Engineering Context ING-INF/01
Department of information engineering,
electronics and telecommunications (DIET)
Università la Sapienza

Video Pattern Generator on VGA using Zybo Board

Andrea Manzini (1597905), Lorenzo Lazzara (1615399), Francesco Arnò (1595382)

Date submitted: July 19, 2017

SAPIENZA
UNIVERSITÀ DI ROMA

Table of Contents

Chapter 1

Overview

This chapter introduces the project and provides its basic features.

About the Design

The aim of the project was to create a VGA Controller which dictates the resolution by producing timing signals to control the raster patterns. Consequently we created a custom block which provide the RGB signals to create a simple video pattern on the screen.
The project has been realized using both the Processing System (PS) and the Programmable Logic (PL) of the Zybo Board [Ref 1]. In the PL side, we configured the FPGA to drive the VGA connector on the board, synthesizing the VGA Controller Module and the Video Pattern Generator. In the PS side, we installed an Embedded Linux developed with Petalinux. This allowed us to create in Linux a script to change the resolution through command line at runtime. The proposed permanent Linux environment can be used to easily add other functionalities.

Feature Summary

- **Four Display Resolutions.** Possibility to choose one of the following resolutions: **VGA** (640x480, 60 Hz), **XGA** (1024x768, 60 Hz), **HD** 720p (1280x720, 60Hz), **FHD** 1080p (1920x1080, 60Hz).

- **Dynamic Reconfiguration**. This feature allows you to change the resolution displayed on the screen through the reconfiguration of two registers by an AXI4-Lite interface.

- **VGA Synchronism Signals Generator.** Generation of vertical synchronization signal and horizontal synchronization signal with the specific timing configurations (Display Time, Back Porch, Pulse Length, Pulse Polarity, Front Porch) for different resolutions.

- **Video Pattern Generator.** Generation of a simple pattern consisting of eight horizontal bars coloured with the combinations of primary colours of the RGB model.

Port Descriptions

Table 1-1 describes the input and output ports provided from the VGA module.

Table 1-1: **VGA Module I/O**

Port	I/O	Description
vga_r[4:0]	Output	**Red colour signal**: conveys picture information for the red component of an image.
vga_g[5:0]	Output	**Green colour signal**: conveys picture information for the green component of an image.
vga_b[4:0]	Output	**Blue colour signal**: conveys picture information for the blue component of an image.
vga_hs	Output	**VGA Horizontal Sync Signal**: sets retrace frequency.
vga_vs	Output	**VGA Vertical Sync Signal**: defines the "refresh" frequency of the display, or the frequency at which all information on the display is redrawn.

Register Space

Table 1-2 shows the set of registers used in the Bus AXI4-Lite. All registers are accessed as 32-bit. The table mentions the default value of the registers and a brief description about them.

Table 1-2: **AXI4-Lite Configurations Registers**

Base Address + Offset (hex)	Register Name	Reset Value (hex)	Access Type	Description
C_BASEADRR + 0x0	reg0	0x00000000	W	Bus AXI4-Lite Register. Contains timing informations of the selected resolution: Bit[10:0]=Horizontal Display Bit[18:11]=Horizontal Back Porch Bit[26:19]=Horizontal Pulse Bit[30:27]=Vertical Front Porch
C_BASEADDR + 0x4	reg1	0x00000000	W	Bus AXI4-Lite Register. Contains timing informations of the selected resolution: Bit[6:0]=Horizontal Front Porch Bit[17:7]=Vertical Display Bit[23:18]=Vertical Back Porch Bit[26:24]=Vertical Pulse Bit[28:27]=Pixel Clock Bit[29]=Pulse Polarity

Chapter 2

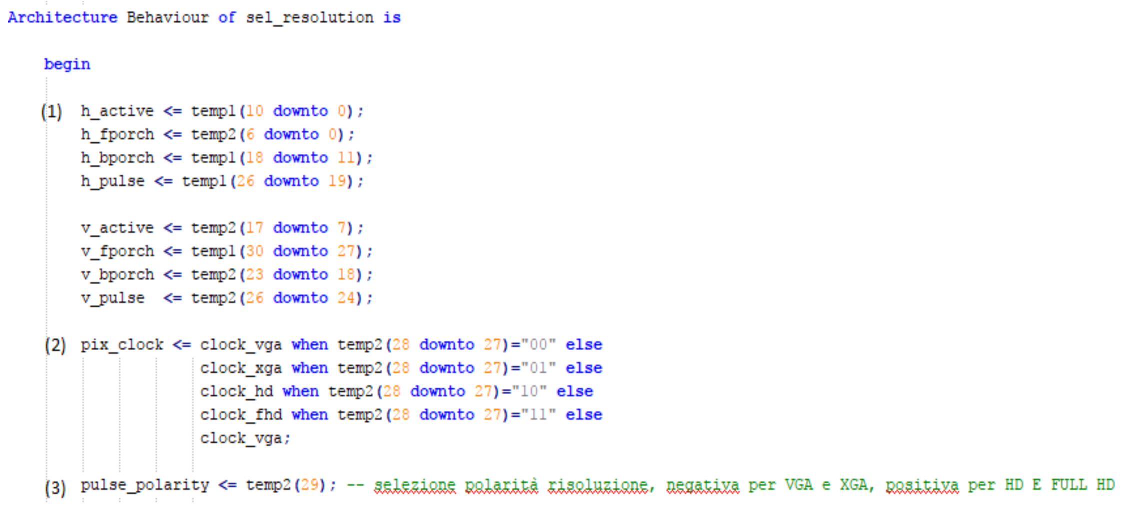
VHDL Code for VGA

This chapter describes the VHDL files for generating VGA timing signals [Ref 6], image generation and resolution selection.

Resolution Selection

This VHDL module is dedicated to set timing parameters, based on the resolution chosen.

Figure 2-1: **Video Parameters Assignment**



```
Architecture Behaviour of sel_resolution is

   begin

  (1) h_active <= temp1(10 downto 0);
      h_fporch <= temp2(6 downto 0);
      h_bporch <= temp1(18 downto 11);
      h_pulse <= temp1(26 downto 19);

      v_active <= temp2(17 downto 7);
      v_fporch <= temp1(30 downto 27);
      v_bporch <= temp2(23 downto 18);
      v_pulse  <= temp2(26 downto 24);

  (2) pix_clock <= clock_vga when temp2(28 downto 27)="00" else
                   clock_xga when temp2(28 downto 27)="01" else
                   clock_hd when temp2(28 downto 27)="10" else
                   clock_fhd when temp2(28 downto 27)="11" else
                   clock_vga;

  (3) pulse_polarity <= temp2(29); -- selezione polarità risoluzione, negativa per VGA e XGA, positiva per HD E FULL HD
```

(1) The module interfaces with the outputs registers of bus AXI, named temp1 and temp2 in the code. Those two registers contain a string of bit that specifies the value of timing parameters about vertical and horizontal scan of image, i.e. active area, sync pulse.

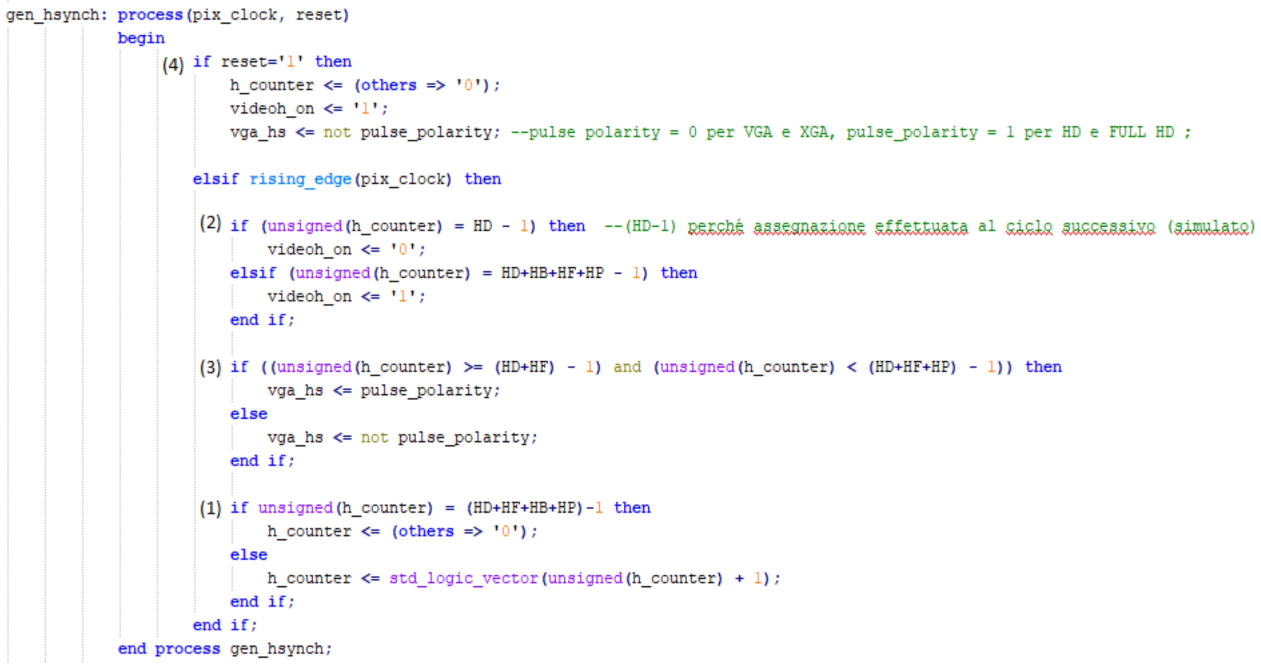
(2) The pixel clocks of the four resolution are the input of a multiplexer. Two bits (28th and 27th) of temp2 are the control input of this multiplexer. They are set in order to choose the correct clock frequency.

(3) The 29th bit of temp2 is used to set the polarity of sync pulse. It is negative for VGA and XGA, positive for HD and FULL HD.

VGA Synchronisms

The generation of synch signals needs two counters, named *h_counter* and *v_counter.* They are used in two different process, one for the horizontal scan and the other one for the vertical scan of the image. The entity outputs are two synch signals, called *vga_hs* e *vga_vs* and the *video_on signal,* active high, which enables the transmission of color signal. *Video_on* is the *AND* output between two internal signals, *videoh_on* and *videov_on ,* refered to horizontal synch and vertical synch, respectively.

Figure 2-2: **Horizontal counter**



```
gen_hsynch: process(pix_clock, reset)
            begin
                (4) if reset='1' then
                        h_counter <= (others => '0');
                        videoh_on <= '1';
                        vga_hs <= not pulse_polarity; --pulse polarity = 0 per VGA e XGA, pulse_polarity = 1 per HD e FULL HD ;

                    elsif rising_edge(pix_clock) then

                     (2) if (unsigned(h_counter) = HD - 1) then  --(HD-1) perché assegnazione effettuata al ciclo successivo (simulato)
                            videoh_on <= '0';
                        elsif (unsigned(h_counter) = HD+HB+HF+HP - 1) then
                            videoh_on <= '1';
                        end if;

                     (3) if ((unsigned(h_counter) >= (HD+HF) - 1) and (unsigned(h_counter) < (HD+HF+HP) - 1)) then
                            vga_hs <= pulse_polarity;
                        else
                            vga_hs <= not pulse_polarity;
                        end if;

                     (1) if unsigned(h_counter) = (HD+HF+HB+HP)-1 then
                            h_counter <= (others => '0');
                        else
                            h_counter <= std_logic_vector(unsigned(h_counter) + 1);
                        end if;
                    end if;
            end process gen_hsynch;
```

Note: HD=Horizontal Active Area, **HF**=Horizontal Front Porch, **HB**=Horizontal Back Porch, **HP**=Horizontal Synch Pulse.

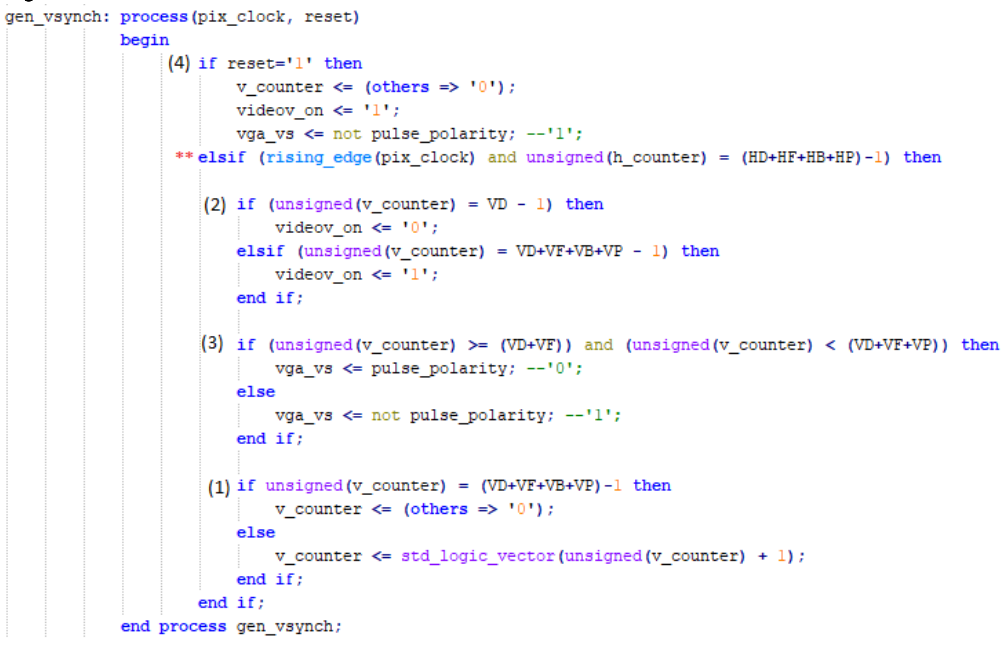
(1) The horizontal counter starts from zero and it increases every pixel clock rising edge. It needs to control the value of *h_counter*, in order to reset it when it reaches the final value of each horizontal line. This value is the sum of four terms: HD, HF, HB, HP. They are provided from the VGA controller, based on the resolution chosen.

(2) When the horizontal counter reaches the end of active area (HD -1), the horizontal enable signal, *videoh_on,* must be deactivated, because the counter will be out of the active zone in the next pixel clock cycle. It will be re-assigned to ‘1’ at the end of the row (HD+HB+HF+HP -1), therefore it will active again at the beginning of the next row.

(3) The *pulse_polarity* signal is an input of the entity and it specifies the polarity of synch pulse. It is based on the resolution chosen. *Vga_hs* is the horizontal synch signal. It is assigned to *pulse_polarity* when the counter reaches the value: HD + HF – 1, and it keeps this value for HP increases of the signal *h_counter*. Then it is re-assigned to *not pulse_polarity* value.

(4) The Asynchronous Reset signal zeros the counter, it deactivates *vga_hs*, by assigning it *not pulse_polarity* value*,* and it enables *videoh_on.*

Figure 2-3: **Vertical Counter**



```
gen_vsynch: process(pix_clock, reset)
            begin
            (4) if reset='1' then
                    v_counter <= (others => '0');
                    videov_on <= '1';
                    vga_vs <= not pulse_polarity; --'1';
             ** elsif (rising_edge(pix_clock) and unsigned(h_counter) = (HD+HF+HB+HP)-1) then

                (2) if (unsigned(v_counter) = VD - 1) then
                        videov_on <= '0';
                    elsif (unsigned(v_counter) = VD+VF+VB+VP - 1) then
                        videov_on <= '1';
                    end if;

                (3) if (unsigned(v_counter) >= (VD+VF)) and (unsigned(v_counter) < (VD+VF+VP)) then
                        vga_vs <= pulse_polarity; --'0';
                    else
                        vga_vs <= not pulse_polarity; --'1';
                    end if;

                (1) if unsigned(v_counter) = (VD+VF+VB+VP)-1 then
                        v_counter <= (others => '0');
                    else
                        v_counter <= std_logic_vector(unsigned(v_counter) + 1);
                    end if;
                end if;
            end process gen_vsynch;
```

Note: VD=Vertical Active Area, **VF**=Vertical Front Porch, **VB**=Vertical Back Porch, **VP**=Vertical Synch Pulse.

(1) The vertical counter starts from zero and it increases at the end of each horizontal line**. *V_counter* must be reset when it reaches the final value of a frame, that is *VD + VF + VB + VP -1.* Those values are provided from the external, based on the chosen resolution.

(2) *Videov_on* is the vertical enable signal. It is assigned to zero when the vertical counter reaches the end of active area*, VD -1*. Therefore, it will be deactivated in the next pixel clock cycle, preventing the transmission of the colour signal. It will be assigned to '1' when *v_counter* is equal to the final value, *VD+VB+VP -1.* Like this it will be active at the beginning of the next frame.

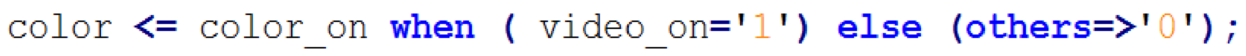
(3) The target of the control on v_counter is to keep active vga_vs, vertical synch signal, by assigning it the value of pulse_polarity, from the next cycle after the vertical front porch (v_counter > = VD + VF), to the previous cycle before the beginning of the back porch (v_counter < VD + VF + VB). After this value of *v_counter* it will be deactived.

(4) The asynchronous Reset signal zeros the vertical counter, it deactives synch vertical signal, *vga_vs*, by assigning it not pulse_polarity, and it enables *videov_on.*

Colour Pattern Generator

The VHDL module for the generation of the image provides a 16-bit output, called colour. The inputs include signal from the module for the generation of synch signal, like the two counters *h_counter and v_counter*, and *video_on signal*, that enables the transmission of color signal.

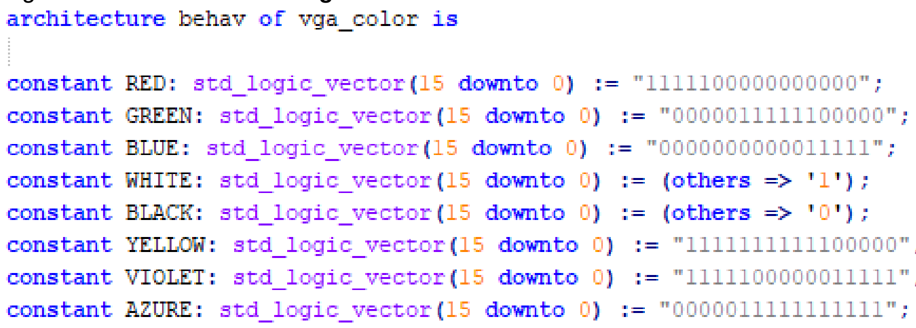
Figure 2-4: **video_on Signal**



```
color <= color_on when ( video_on='1') else (others=>'0');
```

The image displayed on the screen is a series of horizontal slices of eight different colours, encoded like this:

Figure 2-5: **Colours Encoding**

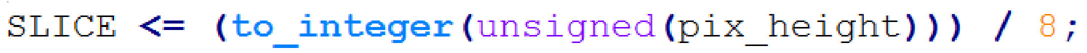


```
architecture behav of vga_color is

constant RED: std_logic_vector(15 downto 0)  := "1111100000000000";
constant GREEN: std_logic_vector(15 downto 0)  := "0000011111100000";
constant BLUE: std_logic_vector(15 downto 0)  := "0000000000011111";
constant WHITE: std_logic_vector(15 downto 0)  := (others => '1');
constant BLACK: std_logic_vector(15 downto 0)  := (others => '0');
constant YELLOW: std_logic_vector(15 downto 0)  := "1111111111100000";
constant VIOLET: std_logic_vector(15 downto 0)  := "1111100000011111";
constant AZURE: std_logic_vector(15 downto 0)  := "0000011111111111";
```

The standard used is **RGB** (Red, Green, Blu), so the most five significant bits are refered to red, the next six to green, and the last five to blue.

The code uses this constant,

Figure 2-6: **SLICE constant**

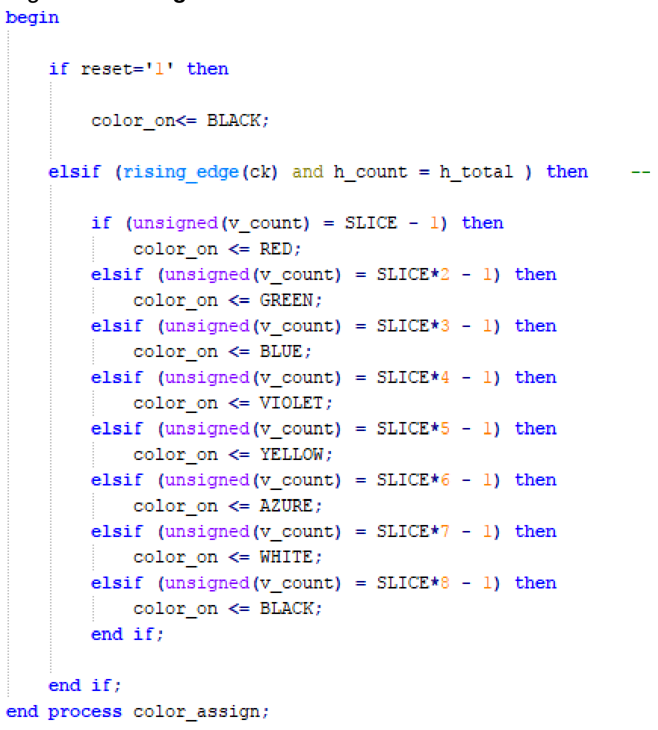


```
SLICE <= (to_integer(unsigned(pix_height))) / 8;
```

Pix_height is an output of *sel_resolution* module (*v_active*), and it refers to vertical width of the resolution chosen. The constant is used to divide the screen in a certain number of horizontal slices, which have the same width.

The algorithm for the generation of the image is shown below:

Figure 2-7: **Image Generation**



```
begin

    if reset='1' then

        color_on<= BLACK;

    elsif (rising_edge(ck) and h_count = h_total ) then    --

        if (unsigned(v_count) = SLICE - 1) then
            color_on <= RED;
        elsif (unsigned(v_count) = SLICE*2 - 1) then
            color_on <= GREEN;
        elsif (unsigned(v_count) = SLICE*3 - 1) then
            color_on <= BLUE;
        elsif (unsigned(v_count) = SLICE*4 - 1) then
            color_on <= VIOLET;
        elsif (unsigned(v_count) = SLICE*5 - 1) then
            color_on <= YELLOW;
        elsif (unsigned(v_count) = SLICE*6 - 1) then
            color_on <= AZURE;
        elsif (unsigned(v_count) = SLICE*7 - 1) then
            color_on <= WHITE;
        elsif (unsigned(v_count) = SLICE*8 - 1) then
            color_on <= BLACK;
        end if;

    end if;
end process color_assign;
```

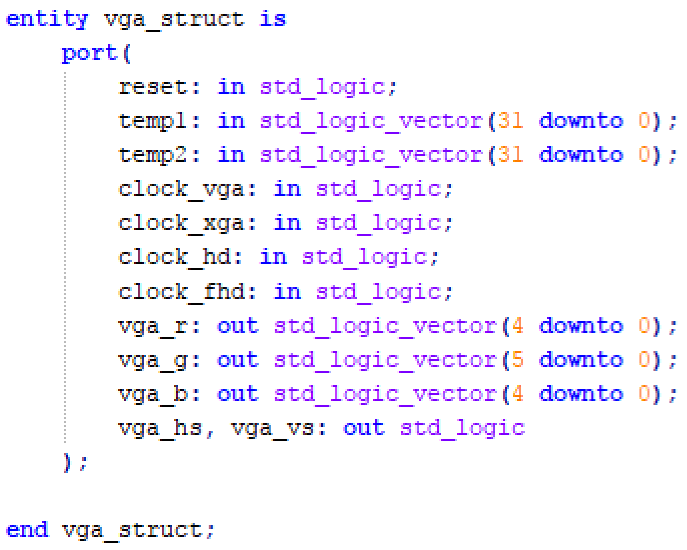
The generation of image is implemented in a unique process, sensitive to pixel clock and reset. At the end of each horizontal line, (*h_count= h_total)*, the internal signal *color_on* is assigned to one of eight colours decoded before. There is a control on *v_counter* to generate the horizontal slices of different colours. Only when *v_counter* reaches the final value of a “slice”, a new colour is assigned to color_on, otherwise *color_on* assumes the same value of the previous cycle, in order to fill up each slice. An asynchronous reset sets black as default colour.

Structural VHDL

This is the structural VHDL that allows the communication between the ‘blocks’ described in the previous paragraphs: **Sel_resolution, Vga_synch** and **Colour_pattern_generation.**

The global inputs and outputs of the entire block are the following:

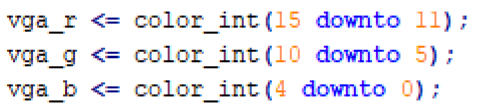
Figure 2-8: **Entity I/O**



```
entity vga_struct is
    port(
        reset: in std_logic;
        temp1: in std_logic_vector(31 downto 0);
        temp2: in std_logic_vector(31 downto 0);
        clock_vga: in std_logic;
        clock_xga: in std_logic;
        clock_hd: in std_logic;
        clock_fhd: in std_logic;
        vga_r: out std_logic_vector(4 downto 0);
        vga_g: out std_logic_vector(5 downto 0);
        vga_b: out std_logic_vector(4 downto 0);
        vga_hs, vga_vs: out std_logic
    );

end vga_struct;
```

The three color signals *vga_r, vga_g* and *vga_b* are taken from the 16-bit output *color* of **Colour_pattern_generation,** in the following way:

Figure 2-9: **VGA RGB Assignment**



```
vga_r <= color_int(15 downto 11);
vga_g <= color_int(10 downto 5);
vga_b <= color_int(4 downto 0);
```

Note: the signal *color* has been assigned to an internal signal *color_int.*

Design Flow Steps

This chapter describes the customization and generation of the design.

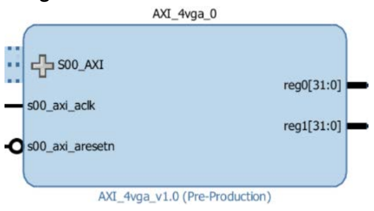
Create an AXI Block IP

After the creation of the VHDL files, the next step is linking the PS side with the PL side through an AXI4-Lite bus interface. To create the custom block with AXI interface use this guide: "*How to Control Peripherals on FPGA by AXI* "[Ref 2] with the following modifications:

- When creating the new project select the Zybo Board not the Zed Board (If the Zybo Board is not listed, follow this guide: "*Vivado Version 2015.1 and Later Board File Installation*" [Ref 3]).

- Don't add the VHDL files created in the previous chapter (you will only create an interface between AXI bus and the VHDL module).

Figure 3-1: **AXI Block**



- In **Tutorial_AXI_IP_v1_0_S00_AXI.vhd** add after
  reg_AXI_0: out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  the declaration of a second port (we need two registers)
  reg_AXI_1: out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);

- In **Tutorial_AXI_IP_v1_0_S00_AXI.vhd** add after
  reg_AXI_0 <= slv_reg0;
  the second linking
  reg_AXI_1 <= slv_reg1;

- In **Tutorial_AXI_IP_v1_0.vhd** add instead of
  LEDs_out: out std_logic_vector(7 downto 0);
  the declaration of the two output ports
  reg0: out std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
  reg1: out std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);

- In **Tutorial_AXI_IP_v1_0.vhd** add after
  reg_AXI_0: out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
  the second port declaration
  reg_AXI_1: out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);

- Now link the two ports adding around line 95 the following lines
  reg_AXI_0 => reg0,
  reg_AXI_1 => reg1,

- Don’t add the custom logic component in structural VHDL and the other following signal. Continue from “*That concludes the VHDL part”.*

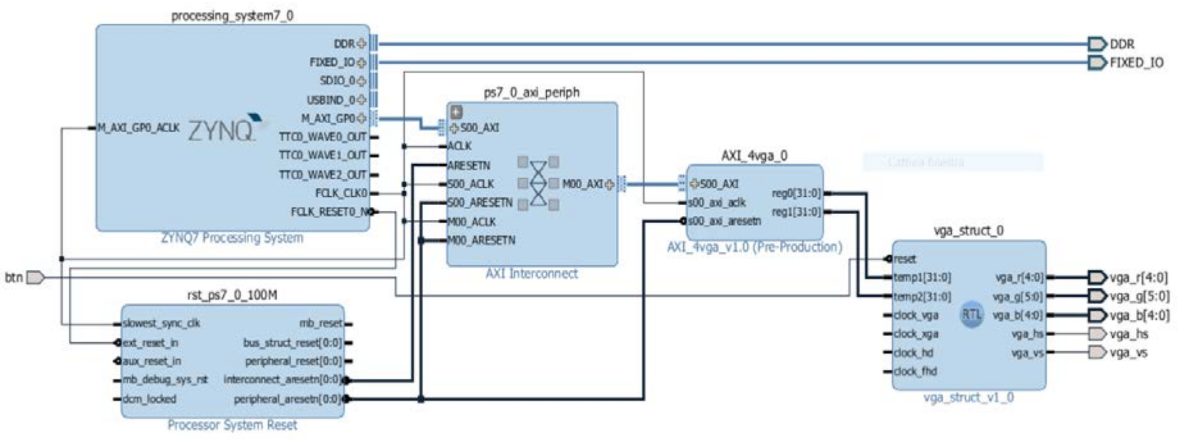
- Continue the paragraph 2.4 and stop at “*The block diagram has also a processor System Reset, able to control resets of our system”.*

Now we have to insert the module described by the VHDL files of the previous chapter. In *flow navigator tab* select **Add Sources,** check **Add or create design sources** and click *next*, select **Add Files** and import the files created in Chapter 2 then click *finish*. The imported files will be displayed in *sources tab, right click* on the structural file and select **Add Module to Block Design**.

Note: If you use the statements of VHDL 2008 in your VHDL files you have to change the file type in Vivado: Expand the struct file*, Right Click* on the file you need to change and select the option **Set File type...**, on the drop-down menu select **VHDL 2008** and then *Click OK*. **Your structural VHDL file must be written in VHDL not VHDL 2008.**

Now make the right connections between the modules as shown in figure below:

Figure 3-2: **Modules interconnection**



Note: To create the ports (vga_hs, vga_vs, ….), select the signal name in the module and press ctrl+T

Clocking Wizard IP

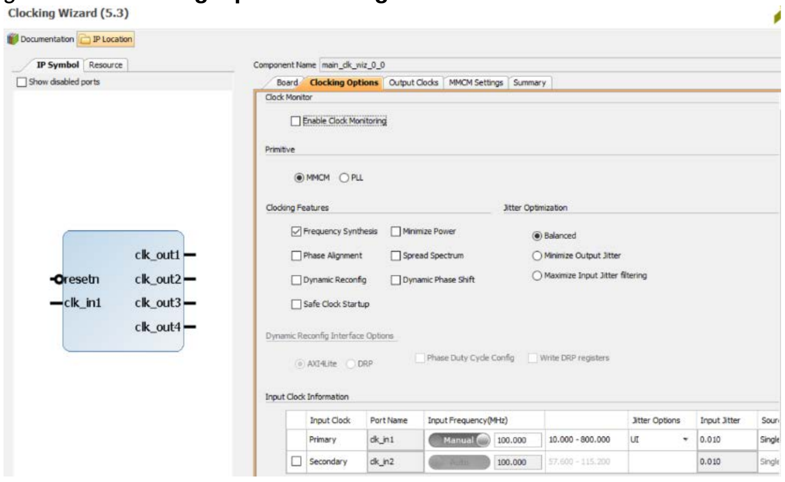
Clocking Wizard IP [Ref 4] is a very useful IP to manage different output clocks for the generation of the pixel clocks.

Select the **Add IP** icon on the right of the block design and search *Clocking Wizard,* double-click on it.
Now we have to configure the Clocking Wizard IP to output four clocks at the requested frequencies for the selected resolutions:

- Double-click on Clocking Wizard in the block design.

- Select the **Clocking Options** tab and set the configuration as shown in figure below:

Figure 3-3: **Clocking Options Configurations**


Clocking Wizard (5.3)

Component Name: main_clk_wiz_0_0

Clock Monitor
- [ ] Enable Clock Monitoring

Primitive: [✓ MMCM] PLL

Clocking Features
- [x] Frequency Synthesis
- [ ] Minimize Power
- [ ] Phase Alignment
- [ ] Spread Spectrum
- [ ] Dynamic Reconfig
- [ ] Dynamic Phase Shift
- [ ] Safe Clock Startup

Jitter Optimization: [✓ Balanced] Minimize Output Jitter Maximize Input Jitter filtering

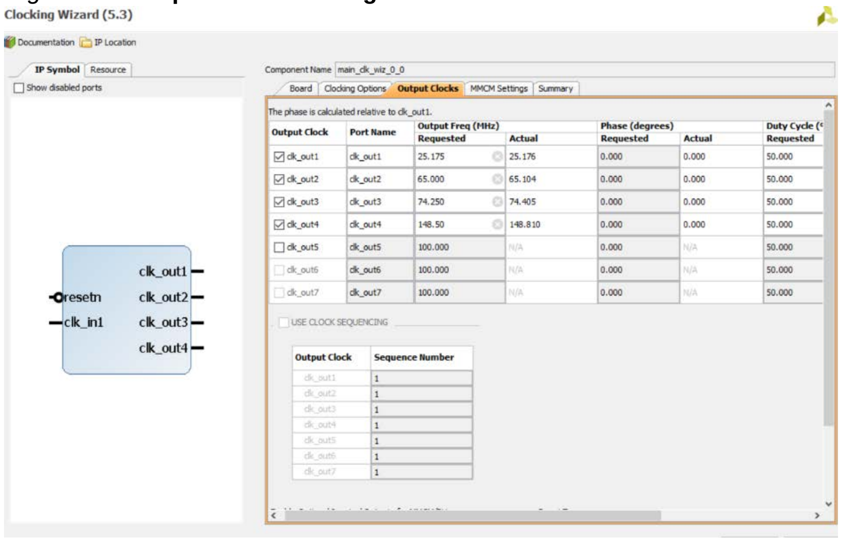
Dynamic Reconfig Interface Options: [✓ AXI4Lite] DRP Phase Duty Cycle Config Write DRP registers

Input Clock Information

	Input Clock	Port Name	Input Frequency(MHz)			Jitter Options	Input Jitter	Sour
	Primary	clk_in1	Manual	100.000	10.000 - 800.000	UI	0.010	Single
☐	Secondary	clk_in2	Auto	100.000	57.600 - 115.200		0.010	Single

- Select the **Output Clocks** tab and set the configuration as shown in figure below:

Figure 3-4: **Output Clocks Configurations**


Clocking Wizard (5.3)

Component Name: main_clk_wiz_0_0

The phase is calculated relative to clk_out1.

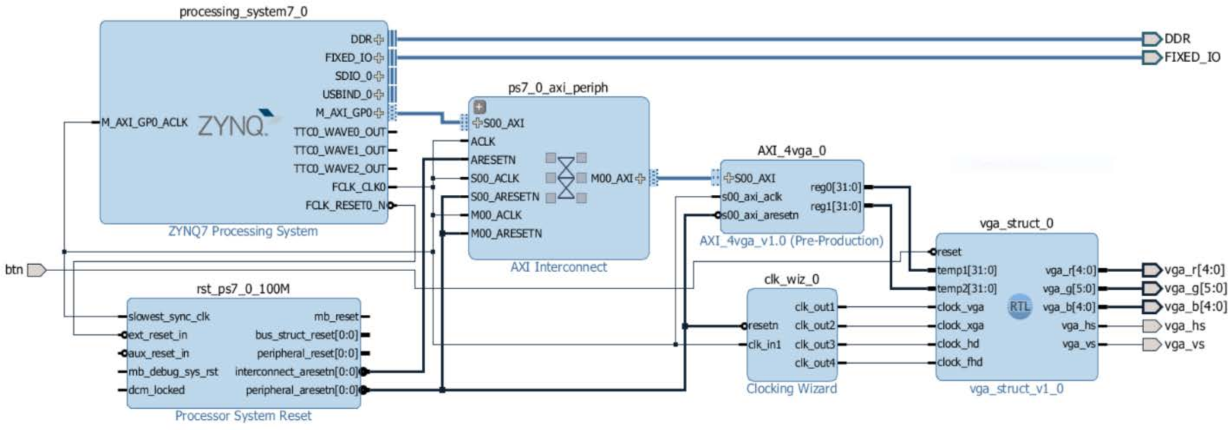
Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%
		Requested	Actual	Requested	Actual	Requested
☑ clk_out1	clk_out1	25.175	25.176	0.000	0.000	50.000
☑ clk_out2	clk_out2	65.000	65.104	0.000	0.000	50.000
☑ clk_out3	clk_out3	74.250	74.405	0.000	0.000	50.000
☑ clk_out4	clk_out4	148.50	148.810	0.000	0.000	50.000
☐ clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000
☐ clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000
☐ clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000

- [ ] USE CLOCK SEQUENCING

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

- Scroll the bar and *uncheck* the **locked** option and *set* the **reset active low.**
- Click *OK* to close the customization window.
- Make connections as shown in figure below:

Figure 3-5: **Modules Interconnection**

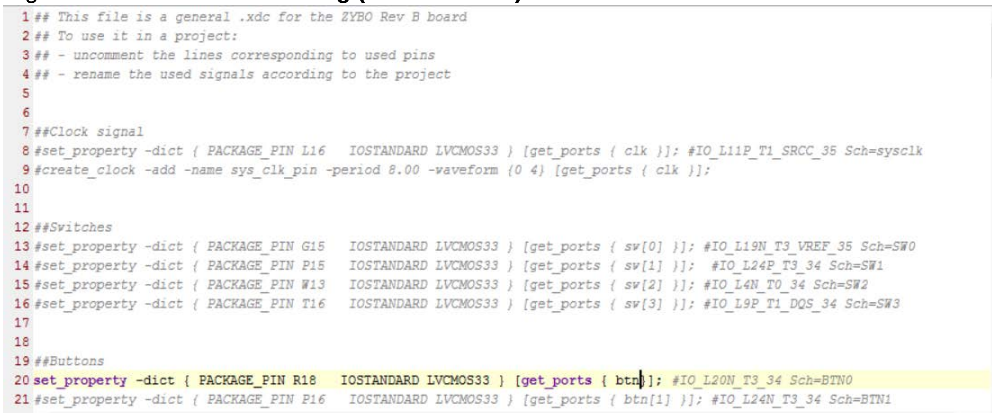


Constraints and Bitstream

Constraint file is a link between FPGA pins and external port in schematic. In this constraint file, the FPGA pins routed to physical VGA pins on Zybo Board are connected to VGA ports in our block diagram.

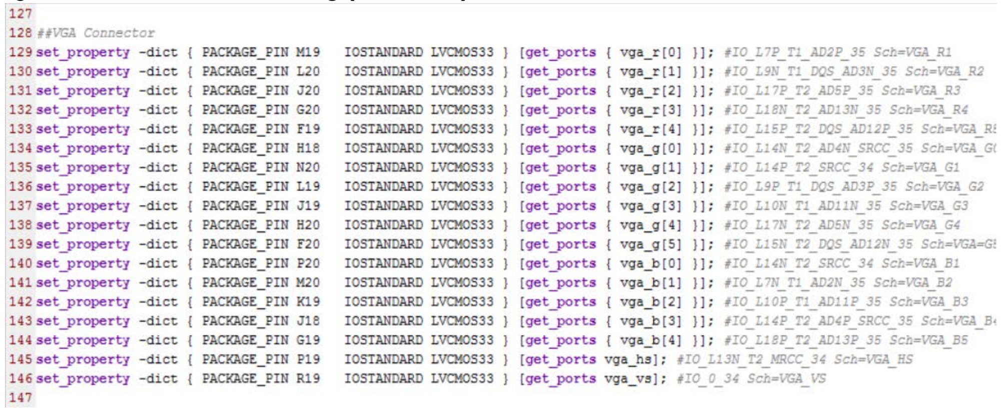
- Download the ZYBO_Master.xdc file [Ref 5].
- In Vivado, *click* the **Add Sources** button under the *Project Manager***,** within the *Flow Navigator tab*. A window will appear, select the option to **Add or create constraints** and *click next***.**
- Select **Add Files** and navigate to the directory that you downloaded the ZYBO XDC file to. *Check* the checkbox labelled **Copy constraints files into project**. *Click* Finish.
- Within the *Sources Tab*, expand the folder **Constraints**. Within it, you should see a file labelled **ZYBO_Master.xdc,** double-click on it. When you open this file, you will notice that all of the lines have been commented out. Leave all of these lines commented **except** the ones that are shown uncommented below.

Figure 3-6: **Constraints Setting (Reset Button)**



```
## This file is a general .xdc for the ZYBO Rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used signals according to the project


##Clock signal
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L11P_T1_SRCC_35 Sch=sysclk
#create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { clk }];


##Switches
#set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { sw[0] }]; #IO_L19N_T3_VREF_35 Sch=SW0
#set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { sw[1] }];  #IO_L24P_T3_34 Sch=SW1
#set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { sw[2] }]; #IO_L4N_T0_34 Sch=SW2
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { sw[3] }]; #IO_L9P_T1_DQS_34 Sch=SW3


##Buttons
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { btn }]; #IO_L20N_T3_34 Sch=BTN0
#set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }]; #IO_L24N_T3_34 Sch=BTN1
```

Figure 3-7: **Constraints Setting (VGA Port)**



```

##VGA Connector
set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports { vga_r[0] }]; #IO_L7P_T1_AD2P_35 Sch=VGA_R1
set_property -dict { PACKAGE_PIN L20   IOSTANDARD LVCMOS33 } [get_ports { vga_r[1] }]; #IO_L9N_T1_DQS_AD3N_35 Sch=VGA_R2
set_property -dict { PACKAGE_PIN J20   IOSTANDARD LVCMOS33 } [get_ports { vga_r[2] }]; #IO_L17P_T2_AD5P_35 Sch=VGA_R3
set_property -dict { PACKAGE_PIN G20   IOSTANDARD LVCMOS33 } [get_ports { vga_r[3] }]; #IO_L18N_T2_AD13N_35 Sch=VGA_R4
set_property -dict { PACKAGE_PIN F19   IOSTANDARD LVCMOS33 } [get_ports { vga_r[4] }]; #IO_L15P_T2_DQS_AD12P_35 Sch=VGA_R
set_property -dict { PACKAGE_PIN H18   IOSTANDARD LVCMOS33 } [get_ports { vga_g[0] }]; #IO_L14N_T2_AD4N_SRCC_35 Sch=VGA_G
set_property -dict { PACKAGE_PIN N20   IOSTANDARD LVCMOS33 } [get_ports { vga_g[1] }]; #IO_L14P_T2_SRCC_34 Sch=VGA_G1
set_property -dict { PACKAGE_PIN L19   IOSTANDARD LVCMOS33 } [get_ports { vga_g[2] }]; #IO_L9P_T1_DQS_AD3P_35 Sch=VGA_G2
set_property -dict { PACKAGE_PIN J19   IOSTANDARD LVCMOS33 } [get_ports { vga_g[3] }]; #IO_L10N_T1_AD11N_35 Sch=VGA_G3
set_property -dict { PACKAGE_PIN H20   IOSTANDARD LVCMOS33 } [get_ports { vga_g[4] }]; #IO_L17N_T2_AD5N_35 Sch=VGA_G4
set_property -dict { PACKAGE_PIN F20   IOSTANDARD LVCMOS33 } [get_ports { vga_g[5] }]; #IO_L15N_T2_DQS_AD12N_35 Sch=VGA=G
set_property -dict { PACKAGE_PIN P20   IOSTANDARD LVCMOS33 } [get_ports { vga_b[0] }]; #IO_L14N_T2_SRCC_34 Sch=VGA_B1
set_property -dict { PACKAGE_PIN M20   IOSTANDARD LVCMOS33 } [get_ports { vga_b[1] }]; #IO_L7N_T1_AD2N_35 Sch=VGA_B2
set_property -dict { PACKAGE_PIN K19   IOSTANDARD LVCMOS33 } [get_ports { vga_b[2] }]; #IO_L10P_T1_AD11P_35 Sch=VGA_B3
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { vga_b[3] }]; #IO_L14P_T2_AD4P_SRCC_35 Sch=VGA_B
set_property -dict { PACKAGE_PIN G19   IOSTANDARD LVCMOS33 } [get_ports { vga_b[4] }]; #IO_L18P_T2_AD13P_35 Sch=VGA_B5
set_property -dict { PACKAGE_PIN P19   IOSTANDARD LVCMOS33 } [get_ports vga_hs]; #IO_L13N_T2_MRCC_34 Sch=VGA_HS
set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [get_ports vga_vs]; #IO_0_34 Sch=VGA_VS

```

- Make sure that the names of the pins in the constraint file correspond to those in the design block. Save.

Last steps are **Validating the Design**, **Create HDL Wrapper**, **Run Synthesis**, **Run Implementation** and **Generate Bitstream**:

- *Click* the **Validate Design** button and then, if validated successfully, select the *Sources Tab.*
  Note: If it is displayed a message like this **"***Reset pin '/.../...' is found to be associated with multiple clock pins, which are not compatible w.r.t frequency, phase and clock-domain...*", ignore it and press *OK*.

- In *Sources Tab*, *Right Click* your block design file (.bd) within the **Design Sources** directory. From the dropdown menu, select the option to **Create HDL Wrapper**. Verify that the radio button **Let Vivado manage wrapper and auto-update** is selected and *click* OK. *Right Click* again on your block design file and select the option **Set as Top**.

- *Click* the **Run Implementation** button under the *Implementation*, within the *Flow Navigator tab*, a **Missing Synthesis Results** window appears. *Click* OK. If your design hasn't been saved, it will prompt you to save again. If so, *click* Save. After waiting, an **Implementation Completed** window will appear. Within the **Implementation Completed** window, select the option to **Generate Bitstream**. *Click* OK.

- After the bitstream has been generated, a **Bitstream Generation Completed** window will appear. Choose the **Open Implemented Design** option and *click* OK.

- Now, it's time to export what we created to SDK in order to program the processor. *Click* **File>Export>Export Hardware...** An *Export Hardware window* will appear, select the **Include bitstream** checkbox option and *click* OK.

Chapter 4

Installing Linux on Zybo

This chapter explains how to create and boot a permanent embedded Linux OS for the Zybo using the Petalinux tool by Xilinx.

Installing Petalinux

- **PREREQUISITES**

  1. In order to install Petalinux you need a working 64-bit Linux Distro. This guide uses **Ubuntu 16.04.2 LTS Desktop**, running in a **VMWare** virtual machine under Windows 10. You can follow VMWare guidelines in order to perform an "Easy Install" of the OS. Remember to previously download the Ubuntu ISO file. Check the *Petalinux System Requirements* to optimize performace (4GB of Ram is recommended).

  2. Donwload **Petalinux 2016.4** from the official download page of Xilinx, under the subsection Embedded Development. You need the file with .run extension (installer). It is important to have a stable internet connection because the file is very large.

- **INSTALLATION**

  Follow the official reference guide to install Petalinux [Ref 7].

  Take particular attention to these points:
  1. Petalinux works only on 64 bit Linux machines.

  2. You can check the external packets you need to install in the *Installation* Requirements or executing the script setting.sh as indicated in the *Setup Working Environment* section of the guide.

  3. Install every packet needed with the command:
     ```
     $ sudo apt-get install <packet_name1> < packet_name1> <..>
     ```
     or with your favourite graphic packet manager.

  4. You may need to manually download and install the Debian packet **zlib1g-dev: i386**.
     Download it from here [Ref 8] and install it with the following command:
     ```
     $ sudo dpkg -i <packet_name>
     ```

  5. After completing all the installations, it is recommended to update dependencies:
     ```
     $ sudo apt-get install -f
     ```

  6. Follow *Setup Working Environment* section of the guide. Remember to call
     ```
     $ source <petalinux_installation_path>/settings.sh
     ```
     anytime you open a new terminal.

Creating, Configuring and Building Project

You need to setup a new boot image specifically compiled for your hardware. In order to do that you can create a new Petalinux project with the Zynq template and then import your hardware description. If you encounter a problem, you can use the option -help to get info or you can consult the official documentation.

1. Create a directory for Petalinux projects and move to it:
```
$ mkdir ~/ZyboProjects
$ cd ~/ZyboProjects/
```

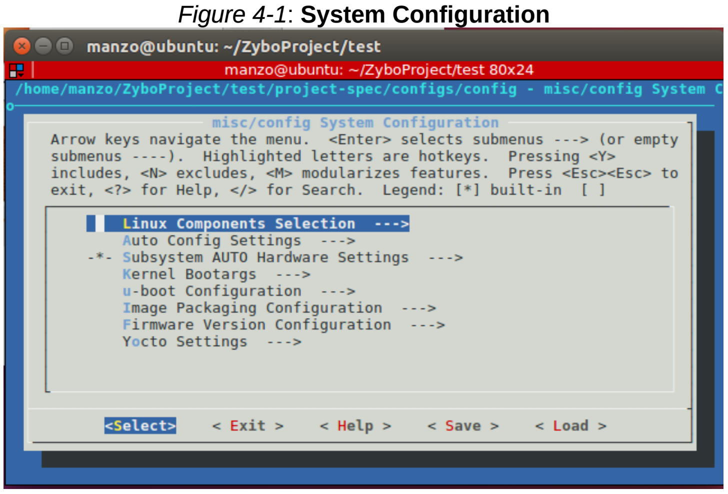
2. Create a project and move to it:
```
$ petalinux-create -t project –n AXI_vga_linux --template zynq
$ cd ./AXI_vga_linux
```

3. In the previous section of this guide you exported your hardware in Vivado. Browse Vivado project directory and open the *.sdk directory. Take the *.hdf file and copy it to your Petalinux project directory. If you are running a virtual machine you can just perform copy and paste as usual.

4. Load the hardware description just copied:
```
$ petalinux-config --get-hw-description
```

5. After some loadings a configuration menu will appear.

Figure 4-1: **System Configuration**

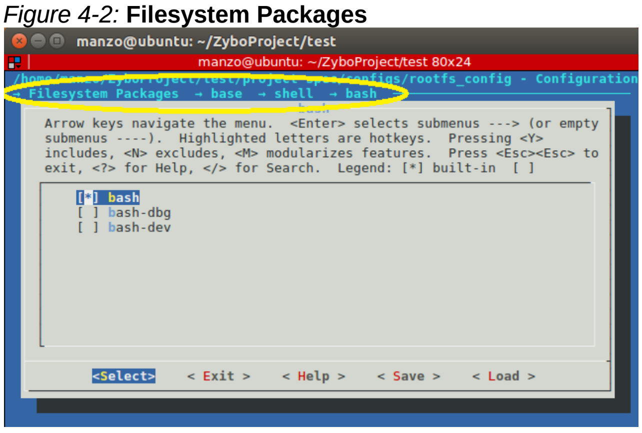


```
/home/manzo/ZyboProject/test/project-spec/configs/config - misc/config System C
                     misc/config System Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

          Linux Components Selection  --->
          Auto Config Settings  --->
      -*- Subsystem AUTO Hardware Settings  --->
          Kernel Bootargs  --->
          u-boot Configuration  --->
          Image Packaging Configuration  --->
          Firmware Version Configuration  --->
          Yocto Settings  --->

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

- Go to *Subsystem AUTO Hardware Settings → Advanced Bootable Images..*
- Make sure that both ***boot*** and ***dtb*** have *primary SD* selected as *image storage media.*
- Go back to the main menu and open *Image Packaging Configuration.*
- Select *SD Card* as ***Root Filesystem Type.***
- Save and exit.

6. Configuration writing can take a while. After it has finished launch the configuration of the kernel.
```
$ petalinux-config -c kernel
```
For the kernel no change is required. Simply exit from the configuration menu.
Launch the configuration of the filesystem
```
$ petalinux-config -c rootfs
```
In the root filesystem settings you can install lots of packets. To complete this project you just need bash. Enable it as shown in the screenshot.

Figure 4-2: **Filesystem Packages**



```
manzo@ubuntu: ~/ZyboProject/test
manzo@ubuntu: ~/ZyboProject/test 80x24
/home/manzo/ZyboProject/test/project-spec/configs/rootfs_config - Configuration
→ Filesystem Packages → base → shell → bash
                                 bash
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

      [*] bash
      [ ] bash-dbg
      [ ] bash-dev

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

7. After everything has been configured, you can start building. This task may take a very long time.
`$ petalinux-build`

8. Package the BOOT.BIN file. It is loaded by zynq at startup and contains the *First Stage Boot Loader*, your custom *Bitstream* and *u-boot*.
`$ petalinux-package --boot --fsbl ./images/linux/zynq_fsbl.elf --fpga ./images/linux/*.bit --u-boot`

Preparing SD Card

In order to use your SD as permanent filesystem drive you need to create an ***ext4* partition**. Read *Configuring SD Card ext filesystem Boot* section of the official reference guide. In particular you need to read and understand *Prerequisites* and *Preparing the SD* subsections.

1. If you are running a virtual machine follow this guide in order to make Ubuntu detect and physically manage your SD Card: *How to use SD Card reader in VMplayer* [Ref 9].

2. In Ubuntu launch GParted from terminal
`$ sudo gparted`

3. Select your SD Card and create the two partitions as previously read in the reference guide.

4. Note down the two partitions path (in this guide they will be referred as /dev/*sdb1* and /dev/*sdb2*).

5. Mount sdb1 and copy files *BOOT.BIN, image.ub and system.dtb* from *<Petalinux_project_path>/images/linux* to root folder of the FAT32 partition. If you are running a virtual machine, the mounted partitions should be visible in the ubuntu main bar or in the folder */media/<user>/* .

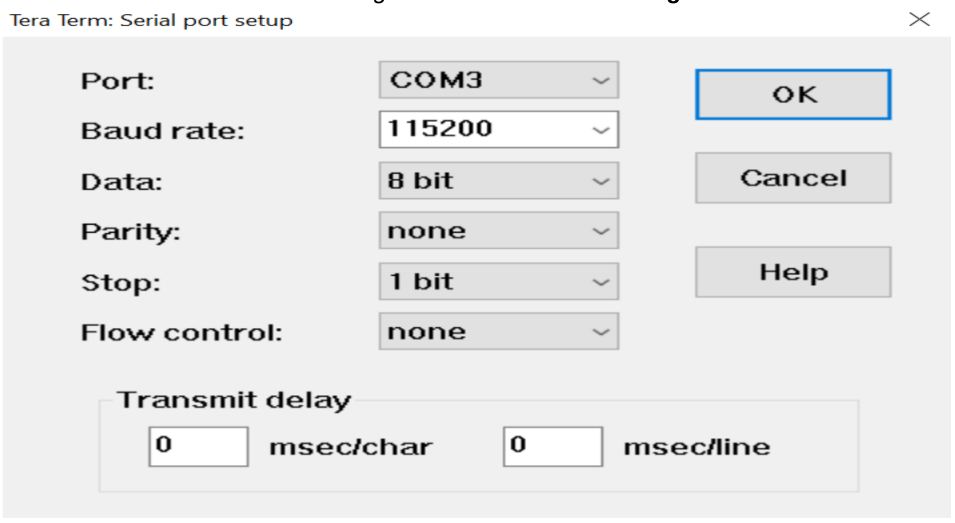
6. Use dd to copy the filesystem image to the ext4 partition.
`$ sudo dd if=<Petalinux_project_path>/images/linux/rootfs.ext4 of=/dev/sdb2`

7. Mount ext4 partition and add any additional user file you need.

Booting Linux on Zybo

1. Insert SD card in the Zybo board.
2. Change the boot jumper to SD mode.
3. Connect the micro USB cable to the UART1 port on the Zybo.
   (The board may absorb a discrete amount of current, so make sure you have a capable USB port in your PC. A 3.0 USB port should be enough. In alternative connect the board to an external power supply and change the power source jumper on the board).
4. Install a serial port terminal in your PC. You can use the open source software "Tera Term". Download it from [Ref 10] and install it.
5. Power on the Zybo and wait for the PC to correctly detect Serial Port.
6. Open Tera Term or your alternative terminal. Setup a serial port (Setup → Serial port..) with these parameters.

Figure 4-3: **Tera Term Settings**



Tera Term: Serial port setup

Port: COM3
Baud rate: 115200
Data: 8 bit
Parity: none
Stop: 1 bit
Flow control: none

Transmit delay
0 msec/char
0 msec/line

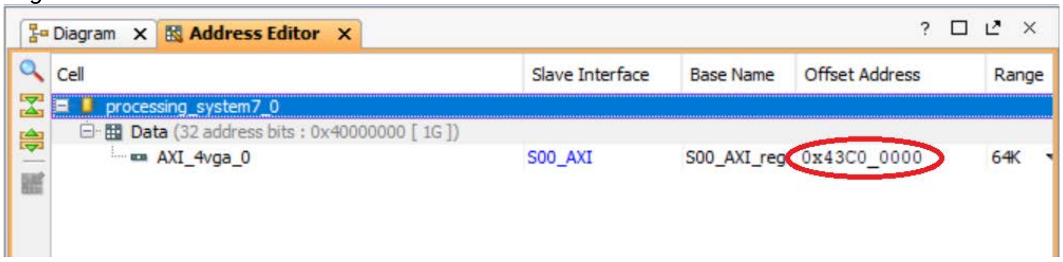
Note: If you don't know the Zybo COM port check it on the *device manager* or try all the ports until you see some activity on terminal.

7. Reset your Zybo (PS-SRST button) to be able to read the boot routine messages.
8. Wait for Linux to load and when requested insert root both as username and password.
9. Test your Linux. Create a file in the filesystem with
   ```
   $ touch /home/root/test
   ```
   and check if it is still there after a reboot.

Controlling VGA Resolutions from Linux

In order to dynamically change VGA resolutions from Linux, you need to write some data to the AXI registers at runtime. They are accessible by the system as physical memory addresses. Physical memory is directly visible to the kernel but not to the user side of Linux. In order to write to registers you can write a kernel module that acts as a driver or use the file */dev/mem* which allows physical memory access from user side. In this guide you will use a script that calls a bash function called *devmem.*

1. Open Vivado project and note down memory addresses of the AXI registers. To see them click on *open block design* and check them in the *address editor* tab as shown below:
   Figure 4-4: **Address Editor**



Cell	Slave Interface	Base Name	Offset Address	Range
processing_system7_0				
Data (32 address bits : 0x40000000 [1G])				
AXI_4vga_0	S00_AXI	S00_AXI_reg	0x43C0_0000	64K

2. Download GitHub zip archive at this link [Ref 11]. Extract it in a folder and open the file *res_choice_script.sh* with a text editor. Change the line 3 with the *Offset Address* you read before.

3. Mount *ext4* partition of the SD Card and copy *res_choice_script.sh* into /home/root/

4. Boot the system on Zybo and launch the script from terminal
```
$ /home/root/res_choice_script.sh
```

5. Test your system typing 1, 2 ,3 or 4 on the terminal.

References

1. ZYBO™ FPGA Board Reference Manual
2. Gianmarco Cerutti, How to Control Peripherals on FPGA by AXI
3. Vivado Version 2015.1 and Later Board File Installation
4. Clocking Wizard IP
5. Zybo Master.xdc
6. VGA Timing Specification
7. How to Install Petalinux
8. Debian Packet Download
9. How to use SD Card reader in VMplayer
10. Tera Term
11. GitHub Download