Ministry of Higher Education and Scientific Research
****
University of Monastir
****
Higher Institute of Informatics and Mathematics of Monastir

Microelectronics Engineering Second Year annual Project report:

Advanced Modular VGA Controller (AMVGA) Written in VHDL and Implemented on FPGA

Realized by:

Kayoum DJEDIDI

Presented on the 03/05/2024.

Supervised by:

VHDL Professor: **Mrs. Marwa Fradi**

College year: 2023-2024

Abstract

This report details the development of the Advanced Modular VGA Controller (AMVGA), a student-led project aimed at creating a versatile VGA controller system. The AMVGA system addresses limitations found in traditional VGA controllers by offering support for multiple display resolutions and embedded output options. It provides users with flexibility, allowing them to select from various resolutions, including 640x480 and 1920x1080, and choose preset embedded outputs.

Key components of the project include the design and implementation of VGA controllers for 640x480 and 1920x1080 resolutions, alongside an output generator component. These controllers generate essential signals such as horizontal sync (HS) and vertical sync (VS) pulses while providing outputs for red, green, and blue (RGB) colors. Additionally, a selector component enables users to seamlessly switch between different outputs and preset functions and modes, enhancing system versatility.

The AMVGA system underwent rigorous simulation and testing using the Vivado Design suite to evaluate its performance in accurately and efficiently driving VGA displays. The modular design approach facilitates easy integration of additional output blocks, enabling future expansion and customization. While the project does not aim for groundbreaking achievements, it serves as a valuable learning experience for students, providing insights into VGA controller technology and fostering skills in hardware design, VHDL programming, and FPGA implementation.

Index Terms: *VGA controller, Versatile display resolutions, Embedded output options, Modular design, VHDL programming, FPGA implementation, Vivado Design suite, Hardware design, Simulation, Testing.*

Table of Contents

Table of Figures

General introduction

The Video Graphics Array (VGA) standard, introduced in the late 1980s, has remained a fundamental aspect of PC graphics hardware and monitors. This report delves into the development of a student-led project focused on creating a versatile VGA controller system. The aim is to address the limitations of traditional VGA controllers by offering support for multiple display resolutions and embedded output options.

The project, named the Advanced Modular VGA Controller (AMVGA), emphasizes modularity and adaptability, allowing users to select from various resolutions, including 640x480 and 1920x1080, and choose preset embedded outputs. Key components of the AMVGA system include the design and implementation of VGA controllers for different resolutions, alongside an output generator component. These controllers generate essential signals such as horizontal sync (HS) and vertical sync (VS) pulses while providing outputs for red, green, and blue (RGB) colors. Additionally, a selector component enables users to seamlessly switch between different outputs and preset functions and modes, enhancing system versatility.

The report documents the thorough simulation and testing of the AMVGA system using the Vivado Design suite to evaluate its performance in accurately and efficiently driving VGA displays. The modular design approach facilitates easy integration of additional output blocks, enabling future expansion and customization. While the project does not aim for groundbreaking achievements, it serves as a valuable learning experience for students, providing insights into VGA controller technology and fostering skills in hardware design, VHDL programming, and FPGA implementation.

I. Overview:

The Video Graphics Array (VGA) controller serves as a cornerstone in the realm of visual display technology, providing the interface between computing systems and monitors. Since its inception in the late 1980s, the VGA standard has played a pivotal role in facilitating the display of graphics and video content on various computing platforms.

In essence, a VGA controller orchestrates the generation of critical signals such as horizontal sync (HS) and vertical sync (VS), which synchronize the display's scanning process. These

signals are fundamental in ensuring the orderly rendering of images and text on the screen, guiding the electron beams in CRT monitors or the pixel matrix in modern displays.
Modular design is pivotal in VGA controllers like our AMVGA project, enabling flexibility and scalability. It streamlines component integration and upgrades without necessitating a full system overhaul, reducing development time and costs.

The development of VGA controllers has evolved over the years, with advancements in hardware design, signal processing, and display technologies. Today, the demand for versatile VGA controllers capable of supporting multiple resolutions, driving various display types, and accommodating diverse output options is ever-growing.

1. Development Board and Design Tools:

1.1. Xilinx ZCU104 Development Board

The ZCU104 development board by Xilinx is integral to our project, featuring the advanced Xilinx Zynq UltraScale+ MPSoC and a dedicated VGA port essential for direct VGA display outputs. It is equipped with high-speed DDR4 memory and a variety of connectivity options, facilitating the development of sophisticated VGA solutions. Although full implementation is beyond the scope of this project, I selected this board due to my familiarity with its capabilities and infrastructure, which will streamline potential future implementations.

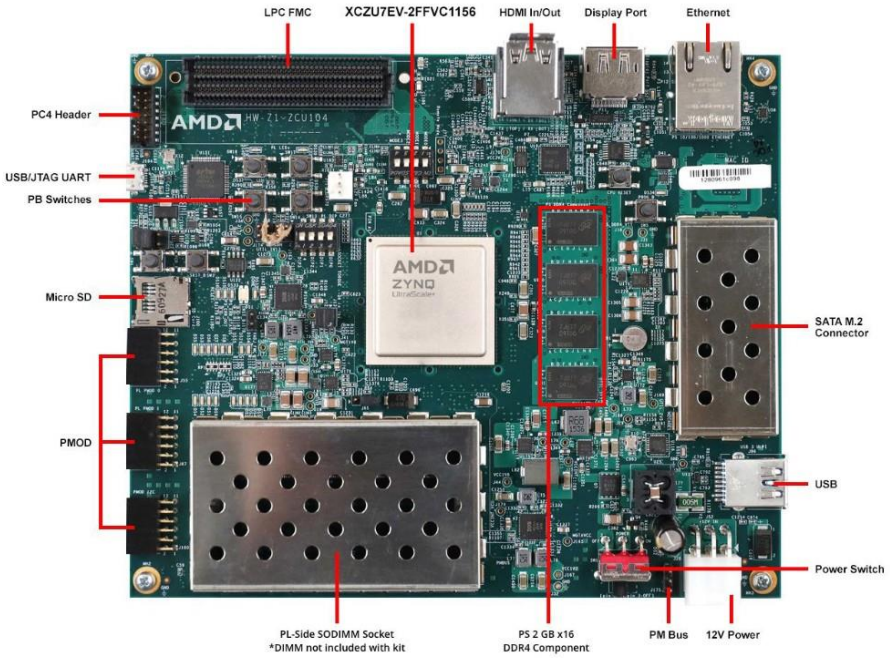


Figure 1 Xilinx ZCU104 development board.

1.1. Xilinx Vivado Design Suite

We use the Xilinx Vivado Design Suite for designing, simulating, and implementing the Advanced Modular VGA Controller (AMVGA). Vivado offers tools for synthesizing VHDL code, visual block design, and robust simulation capabilities. These features allow us to efficiently test and optimize our VGA controller designs on the ZCU104 board, ensuring they meet the required performance specifications.

1.2. The VGA Port:

The VGA port on the ZCU104 development board is a critical interface for our project, featuring five active signals: horizontal and vertical synchronization signals (hsync and vsync), along with three video signals for the red, green, and blue components. These signals are facilitated through a 15-pin D-subminiature connector, typical for VGA interfaces. The video signals are inherently analog; therefore, a digital-to-analog converter (DAC) within the video controller is used to convert digital outputs into corresponding analog levels. For a video signal represented by an N-bit word, the DAC can produce 2^N distinct analog levels. Consequently, the three RGB video signals together can generate 2^(3N) different colors, providing what is referred to as 3N-bit color depth. This capability allows for a broad spectrum of color displays, ranging from simple monochrome to complex multi-color output depending on the bit depth used per video signal.

2. Technical Background

2.1. Explanation of VGA Signals and Operation

2.1.1. Overview

The VGA (Video Graphics Array) standard utilizes several signals to facilitate communication between the graphics hardware and the display. Understanding these signals is crucial for designing and implementing a functional VGA controller system like the Advanced Modular VGA Controller (AMVGA) project. Let's delve into the explanation of VGA signals and their operation:

- Horizontal Sync (HSync) and Vertical Sync (VSync):

The Horizontal Sync (HSync) signal indicates the start of each new scan line on the display. It synchronizes the horizontal scanning process, ensuring that each line is drawn accurately without overlap or gaps.

Vertical Sync (VSync) signal marks the beginning of a new frame or screen refresh. It synchronizes the vertical scanning process, indicating when the electron beam (in CRT monitors) or the pixel matrix (in modern displays) should return to the top of the screen to begin drawing the next frame.

- Video Signals (RGB):

Red, Green, and Blue (RGB) signals represent the primary color components of the display. These analog signals determine the intensity of each color on the screen, combining to produce a wide range of colors and hues.

Each RGB signal varies in voltage level to represent different color intensities. For example, higher voltage levels indicate brighter colors, while lower levels represent darker shades.

Sync Pulses and Back Porch/ Front Porch:

Sync pulses are brief signals embedded within the HSync and VSync signals, indicating the start of each scan line and frame, respectively.

Back Porch and Front Porch refer to the intervals between the sync pulses and the active video signal. These intervals allow for blanking periods to ensure smooth transitions between scan lines and frames without visual artifacts.

- Composite Sync (CSync):

Composite Sync (CSync) combines the HSync and VSync signals into a single synchronization signal. It simplifies the synchronization process, particularly in modern display systems, and is beyond the scope of this project.

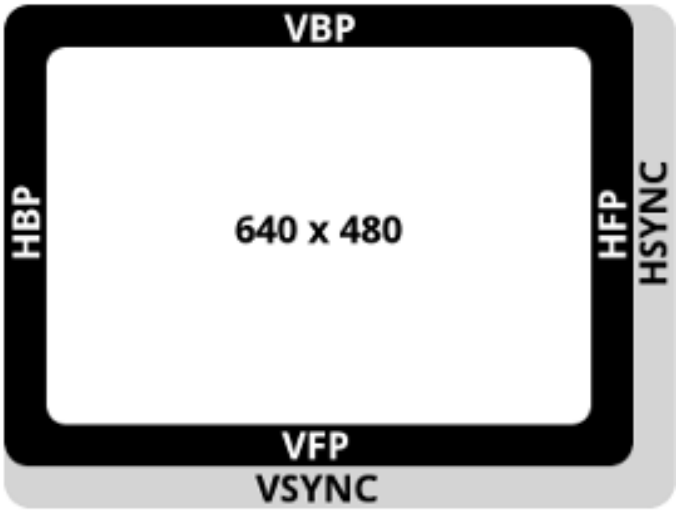


Figure 2 Parts of a screen.

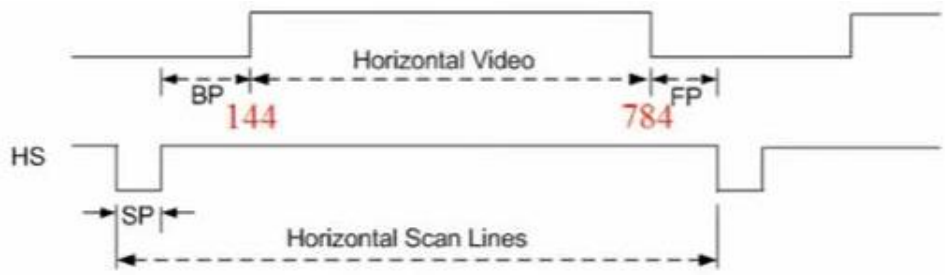
2.1.2. VGA Timing

2.1.2.1. **Horizontal Timing**

A standard monitor operates at 60 Hz. θ That is, the screen is refreshed 60 times per second, which is equal to once every 1/60th of a second. Therefore, given a target resolution supported by the monitor, timing for each pixel, horizontal line, and vertical retrace can be derived. We will design a VGA controller for a standard screen displaying a resolution of 640*480, a common, lower resolution for a VGA monitor. Then use the same principals for a 1920*1080

resolution. A 25 MHz clock will be used to drive the controller and as we will later discover, it is sufficient to accomplish the target resolution. We will refer to this as the pixel clock.

- First, we must derive the timing for the horizontal sync signal, HS.
- This signal consists of four regions, the sync pulse (SP), back porch (BP), horizontal video (HV), and front porch (FP).
- The sync pulse signals the beginning of a new line and is accomplished by bringing HS low. The signal is then brought high for the back porch where pixels are not yet written to the screen at the left.
- After the back porch, the HS signal remains high during the horizontal video period where pixels are written to the screen proceeding from left to right.
- Finally, the HS signal also remains high during the front porch where no pixels are written to the screen at the right.
- Figure 3 shows the horizontal sync regions and timing, while table one summarizes the timings.



Pixel clock = 25 MHz Pixel time = 0.04 µs
Horizontal video = 640 pixels x 0.04 µs = 25.60 µs
Back porch, BP = 16 pixels x 0.04 µs = 0.64 µs
Front porch, FP = 16 pixels x 0.04 µs = 0.64 µs
Sync pulse, SP = 128 pixels x 0.04 µs = 5.12.µs
Horizontal Scan Lines = SP + BP + HV + FP
= 128 + 16 + 640 + 16
= 800 pixels x 0.04 µs = 32 µs
1/60 Hz = 16.67 ms / 32 µs = 521 horizontal scan lines per frame

Figure 3 Horizontal timing.

25MHz pixel clock	Active video	Front porch	Sync pulse	Back porch
Horizontal	640	16	96	48
Vertical	480	11	2	31

Table 1: VGA Timings

- At a 25 MHz pixel clock, each pixel will take 1/25*106 = 0.04µs.
- Next, we should consider the horizontal video period since we desire 640 pixels per line for a 640*480 resolution.
- Six hundred forty pixels requires 640*0.04µs =25.60 µs to display across one line.
- According to specification, the length of the sync pulse, SP, should be approximately one-fifth of the horizontal video time.
- Therefore, SP equals 25.60 µs /5 = 5.12 µs. θ Using our 25 MHz clock, this means that SP requires 5.12 µs /0.04 µs =128 clock ticks or pixels.
- Also, according to specification, the back and front porches should each be approximately one fortieth of the time required for the horizontal video.
- Therefore, the porches require 25.60 µs / 40 = 0.64 µs each. θ Using a 25 MHz pixel clock, each porch requires 0.64 µs / 0.04 µs = 16 clock ticks or pixels. θ Finally, calculating the total number of clock pulses for a line we have SP + BP + HV + FP = 128 + 16 + 640 + 16 = 800 pixels clock pulses at 0.04 µs per pulse yields 32 µs for the entire line.
- Considering the line by counting pixels, we have 128 invisible pixels for the sync pulse where HS is low.
- The horizontal video region, where pixels are visible, starts at 128 + 16 =144 after the sync pulse and invisible back porch and continues for 640 pixels stopping at 144 + 640 =784. θ At this point, there is no visible video again at the front porch for 16 pixels until 784 + 16 = 800 where the sequence starts over again.
- Given that one entire screen, or frame, must be written in 1/60th of a second or 16. 67 ms and each line requires 32 µs, it is possible to write 16.67 ms / 32 µs =521 horizontal lines per screen. This is consistent with our target resolution of 640*480 which requires 480 lines per screen.
- Since 521 is slightly greater than 480, it is perfect for synchronizing the vertical sync signal since it also has a vertical sync pulse, back porch, vertical video, and front porch region.
- Of the four regions that make up the vertical sync signal, the sync pulse (SP), back porch (BP), vertical video (VV), and front porch (FP), we already know that the vertical video region must be 480 lines.

Figure 4 shows the vertical sync signal and timing.

2.1.2.2. **Vertical Timing**

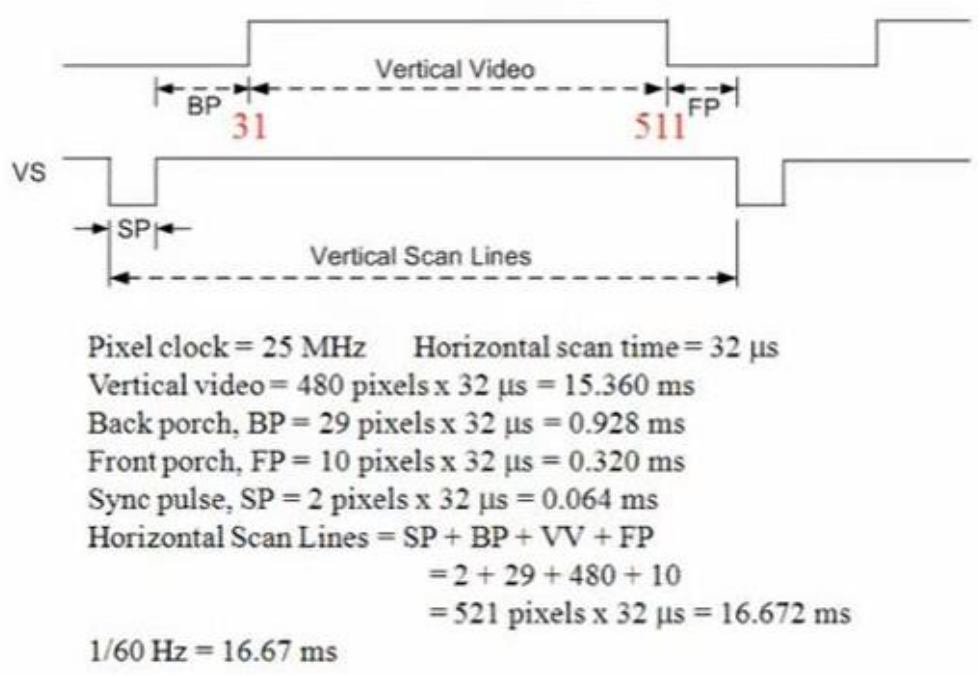


Figure 4 Vertical Timing

- The vertical video region requires 480 lines at 32 µs per line = 15.360 ms.
- According to specification, the vertical sync pulse should be approximately 1/240th of the vertical video timing.
- Therefore, 15.360 ms / 240 = 0.064 ms for the vertical SP.
- At 32 µs per line, this requires 0.064 ms /32 µs = 2 lines.
- Finally, splitting the remaining 39 lines between the back porch and the front porch using a 75%, 25% split as per specification, the back porch will be 29 lines and the front porch will be 10 lines.
- Finally, the 521 lines are split into a vertical SP + BP + VV + FP = 2 + 29 + 480 + 10.
- Considering the frame per counting lines, the VS signal is low for 2 lines to set the sync pulse SP.
- Then, VS is brought high for the remaining 519 lines.
- There is no visible video for the 29 lines back porch followed by 480 lines of visible video until the counter has reached 31 +480 = 511 where there is no visible video again for the 10 lines front porch.

- The counter continues until it reaches 511 +10 = 521 where the sequence starts over again and draws a new frame.

2.1.2.3. **Other resolutions**

There are many other standard resolutions supported by most modern screens, such as 800x600 and 1024x768. However, **achieving higher resolutions, like 1920x1080@60Hz, requires a significantly higher pixel clock frequency, in this case, 148.5MHz** - a fairly demanding requirement. To accommodate this resolution, we upgraded the VHDL component clkdiv to include a resolution select input, allowing for switching between output clock frequencies of 25MHz and 148.5MHz based on the selected resolution. This enhancement involved modifying the entity to incorporate the new input and adjusting the clock division logic accordingly. With the original 25MHz clock, supporting a resolution like 800x600 would be insufficient due to the lower frequency. For instance, expanding the visible pixels across to 800 would drastically reduce the number of vertical lines, potentially falling short of the desired 600 visible vertical lines. This underscores the necessity of adjusting clock frequencies to meet the demands of various resolutions, ensuring optimal VGA controller performance across different display configurations.

For any desired resolution, the pixel clock frequency must be calculated such that the following are true:

Given: a target HV and VV pixel resolution

Horizontal constraints

HSP = 1/5 * HV	(horizontal sync pulse pixels)
HBP = 1/40 * HV	(horizontal back porch pixels)
HFP = 1/40 * HV	(horizontal front porch pixels)

Vertical constraints

VSP = 1/240 * VV	(vertical sync pulse lines)
VFP+VBP= 0.08125 * VV	
VBP = .75 * (VFP+VBP)	(vertical back porch lines)
VFP = .25 * (VFP+VBP)	(vertical front porch lines)

where HSP+HBP+HV+HFP = HP_{total} pixels and VSP+VBP+VV+VFP = VP_{total} lines.
Recall that VP_{total} must occur in $1/60^{th}$ of a second. Then,

16.67ms / VP_{total} = T_{line} ms

and T_{line} / HPtotal = T_{pixel} ms

Figure 5 Clock timing guide.

II. Design and implementation.

1.1. Insights and workflow

The AMVGA project began with a simpler concept, but through ongoing upgrades and refinements, it evolved into the sophisticated system depicted in the top-level diagram. As we explore this report, I'll provide insights into the project's development stages and the enhancements made along the way, showcasing the iterative nature of the design process.

1.2. Top Level Block

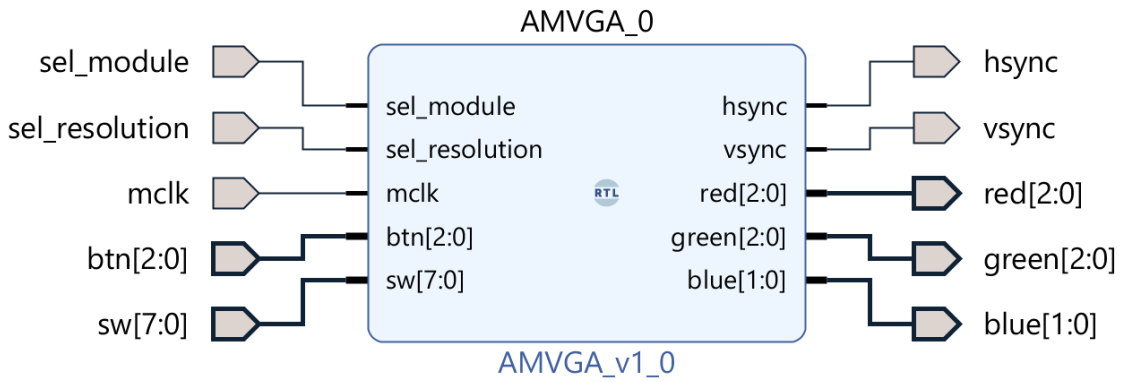


Figure 6 Top Level Block

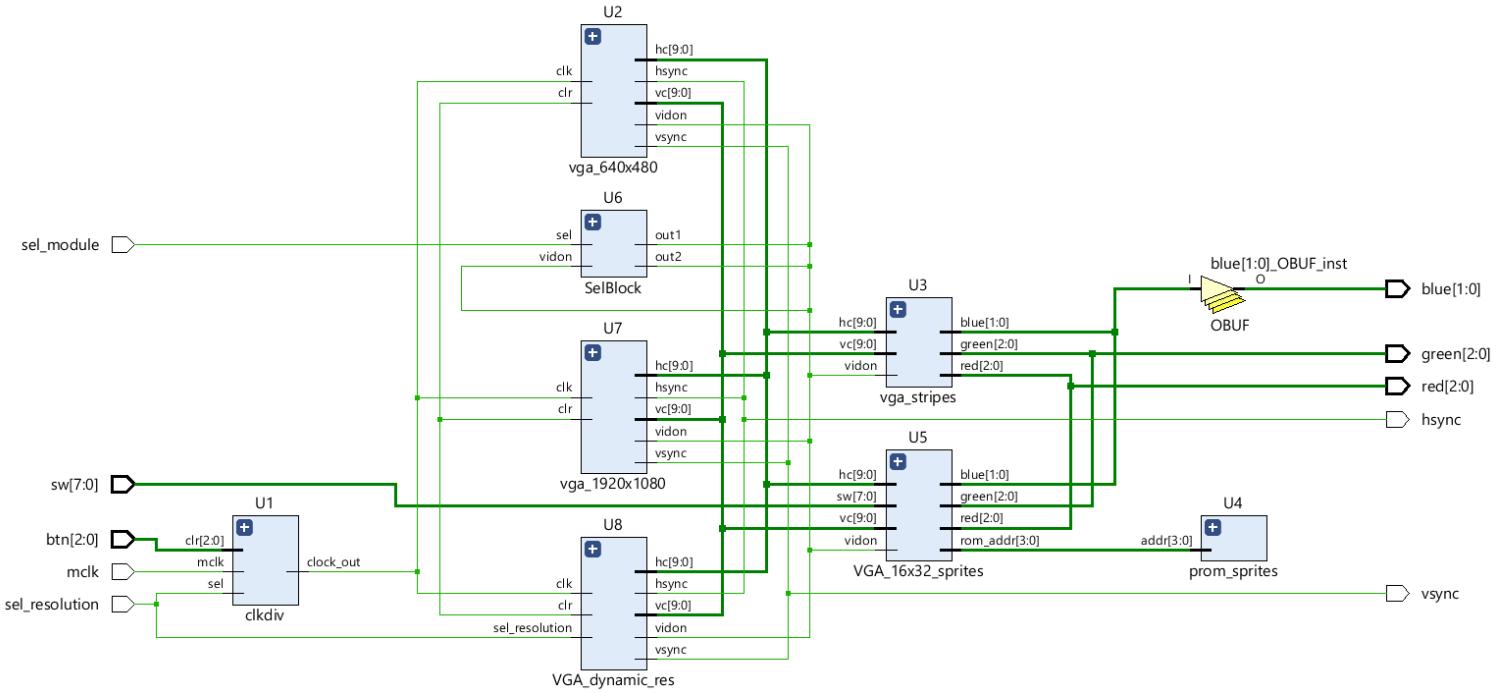


Figure 7 RTL Elaborated Design

The VHDL code provided describes the top-level architecture for the Advanced Modular VGA Controller (AMVGA), integrating various components to enable versatile VGA display functionality. This design leverages multiple specialized subcomponents to handle different aspects of VGA processing, allowing for dynamic resolution changes and customizable graphic displays. Below is a breakdown of how each component fits into the system and their interconnections:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.all;
4
5  entity AMVGA is
6         port(
7         sel_module : in std_logic;
8         sel_resolution: in std_logic;
9         mclk : in std_logic ;
10        btn : in std_logic_vector(2 downto 0);
11        sw : in std_logic_vector(7 downto 0);
12        hsync : out std_logic ;
13        vsync : out std_logic ;
14        red : out std_logic_vector(2 downto 0);
15        green : out std_logic_vector(2 downto 0);
16        blue : out std_logic_vector(1 downto 0)
17        );
18 end AMVGA;
19 architecture arch_stripes_top of AMVGA is
20
21 component clkdiv port ( mclk, sel : in std_logic ;
22 clr : in std_logic_vector (2 downto 0);
23 clock_out: out std_logic);
24 end component;
25
26 component vga_640x480 port (clk, clr : in std_logic;
27        hsync : out std_logic;
28        vsync : out std_logic;
29        hc : out std_logic_vector(9 downto 0);
30        vc : out std_logic_vector(9 downto 0);
31        vidon : out std_logic);
32
33     end component;
```

```
34  component prom_sprites port (
35          addr: in std_logic_vector (3 downto 0);
36          Output_sprite: out std_logic_vector (0 to 31)
37      );
38     end component;
39
40  component VGA_16x32_sprites  port (
41          sw : in std_logic_vector(7 downto 0);
42          hc : in std_logic_vector(9 downto 0);
43          vc : in std_logic_vector(9 downto 0);
44          vidon : in std_logic;
45          rom_data : in std_logic_vector(31 downto 0);
46          rom_addr : out std_logic_vector(3 downto 0);
47          red : out std_logic_vector(2 downto 0);
48          green : out std_logic_vector(2 downto 0);
49          blue : out std_logic_vector(1 downto 0)
50      );
51      end component;
52
53  component SelBlock Port ( sel : in STD_LOGIC;
54             vidon : in STD_LOGIC;
55             out1 : out STD_LOGIC;
56             out2 : out STD_LOGIC);
57     end component;
58
59  component InputSelector Port ( sel,clk,clr : in std_logic;
60             inputs : in std_logic_vector(9 downto 0);
61             out1, out2, out3, out4, out5 : out std_logic;
62             out6, out7, out8, out9, out10 : out std_logic);
63    end component;
64
65
66   component vga_dynamic_res Port (
67          sel_resolution : in std_logic;
68          clk, clr : in std_logic;
69          hsync : out std_logic;
70          vsync : out std_logic;
71          hc : out std_logic_vector(9 downto 0);
72          vc : out std_logic_vector(9 downto 0);
73          vidon : out std_logic
74      );
75
76     end component;
```

```
77
78  signal clk25, vidon,clr1,clr: std_logic;
79  signal hc, vc : std_logic_vector(9 downto 0);
80  signal rom_data :  std_logic_vector(31 downto 0);
81  signal rom_addr :  std_logic_vector(3 downto 0);
82  signal Output_sprite: std_logic_vector (0 downto 31);
83  signal in0,in1,out_data: std_logic_vector(7 downto 0);
84  signal hsync1,hsync2,vsync1,vsync2 : std_logic;
85  signal hc1,hc2,vc1,vc2 : std_logic_vector(9 downto 0);
86  signal vidon1,vidon2 : std_logic;
87  begin
88
89  U1: clkdiv
90      port map (
91                  mclk => mclk,
92                  sel=> sel_resolution,
93                  clr => btn,
94                  clock_out => clk25
95      );
96
97  U3 : vga_stripes
98          port map (
99                  vidon => vidon,
100                 hc => hc,
101                 vc => vc,
102                 red => red,
103                 green => green,
104                 blue => blue
105     );
106 U4: prom_sprites
107     port map(
108         addr =>rom_addr,Output_sprite=>rom_data
109    );
110
111 U5: VGA_16x32_sprites
112 port map( sw => sw,
113         hc => hc,
114         vc => vc,
115         vidon => vidon,
116         rom_data => Output_sprite,
117         rom_addr => rom_addr,
118         red => red,
119         green  => green,
120         blue  => blue );
```

```
120         blue  => blue );
121
122 U6: SelBlock
123 port map (sel=>sel_module, vidon=>vidon, out1=>vidon, out2=>vidon);
124
125  U8: vga_dynamic_res
126     Port map (
127         sel_resolution =>sel_resolution,
128         clk => clk25, clr =>clr1,
129         hsync => hsync,
130         vsync => vsync,
131         hc => hc,
132         vc => vc,
133         vidon => vidon );
134
135 end arch_stripes_top;
136
```

1.3. Components and Connectivity

1.3.1. Clock Divider (clkdiv)

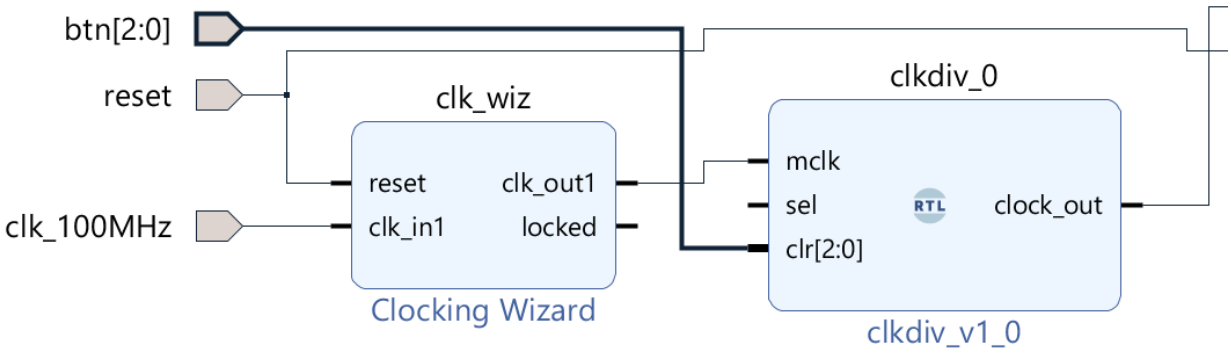


Figure 8 Clock Divider (clkdiv)

The "clkdiv" component is a clock divider module designed to generate an output clock signal at either 25MHz or 148.5MHz based on the selection input "sel". It takes a master clock signal "mclk" as input and produces the divided clock output "clock_out".

The "sel" input determines which clock frequency to use when "sel" is '1', the output clock frequency is set to 148.5MHz, and when "sel" is '0', it defaults to 25MHz.

Internally, the component counts the number of master clock cycles and toggles the output signal accordingly. When the count reaches the maximum count value corresponding to the selected frequency, the output signal is toggled, and the count is reset to 1.

This component enables the VGA controller to switch between the required clock frequencies seamlessly, allowing for the proper synchronization of display signals with different resolutions. It's also coupled with the predefined Clocking Wizard.

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

entity clkdiv is
  port (
    mclk : in std_logic;
    sel : in std_logic; -- New input for selecting clock frequency
    clr : in std_logic_vector (2 downto 0);
    clock_out: out std_logic
  );
end clkdiv;

architecture bhv of clkdiv is

  constant MCLK_FREQ : real := 100.0e6; -- Master clock frequency
  constant FREQ_25MHZ : real := 25.0e6;
  constant FREQ_148_5MHZ : real := 148.5e6;
  signal count: integer := 1;
  signal tmp : std_logic := '0';
  signal max_count : integer := integer(MCLK_FREQ / (2.0 * FREQ_25MHZ)); -- Default to 25MHz

begin

  process(mclk, clr, sel)
  begin
    if(clr = "111") then
      count <= 1;
      tmp <= '0';
    elsif(mclk'event and mclk = '1') then
      if sel = '1' then
        max_count <= integer(MCLK_FREQ / (2.0 * FREQ_148_5MHZ)); -- Set for 148.5 MHz
      else
```

1.3.2. VGA Stripes Generator (vga_stripes)

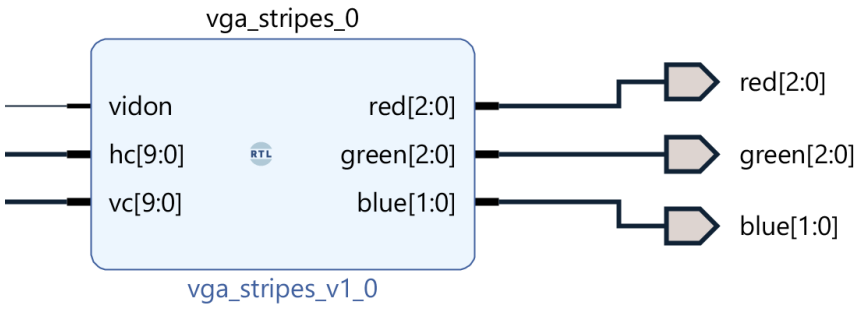


Figure 9 VGA Stripes Generator (vga_stripes)

The "vga_stripes" component is a module responsible for generating VGA signals to produce vertical stripes on the display screen. It takes inputs such as "vidon" (video on/off signal), "hc"

(horizontal counter), and "vc" (vertical counter), and produces output signals for red, green, and blue components of the display.

Within the process, when "vidon" is high (indicating video output is active), the component generates alternating red and green stripes vertically across the screen. It achieves this by setting the red signal to a constant value and toggling the green signal. The "vc(4)" signal is used to create the alternating pattern, with "red" and "green" signals being assigned accordingly.

The "blue" signal is kept constant, indicating no blue component in the generated stripes.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.all;
4  entity vga_stripes is
5      port ( vidon : in std_logic;
6      hc : in std_logic_vector(9 downto 0);
7      vc : in std_logic_vector(9 downto 0);
8      red : out std_logic_vector(2 downto 0);
9      green : out std_logic_vector(2 downto 0);
10     blue : out std_logic_vector(1 downto 0));
11 end vga_stripes;
12 architecture arch_stripes of vga_stripes is
13 begin
14       process(vidon, vc)
15       begin
16            red <="000";
17            green <="000";
18            blue <= "00";
19            if vidon = '1' then
20                    red <= vc(4) & vc (4) & vc (4);
21                    green <= not (vc(4) & vc (4) & vc (4));
22             end if;
23         end process;
24 end arch_stripes;
```

1.3.3. PROM Sprites (prom_sprites)

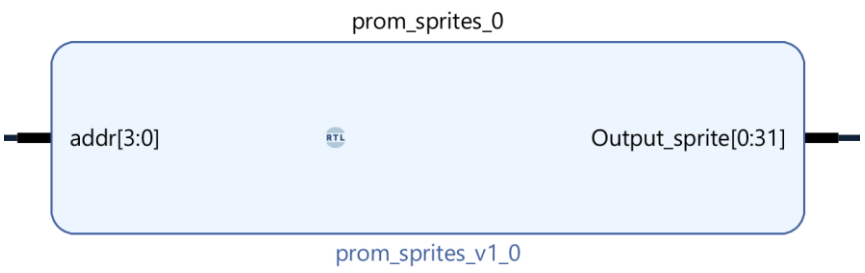


Figure 10 PROM Sprites (prom_sprites)

The "prom_sprites" component serves as a programmable read-only memory (ROM) for storing sprite data. It takes an address input "addr" to select the desired sprite from memory and outputs the corresponding sprite data through "Output_sprite".

Internally, the component defines two ROM arrays, "rom" and "rom_1", each containing 16 sprite patterns represented as binary strings. The process statement reads the address input "addr" and converts it to an integer value "j". It then uses this value to index into the ROM arrays and assigns the selected sprite data to "Output_sprite".

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4
5  entity prom_sprites is
6      port (
7          addr: in std_logic_vector (3 downto 0);
8          Output_sprite: out std_logic_vector (0 to 31)
9      );
10 end prom_sprites;
11
12 architecture arch of prom_sprites is
13     type rom_array is array (natural range <>) of std_logic_vector(0 to 31);
14     constant rom: rom_array := (
15         "01111110001011000001101000000010", --0
16         "01000001000011000001101000000010", --1
17         "01000001000101000101010000000010", --2
18         "01000000100101000101010000000010", --3
19         "01000000001010100010101000000010", --4
20         "01000000001010010100101000000010", --5
21         "01000000001010010100101000000010", --6
22         "01000000001010010100101111111110", --7
23         "01000000001010001000101000000010", --8
24         "01000000001010001000101000000010", --9
25         "01000000001010001000101000000010", --10
26         "01000000001010000000101000000010", --11
27         "01000000100100000000101000000010", --12
28         "01000001000100000000101000000010", --13
29         "01000001000100000000101000000010", --14
30         "01111110000100000000101000000010"  --15
31 );
32
33  type rom_array_1 is array (natural range <>) of std_logic_vector(0 to 31);
```

1.3.4. VGA 16x32 Sprites (VGA_16x32_sprites)

The "VGA_16x32_sprites" component is responsible for displaying sprites on a VGA screen with a resolution of 16x32 pixels. It takes inputs such as "sw" (switch settings), "hc" (horizontal counter), "vc" (vertical counter), "vidon" (video on/off signal), and "rom_data" (sprite data from

ROM). Additionally, it outputs the sprite address "rom_addr" and color signals "red", "green", and "blue" for display.

Within the process, when "vidon" is high (indicating video output is active), the component checks if the current horizontal and vertical counter values fall within the active display region for the sprite. If they do, it calculates the sprite address based on the vertical counter and sets the color signals based on the corresponding data from the ROM.

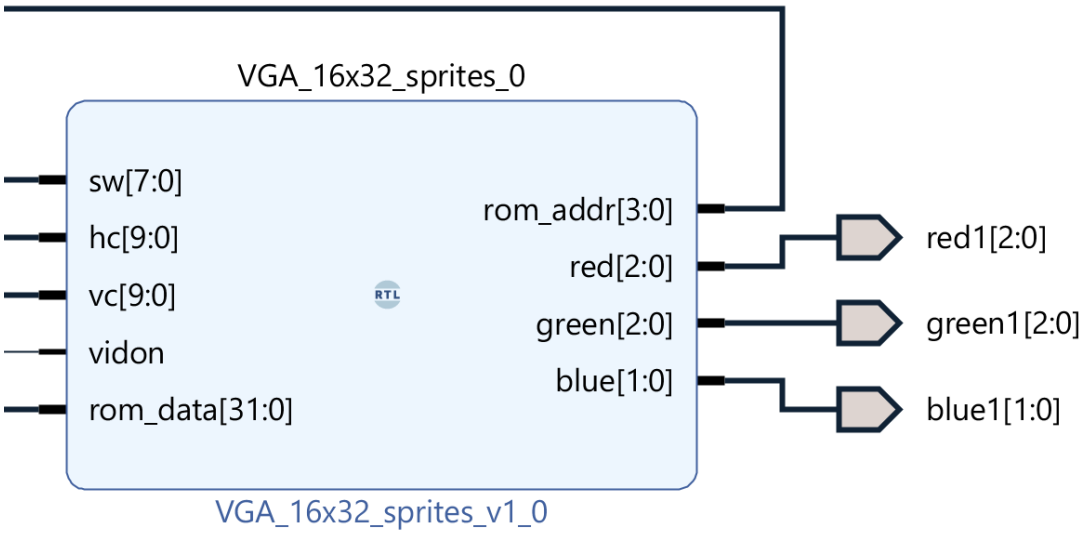


Figure 11 VGA 16x32 Sprites (VGA_16x32_sprites)

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4
5  entity VGA_16x32_sprites is
6      port (
7          sw : in std_logic_vector(7 downto 0);
8          hc : in std_logic_vector(9 downto 0);
9          vc : in std_logic_vector(9 downto 0);
10         vidon : in std_logic;
11         rom_data : in std_logic_vector(31 downto 0);
12         rom_addr : out std_logic_vector(3 downto 0);
13         red : out std_logic_vector(2 downto 0);
14         green : out std_logic_vector(2 downto 0);
15         blue : out std_logic_vector(1 downto 0)
16     );
17 end entity VGA_16x32_sprites;
18
19 architecture behavioral of VGA_16x32_sprites is
20     signal C1, R1 : natural := 0;
21     signal rom_pix : natural := 0;
22
23     constant w : natural := 32;
24     constant h : natural := 16;
25     constant hbp : natural := 96;
26     constant vbp : natural := 2;
27
28 begin
29     process (hc, vc, vidon, rom_data)
30     begin
31         if vidon = '1' then
32             if (to_integer(unsigned(hc)) >= hbp + C1 and to_integer(unsigned(hc)) < hbp + C1 + w) and
33                (to_integer(unsigned(vc)) >= vbp + R1 and to_integer(unsigned(vc)) < vbp + R1 + h) then
```

1.3.5. Selection Block (SelBlock)

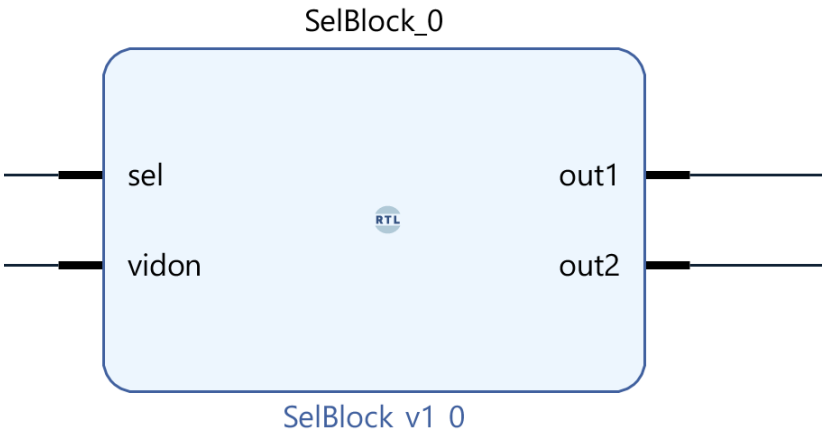


Figure 12 Selection Block (SelBlock)

The "SelBlock" component is a selector block module designed to switch between two input signals based on the value of the selection signal "sel". It has inputs "sel" and "vidon" and outputs "out1" and "out2".

Within the process, when "sel" is low ('0'), "out1" receives the value of "vidon", while "out2" is set to low ('0'). Conversely, when "sel" is high ('1'), "out1" is set to low ('0'), and "out2" receives the value of "vidon".

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity SelBlock is
5      Port ( sel : in STD_LOGIC;
6             vidon : in STD_LOGIC;
7             out1 : out STD_LOGIC;
8             out2 : out STD_LOGIC);
9  end SelBlock;
10
11 architecture Behavioral of SelBlock is
12 begin
13     process(sel, vidon)
14     begin
15         if sel = '0' then
16             out1 <= vidon;
17             out2 <= '0';
18         else
19             out1 <= '0';
20             out2 <= vidon;
21         end if;
22     end process;
23 end Behavioral;
```

1.3.6. Dynamic Resolution VGA Controller (vga_dynamic_res)

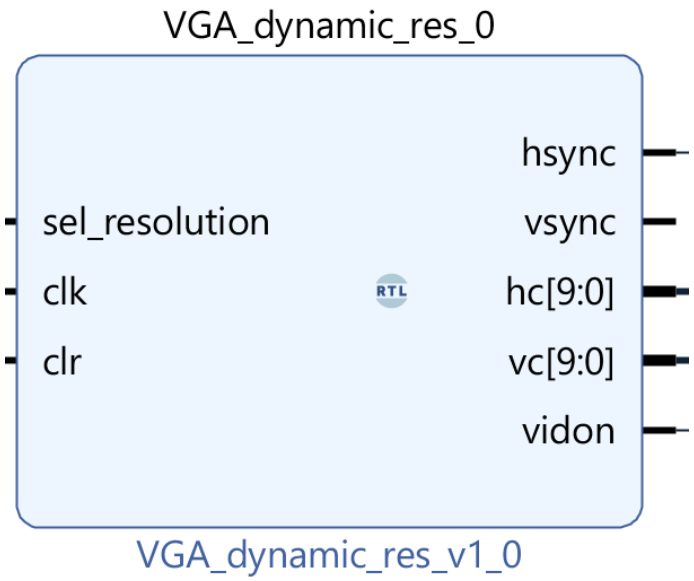


Figure 13 Dynamic Resolution VGA Controller (vga_dynamic_res)

The "VGA_dynamic_res" component is a VGA controller module capable of dynamically switching between two different resolutions: 640x480 and 1920x1080. It takes inputs such as "sel_resolution" (resolution selection signal), "clk" (clock signal), and "clr" (clear signal), and outputs VGA signals such as "hsync" (horizontal synchronization), "vsync" (vertical synchronization), "hc" (horizontal counter), "vc" (vertical counter), and "vidon" (video on/off signal).

Internally, it instantiates two VGA controller components, "vga_640x480" and "vga_1920x1080", representing the two supported resolutions. Based on the value of "sel_resolution", it selectively connects the output signals of the appropriate VGA controller to the corresponding output ports.

When "sel_resolution" is low ('0'), indicating selection of the 640x480 resolution, the component routes the VGA signals from the "vga_640x480" component to the output ports. Otherwise, when "sel_resolution" is high ('1'), indicating selection of the 1920x1080 resolution, it sets all output signals to low ('0').

This component provides flexibility in supporting different display resolutions, allowing for seamless integration into systems requiring dynamic resolution switching.

- Handles dynamic switching between resolutions based on `sel_resolution`.
- Outputs sync signals (`hsync`, `vsync`), video enable (`vidon`), and coordinates (`hc`, `vc`), managing the overall display timing and synchronization.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.all;
4
5  entity VGA_dynamic_res is
6      Port (
7          sel_resolution : in std_logic;
8          clk, clr : in std_logic;
9          hsync : out std_logic;
10         vsync : out std_logic;
11         hc : out std_logic_vector(9 downto 0);
12         vc : out std_logic_vector(9 downto 0);
13         vidon : out std_logic
14     );
15 end VGA_dynamic_res;
16
17 architecture Behavioral of VGA_dynamic_res is
18     component vga_640x480 is
19         Port (
20             clk, clr : in std_logic;
21             hsync : out std_logic;
22             vsync : out std_logic;
23             hc : out std_logic_vector(9 downto 0);
24             vc : out std_logic_vector(9 downto 0);
25             vidon : out std_logic
26         );
27     end component;
28
29     component vga_1920x1080 is
30         Port (
31             clk, clr : in std_logic;
32             hsync : out std_logic;
33             vsync : out std_logic;
34             hc : out std_logic_vector(9 downto 0);
35             vc : out std_logic_vector(9 downto 0);
36             vidon : out std_logic
37         );
38     end component;
39
40     signal hsync_int, vsync_int : std_logic;
41     signal hc_int, vc_int : std_logic_vector(9 downto 0);
42     signal vidon_int : std_logic;
43 begin
44     VGA640: vga_640x480
45         port map (
46             clk => clk,
```

```
47              clr => clr,
48              hsync => hsync_int,
49              vsync => vsync_int,
50              hc => hc_int,
51              vc => vc_int,
52              vidon => vidon_int
53          );
54
55      VGA1920: vga_1920x1080
56          port map (
57              clk => clk,
58              clr => clr,
59              hsync => hsync_int,
60              vsync => vsync_int,
61              hc => hc_int,
62              vc => vc_int,
63              vidon => vidon_int
64          );
65
66      process (sel_resolution)
67      begin
68          if sel_resolution = '0' then
69              hsync <= hsync_int;
70              vsync <= vsync_int;
71              hc <= hc_int;
72              vc <= vc_int;
73              vidon <= vidon_int;
74          else
75              hsync <= '0';
76              vsync <= '0';
77              hc <= (others => '0');
78              vc <= (others => '0');
79              vidon <= '0';
80          end if;
81      end process;
82  end Behavioral;
```

1.3.7. Input Selector (InputSelector) *(Defined but not used in the connection map)

The "InputSelector" component acts as a multiplexer for selecting between two sets of inputs based on the value of the selection signal "sel". It has inputs "sel", "clk0", "clr0", "clk1", "clr1", and "inputs", and outputs "out1" through "out10".

This component would typically be used to select between different input signals based on a selection control and resolution of choice, though it's not connected in the provided architecture. Since I merged both resolution options and selection into one component.

```
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity InputSelector is
6      Port ( sel,clk0,clr0,clk1,clr1 : in std_logic;
7             inputs : in std_logic_vector(9 downto 0);
8             out1, out2, out3, out4, out5 : out std_logic;
9             out6, out7, out8, out9, out10 : out std_logic);
10 end InputSelector;
11
12 architecture Behavioral of InputSelector is
13 begin
14     process(sel, inputs)
15     begin
16         if sel = '0' then
17             out1 <= inputs(9);
18             out2 <= inputs(8);
19             out3 <= inputs(7);
20             out4 <= inputs(6);
21             out5 <= inputs(5);
22             out6 <= '0';
23             out7 <= '0';
24             out8 <= '0';
25             out9 <= '0';
26             out10 <= '0';
27         else
28             out1 <= '0';
29             out2 <= '0';
30             out3 <= '0';
31             out4 <= '0';
32             out5 <= '0';
33             out6 <= inputs(4);
```

1.4. Other aspects

1.4.1. Signals

- Various control signals like `clr` (clear), `vidon` (video enable), `hc` and `vc` (horizontal and vertical counters) are used to coordinate and synchronize the video output across components.
- `hsync` and `vsync` are the synchronization pulses for the VGA output.
- RGB signals (`red`, `green`, `blue`) are derived from the VGA stripes and sprite components to form the final video output.

1.4.2. Clock Generation

- The `clkdiv` component generates a clock signal suitable for the selected resolution, toggling between frequencies suitable for standard VGA (25 MHz) and potentially higher resolutions (e.g., 148.5 MHz).

1.4.3. Video Signal Generation

- `vga_dynamic_res` sets the base video timing and synchronization based on the selected resolution.

- `vga_stripes` and `VGA_16x32_sprites` generate graphical content, which is then managed based on the output from `SelBlock` to enable or disable video output dynamically.

1.4.4. Sprite Handling

- The `prom_sprites` component serves as a lookup table for sprite graphics, which are then processed and mapped onto the display by the `VGA_16x32_sprites`.

1.5. Architectural Integration

Below is the generated block diagram for this syste, this architecture effectively integrates various functional blocks to support a modular and dynamic VGA display system. It demonstrates flexibility in managing different display modes and resolutions, driven by external inputs like buttons (`btn`), switches (`sw`), and selection signals (`sel_module`, `sel_resolution`). Each component is designed to interact seamlessly, providing a cohesive and configurable VGA output capable of supporting educational and developmental applications in FPGA and VHDL training environments.

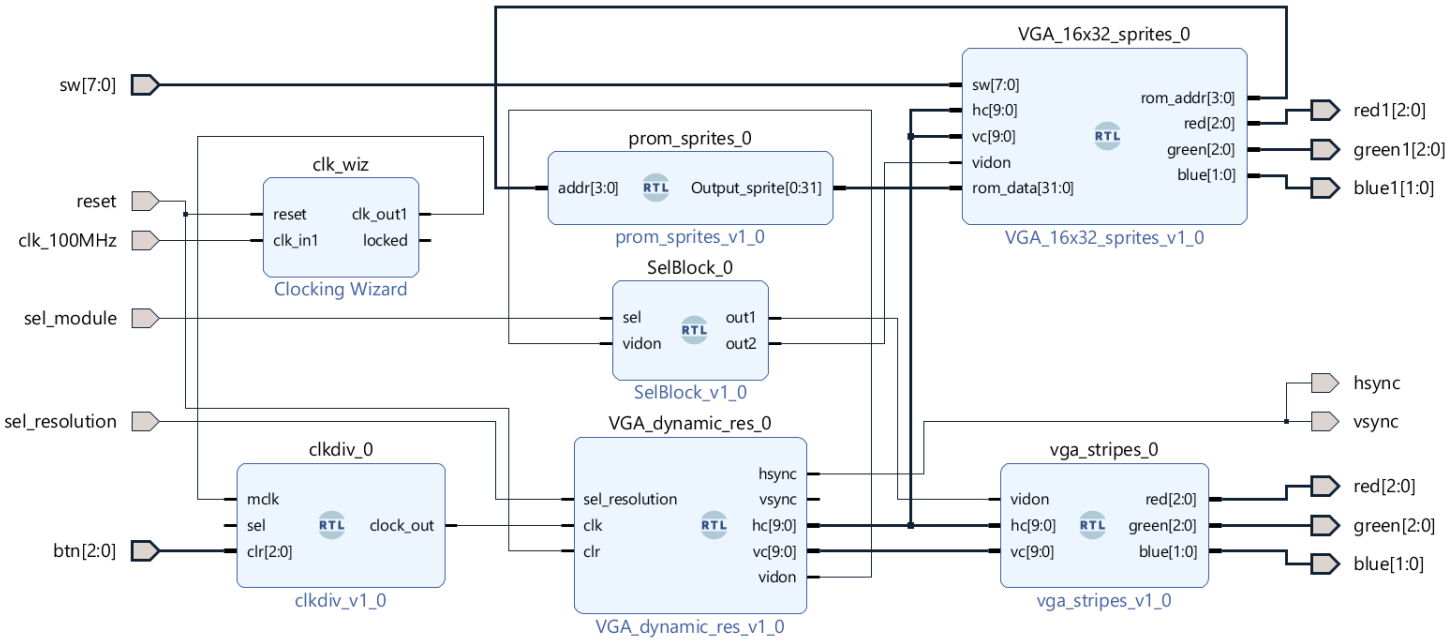


Figure 14 Architectural Integration Block diagram.

2. Validation

2.1. Test Bench and simulation

In our Advanced Modular VGA Controller (AMVGA) project, the use of simulation and test benches is instrumental in validating the intricate functionalities of our VHDL-designed FPGA system. The test bench specifically crafted for this project is tailored to simulate every component of the AMVGA, from clock division and resolution switching to the generation and handling of VGA signals such as horizontal sync (hsync), vertical sync (vsync), and RGB color outputs.

This simulation framework allows us to meticulously test the dynamic resolution adjustments that our VGA controller supports—switching between different predefined resolutions to ensure the VGA output adheres to the expected timing and color specifications. We apply a variety of digital inputs, including module selection and resolution change signals, to evaluate how our design handles different user interactions and display settings.

Additionally, the test bench verifies the integration of sprite and pattern generation modules to ensure that graphical outputs are correctly overlaid onto the VGA signal without disruptions. By emulating real-world scenarios and edge cases within this controlled virtual environment, we can closely monitor the system's response and fine-tune its performance. This rigorous simulation process is crucial in ensuring that our VGA controller is both robust and versatile, ready for deployment in educational settings or as a foundation for further research and development in digital display technology.

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.ALL;
3   USE ieee.numeric_std.ALL;
4
5   ENTITY AMVGA_tb IS
6   END AMVGA_tb;
7
8   ARCHITECTURE behavior OF AMVGA_tb IS
9       -- Component Declaration for the Unit Under Test (UUT)
10      COMPONENT AMVGA
11      PORT(
12           sel_module : IN std_logic;
13           sel_resolution : IN std_logic;
14           mclk : IN std_logic;
15           btn : IN std_logic_vector(2 downto 0);
16           sw : IN std_logic_vector(7 downto 0);
17           hsync : OUT std_logic;
18           vsync : OUT std_logic;
19           red : OUT std_logic_vector(2 downto 0);
20           green : OUT std_logic_vector(2 downto 0);
21           blue : OUT std_logic_vector(1 downto 0)
22      );
23      END COMPONENT;
24
25      -- Inputs
26      signal sel_module : std_logic := '0'; -- Default low setting for simple scenario
27      signal sel_resolution : std_logic := '0'; -- Default low setting for simple scenario
28      signal mclk : std_logic := '0';
29      signal btn : std_logic_vector(2 downto 0) := "000"; -- Ensure it is not '111'
30      signal sw : std_logic_vector(7 downto 0) := (others => '0');
```

```
31
32          -- Outputs
33          signal hsync : std_logic;
34          signal vsync : std_logic;
35          signal red : std_logic_vector(2 downto 0);
36          signal green : std_logic_vector(2 downto 0);
37          signal blue : std_logic_vector(1 downto 0);
38
39          -- Clock definition for 50 MHz
40          constant clk_period : time := 20 ns; -- 50 MHz clock frequency
41
42      BEGIN
43          -- Instantiate the Unit Under Test (UUT)
44          uut: AMVGA PORT MAP (
45              sel_module => sel_module,
46              sel_resolution => sel_resolution,
47              mclk => mclk,
48              btn => btn,
49              sw => sw,
50              hsync => hsync,
51              vsync => vsync,
52              red => red,
53              green => green,
54              blue => blue
55          );
56
57          -- Clock process
58          mclk process: process
59      begin
60          while TRUE loop
61              mclk <= '0';
62              wait for 10 ns;
63              mclk <= '1';
64              wait for 10 ns;
65          end loop;
66      end process;
67
68      -- Test Stimulus process
69      stim_proc: process
70      begin
71          -- Initially trigger the clear signal then deactivate
72          btn <= "010";  -- Briefly activate the clear to test the response, not setting to '111'
73          wait for 100 ns;
74          btn <= "000";  -- Ensure clear is not active during normal operation
75
76          -- Hold the initial settings
77          wait for 20 ms;  -- Observe the behavior for 20 milliseconds
78
79          -- Modify control signals as needed
80              sel_module <= '0';
81              sel_resolution <= '0';
82              wait for 500 ns;  -- Allow the system to stabilize
83          -- Simulation ends after the observation
84          wait;
85      end process;
86  END behavior;
```

The waveform results from the simulation indicate that many critical output signals, such as `hsync`, `vsync`, `red`, `green`, and `blue`, remain undefined (`U`) throughout the simulation period. This suggests potential issues with signal initialization, component integration, or the propagation of signals within the system. To address these challenges effectively, it would be prudent to adopt a systematic approach by validating each component individually before integrating them into the larger system. This method would allow for isolating specific issues

within each component, ensuring that each module functions correctly under defined test conditions. Validating components individually also simplifies troubleshooting by narrowing down the source of errors and ensuring compatibility among all modules. Once each component is verified to operate as expected, integrating them into the full system will likely result in a more stable and predictable overall system behavior, significantly reducing the complexity of debugging and enhancing the reliability of the simulation outcomes.

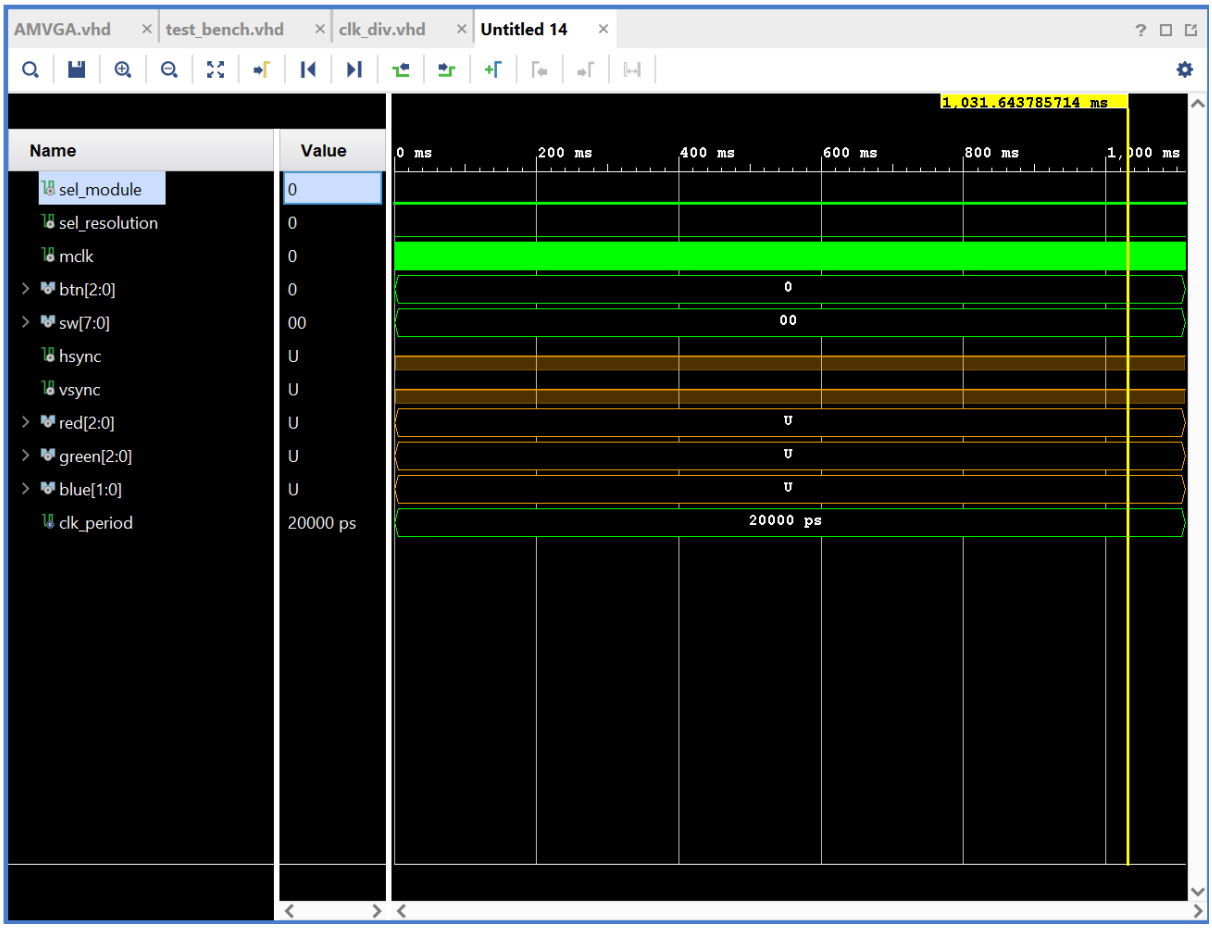


Figure 15 Simulation results.

III. Conclusions

The development of the Advanced Modular VGA Controller (AMVGA) project represents a significant educational endeavor in understanding and implementing complex digital systems using VHDL and FPGA technology. Throughout the project, I engaged in a comprehensive design and simulation process, utilizing the Vivado Design Suite to create and refine multiple VGA controller modules capable of supporting various display resolutions.

Key components of the system, including the VGA controllers for 640x480 and 1920x1080 resolutions, an output generator, and additional modules like `vga_stripes`, `prom_sprites`, and dynamic resolution selectors, were individually developed and tested. This modular approach not only streamlined the integration process but also enhanced the system's flexibility and scalability, allowing for easy updates and modifications.

The project faced challenges, particularly in the integration and simulation phases, where issues such as signal initialization, synchronization, and component compatibility were encountered. The use of a systematic testing approach, where each component was validated individually before integration, proved invaluable. This strategy significantly reduced complexity in debugging and ensured that each part functioned correctly within the system.

In conclusion, the AMVGA project successfully demonstrated the practical application of VHDL in FPGA design for VGA controllers. It highlighted the importance of a methodical approach to design and testing, particularly the effectiveness of breaking down the system into manageable modules and verifying each independently. This project not only fulfilled its educational objectives by providing deep insights into VGA controller technology but also equipped me with valuable skills in hardware design and simulation that are critical for future digital design projects. The experience gained and the knowledge developed from this project lay a solid foundation for further exploration and innovation in the field of digital systems design.