DIGITAL SYSTEM DESIGN

PROJECT DOCUMENTATION

VGA CONTROLLER

Table of Contents

[1. Specification 3](#_Toc9163930)

[2. General considerations 3](#_Toc9163931)

[3. Block diagram 4](#_Toc9163932)

[4. Project implementation 5](#_Toc9163933)

[I. Basic MSI components 5](#_Toc9163934)

[a) AND\_GATE 5](#_Toc9163935)

[b) D Flip Flop 6](#_Toc9163936)

[c) Counter 6](#_Toc9163937)

# Specification

The requirement of the project is to implement a VGA controller using an FPGA board. There must be four different images (square, circle, triangle, vertical lines) displayed on the screen which must show the ability of the controller to choose at least four different colors for the images. A button must control the switching of the images. The user should be able to move the objects on the screen along the X-axis and Y-axis using other buttons.

# General considerations

The acronym VGA stands for Video Graphic Array and it represents a video display standard first introduced in 1987 by the IBM PS/2 line of computers. The general implementation of a VGA controller consists of 2 timing signals (Horizontal and Vertical Sync) and 3 busses that represent the bit-encodings for colors red, green and blue.

This project was implemented using a Nexys 4 DDR FPGA board and therefore each color buss has 4 bits creating a 12-bit unique pattern which can display 4096 different colors. Furthermore, the resolution considered is 640x480 pixels.

A close up of a map

Description automatically generatedThe rest of this section will present the basic functioning principles of the VGA port and the VGA timing constraints according to the FPGA board reference manual.

The first displays to use the VGA standard were CRT-based (cathode-ray tube). The main components of a CRT-display are the signals, the electron guns, the deflection coils, a high voltage source connected to the screen and the cathode-rays themselves (as seen in the image on the right). The cathode ray ‘hits’ the screen with electrons controlled by the electron guns. As a result of this collision, the inner phosphor coat of the display glows at the exact point where the impact happened. The color emanated depends on the RGB signals which are driven to the electron guns. The horizontal and vertical sync (HS and VS) signals control the movement of the electron beam through the deflection coils, making the cathode-ray traverse the screen on a pixel-by-pixel basis.

HS and VS have the role of synchronizing the electron beam with the display. A pixel clock of 25MHz represents the time available for the controller to print one pixel; it also coordinates HS and VS. An entire time period of the horizontal sync signal can be split into 4 distinct frames: display time (640 pixel clocks) – effective writing of pixels, front-porch (16 pixel clocks) – changing direction from forward moving to backward, sync pulse (96 pixel clocks) – traversing the screen from right to left without displaying anything on the screen, it also coincides with the only time when HS is not active – and back-porch (48 pixel clocks) – changing direction from backward moving to forward. A similar logic is used for VS: display time (480 rows), front-porch (1 row), sync pulse (3 rows), back porch (16 rows).

The LCD displays have adapted their functioning to work with the same timing signal as the CRT ones, so the same implementation logic can be used.

# Block diagram

A picture containing wall, indoor

Description automatically generated

Input and output description:

* fpga\_clock: input, internal clock of the FPGA board, used to obtain the pixel clock for the VGA controller and to control the debouncers linked to the buttons
* imgBtn: input, button, controls the switching of the images
* btnLeft: input, button, used for moving the image along the X-axis one pixel left on each press
* btnRight: input, button, used for moving the image along the X-axis one pixel right

on each press

* btnUp: input, button, used for moving the image along the Y-axis one pixel up on each press
* btnDown: input, button, used for moving the image along the Y-axis one pixel down on each press
* RIn [3:0]: bus input, switches, receives the red component of the pixel color
* GIn [3:0]: bus input, switches, receives the green component of the pixel color
* BIn [3:0]: bus input, switches, receives the blue component of the pixel color
* H\_Sync: output, horizontal sync signal
* V\_Sync: output, vertical sync signal
* R [3:0]: bus output, red component of the color
* G [3:0]: bus output, green component of the color
* B [3:0]: bus output, blue component of the color

Note: All outputs of the controller are sent directly to the VGA port, which will then display an image on the screen.

# Project implementation

The project contains a total of 12 modules: 5 basic MSI components, 6 major components and a top module file in which the major components are instantiated and connected.

## Basic MSI components

### AND\_GATE

A close up of a logo

Description automatically generated

Generic Parameters:

* N: size of the input bus

Inputs and Outputs:

* Input [N-1:0]: std\_logic\_vector input of variable length; generic parameter N gives the number of wires in the bus
* Gate\_output: std\_logic output

This component implements a basic and-gate with a variable number of inputs which are received through a bus. This is done using a generic parameter *N* which gives the number wires of the input bus. The output represents a single standard logic signal: the result of the and logic operator ‘and’ applied to all of *input*’s terms.

### D Flip Flop

A close up of a logo

Description automatically generated

Generic Parameters: none

Inputs and Outputs:

* D: std\_logic input
* CLK: std\_logic input, clock signal of the component
* Q: std\_logic output

This component implements a basic D Flip Flop that changes its state on the rising edge of the clock pulse. The output *Q* will take the value of input *D* when the clock signal changes from ‘0’ to ‘1’.

### D Flip Flop for Integers

Generic Parameters:

* Nr\_range: range of integer signals

Inputs and Outputs:

* D: integer input
* Ce: std\_logic input
* Clk: std\_logic input
* Q integer output

It has a similar behavior to the previous component, the only differences are the type if *D* and *Q* signals, which are integers, and the feature of having a clock enable (*ce*) signal. If *ce* has value ‘1’ then the output takes the value of input *D* on the rising edge of the clock signal *clk*, otherwise the counter is inactive and it maintains its previous state.

### Offset Counter

A picture containing object

Description automatically generated

Generic Parameters:

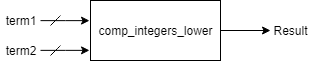
* numberRange: maximum value of output

Input and Output:

* enable: std\_logic input
* plus: std\_logic input, clock signal of the component
* offset: integer output, its range is given by the generic parameter

This component implements a counter which has the clock enable feature implemented and whose output variable is of type integer. If the input *enable* is active (has value ‘1’) then the counter increases the output value by one unit on every rising edge of the clock, *pulse*. If *enable* is not active, the counter does not work. The range of the output is given by the generic parameter *numberRange*: it can take values between 0 and *numberRange* – 1. When *offset* reaches its upper limit the counter resets automatically.

### Integer comparator



Generic parameters:

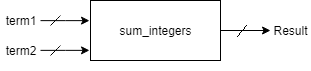
* Number\_range: maximum value of input variables

Inputs and Outputs:

* Term1: integer input
* Term2: integer input
* Result: std\_logic output

This component implements a basic comparator between two variables of integer type. It returns an std\_logic signal as an output which is ‘1’ if *term1* is strictly lower than *term2*, otherwise *result* is 0.

### Integer adder



Generic parameters:

* Nr\_range: range of the variables

Inputs and outputs:

* Term1: integer input
* Term2: integer input
* Result: integer output

This component is a basic adder with a simple behavior: it returns the sum of the two inputs. All signals are integers.

## Main Project Modules

### Frequency divider

Block diagram:

Internal schematic:

A screenshot of a cell phone

Description automatically generated

Generic Parameters: none

Inputs and Outputs:

* Clk\_in: std\_logic input
* resetDivider: std\_logic input
* clk\_out: std\_logic input

Notations used:

* Inter [1:0]: std\_logic\_vector internal signal, represent the output of the counter component inside the frequency divider

This module implements a frequency divider using as an internal component a 2-bit counter. The aim of the module is to obtain the pixel clock (which for the 640x480 resolution should have a value of 25MHz). In order to do that, the *clk\_in* signal is connected to the internal clock of the FPGA board. For a Nexys 4 DDR board, this clock has a frequency of 100MHz, which means it should be divided by 4 to obtain a frequency of 25MHz. This is done through the usage of the 2-bit counter. Since 4 = 22, we only need to connect to most important bit of the output of the counter to *clk\_out*. The output of the frequency divider will therefore be the pixel clock. The *resetDivider* input of the frequency divider is connected to the *reset* of the counter and when it has value ‘1’ it would reset the counter to 0. However, in this project the divider should work continuously without interruption so *resetDivider* is connected to the ground.

### Debouncer

Block diagram:

A screenshot of a social media post

Description automatically generatedInternal schematic:

Generic parameters:

* Nr\_bistabile: gives the number of flip flops inside the debouncer

Inputs and Outputs:

* Btn\_in: std\_logic input
* Clk: std\_logic input, clock signal of component
* Btn\_out: std\_logic output

Notations used:

* Inter [1:nr\_bistabile]: std\_logic\_vector, each element of the vector represents the output of a flip flop

Note: in the internal schematic scheme, *n* is used instead of *nr\_bitabile* for

The term ‘bouncing’ refers to the situation when a button is pressed and its corresponding signal switches from a low voltage to a high voltage and then back several times before it steadies on the high value. This happens due to mechanical reasons, since the contact between the metal pieces of the button is not perfect. If, for example, the button would be connected to a counter as the clock signal, it would cause the counter to jump several states on a single button press instead of just one state, which is the desired effect. In order to avoid this situation, the button signal must first go through a debouncer.

The debouncer implemented for this project consists of a several D-flip flops and an and-gate. The number of flip flop inside the deboucer is given by a generic parameter, *nr\_bistabile*, which would also give the size of the input bus of the and-gate. All flip flops have the same clock signal, *clk*. The input of the first one is connected to *btn\_in*, which is the button pressed. For all others, the input is connected to the output of the previous flip flop. All the outputs represent wire of *inter*, which will then be linked to an and-gate, whose output will be the output of the entire module. Basically, *btn\_out* will be ‘1’ only when all of the flip flops will have value ‘1’, which means the button signal stopped switching between high and low voltages.