DIGITAL SYSTEM DESIGN

PROJECT DOCUMENTATION

VGA CONTROLLER

Table of Contents

[1. Specification 3](#_Toc9461744)

[2. General Considerations 3](#_Toc9461745)

[3. Block Diagram 4](#_Toc9461746)

[4. Project Implementation 5](#_Toc9461747)

[I. Basic MSI components 5](#_Toc9461748)

[a) AND\_GATE 5](#_Toc9461749)

[b) D Flip Flop 6](#_Toc9461750)

[c) D Flip Flop for Integers 6](#_Toc9461751)

[d) Offset Counter 7](#_Toc9461752)

[e) Counter 8](#_Toc9461753)

[f) Integer Counter 8](#_Toc9461754)

[g) Integer comparator 9](#_Toc9461755)

[II. Main Project Modules 9](#_Toc9461756)

[a) Frequency divider 9](#_Toc9461757)

[b) Debouncer 11](#_Toc9461758)

[c) DU – debounce unit 13](#_Toc9461759)

[d) Image Generator 14](#_Toc9461760)

[e) Position calculator 21](#_Toc9461761)

[f) VGA Controller 22](#_Toc9461762)

[g) Top Module 26](#_Toc9461763)

[5. Justification of the Chosen Solution 28](#_Toc9461764)

[6. Mapping on the FPGA board 29](#_Toc9461765)

[7. Usage and Maintenance Instructions 30](#_Toc9461766)

[8. Further Development Possibilities 31](#_Toc9461767)

# 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 no longer have cathode rays. The acronym stands for liquid-cristal display; the colors displayed on the screen are controlled through voltage imposed across the liquid cristal to change the light permittivity of a very small area. The crystal properties are change on pixel-by-pixel basis. LCD’s have adapted their functioning to work with the same timing signals as the CRT displays, 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 14 source files which can be divided into 7 basic MSI components, 6 major modules built from the MSI components and a top module file in which the major modules 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 on the next clock cycle.

If we assume *ce* is ‘1’ and that the generic parameter has value 10 then the following table show the behavior of the counter:

|  |  |
| --- | --- |
| CURRENT STATE | NEXT STATE |
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |
| 5 | 6 |
| 6 | 7 |
| 7 | 8 |
| 8 | 9 |
| 9 | 0 |

### A picture containing object Description automatically generatedCounter

Generic Parameters:

* nr\_bits: gives the size of the output bus

Inputs and Outputs:

* reset: std\_logic input
* clock: std\_logic input, clock signal of component
* Q [nr\_bits – 1: 0]: std\_logic\_vector output

This is a basic counter on a variable number of bits with an asynchronous reset. On the rising edge of the clock signal, the output *Q* will increase with one unit.

If *reset* is active (has value ‘1’) then regardless of the previous state *Q* will be set to 0. If *reset* is ‘0’ and assuming the generic parameter has value 3 the following table shows the behavior of the counter:

|  |  |
| --- | --- |
| CURRENT STATE | NEXT STATE |
| 000 | 001 |
| 001 | 010 |
| 010 | 011 |
| 011 | 100 |
| 100 | 101 |
| 101 | 110 |
| 110 | 111 |
| 111 | 000 |

### Integer Counter

Generic parameters:

* max\_value: gives the range of the output

Inputs and Outputs:

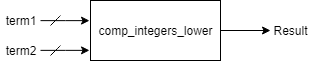
* clk: std\_logic input, clock signal of the component
* count: integer output

This component implements a counter whose output is and integer. The output’s maximum value is given by the generic parameter. The counter goes to the next state on the rising edge of the clock cycle. If the value of the output has reached the maximum value, the counter will reset automatically on the next clock cycle.

If we assume that *max\_value* is given value 5 the counter has the following behavior.

|  |  |
| --- | --- |
| CURRENT STATE | NEXT STATE |
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 0 |

### Integer comparator



Generic parameters:

* number\_range: maximum value of input signals

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.

## 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.

### 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\_bistabile* for simplicity

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 expected effect.

In an ideal situation where all the physical circuits would only have perfect contacts between their components the signal triggered by a button press would look something like this:

A close up of a logo

Description automatically generated

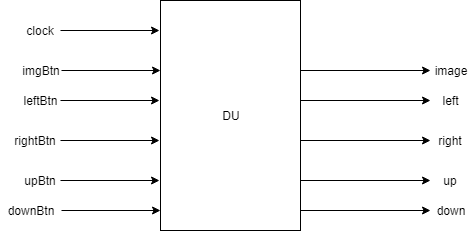
A close up of a logo

Description automatically generatedThis is what we expect to happen when a button is pressed, but what the waveform of the signal looks like in reality is more like the next graphic:

In order to avoid this situation, the button signal must first go through a debouncer – a component whose purpose is to ‘clear’ the signal and avoid the bouncing phenomena. The output of the debouncer should have a waveform like the first one presented previously (the one without any bounce periods), but with a small delay between the moment the button is pressed and the moment when the debouncer’s output switches from ‘0’ to ‘1’.

The debouncer implemented for this project consists of a several D-flip flops and an and-gate. The number of flip flop inside the debouncer 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 the flip flops will have value ‘1’, which means the button signal stopped switching between high and low voltages.

### DU – debounce unit

Block diagram:

Internal schematic: A close up of text on a black background

Description automatically generated

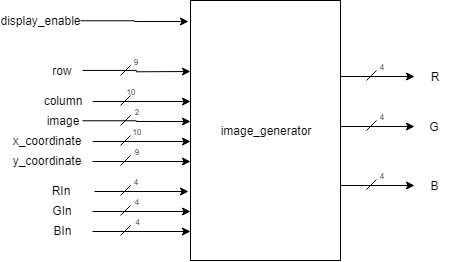
Generic parameters: none

Inputs and Outputs:

* clock: std\_logic input, clock signal of the component
* imgBtn: std\_logic input, button
* leftBtn: std\_logic input, button
* rightBtn: std\_logic input, button
* upBtn: std\_logic input, button
* downBtn: std\_logic input, button
* image: std\_logic output, debounced signal corresponding to imgBtn
* left: std\_logic output, debounced signal corresponding to leftBtn
* right: std\_logic output, debounced signal corresponding to rightBtn
* up: std\_logic output, debounced signal corresponding to upBtn
* down: std\_logic output, debounced signal corresponding to downBtn

This component represents a series of debouncers, one for each button from the user interface of the project. An internal constant called *nr\_bistabile* will give the number of flip flops inside the debouncers. Thus, this value can be easily changed.

### Image Generator

Block diagram:

Internal schematic:

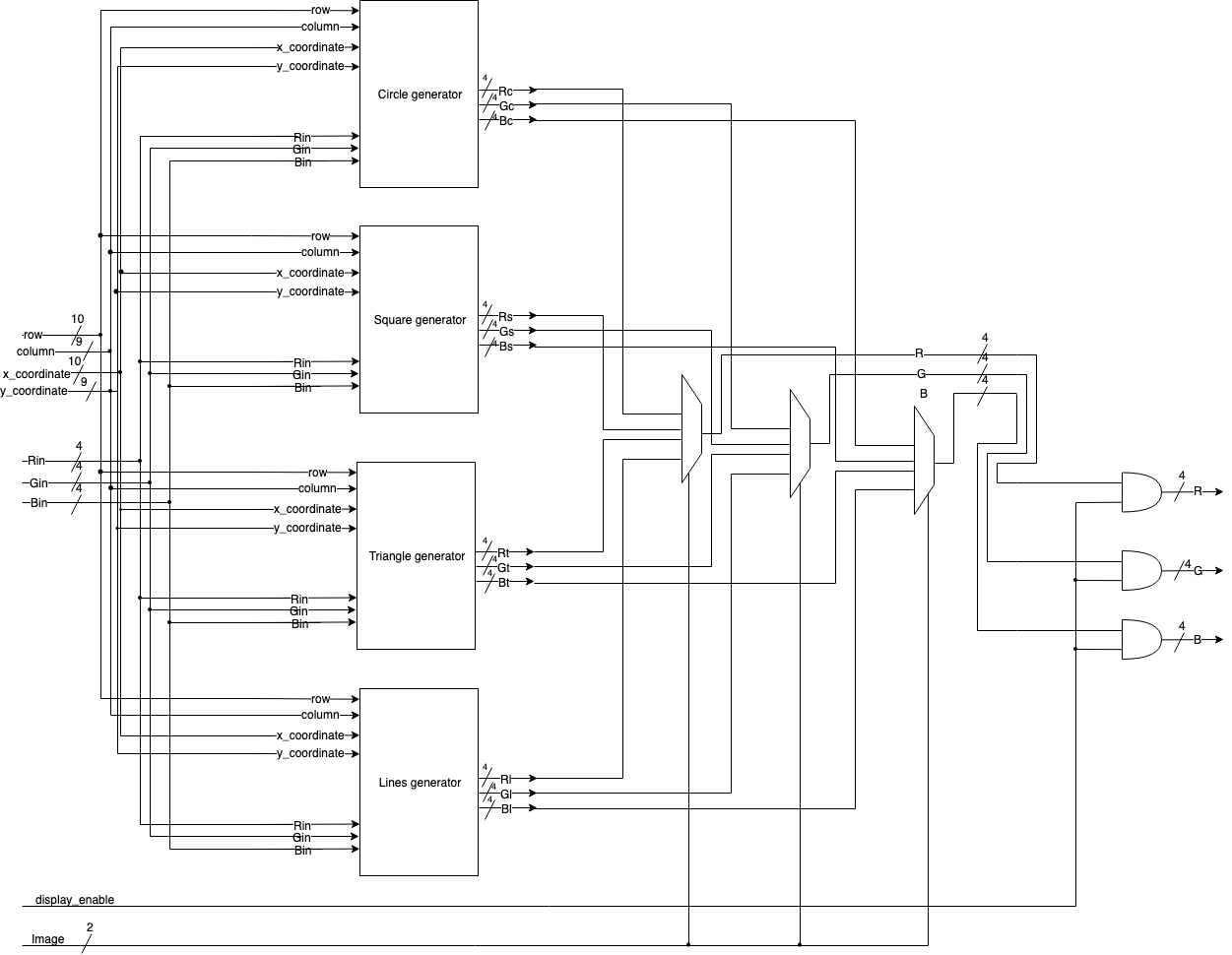


Image generator’s inputs and outputs:

* display\_enable: std\_logic; used for when the pointer of the display is out of the bounds set by resolution. If it is out of bound simply writes the color black in order not to have artifacts.
* row: integer
* column: integer
  + Both row and column point to the point on the screen where the pixel is going to be colored by Rin, Gin, Bin.
* image: integer; selects with numbers from 0 to 3 the image that will be displayed

* x\_coordinate: integer
* y\_coordinate: integer
* The coordinates for the origin points of the image that will be drawn on the screen.

* Rin, Gin, Bin:
* The RGB values represented on 4 bits each for the colors that the pixels within the image that will be drawn.

* R, G, B:
* The RGB values represented on 4 bits each that, depending on the inputs [row], [column], will be either the interior of the image or the background.

The ports *R, G, B, Rin, Bin, Gin* are 4-bit busses since each component of the colors is represented in numbers from 0 to 255.

Mention:

* Every module has the constant *size* which is the size of the image.
* The formulas for the shapes will have a true or false value. If the condition is met, the pixel will take the color given by *Rin, Gin, Bin,* otherwise it will be part of the background.

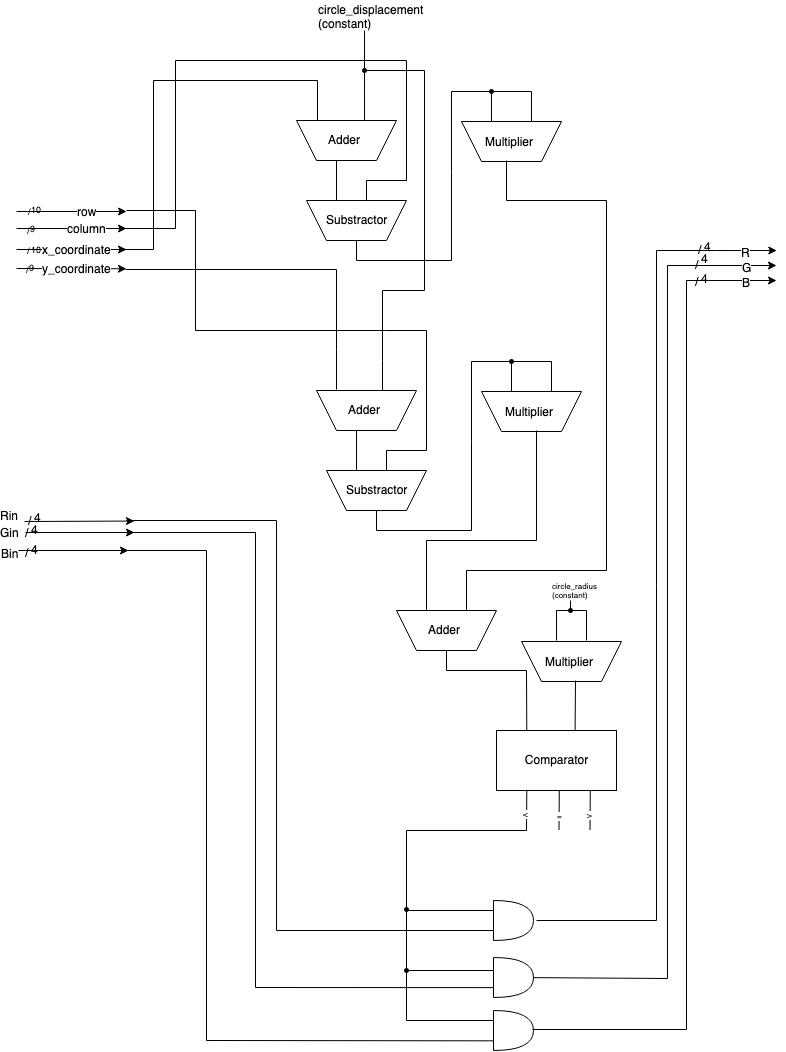
The way that the image generator figures out which pixel should take the *Rin, Bin, Gin* values or the background color is through mathematical formulas described here:

1. Circle

((*column* - (*x\_coordinate* + *circle\_displacement)*) \* (*column* - (*x\_coordinate* + *circle\_displacement*)) +  (*row* - (*y\_coordinate* + *circle\_displacement*)) \* (*row* - (*y\_coordinate* + *circle\_displacement*))) < *circle\_radius* \* *circle\_radius*

This checks if the distance between the origin point and the other points is less than the desired size of the circle and if yes, the pixels are colored.

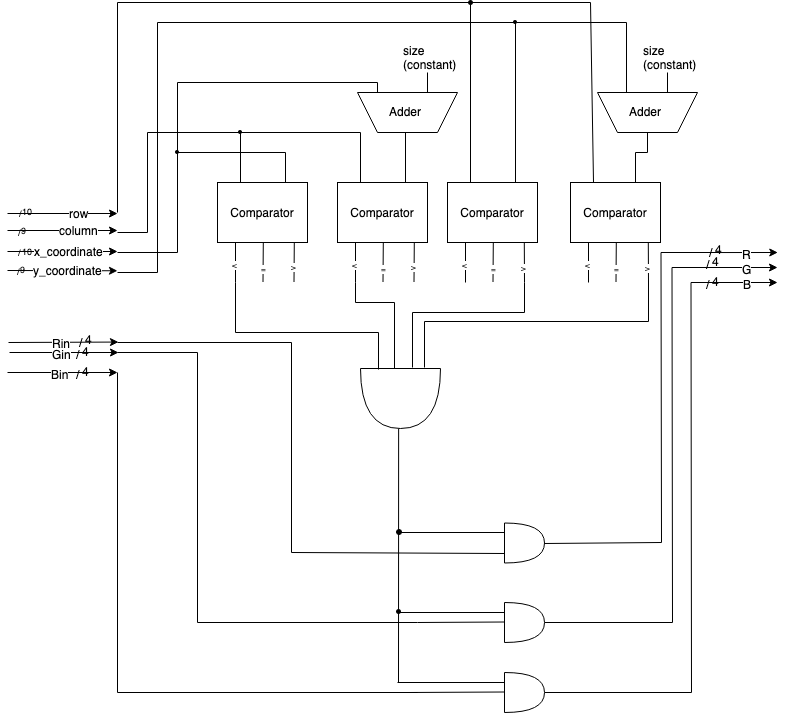
*circle\_displacement* is a constant used to offset the circle so that the circle image not to go off the screen because its origin is the center rather the top left.



1. Square

*column* > *x\_coordinate* and *column* < *x\_coordinate* + *size* and *row* > *y\_coordinate* and *row* < *y\_coordinate* + *size*

This checks if the pixels are in the bounds of the left and right sides of the square, same for the lower and higher sides, if yes, the pixels are colored.



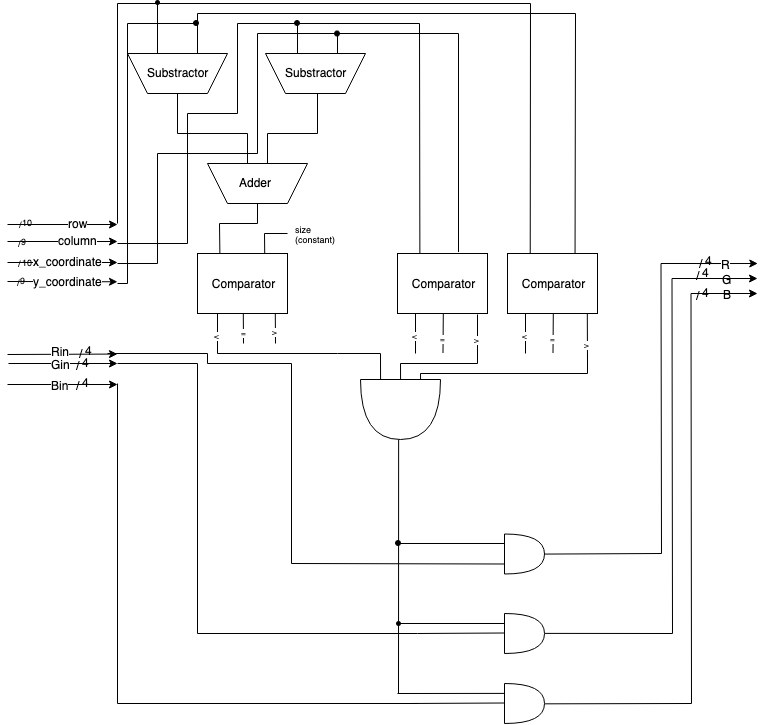
1. Triangle

(*row* - *y\_coordinate*) + (*column* - *x\_coordinate*) <= *size* and

*column* > *x\_coordinate* and

*row* > *y\_coordinate*

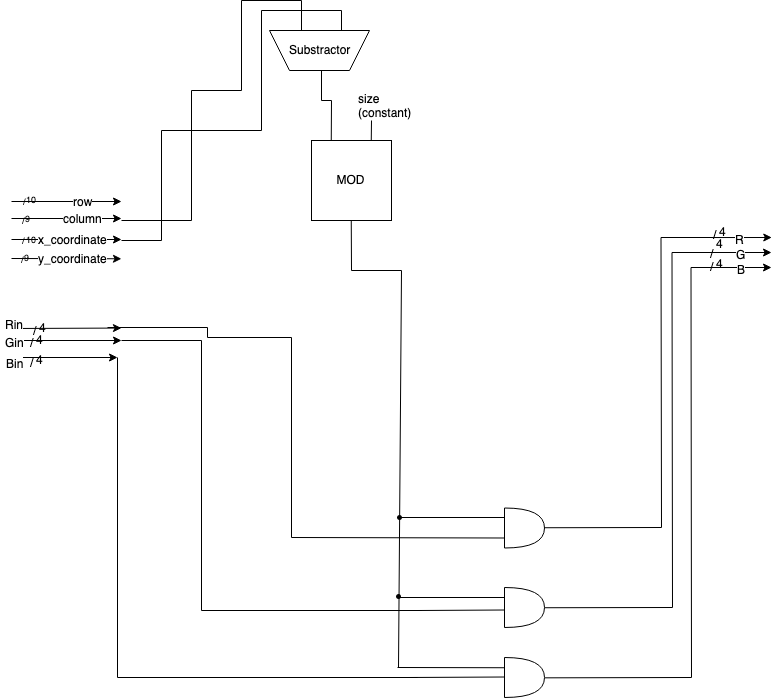
This checks if the pixels are underneath the first diagonal (relative to the origin) and if they are higher and greater than the bottom side respectively the left side, if yes, the pixels are colored.



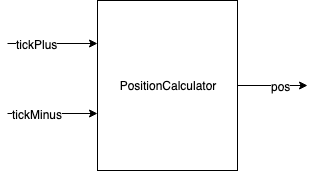
1. Vertical lines

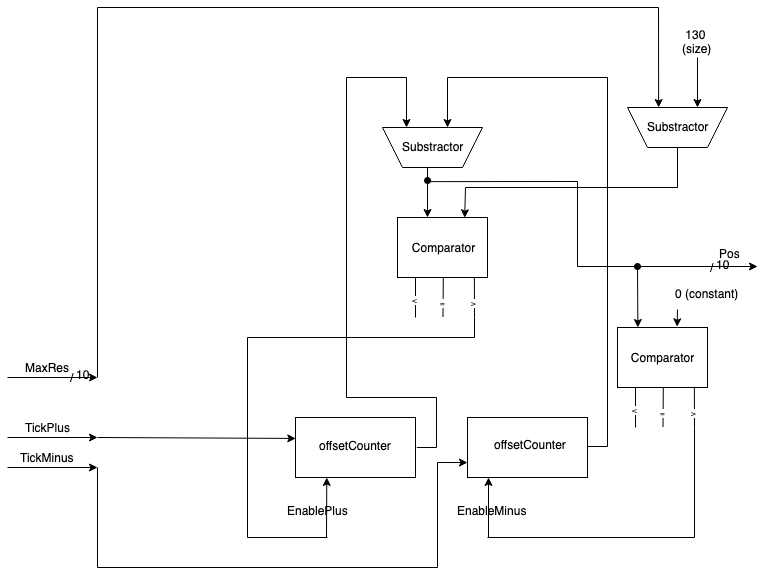
(*column* - *x\_coordinate*) mod *size* = 0

This checks if the pixels, in every line, are divisible by the size, in this case size means the distance between the lines, if yes, the pixels are colored.



### Position calculator

Block diagram:

Internal schematic:

*PositionCalculator* calculates the position for the x and y coordinates for the origin of each image that will be drawn on the screen. It does that through 2 counters, each for the 2 opposing direction. We use one *positionCalculator* for the x axis and for the y axis.

It takes a generic argument *maxRes* that represents the maximum amount the image can be moved in each direction.

The other 2 inputs are *tickPlus* and *tickMinus* which move the origin position in the 2 directions specified.

The module outputs signal *pos* which is the origin position for the images on one axis.

### VGA Controller

Block diagram:

A close up of text on a white background

Description automatically generatedInternal schematic:

Inputs and Outputs:

* pixel\_clock: std\_logic input, clock signal of the unit
* h\_sync: std\_logic output, horizontal sync
* v\_sync: std\_logic output, vertical sync
* display\_enable: std\_logic output
* row: integer output, y-coordinate of the pixel that should be drawn during the current clock cycle
* column: integer output, x-coordinate of the pixel that should be drawn during the current clock cycle

Note: The names written in blue on the schematic represent constants, and therefore the corresponding busses should also have constant value. This is achieved by connecting all the wires of the bus to either the ground or the current source. No extra memory is needed for this.

A close up of a map

Description automatically generatedIn order to understand the way this component functions we firs need to understand the timing constraints of a VGA. This topic was also discussed in the second part of this documentation. The following graphic shows how the horizontal sync waveform should look like. It also shows in parallel what is the exact behavior of the cathode-ray during a total period of the horizontal sync signal. The source of the graphic is the reference manual of the NEXYS 4 DDR FPGA board.

We will use a counter to keep track of how many *pixel\_clock* impulses have past since the horizontal sync signal has started a new period. The output of this counter (which is a *counter\_for\_integers* component) will be called *h\_count*. An important observation is that if we consider the display time to be the beginning of the total period for horizontal sync then the output of the counter will also be equal to the column number of the pixel that should be displayed in the current *pixel\_clock* cycle. However, this is only valid for the display time period, then the cathode-ray is actively influencing the phosphor coat of the display (or when the liquid-crystal light permittivity is changed for LCD displays). So, we will use a flip flop with clock enable to store the value of the column (*integer\_bist\_d\_ce*). We can only store the value if *h\_count* is smaller than *h\_display* (a constant that has the value 640 – the number of pixels in a row of the screen). This condition will be checked with a comparator (*comp\_integers\_lower*) whose output, *column\_enable*, will be linked as clock enable of the flip flop for the columns.

Using the value of *h\_count* we can also determine the value of the *h\_sync* output. To do that there are two mode constants defined:

*H\_first\_part* = *h\_display* + *h\_front\_porch*

*H\_last\_part* = *h\_display* + *h\_front\_porch* + *h\_sync\_pulse*

If *h\_count* is lower that *h\_first\_part* or if it is higher than *h\_last\_part*, then *h\_sync* should be ‘1’. Otherwise it should be ‘0’. With other words, *h\_sync* is ‘0’ only during the syncing period. We can notice that the second condition is equivalent to *h\_last\_part* being lower than *h\_count* and for both we can use the *comp\_integers\_lower* module. If either one of these conditions is satisfied then *h\_sync* will be active, otherwise it is inactive. This translates into an or-gate at the level of hardware components. The output of this gate will be *h\_sync\_input* and it will represent the input of a flip flop that store the value of *h\_sync*.

For vertical sync signal the logic is similar, only that instead of *pixel\_clock* cycles we will count rows of the screen. The moment when *h\_sync* switches from ‘1’ to ‘0’ indicates the moment a row of the screen is finished so the counter for the vertical sync should be incremented. Since this counter works on the rising edge of the clock pulse, we will consider an intermediary signal that is always the opposite of *h\_sync* (*h\_sync* is negated with the use of an inverter) as the clock signal of the counter. To ensure that the entire project is synchronized we will for pass this signal through a flip flop before it is connected to the vertical sync counter. The output of the counter will be called *v\_count* and it is also equal to the number of the current row if it is lower than *v\_display* (constant equal to 480 = number of pixels on a column). We will store the value of the row into a flip flop with clock enable. The condition under which we load data into this flip flop will also be computed by means of a comparator whose output will be called *row\_enable*. We will also need second flip flop for the *v\_sync* signal. This will use another constant *v\_first\_part* defined like *h\_first\_part*. If *v\_count* is lower than *v\_first\_part* (condition checked with a comparator) then *v\_sync* is ‘1’.

The only aspect left to discuss is the *display\_enable* signal. There are two different states that can be distinguished. The first one is the active part where the screen is traversed on a pixel-by-pixel basis and the blanking time when the cathode-rays change their position without displaying any pixel. *Display\_enable* is ‘1’ for the active time and ‘0’ for the blanking time. To determine in which state the VGA is it is enough to check is both *row\_enable* and *column\_enable* are ‘1’ – which is a normal and-gate. The value of *display\_enable* will be stored into a flip flop whose input (*display\_enable\_input*) is the output of the simple and-gate.

All flip flop from the schematic are connected to *pixel\_clock* to ensure the synchronization of the entire unit.

### Top Module

A close up of text on a white background

Description automatically generatedInternal schematic:

Notations used:

* image: std\_logic, debounced signal corresponding to imgBtn
* left: std\_logic, debounced signal corresponding to btnLeft
* right: std\_logic, debounced signal corresponding to btnRight
* up: std\_logic, debounced signal corresponding to btnUp
* down: std\_logic, debounced signal corresponding to btnDown
* image\_number: integer, number of the image currently displayed (0 – circle, 1 – square, 2 – triangle, 3 – vertical lines)
* x: integer, coordinate on x-axis of the object’s origin (center for circle, top left corner for square and triangle, x-coordinate of first vertical line)
* y: integer, coordinate on y-axis of the object’s origin (center for circle, top left corner for square and triangle, not taken into consideration for vertical lines)
* pixel\_clock: std\_logic, clock signal
* display\_enable: std\_logic, its meaning was discussed in detail in the description of the vga\_controller module
* row: integer, y-coordinate of pixel to be displayed during current clock cycle
* column: integer, x-coordinate of pixel to be displayed during current clock cycle

The inputs and outputs of this module were presented at the beginning of the documentation, in the block diagram section, and will not be reviewed here.

This component has the role of instantiating and connecting all the other major modules so that the final project will have the right functioning.

The first module is the frequency divider. This is connected to the internal clock of the FPGA board which has a frequency of 100MHz. The output will be another clock signal with a frequency of 25MHz and a duty cycle of 50%. It is denoted by *pixel\_clock* since within this project it represents the time available to write a single pixel on the screen. The divider also has a reset function. However, in this project it should work continuously, without any interruptions so the input *reset Divider* is connected to the ground.

The next module is the debounce unit (*DU*). Its purpose is to ‘clean’ all the button signals of the bouncing times. It gets as inputs al the button inputs of the project (which are also inputs of the *top\_module* component) and as a clock signal it is connected to *pixel\_clock*. The 25MHz frequency of this clock should ensure that the debouncers inside the unit create a delay long enough to avoid bouncing. The outputs of this unit are the signals that should be used by other internal modules.

The image signal is connected to a counter as its clock signal. The output of the counter increases with one unit for every button press and it stores the number of the image that is currently displayed. This information will be passed to the *image\_generator* module.

There are two instances of the same component, *positionCalculator*. Each of these instances will compute the position of the object’s origin by computing some intermediary values representing the displacement between the object’s origin and the screen origin (top left corner). One component computes the displacement on the x-axis by taking into consideration the number of times the buttons left and right are pressed: left will decrease the displacement while right will increase the displacement within the screen bounds (this means that the object cannot be moved outside the limits of the screen even if the user tries to do so; e.g. is the square touches the left edge of the screen is can no longer be moved left). The output is the x-coordinate of the origin. A similar logic is used for the other instance: it will compute the y-coordinate of the origin by checking the up and down signals, where up will decrease the displacement and down will increase it. This is also done within the bounds of the screen. Initially both displacements are 0. The outputs are suggestively name x and y.

The *vga\_controller* unit receives as the only input the *pixel\_clock* signal. It will give as outputs horizontal and vertical sync, *display\_enable* signal and the row and column of the current pixel. Out the these, horizontal and vertical sync will also the outputs of the top\_module component, the rest will the linked as inputs for the *image\_generator*.

Finally, *image\_generator* requires as inputs *image\_number, row, column, x, y, Rin, Gin* and *Bin* and based on their value it will compute the color of the current pixel. The output of this unit will be the 12-bit RGB encoding of the color; it also coincides with the rest of the outputs of *top\_module*.

All the outputs of the *top\_module* component will be routed directly to the pins corresponding to the VGA port on the FPGA board. Out of the inputs, *fpga\_clock* is an internal signal of the board. The rest are buttons.

# Justification of the Chosen Solution

We chose to implement the project in a modular and easy to understand fashion. The solution is optimal for several reason. Firstly, the modules are built in such a way that any future improvement or modification would need a minimum amount of changes. Depending on what we want to change, we might only need to update some values used in the controller of maybe some lines of code. At most it might be necessary to change or implement new modules, but since all the major components of the project are independent of each other it is not needed to build a new project from scratch.

Secondly, the code written has a general character meaning that it could be used as it is for similar projects with only few adjustments. The algorithm implemented is the same for any VGA controller no matter what resolution, FPGA board or what image generator module is used, so the project has a high portability.

To sum up, the solution this project gives for a VGA controller is modular, easy to modify and portable to other types of FPGA boards; all these characteristics justify why we chose to implement the project in the manner presented in this documentation.

# Mapping on the FPGA board

We are using the IO of the NEXYS 4 DDR FPGA board for interaction with the outside word.

The outputs we have are connected directly to the VGA port. The pins are mapped as presented below.

The color encoding:

* R bit number 0 is mapped to A3
* R bit number 1 is mapped to B4
* R bit number 2 is mapped to C5
* R bit number 3 is mapped to A4
* G bit number 0 is mapped to C6
* G bit number 1 is mapped to A5
* G bit number 2 is mapped to B6
* G bit number 3 is mapped to A6
* B bit number 0 is mapped to B7
* B bit number 1 is mapped to C7
* B bit number 2 is mapped to D7
* B bit number 3 is mapped to D8

And the horizontal and vertical sync:

* H\_SYNC mapped to B11
* V\_SYNC mapped to B12

For inputs we have switches for the input-color encoding:

* RIN bit number 0 is mapped to T13
* RIN bit number 1 is mapped to R16
* RIN bit number 2 is mapped to U8
* RIN bit number 3 is mapped to T8
* GIN bit number 0 is mapped to R13
* GIN bit number 1 is mapped to U18
* GIN bit number 2 is mapped to T18
* GIN bit number 3 is mapped to R17
* BIN bit number 0 is mapped to R15
* BIN bit number 1 is mapped to M13
* BIN bit number 2 is mapped to L16
* BIN bit number 3 is mapped to J15

For the clock (fpga\_clock) we use the internal clock from the FPGA board mapped by default to pin E3.

The rest of the inputs are buttons.

* IMGBTN mapped to N17: this button when pressed iterates through the images that we draw on the screen.

Finally, the buttons for moving the origin position on the x- and y-axis.

* BTNLEFT mapped to P17
* BTNRIGHT mapped to M17
* BTNUP mapped to M18
* BTNDOWN mapped to P18

# Usage and Maintenance Instructions

A circuit board

Description automatically generated

If we want to see the way the project functions, the first step that must be done is to connect the FPGA board to a display through the VGA port. After that we can load the code on the FPGA board and after a few moments the image of a circle will appear on the screen. From this point forward the user will control what happens next using the buttons and switches available on the board.

There are four different images to be displayed: a circle, a square, a triangle and vertical lines. To switch from image to the next one, button 1 must be pressed once.

The images can be moved within the bounds of the display using four more buttons: buttons 3 and 5 for the x-axis (left and right respectively) and buttons 2 and 4 for the y-axis (up and down respectively).

At any time while the project is active, the color of the objects can be changed using 4 sets of switches. The background will always remain white. The rightmost four switches of the board correspond to the blue component of the color, the next four switches control the green component and the next four control the red component. (Note that the leftmost four switches are not used so changing their state will not have any effect). Within each of these sets of four, the leftmost switch controls the least significant bit of the color encoding, while the rightmost switch controls the most significant bit of the color. Having all the switches set on ‘0’ will make the object black, while having all the switched on ‘1’ will make the object white, which is the same color as the background so in this case the object will no longer be visible.

# Further Development Possibilities

There are several improvements that could be brought to this project. For examples, instead of having a component that decides the color of each pixel based on its position on the screen, the images could be stored into a memory and then received from there when they should be displayed. Since no more computations would be needed in this case, the images could also be more complex, not only geometric shapes. Furthermore, depending on the capacity of the memory used, there could be more images, not just four. Of course, with enough images and with the right timing, instead of displaying independent images, the controller could be programmed to display a movie by displaying related and nearly identical images fast enough so that it gives the viewer the impression of movement. All these changes could be achieved by replacing the image\_generator module from the project with another appropriate component. If the user is no longer given the opportunity to move the images in the x- and y-axis, then the two instances of positionCalculator from the top module could also be removed. This could be the case when the VGA controller projects a movie on the screen and the user does no longer interact with it, they only watch the images displayed.

Another idea of development would be to change the resolution of the images displayed. In order to do this, the only changes needed are the values of some constants inside the vga\_controller module so that they match the characteristics and the timing constraints of the resolution we want to use.

Finally, this project only implements a module that displays images on a screen. It could be integrated into a bigger and more complex project that needs a VGA controller to display images on a screen.