binary type address is converted to integer type index of array
    M <= rom(j);
  end process;
end prom;
```

Block RAM

Block RAM IP can be used through the IP catalog.

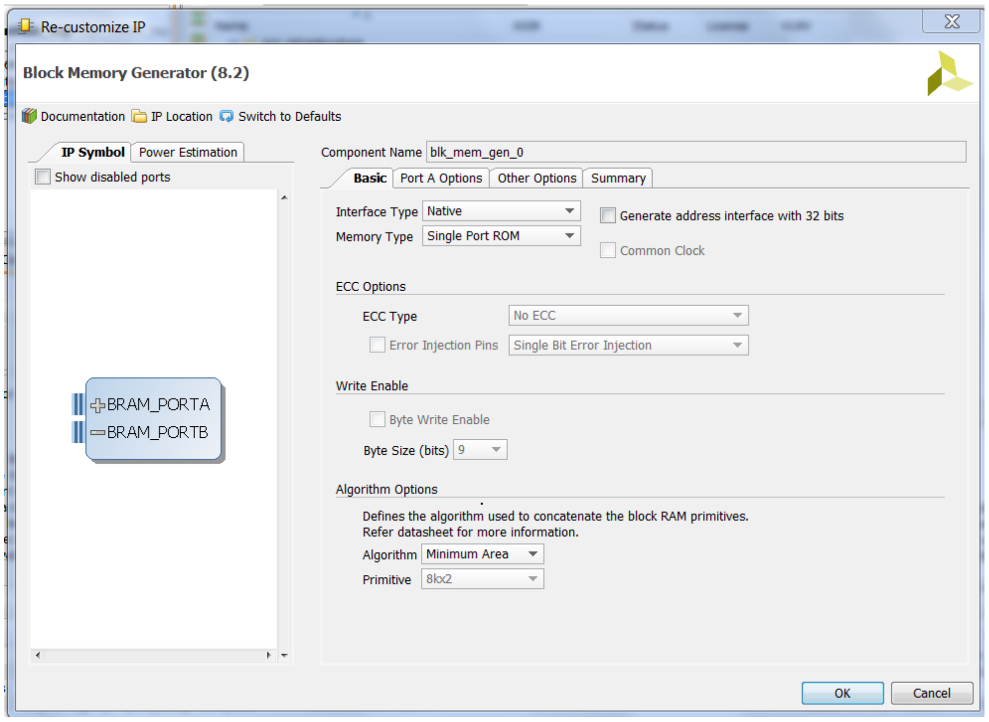




Name	AXI4	Status	License	VLNV
AXI Infrastructure				
BaseIP				
Basic Elements				
Accumulators				
Concat		Production	Included	xilinx.com...
Constant		Production	Included	xilinx.com...
Counters				
DSP48 Macro		Production	Included	xilinx.com...
Memory Elements				
Block Memory Generator	AXI4	Production	Included	xilinx.com...
Distributed Memory Generator		Production	Included	xilinx.com...
Registers, Shifters & Pipelining				
Slice		Production	Included	xilinx.com...
Communication & Networking				
Debug & Verification				
Digital Signal Processing				
Embedded Processing				
FPGA Features and Design				
Math Functions				
Memories & Storage Elements				
Partial Reconfiguration				
Standard Bus Interfaces				
Video & Image Processing				

We use a read-only memory in which we place the sprites. To do so we select Single Port ROM.

Next we determine the width and height of the sprite we want to draw, in this case 16 pixels x 16 lines, so each memory location contains a binary word of 16 bits wide, 16 address locations in total.



Re-customize IP

Block Memory Generator (8.2)

Documentation | IP Location | Switch to Defaults

IP Symbol | Power Estimation

☐ Show disabled ports

Component Name blk_mem_gen_0

Basic | Port A Options | Other Options | Summary

Interface Type Native

Memory Type Single Port ROM

☐ Generate address interface with 32 bits

☐ Common Clock

ECC Options

ECC Type No ECC

☐ Error Injection Pins Single Bit Error Injection

Write Enable

☐ Byte Write Enable

Byte Size (bits) 9

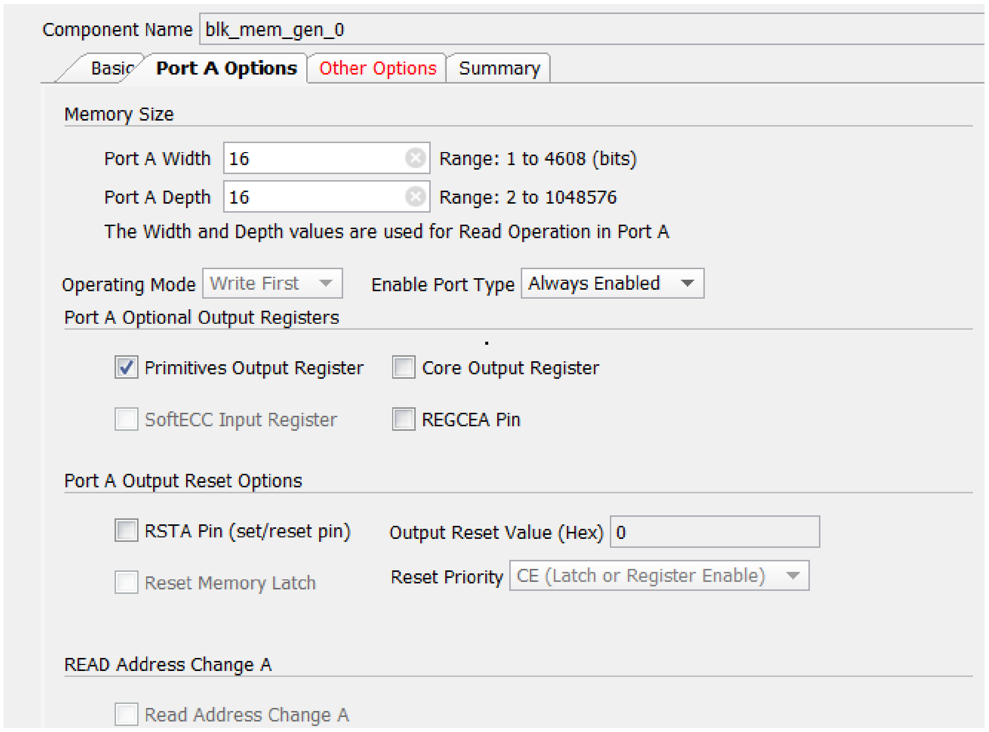
Algorithm Options

Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.

Algorithm Minimum Area

Primitive 8kx2

OK | Cancel



Component Name blk_mem_gen_0

Basic | **Port A Options** | Other Options | Summary

Memory Size

Port A Width 16 Range: 1 to 4608 (bits)

Port A Depth 16 Range: 2 to 1048576

The Width and Depth values are used for Read Operation in Port A

Operating Mode Write First Enable Port Type Always Enabled

Port A Optional Output Registers

☑ Primitives Output Register ☐ Core Output Register

☐ SoftECC Input Register ☐ REGCEA Pin

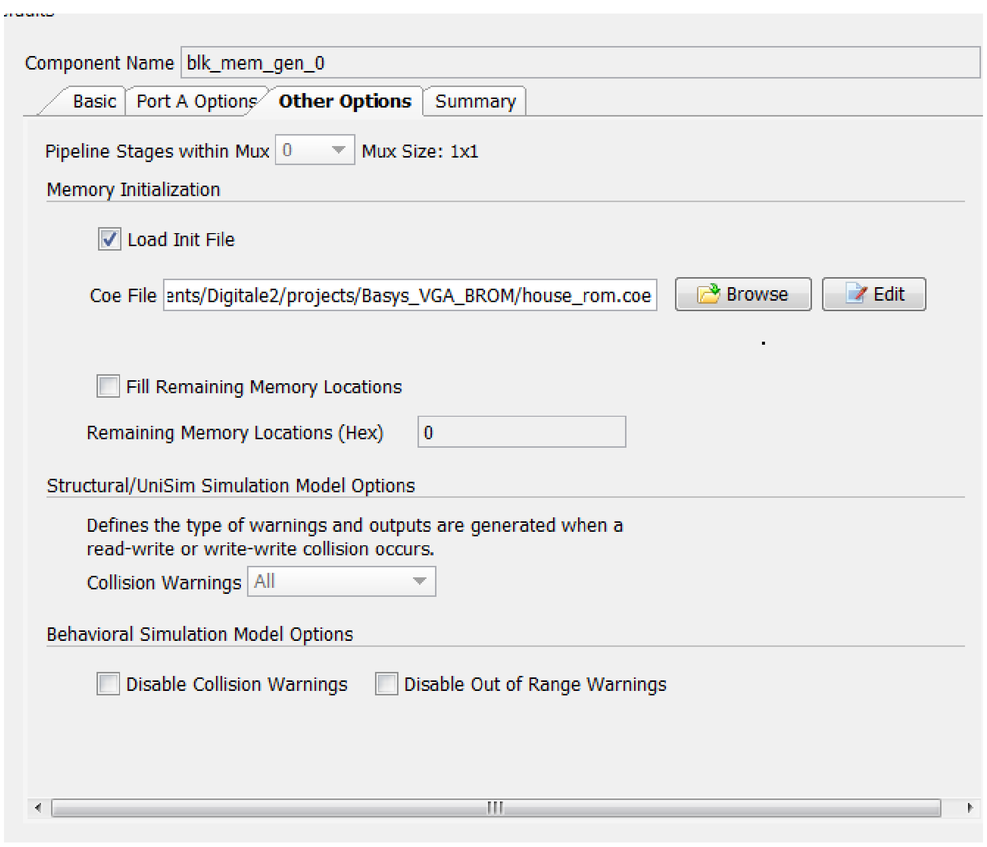
Port A Output Reset Options

☐ RSTA Pin (set/reset pin) Output Reset Value (Hex) 0

☐ Reset Memory Latch Reset Priority CE (Latch or Register Enable)

READ Address Change A

☐ Read Address Change A



Component Name blk_mem_gen_0

Basic | Port A Options | **Other Options** | Summary

Pipeline Stages within Mux 0 Mux Size: 1x1

Memory Initialization

[✓] Load Init File

Coe File ents/Digitale2/projects/Basys_VGA_BROM/house_rom.coe Browse Edit

[ ] Fill Remaining Memory Locations

Remaining Memory Locations (Hex) 0

Structural/UniSim Simulation Model Options

Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.

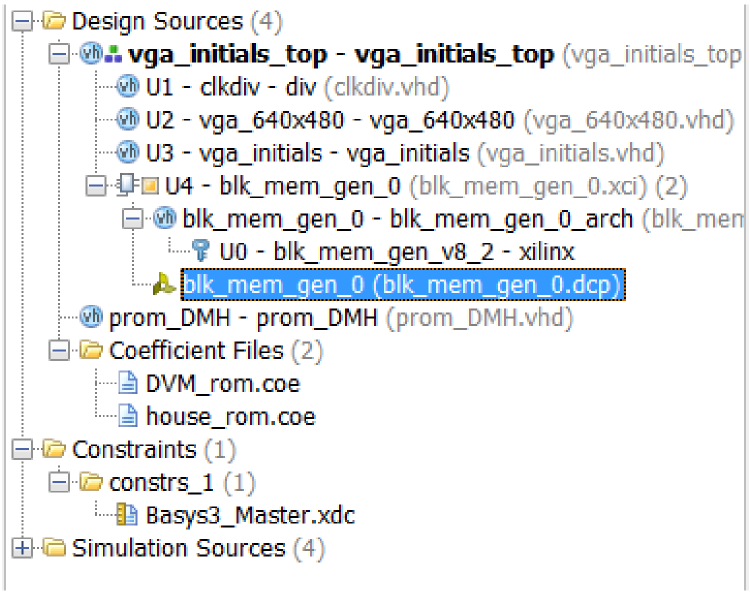
Collision Warnings All

Behavioral Simulation Model Options

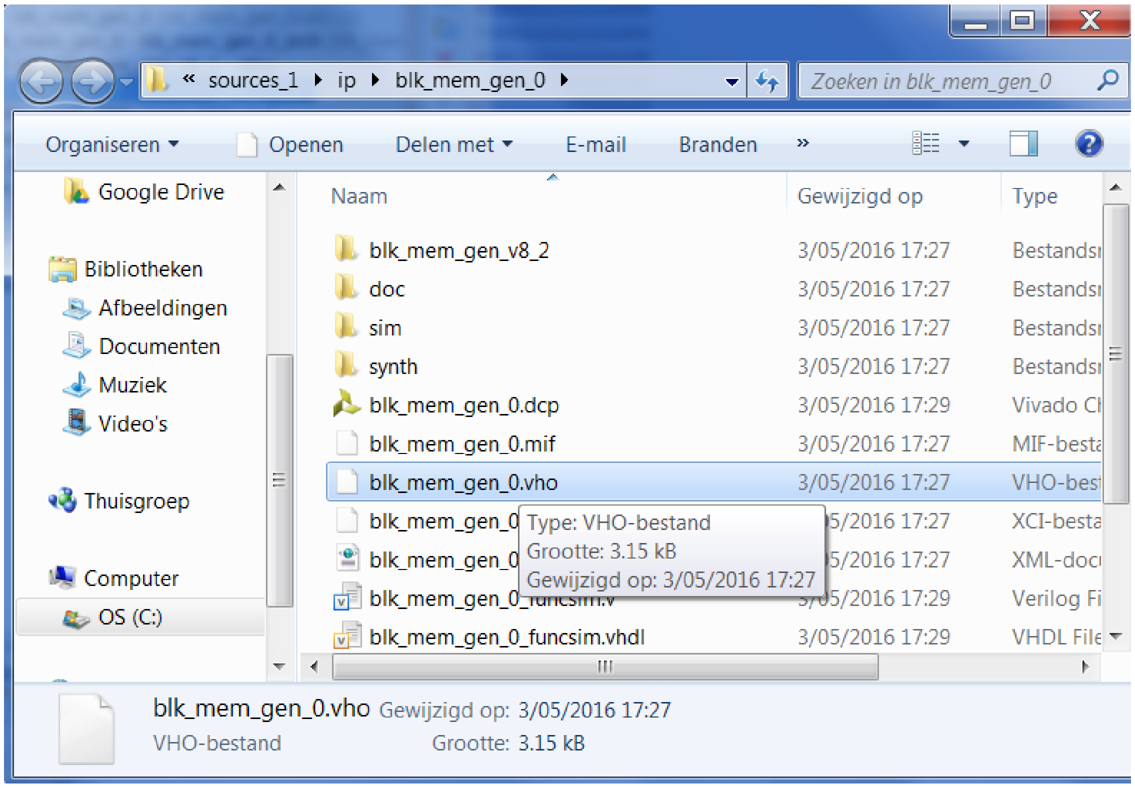
[ ] Disable Collision Warnings [ ] Disable Out of Range Warnings

```
memory_initialization_radix = 2;
memory_initialization_vector =
0000000110000000
0100001111000010
0000011111100000
0000111111110000
0001111111111000
0011111111111100
0111111111111110
1111111111111111
1100000000000011
1100000001110011
1100000001110011
1100000001110011
1100111000000011
1100111000000011
1100111000000011
1100111000000011
0000000000000000
1100001111000011
1000010000100001
1000100000010001
0001000000001000
0010000000000100
0100000000000010
1111111111111111
1100000000000011
1100000000000011
1100000001110011
1100000001110011
1100000000000011
1100111111000011
1100111111000011
1100111111000011
;
```

After the generation of the memory IP, we should implement this in the parent code, by copy-pasting the component and the instantiation at the right spot. We need to open block_mem_gen_0.vho to find the component and the instantiation templates. Finally the ports of the instantiation template should be connected to the signal of the top design.



```
Design Sources (4)
  vga_initials_top - vga_initials_top (vga_initials_top
    U1 - clkdiv - div (clkdiv.vhd)
    U2 - vga_640x480 - vga_640x480 (vga_640x480.vhd)
    U3 - vga_initials - vga_initials (vga_initials.vhd)
    U4 - blk_mem_gen_0 (blk_mem_gen_0.xci) (2)
      blk_mem_gen_0 - blk_mem_gen_0_arch (blk_mem
        U0 - blk_mem_gen_v8_2 - xilinx
      blk_mem_gen_0 (blk_mem_gen_0.dcp)
  prom_DMH - prom_DMH (prom_DMH.vhd)
  Coefficient Files (2)
    DVM_rom.coe
    house_rom.coe
Constraints (1)
  constrs_1 (1)
    Basys3_Master.xdc
Simulation Sources (4)
```



Naam	Gewijzigd op	Type
blk_mem_gen_v8_2	3/05/2016 17:27	Bestandsı
doc	3/05/2016 17:27	Bestandsı
sim	3/05/2016 17:27	Bestandsı
synth	3/05/2016 17:27	Bestandsı
blk_mem_gen_0.dcp	3/05/2016 17:29	Vivado Cl
blk_mem_gen_0.mif	3/05/2016 17:27	MIF-besta
blk_mem_gen_0.vho	3/05/2016 17:27	VHO-best
blk_mem_gen_0	5/2016 17:27	XCI-besta
blk_mem_gen_0	5/2016 17:27	XML-docı
blk_mem_gen_0_funcsim.v	3/05/2016 17:29	Verilog Fi
blk_mem_gen_0_funcsim.vhdl	3/05/2016 17:29	VHDL File

Type: VHO-bestand
Grootte: 3.15 kB
Gewijzigd op: 3/05/2016 17:27

blk_mem_gen_0.vho Gewijzigd op: 3/05/2016 17:27
VHO-bestand Grootte: 3.15 kB

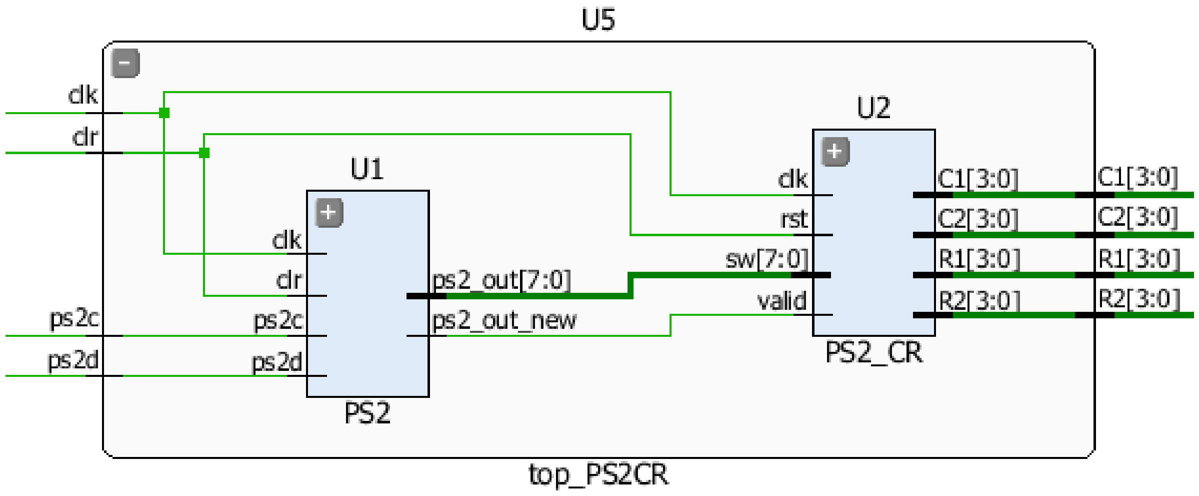
```
COMPONENT blk_mem_gen_0  PORT (
        clka : IN STD_LOGIC;
        addra : IN STD_LOGIC_VECTOR(3 DOWNTO 0);          douta : OUT STD_LOGIC_VECTOR(15
DOWNTO 0)  );
END COMPONENT;

your_instance_name : blk_mem_gen_0
PORT MAP (    clka => clka,    addra => addra,    douta => douta  );
```

```
U4 : blk_mem_gen_0
     PORT MAP (
      clka => clk,
      addra => rom_addr4,
      douta => M
     );
```

U5: PS2 to CR information

This component exists of two sub-components. One component U1: PS2 converts the PS2 clock and PS2 data into the scan code output, ps2_out, and gives a flag if the data is valid, ps2_out_new.

PS2_CR converts the scan code from keyboard into a value for column and row for sprite 1 (C1 & R1) and sprite 2 (C2 & R2).



PS2

This component consists of three processes.

The first filter process is used for debouncing. The input needs to be stable 1 or stable 0 for 8 system clock pulses to put out a 1 or 0 respectively.

The second process shifts in the PS2 data, signal ps2d, in an internal shift register, at each falling edge of the PS2 clock, signal ps2c.

In the third process an idle counter determines when the transaction is finished, defined by the PS2 clock remaining at a high logic level for more than 55us. If we divide the system clock 100,000,000 by 18,000 we get 5555.5555. If we multiply this with 10ns period of the system clock we get 5555.5555*10ns = 55.55us.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity PS2 is
   Generic(clk_freq: INTEGER := 100_000_000); --system clock frequency in Hz
   Port ( clk : in STD_LOGIC;                          --system clock
          clr : in STD_LOGIC;                          --clr
          ps2d : in STD_LOGIC;                         --input data from PS2
          ps2c : in STD_LOGIC;                         --input clock from PS2
          ps2_out_new : out STD_LOGIC;                 --flag that new PS2 output is available on ps2_code
bus
          ps2_out : out STD_LOGIC_VECTOR (7 downto 0));    --output received from PS2
end PS2;

architecture Behavioral of PS2 is

signal ps2c_int  : STD_LOGIC;                         --debounced clock signal from PS/2 keyboard
signal ps2d_int : STD_LOGIC;                          --debounced data signal from PS/2 keyboard
signal ps2c_filter: STD_LOGIC_VECTOR(7 DOWNTO 0);
signal ps2d_filter: STD_LOGIC_VECTOR(7 DOWNTO 0);
signal ps2_word     : STD_LOGIC_VECTOR(10 DOWNTO 0);      --stores the ps2 data word
signal count_idle   : INTEGER RANGE 0 TO clk_freq/18_000; --counter to determine PS/2 is idle
```

```
begin

filter: process(clk, clr)
  begin
    if clr = '1' then
      ps2c_filter <= (others => '1');
      ps2d_filter <= (others => '1');
    elsif rising_edge (clk) then
      ps2c_filter(7) <= ps2c;                 --shift regsiter to shift in PS2 clock
      ps2c_filter(6 downto 0) <= ps2c_filter(7 downto 1);
      ps2d_filter(7) <= ps2d;                 --shift regsiter to shift in PS2 data
      ps2d_filter(6 downto 0) <= ps2d_filter(7 downto 1);
      if ps2c_filter = X"FF" then             --if "11111111" is shift register: ps2c_int = '1'
        ps2c_int <= '1';
      elsif ps2c_filter = X"00" then          --if "00000000" is shift register: ps2c_int = '0'
        ps2c_int <= '0';
      end if;
      if ps2d_filter = X"FF" then             --if "11111111" is shift register: ps2d_int = '1'
        ps2d_int <= '1';
      elsif ps2d_filter = X"00" then          --if "00000000" is shift register: ps2d_int = '0'
        ps2d_int <= '0';
      end if;
    end if;
  end process filter;
```

```
--input PS2 data
input:  process(ps2c_int)
  begin
    if(ps2c_int'event and ps2c_int = '0') then    --falling edge of PS2 clock
      ps2_word <= ps2d_int & ps2_word(10 downto 1);   --shift in PS2 data bit
    end if;
  end process;

--determine if PS2 port is idle (i.e. last transaction is finished) and output result
output:  process(clk)
  begin
    if rising_edge (clk) then           --rising edge of system clock
      if(ps2c_int = '0') then                 --low PS2 clock, PS/2 is active
        count_idle <= 0;                           --reset idle counter
         ps2_out_new <= '0';
      elsif(count_idle /= clk_freq/18_000) then
--PS2 clock has been high less than a half clock period (<55us)
          count_idle <= count_idle + 1;           --continue counting
           ps2_out_new <= '0';
       else
            ps2_out_new <= '1';                 --set flag that new PS/2 code is available, when counter >55us
               ps2_out <= ps2_word(8 DOWNTO 1);
      end if;
    end if;
  end process;
end Behavioral;
```

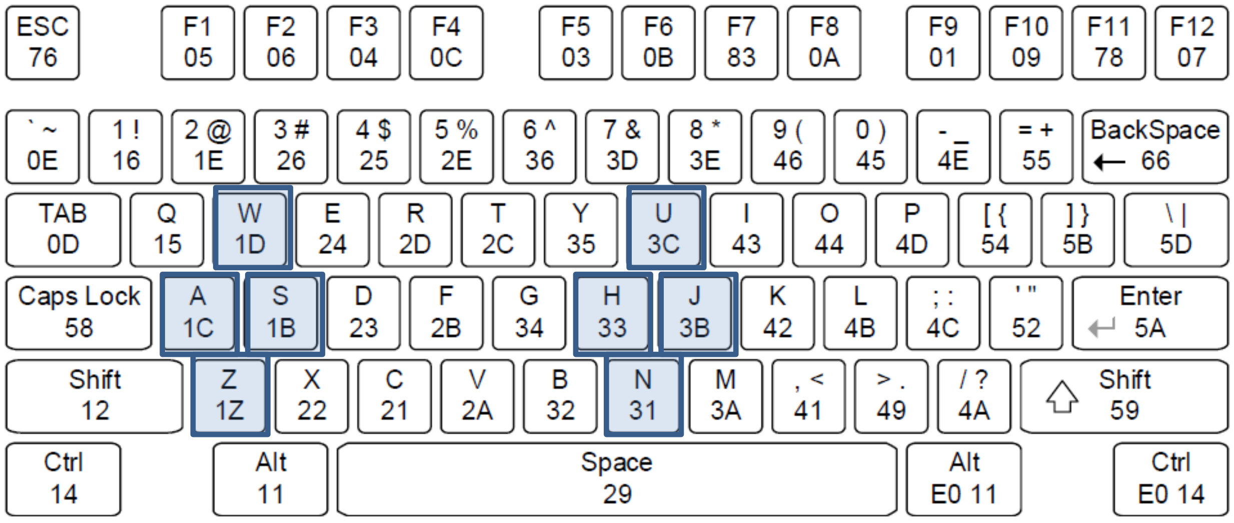
PS2_CR

This component looks at the falling edge of the ps2_out_new of the PS2 component, which is connected to valid of the PS2_CR component.

When the right scan code is received, the R1C1 or R2C2 is increased or decreased, if the flag is still set to ‘0’.

We use scan codes

- “1D” for up1
- “1C” for left1
- “1B” for right1
- “1B” for down1
- “3C” for up2
- “33” for left2
- “3B” for right2
- “31” for down2



We only want to allow a change for R1C1 or R2C2 when we push the key, not when we release the key. So if the scan code gives “F0”, the flag is set to ‘1’ and the next scan code is ignored. This means R1C1 and R2C2 remain the same for once and the flag is put back to ‘0’.

```
entity PS2_CR is
    Port ( data : in STD_LOGIC_VECTOR (7 downto 0);
           valid : in STD_LOGIC;
           rst : in STD_LOGIC;
           clk : in STD_LOGIC;
           btns : out STD_LOGIC_VECTOR (3 downto 0);
           C1 : out STD_LOGIC_VECTOR (3 downto 0);
           R1 : out STD_LOGIC_VECTOR (3 downto 0);
           C2 : out STD_LOGIC_VECTOR (3 downto 0);
           R2 : out STD_LOGIC_VECTOR (3 downto 0));
end PS2_CR;

architecture Behavioral of PS2_CR is
signal prev_valid, flag: std_logic;
signal C1_int : STD_LOGIC_VECTOR (3 downto 0);
signal R1_int : STD_LOGIC_VECTOR (3 downto 0);
signal C2_int : STD_LOGIC_VECTOR (3 downto 0);
signal R2_int : STD_LOGIC_VECTOR (3 downto 0);
begin

process (clk, rst) begin
    if rst = '1' then
        flag <= '0';
        R1_int <= "0000";
        C1_int <= "0000";
        R2_int <= "0000";
        C2_int <= "0000";
    elsif rising_edge (clk)then
        prev_valid <= valid;
        if (prev_valid = '1' and valid = '0' and flag = '0') then
--push key the first time check data at falling edge
                case (data) is
                    when X"1D" =>   R1_int <= R1_int - 1;  --up1
                                    C1_int <= C1_int;
                                    R2_int <= R2_int;
                                    C2_int <= C2_int;
                                    flag <= '0' ;
                     when X"1C" =>   C1_int <= C1_int - 1; --left1
                                    R1_int <= R1_int;
                                    R2_int <= R2_int;
                                    C2_int <= C2_int;
                                    flag <= '0' ;
        when X"1B" =>   C1_int <= C1_int + 1; --right1
                        R1_int <= R1_int;
                        R2_int <= R2_int;
                        C2_int <= C2_int;
                        flag <= '0' ;
        when X"1A" =>   R1_int <= R1_int + 1; --down1
                        C1_int <= C1_int;
                        R2_int <= R2_int;
                        C2_int <= C2_int;
                        flag <= '0' ;
        when X"3C" =>    R2_int <= R2_int - 1; --up2
                         R1_int <= R1_int;
                         C1_int <= C1_int;
                         C2_int <= C2_int;
                         flag <= '0' ;
        when X"33" =>   C2_int <= C2_int - 1;  --left2
                        R1_int <= R1_int;
                        R2_int <= R2_int;
                        C1_int <= C1_int;
                        flag <= '0' ;
        when X"3B" =>   C2_int <= C2_int + 1;  --right2
                        R1_int <= R1_int;
                        C1_int <= C1_int;
                        R2_int <= R2_int;
                        flag <= '0' ;
        when X"31" =>   R2_int <= R2_int + 1; --down2
                        R1_int <= R1_int;
                        C1_int <= C1_int;
                        C2_int <= C2_int;
                        flag <= '0' ;
        when X"F0" =>   R2_int <= R2_int;  --release code
                        R1_int <= R1_int;
                        C1_int <= C1_int;
                        C2_int <= C2_int;
                        flag <= '1' ;       --set flag at release button
        when others =>  R1_int <= R1_int;
                        C1_int <= C1_int;
                        R2_int <= R2_int;
                        C2_int <= C2_int;
                        flag <= '0' ;
    end case;
```

```
            elsif (prev_valid = '1' and valid = '0' and flag = '1') then -- if flag is set, ignore the next button
            case (data) is
               when others =>  R1_int <= R1_int;
                               C1_int <= C1_int;
                               R2_int <= R2_int;
                               C2_int <= C2_int;
                               flag <= '0' ;                -- set flag back to '0'
       end case;
       end if;
   end if;
end process;

C1 <= C1_int;
R1 <= R1_int;
C2 <= C2_int;
R2 <= R2_int;

end Behavioral;
```

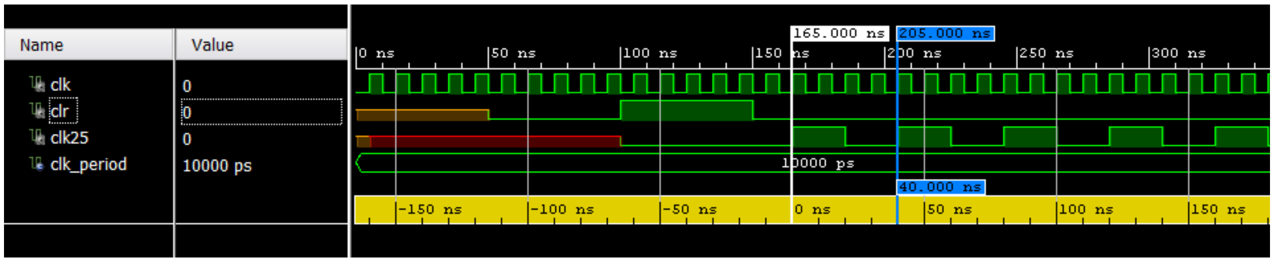
Test benches & Diagrams

U1: 25MHz clock signal

Testing the 25MHz generation is done by generating a 100MHz, period 10ns, clock signal. The component should put out a 25MHz, 40ns period, clock period.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity vga_640x480_tb is
end vga_640x480_tb;

architecture Behavioral of vga_640x480_tb is
        component vga_640x480 is
          port ( clk, clr : in std_logic;
                 hsync : out std_logic;
                 vsync : out std_logic;
                 hc : out std_logic_vector(9 downto 0);
                 vc : out std_logic_vector(9 downto 0);
                 vidon : out std_logic
                   );
        end component vga_640x480;
        signal clk, clr, hsync, vsync, vidon  : STD_LOGIC;
        signal hc, vc :  std_logic_vector(9 downto 0);
        constant clk_period: time := 40ns;
begin

UUT: vga_640x480  port map (clk => clk, clr => clr, hsync => hsync, vsync => vsync, vidon => vidon, hc
=> hc, vc => vc );

clk_proc: process begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
end process;

stim_proc: process begin
        wait for 50ns;
        clr <= '0';
        wait for 50ns;
        clr <= '1';
        wait for 50ns;
        clr <= '0';
        wait for 50ns;
        wait;
end process;

end Behavioral;
```

It is important to give a clr signal, as the 25MHz signal is derived from a counter in the clk_div component. It is clear that the simulator does not know what the outcome is for an addition to an unknown signal.

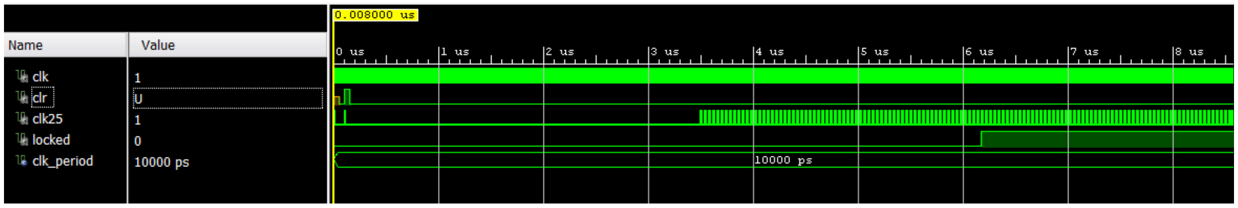


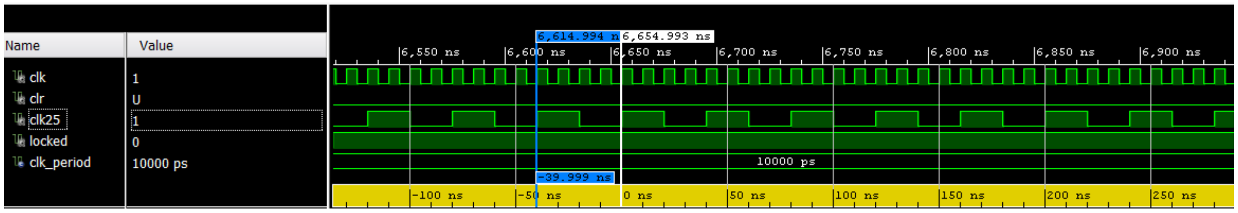
In the diagram it is clear that the timing between the two markers is correct, being a period of 40ns.

The signal clr is uninitialized (orange) until 50ns, when it is made ‘0’. After 50ns it is made ‘1’ and after 50ns it is ‘0’ again.

Until the first clk flank clk25 is uninitialized. Starting from this first flank a mathematical operation is performed internally in the clk_div component, which results in the red output, because adding to “unknown” results in a X (strong unknown). This is resolved after the first clr signal.

If we use the Multi Media Clock Manager, through the clocking wizard we see that it takes some time for the output clock to appear. And it takes even more time for the locked signal to be one. This is an estimate made by the tools, on the actual physical behavior of the MMCM. The period of the output clock of 25MHz is of course as it should be. This is visible in the diagram below.



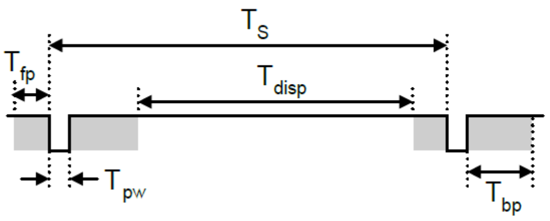


U2: Synchronization

Synchronization is tested by adding a 25MHz, period 40ns, to the component vga_640x480. To do so, we change the constant clk_period to 40ns and make the clk half a period ‘0’ and half a period ‘1’.

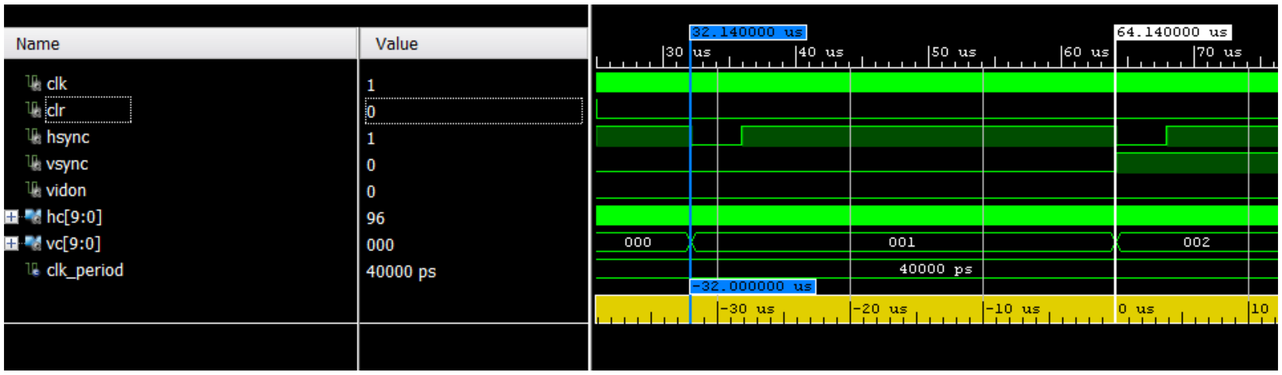
We wait for 50ns to make the clr signal ‘0’, after 50ns we make it ‘1’ and again after 50ns ‘0’ again.

Correct operation of this block is tested by checking the timing of the vsync and hsync, and to check if vidon is 1 during display time.

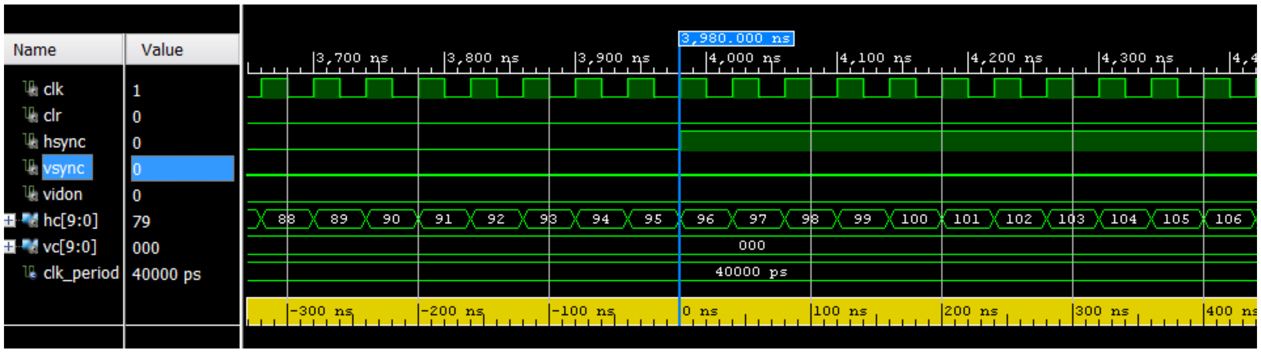


Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

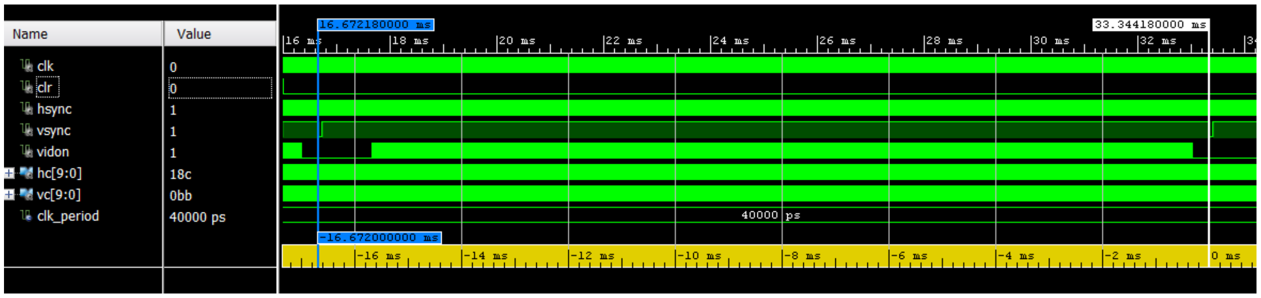
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity vga_640x480_tb is
end vga_640x480_tb;

architecture Behavioral of vga_640x480_tb is
            component vga_640x480 is
               port ( clk, clr : in std_logic;
                     hsync : out std_logic;
                     vsync : out std_logic;
                     hc : out std_logic_vector(9 downto 0);
                     vc : out std_logic_vector(9 downto 0);
                     vidon : out std_logic);
            end component vga_640x480;

            signal clk, clr, hsync, vsync, vidon  : STD_LOGIC;
            signal hc, vc :  std_logic_vector(9 downto 0);
            constant clk_period: time := 40 ns;
begin
UUT: vga_640x480  port map (clk => clk, clr => clr, hsync => hsync, vsync => vsync, vidon => vidon, hc =>
hc, vc => vc );

clk_proc: process begin
            clk <= '0';
            wait for clk_period/2;
            clk <= '1';
            wait for clk_period/2;
end process;

stim_proc: process begin
wait for 50ns;
clr <= '0';
wait for 50ns;
clr <= '1';
wait for 50ns;
clr <= '0';
wait for 50ns;
wait;
end process;
end Behavioral;
```



In this diagram the timing of the horizontal pulses is checked. With a 40ns period and 800 pixels per line we arrive indeed at 32 µs period.

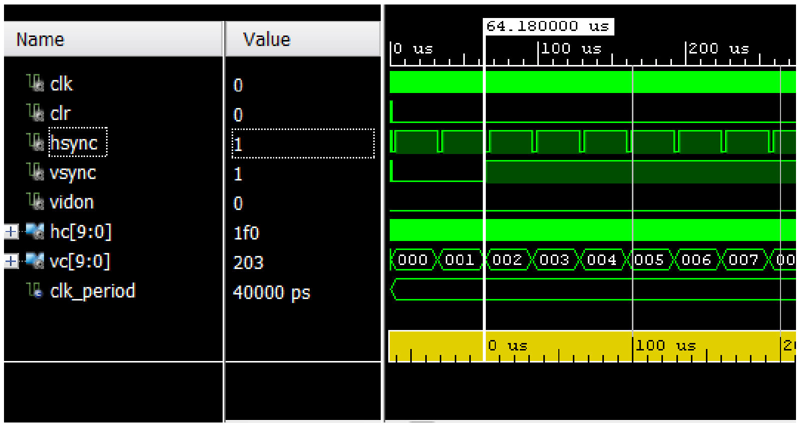


In this diagram it is clear that hsync goes high if the horizontal counter reaches 96

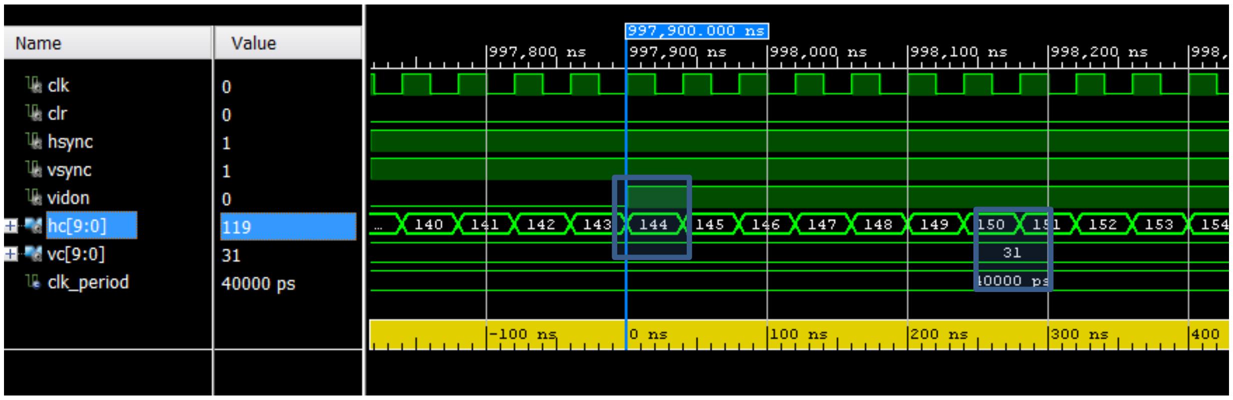


In this diagram the timing vsync is displayed between the two markers, being 16.67ms, according to the VGA 640x480 60Hz standard.

In between vsync pulses a complete frame is put on the display. The vidon signal is clearly 1 during a limited time during frame time.



If we look more closely at the vsync, we see that it goes high when the vertical counter vc is 2, after 2 lines.



Zoming on the vidon signal we see that vertical counter vc needs to be 31 (2 + 29) and that horizontal counter hc needs to be 144 for vidon to go ‘1’.

U3: VGA output

As this component is the core component for determining the autput of the RGB signal, its operation is best tested with testing of the complete top design.

U4: Block ROM

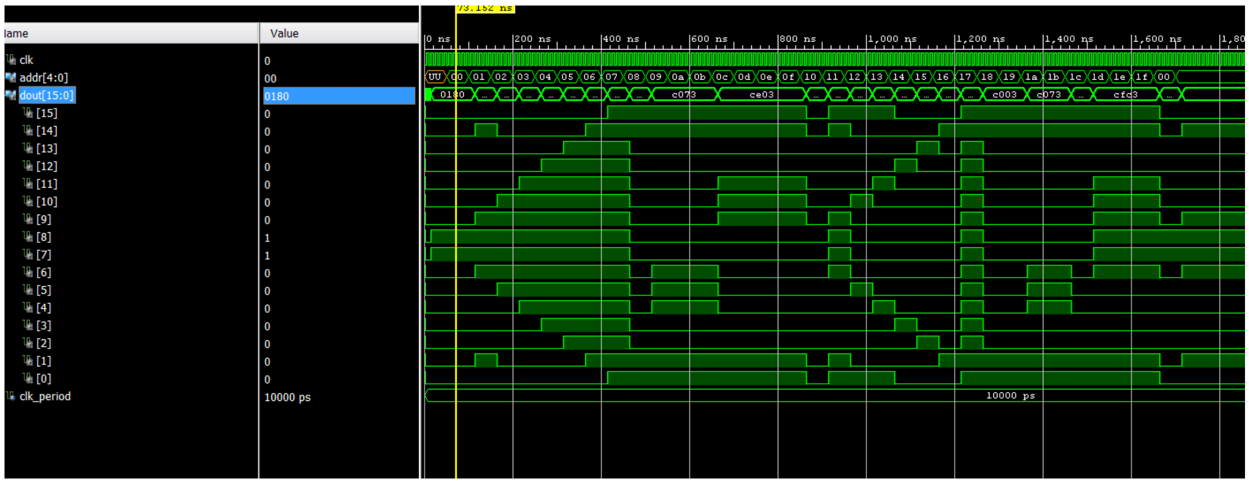
This component is tested by running through all different addresses of the memory. The content of the memory initialization file, the COE-file, is the following:

```
memory_initialization_radix = 2;
memory_initialization_vector =
0000000110000000
0100001111000010
0000011111100000
0000111111110000
0001111111111000
0011111111111100
0111111111111110
1111111111111111
1100000000000011
1100000001110011
1100000001110011
1100000001110011
1100111000000011
1100111000000011
1100111000000011
1100111000000011
0000000000000000
1100001111000011
1000010000100001
1000100000010001
0001000000001000
0010000000000100
0100000000000010
1111111111111111
1100000000000011
1100000000000011
1100000001110011
1100000001110011
1100000000000011
1100111111000011
1100111111000011
1100111111000011
```

In this case it is used to draw the pictures of two houses, sprite 1 and sprite 2.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity blk_mem_gen_0_tb is
end blk_mem_gen_0_tb;

architecture Behavioral of blk_mem_gen_0_tb is

        COMPONENT blk_mem_gen_0
          PORT (
            clka : IN STD_LOGIC;
            addra : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
            douta : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
          );
         END COMPONENT;

        signal clk: STD_LOGIC;
        signal addr : STD_LOGIC_VECTOR(4 DOWNTO 0);
        signal dout : STD_LOGIC_VECTOR(15 DOWNTO 0);

        constant clk_period: time := 10 ns;
```

```
begin

UUT: blk_mem_gen_0  port map (clka => clk, addra => addr, douta => dout);

clk_proc: process begin
            clk <= '0';
            wait for clk_period/2;
            clk <= '1';
            wait for clk_period/2;
end process;

stim_proc: process begin
            wait for 50ns;
            addr <=  "00000";
            for i in 0 to 32 loop
                        wait for 50ns;
                        addr <=  addr +1;
            end loop;
            wait;
end process;

end Behavioral;
```

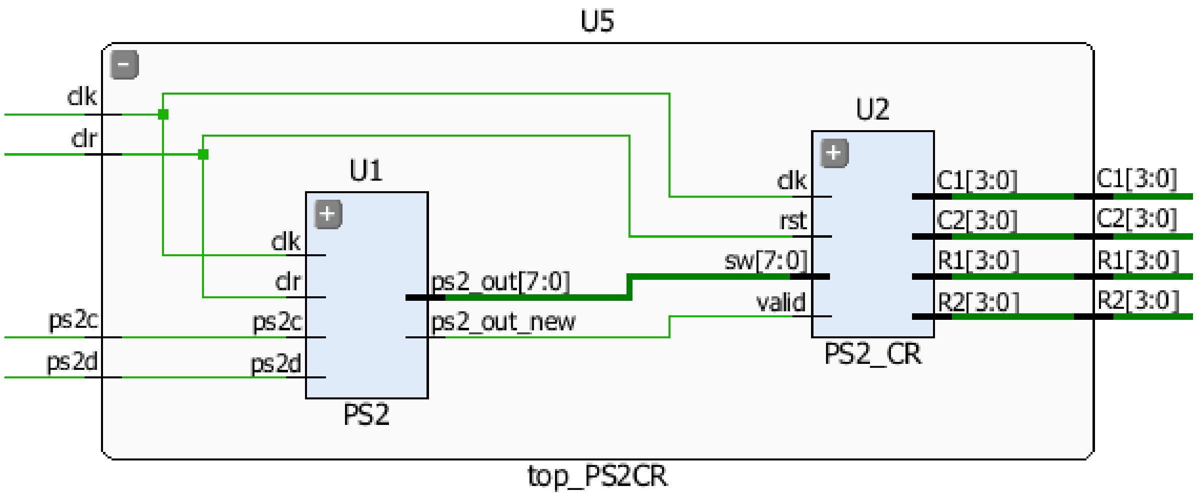


If we look at the address in the diagram it shows that the address starts with 50ns being “initialized” (orange), the it is put on “00000” and each 50ns one is added to the address until the output is “1 1111 1111” or X”1F”.

Looking at the diagram, we can clearly see the both figures from the COE file being drawn from left to right.

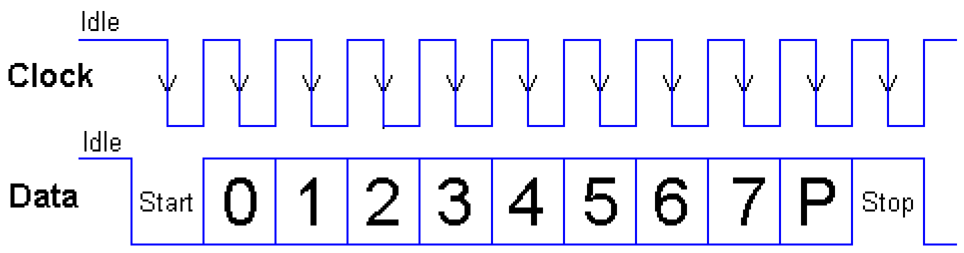
U5: PS2 to CR information

This component exists of two sub-components. One component U1: PS2 converts the PS2 clock and PS2 data into the scan code output, ps2_out, and gives a flag if the data is valid, ps2_out_new.



To test the code we need to send the proper the PS2 clock and PS2 data to the component. The keyboard provides both the clock and data. The clock has a frequency between 10 kHz and 16.7 kHz (i.e. a 60-100us period). The data begins with a start bit (logic low), followed by one byte of data, a parity bit, and finally a stop bit (logic high). The data is sent LSB first.

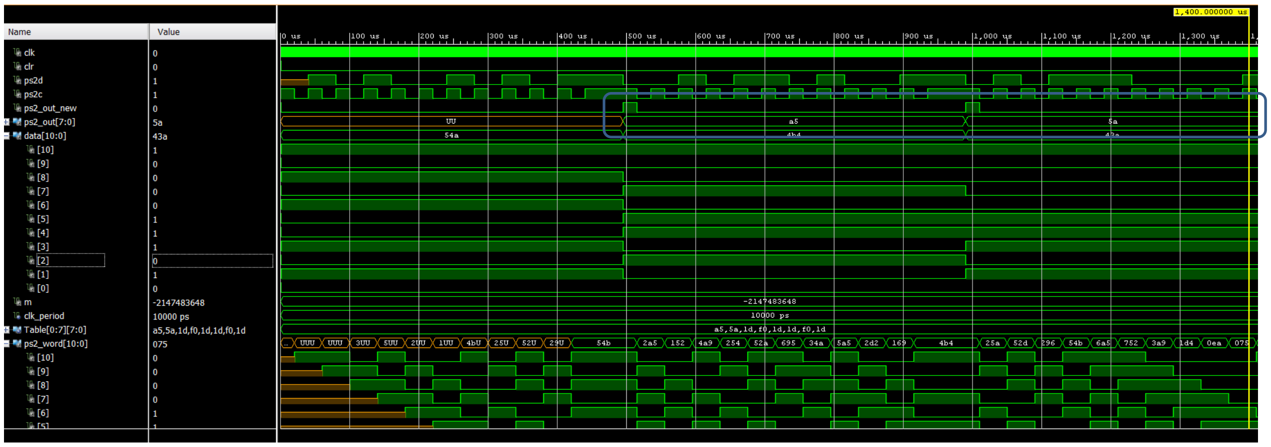
We will use a 40 us period and keep the clock idle after transfer during 55us.



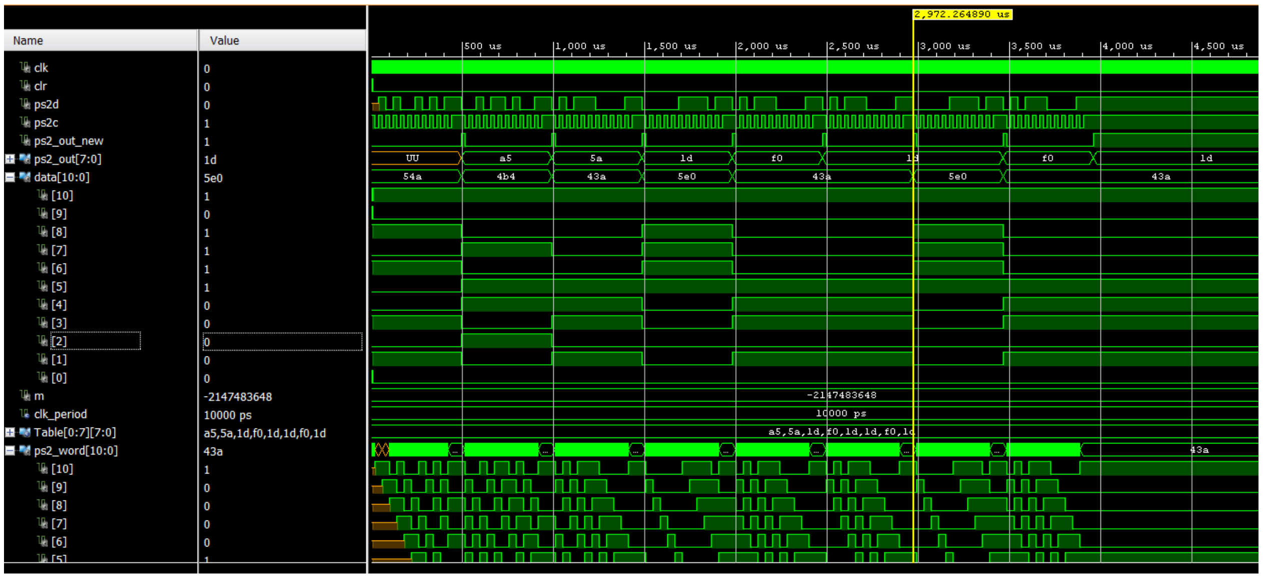
Correct operation is proven if the ps2_out displays the correct output data and the ps2_out_new flag is set. For generic reason, an array is added with test values of the PS2 data.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity PS2_tb is
end PS2_tb;

architecture Behavioral of PS2_tb is

component PS2 is
   Generic(clk_freq: INTEGER := 100_000_000); --system clock frequency in Hz
   Port ( clk : in STD_LOGIC;
          clr : in STD_LOGIC;
          ps2d : in STD_LOGIC;
          ps2c : in STD_LOGIC;
--flag that new PS2 output is available on ps2_code bus
          ps2_out_new : out STD_LOGIC;
--output received from PS2
          ps2_out : out STD_LOGIC_VECTOR (7 downto 0));
end component PS2;

signal clk, clr, ps2d, ps2c, ps2_out_new: STD_LOGIC;
signal ps2_out : STD_LOGIC_VECTOR (7 downto 0);
signal data : STD_LOGIC_VECTOR (10 downto 0);
signal m : integer;

constant clk_period: time := 10 ns;

type Codes_Table is array (natural range <>) of STD_LOGIC_VECTOR (7 downto 0);
constant Table : Codes_Table := ( X"A5", X"5A", X"1D", X"F0", X"1D",  X"1D", X"F0", X"1D" );
--table of different keys to be pushed
```

```
begin

UUT: PS2  port map (clk => clk, clr => clr, ps2d => ps2d, ps2c => ps2c, ps2_out_new => ps2_out_new,
ps2_out => ps2_out);

clk_proc: process begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
end process;

stim_proc: process begin
ps2c <= '1';
wait for 50ns;
clr <=  '0';
wait for 50ns;
clr <=  '1';
wait for 50ns;
clr <=  '0';
wait for 50ns;

for j in 0 to 7 loop                        --amount of keys to be pushed
data <= '1'& '0' & table (j)  & '0';        --stop - parity - data - start
   for i in 0 to 10 loop                    --11 bit word coming from PS2 keyboard
      ps2d <= data (i);
      wait for 20us;
      ps2c <= '0';
      wait for 20us;
      ps2c <= '1';
   end loop;
wait for 55us;
end loop;

wait;

end process;
end Behavioral;
```



In the diagram it is clear that new data is made available on ps2_out and the ps2_out_new flag is set for a certain amount of time. To perform an action on the incoming scan code, the user should poll on rising or falling edge of ps2_out_new and look at the momentary value of ps2_out.



In this diagram we can see the output at ps2_out of the different values from testbench.

```
type Codes_Table is array (natural range <>) of STD_LOGIC_VECTOR (7 downto 0);
constant Table : Codes_Table := ( X"A5", X"5A", X"1D", X"F0", X"1D",  X"1D", X"F0", X"1D" );
```

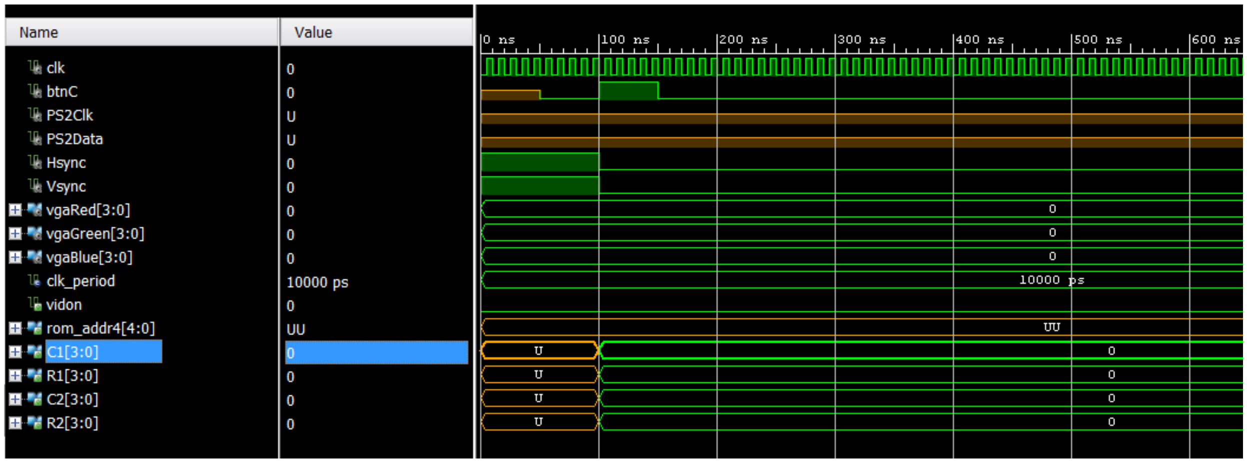
This can be used to perform actions on this output.

Top design

In this simulation we simulate the complete operation of the design. We leave the input of the keyboard as it is, we do not change anything in this, so the position of the two sprites is not changed.

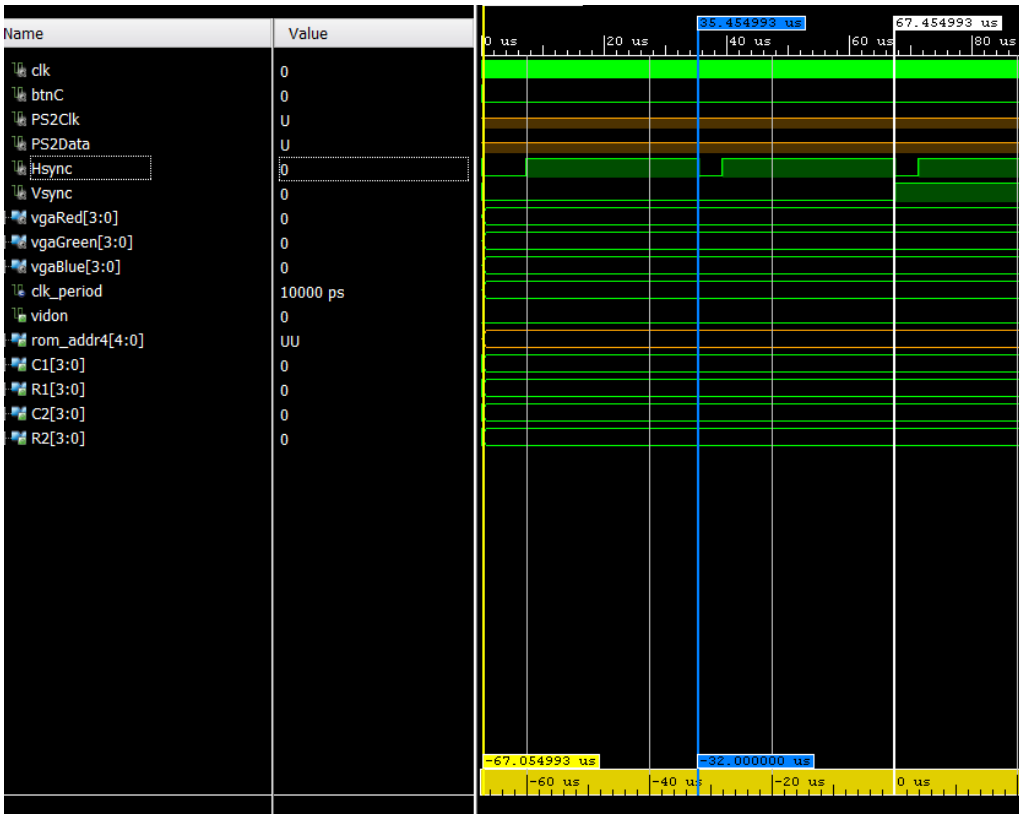
If we look at the port of the top design we see that after an initial reset and applying the clock signal, the complete design should run automatically.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity top_tb is
end top_tb;

architecture Behavioral of top_tb is

component vga_initials_top is
        port(
                clk : in STD_LOGIC;
                btnC : in STD_LOGIC;
                PS2Clk : in STD_LOGIC;
                PS2Data : in STD_LOGIC;
                 Hsync : out STD_LOGIC;
                Vsync : out STD_LOGIC;
                vgaRed : out std_logic_vector(3 downto 0);
                 vgaGreen : out std_logic_vector(3 downto 0);
                vgaBlue : out std_logic_vector(3 downto 0)
          );
end component vga_initials_top;
```



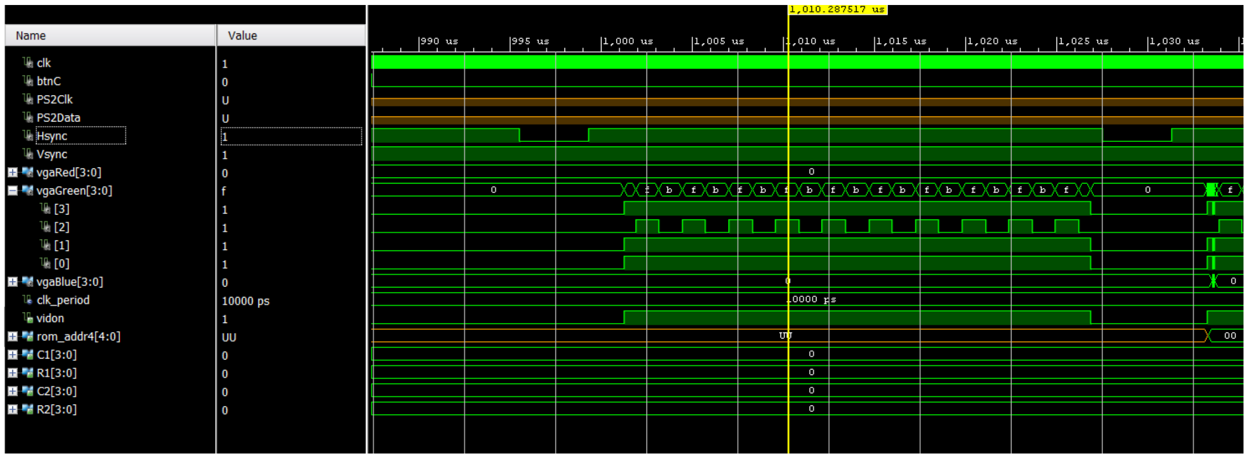
Name	Value
clk	0
btnC	0
PS2Clk	U
PS2Data	U
Hsync	0
Vsync	0
vgaRed[3:0]	0
vgaGreen[3:0]	0
vgaBlue[3:0]	0
clk_period	10000 ps
vidon	0
rom_addr4[4:0]	UU
C1[3:0]	0
R1[3:0]	0
C2[3:0]	0
R2[3:0]	0

In this diagram we see the clock and the reset, which has its effect on Hsyn, Vsync, C1-R1 and C2-R2.

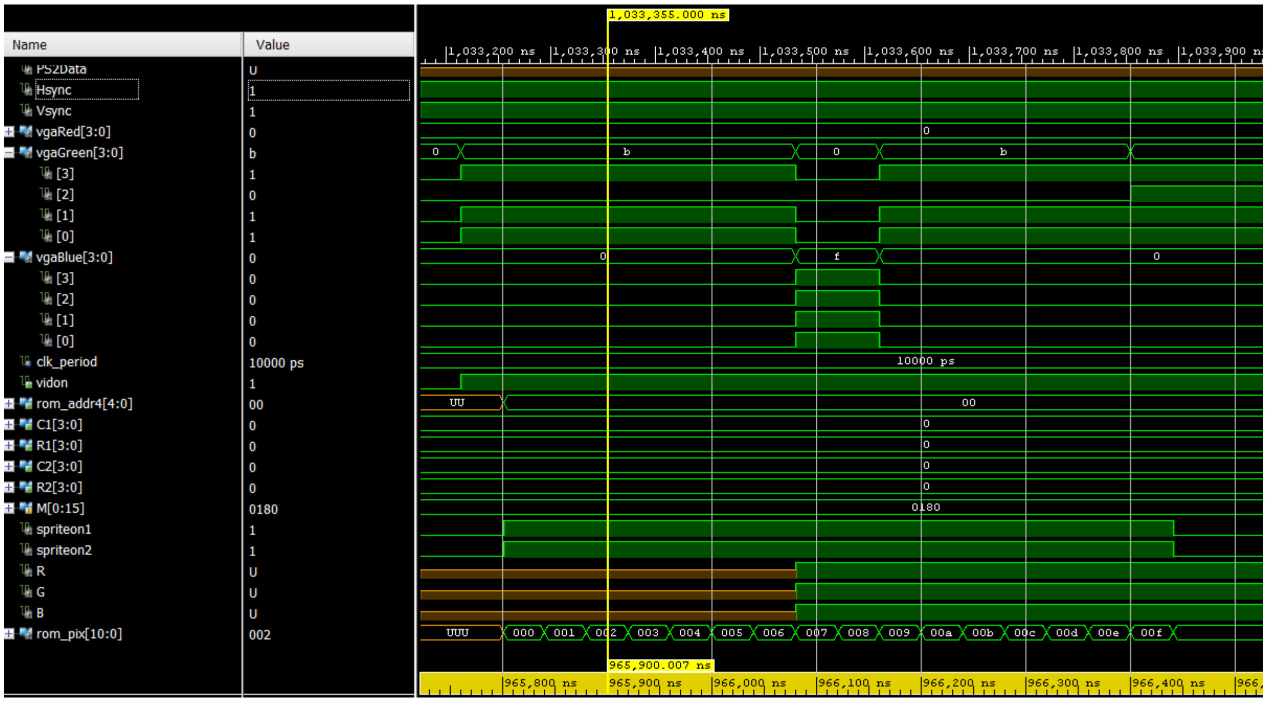


Name	Value
clk	0
btnC	0
PS2Clk	U
PS2Data	U
Hsync	0
Vsync	0
vgaRed[3:0]	0
vgaGreen[3:0]	0
vgaBlue[3:0]	0
clk_period	10000 ps
vidon	0
rom_addr4[4:0]	UU
C1[3:0]	0
R1[3:0]	0
C2[3:0]	0
R2[3:0]	0

We see the Vsync and Hsync reappear with the proper timing.



In this diagram we can clearly see that during the time the vidon is ‘1’ the green background is drawn, from completely green “1111” or HEX ”f”, to a little darker “1011” or HEX ”b”.



In this diagram you can see that rom_pix counts until it reaches the first ‘1’ in M, which is “00011000”. This happens at 7 pulses and this makes vgaBlue completely blue “1111” or HEX “f”. Next step is also ‘1 at the next position. Then vgaBlue becomes “0000” again.

Sources

Digital Design Using Digilent FPGA Boards *Richard E. Haskell / Darrin M. Hanna* LBE Books – Third Edition, 2014

http://www.allaboutcircuits.com/textbook/digital/chpt-4/contact-bounce/

https://eewiki.net/pages/viewpage.action?pageId=28278929&preview=/28278929/28508225/ps2_keyboard.vhd#PS/2KeyboardInterface%28VHDL%29-CodeDownloads