University of Balamand

Faculty of Engineering

ELCP392:

Senior Design II Report

Submitted by:

Wassim Moubayed(A1811062)

Ghady Ziadeh (A1810978)

Fadi Nabbout(A1810332)

Submitted to:

Dr. Rafic Ayoubi

Due Date

12/17/2021

Contents

1. Abstract

With the advancement of technology, displaying data on a screen has improved in the past decades, and the VGA display has been considered as a revolutionary change at the time of its creation. The use of VGA was important for scientist and displaying pixels as fast as possible for accurate, fast and reliable results. For a board like the FPGA using the VGA to display results, there are different methods to achieve good results. Two of the most important methods of displaying VGA are either by the HPS or using a bus master.

2. Introduction:

i. Topic:

Hardware and software interface for video graphics display using Intel FPGA with ARM processor using SDRAM (since the FPGA cannot handle big memory on the chip RAM) and VGA ports

ii. FPGA:

FPGA is a short for field programmable gate array, It is a hardware circuit that a user can program to do one or multiple logical operations, the board contains ICs (integrated circuits on the board). FPGA is an integrated circuit designed to be reprogrammable using VHDL language based on what task is requested by the user, it can be reprogrammed as many times as desired. Due to its programmable ability, it makes it ideal for many markets, such as: ASIC Prototyping, Automotive, High performance Computing and Data storage, industrial, medical, security, Video and Image Processing, Wired/Wireless communications.

History about FPGA:

- FPGA industry started from programmable read only memory.
- ALTERA delivered the industry's first reprogrammable logic device in 1984 the EP300
- Xilinx invented the first commercially viable FPGA in 1985 – the XC2064 which had 64 configurable logic blocks and 3 input lookup tables.
- Microsoft began using FPGAs to accelerate Bing in 2014, and in 2018 began deploying FPGAs across other data center workloads for their Azure cloud computing platform.
- In 2012 logic blocks and interconnects of traditional FPGA were combined with embedded microprocessors and peripherals to form a full "system on programmable chip"

iii. FPGA VS Microcontroller:

Criteria	FPGA	Microcontroller
Flexibility	Both hardware and firmware are reprogrammable	Only firmware is reprogrammable
Programming	Complex	Simple
Cost	Costly	Cost effective
Power consumption	High	Low
Processing power	Very High	High

In conclusion, FPGA is best when doing a design that requires complex logic and high processing power while the Microcontroller is used for simple designs.

iv. DE1-SoC:

What we will use in our project is the DE1-SoC development kit, which is an FPGA by ALTERA that contains dual-core Cortex A9 embedded cores, SDRAM(1GB DDR3 for HPS, 64 MB DDR3 for FPGA), USB ports, Ethernet, VGA ports for video capabilities and other features shown in the figure below. DE1-SoC board comes with the NIOS II processor and an external processor which is the ARM processor which is more complex and powerful than the NIOS II processor. For our project, we will start by implementing the ARM processor to achieve more complex, faster, and accurate results.

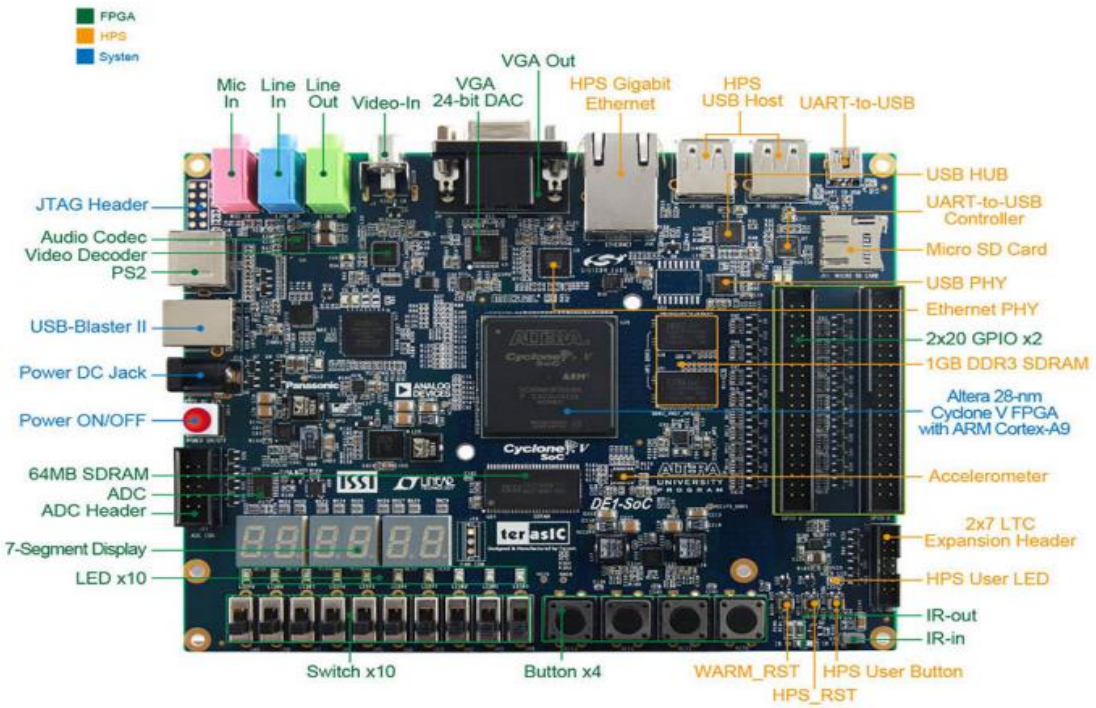


Figure 6-4. Front [1]

- Green for peripherals directly connected to the FPGA
- Orange for peripherals directly connected to the HPS
- Blue for board control

v. ARM Processor:

We are starting our project with the ARM processor, which is a reduced instruction set computer (RISC) architecture the base processor in the DE1-SoC board. The ARM based hard processor system (HPS) consist a processor, peripherals, and memory interface with the FPGA. The arm processor is powerful enough to handle very complex operations and at a good speed.

3. Components/Software Used

i. Software used:

- Quartus Prime Lite Edition Software (to write the code/pin assignment). We decided to go with the Quartus software since we are familiar with it and it supports our DE1-SoC board

- Quartus Platform Designer (to add components/wiring in the project) is one of the most advanced system integration tools for processor system design, according to Intel. You can configure a wide set of components using a graphical interface. After finishing the configuration part, the system translates the graphical interface of the connections set up to a Verilog/SystemVerilog code to be used later on in the project.

- Sublime Text is a software that we used as a source code editor to be able to write our C code in order to make a communication between the HPS and the peripherals of the FPGA

ii. SDRAM:

What is an SDRAM?

Current SDRAM, DDR, DDR2, DDR3, and so forth are intended for present day PC frameworks and require a memory controller. The memory controller will acknowledge memory demands from the CPU, dissect the solicitations, modify them, line them up, and dispatch them to the SDRAM in the most effective way. While fine for a cutting edge PC, memory controllers like that are extremely muddled for somebody who simply needs an essential regulator to permit utilizing a SDRAM with their easier FPGA projects.

SDRAM isn't costly at all they cost roughly 3.5$ for the 32Mib and most FPGA boards accompany a type of SDRAM on-board with them, so it bodes well to use this memory when the FPGA's Block-RAM limit isn't adequate. Quicker and simpler to utilize SRAM can likewise be utilized if your improvement board has it, however I would say most FPGA boards don't accompany SRAM, presumably because of the greater expense.

Why use the SDRAM instead of the internal memory?

Albeit present day FPGAs contain inside of them internal memories, the measure of memory accessible is consistently significantly degrees beneath what is conceivable with devoted memory chips. So it isn't astonishing that numerous FPGA originators append some sort of memory to their FPGA. Specifically, SDRAMs are extremely famous recollections because of their fast and minimal expense. Tragically, they are not as simple to control as static memories, so a SDRAM controller is regularly utilized.

Connections between the FPGA and SDRAM

Pin Assignment of SDRAM

Signal Name	FPGA Pin No.	Description	I/O Standard
DRAM_ADDR[0]	PIN_AK14	SDRAM Address[0]	3.3V
DRAM_ADDR[1]	PIN_AH14	SDRAM Address[1]	3.3V
DRAM_ADDR[2]	PIN_AG15	SDRAM Address[2]	3.3V
DRAM_ADDR[3]	PIN_AE14	SDRAM Address[3]	3.3V
DRAM_ADDR[4]	PIN_AB15	SDRAM Address[4]	3.3V
DRAM_ADDR[5]	PIN_AC14	SDRAM Address[5]	3.3V
DRAM_ADDR[6]	PIN_AD14	SDRAM Address[6]	3.3V
DRAM_ADDR[7]	PIN_AF15	SDRAM Address[7]	3.3V
DRAM_ADDR[8]	PIN_AH15	SDRAM Address[8]	3.3V
DRAM_ADDR[9]	PIN_AG13	SDRAM Address[9]	3.3V
DRAM_ADDR[10]	PIN_AG12	SDRAM Address[10]	3.3V
DRAM_ADDR[11]	PIN_AH13	SDRAM Address[11]	3.3V
DRAM_ADDR[12]	PIN_AJ14	SDRAM Address[12]	3.3V
DRAM_DQ[0]	PIN_AK6	SDRAM Data[0]	3.3V
DRAM_DQ[1]	PIN_AJ7	SDRAM Data[1]	3.3V
DRAM_DQ[2]	PIN_AK7	SDRAM Data[2]	3.3V
DRAM_DQ[3]	PIN_AK8	SDRAM Data[3]	3.3V
DRAM_DQ[4]	PIN_AK9	SDRAM Data[4]	3.3V
DRAM_DQ[5]	PIN_AG10	SDRAM Data[5]	3.3V

DRAM_DQ[6]	PIN_AK11	SDRAM Data[6]	3.3V
DRAM_DQ[7]	PIN_AJ11	SDRAM Data[7]	3.3V
DRAM_DQ[8]	PIN_AH10	SDRAM Data[8]	3.3V
DRAM_DQ[9]	PIN_AJ10	SDRAM Data[9]	3.3V
DRAM_DQ[10]	PIN_AJ9	SDRAM Data[10]	3.3V
DRAM_DQ[11]	PIN_AH9	SDRAM Data[11]	3.3V
DRAM_DQ[12]	PIN_AH8	SDRAM Data[12]	3.3V
DRAM_DQ[13]	PIN_AH7	SDRAM Data[13]	3.3V
DRAM_DQ[14]	PIN_AJ6	SDRAM Data[14]	3.3V
DRAM_DQ[15]	PIN_AJ5	SDRAM Data[15]	3.3V
DRAM_BA[0]	PIN_AF13	SDRAM Bank Address[0]	3.3V
DRAM_BA[1]	PIN_AJ12	SDRAM Bank Address[1]	3.3V
DRAM_LDQM	PIN_AB13	SDRAM byte Data Mask[0]	3.3V
DRAM_UDQM	PIN_AK12	SDRAM byte Data Mask[1]	3.3V
DRAM_RAS_N	PIN_AE13	SDRAM Row Address Strobe	3.3V
DRAM_CAS_N	PIN_AF11	SDRAM Column Address Strobe	3.3V
DRAM_CKE	PIN_AK13	SDRAM Clock Enable	3.3V
DRAM_CLK	PIN_AH12	SDRAM Clock	3.3V
DRAM_WE_N	PIN_AA13	SDRAM Write Enable	3.3V
DRAM_CS_N	PIN_AG11	SDRAM Chip Select	3.3V

SDRAM Controllers

Albeit present day FPGAs contain inside of them internal memories, the measure of memory accessible is consistently significantly degrees beneath what is conceivable with devoted memory chips. So it isn't astonishing that numerous FPGA originators append some sort of memory to their FPGA. Specifically, SDRAMs are extremely famous recollections because of their fast and minimal expense. Tragically, they are not as simple to control as static memories, so a SDRAM controller is regularly utilized.

The Controller's responsibility is to manage every one of the terrible pieces of SDRAM and to break out a basic interface. This interface by and large comprises of a address input, an data input, an data output, and some control signs to indicate a read/write signal, to tell when data is prepared, and if the RAM is occupied. That is it. No banks, no columns, no precharge, no opening, no issue

Dynamic memories are more confounded to drive than static ones. We have rows and columns and banks and refresh cycles to deal with. In any case, SDRAMs are convincing on account of their fast and minimal expense per bit.

So what we need is an approach to get to a SDRAM, however without hardly lifting a finger of utilization of a static memory. That is the reason why memory controllers are made. They go about as translation layers: on one side, they give the client a simple to utilize memory interface, and afterward accomplish the dirty work to drive the real SDRAM signals.

Basic SDRAM architecture:

The SDRAM chip architecture is organized with the memory cells organized into a two dimensional array of rows and columns.

To address a particular memory cell within the overall SDRAM, it is necessary first to address the required row, and then the specific column. This selects the column within the row. This isolates the data storage elements to be read from or written to.

An SDRAM row is called a page. Once the row is open it is possible to address multiple columns addresses on the row. Using this technique improves the memory access speed, reducing latency because the row address does not have to be re-sent and set-up. Each time the row is opened it naturally takes time.

As a result, the row address is taken as the higher order address bit elements and the column as the lower ones.

The row and column elements are sent separately for a variety of reasons including the successive addressing of column elements once a row is open. As a result, the row and column addresses are multiplexed onto the same lines - this significantly reduces the package pin count, and this has a major impact on the overall chip cost as one major element of the chip cost is its package.

It should be noted, though, that the row address size is normally larger than the column address because the power of the chip is not related to the number of columns, but the number of rows does impact this figure.

SDRAM chip architecture:

The circuit architecture of the SDRAM chip is one part of the SDRAM design. There are likewise the chip architecture aspects.

The actual chip SDRAM chip architecture will change as per the producer, and it will likewise depend somewhat on the size of the SDRAM.

The SDRAM architecture can be parted into two principle regions:

- **Array:**

This component of the SDRAM design is the space of the chip where the memory cells are carried out. It is ordinarily partitioned into various banks, which thusly is parted into more modest areas which are named segments.

- **Periphery:**

This is the space of the chip where control and addressing to hardware is situated, and other things such as line drivers and sense amplifiers. The chip periphery regularly isolates array banks and segments from one another.

Taking a look at the relative areas occupied by the array and the periphery it is feasible to decide a figure of legitimacy for the extent of the general territory involved by the real memory. This is regularly named the array or cell efficiency in light of the fact that the point of the chip is to give memory - the periphery, albeit significant doesn't expand the size of memory.

The array or cell efficiency for the chip is normally expressed as a percentage:

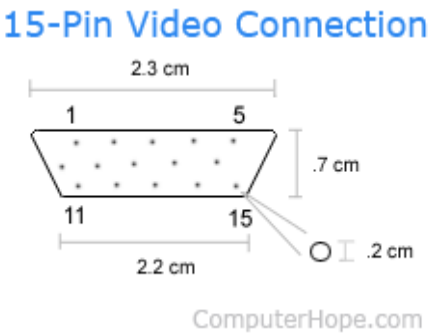
$$\text{Array / cell efficiency (\%)} = (\text{Array area/Overall})*100$$

As the periphery doesn't add to the real measure of memory on board, organizations attempt to build the array proficiency. Figures are normally in the locale of 60 - 70%.

iii. VGA:

The VGA Adapter is utilized to draw pictures on your PC screen. A picture comprises of a rectangular array of picture components, called pixels. Every pixel shows up as a speck on the screen, and the whole screen comprises of 320 columns by 240 rows of pixels. Pixels are orchestrated in a rectangular grid, with the coordinate (x,y) = (0, 0) at the upper left corner of the screen, and the coordinate (x,y) = (319, 239) at the base right corner of the screen.

VGA utilizes analog signals, which implies it is only capable of lower quality and a lower resolution display on screens.



VGA pins:

Pin	Function
1	Red Video
2	Green Video
3	Blue Video
4	Monitor ID 2
5	TTL Ground (monitor self-test)
6	Red Analog Ground
7	Green Analog Ground
8	Blue Analog Ground
9	Key (Plugged Hole)
10	Sync Ground
11	Monitor ID 0
12	Monitor ID 1
13	Horizontal Sync
14	Vertical Sync
15	Monitor ID 3

The VGA Adapter interfaces the Nios II processor to the DE1-SoC Video DAC chip which outputs to your screen. The interface to the adapter is indistinguishable from that of a memory: the address corresponds to the pixel you want to read/write, and the data you read/write from/to that address is the color for that pixel.

Device	VGA Adapter
Configuration	320x240 pixel resolution, 80x60 character resolution, 16-bit color
Input/Output	Input and Output
Address Base	Pixel Buffer: 0x08000000, Character Buffer: 0x09000000
Address Map	pixel base+{y[7:0],x[8:0],1'b0} corresponds to pixel (x,y) character base+{y[5:0],x[6:0]} corresponds to character (x,y)
Initialization	None needed
Interrupts	None
Hardware Setup	Connect the VGA plug of your monitor to the VGA port of the DE1-SoC

Pixel Color - The color of a pixel is a mixture of the three primary colors: blue, red and green. By altering the presence of each primary color, other colors can be made. A 16-bit halfword is used to represent the presence of each primary color of a pixel. The five most-significant bits [15:11] represent the intensity of the red color, the following six bits [10:5] represent the intensity of the green color, finally the four least significant bits [4:0] represent the intensity of the color blue, as shown in the table below.

Red 15...11	Green 10...5	Blue 4...0

Example: 15-11 10-5 4-0

Red color: 0xF800 0b1111100000000000

Purple color: 0xF81F 0b1111100000011111

White color: 0xFFFF 0b1111111111111111

Pixel Address - The color of each pixel in an image is stored at an address in the memory. The address of a pixel is the addition of the offset and the base address. In the DE1-SoC, the base address is 0x08000000. The (x, y) offset is determined by concatenating the 9-bit x coordinate starting at the 1st bit [9:1] and the 8-bit y coordinate starting at the 10th bit [17:10], as shown below.

Pixel Address				
Bits	31:18	17:10	9:1	0
Function	00001000000000	y[7:0]	x[8:0]	0

To find each pixel in memory, we add the base address to the (x, y) offset. A formula offset = (2 * x) + (1024 * y), is utilized to get the offset.

Example:

Offset (0, 0) => (base + 0x00000000) = 0x08000000.

Offset (1, 0) => (base + 0x00000002) = 0x08000002.

Offset (0, 1) => (base + 0x00000400) = 0x08000400.

Offset (319, 239) => (base + 0x0003BE7E) = 0x0803BE7E.

Drawing Shapes - The pixel buffer can be used to draw only singular pixels, thus in order to draw lines or shapes, it is needed to draw each pixel of the line or shape desired.

Characters - Besides drawing each pixel individually, there is also a character buffer that permits the placement of texts on the screen. It is also possible to write characters alongside with the pixels and shapes drawn using the pixel buffer. Similar to the pixel buffer, the entire screen is represented as a rectangular grid of 80 columns by 60 rows of characters. The character coordinate (x, y) = (0, 0) represents a character at the upper-left corner of the screen, and the coordinate (x, y) = (79, 59) represents a character at the lower-right corner of the screen. Characters are represented by their specific ASCII codes. Each character occupies one byte of memory.

Character Address - Each character occupies one byte of memory. The address of a character is the addition of the (x, y) offset and the base Character address. In the DE1-SoC, the base char address is 0x09000000. The (x, y) offset is determined by concatenating the 7-bit x [6:0] coordinate and the 6-bit y [12:7] coordinate, as shown below.

Character Address

Bits	31:13	12:7	6:0
Function	00001001000000000000	y[5:0]	x[6:0]

To determine the location of each character in memory, we add the base character address to the (x, y) offset. A formula offset = x + 128*y, is utilized to get the offset. Example:

Offset (0, 0) => (base + 0x00000000) = 0x09000000.

Offset (1, 0) => (base + 0x00000001) = 0x09000001.

Offset (0, 1) => (base + 0x00000080) = 0x09000080.

Offset (79, 59) => (base + 0x00001DCF) = 0x09001DCF.

Customization for our project

In our project, we changed the resolution from 320x240 to 640x480.The span of the addresses in the memory map was doubled, the addressing and colors of pixels had to be modified in the main program. No changes needed for the character buffer. The color encoding is now 8-bit with top 3 bits red, next 3 green, lower 2 bits blue.

```
#define BOARD                 "DE1-SoC"

/* Memory */
#define DDR_BASE              0x00000000
#define DDR_END               0x3FFFFFFF
#define A9_ONCHIP_BASE        0xFFFF0000
#define A9_ONCHIP_END         0xFFFFFFFF
#define SDRAM_BASE            0xC0000000
#define SDRAM_END             0xC3FFFFFF
//
#define FPGA_ONCHIP_BASE      0xC8000000
#define FPGA_ONCHIP_END       0xC803FFFF
// modified for 640x480
// #define FPGA_ONCHIP_SPAN      0x00040000
#define FPGA_ONCHIP_SPAN      0x00080000
//
#define FPGA_CHAR_BASE        0xC9000000
#define FPGA_CHAR_END         0xC9001FFF
#define FPGA_CHAR_SPAN        0x00002000

/* Cyclone V FPGA devices */
#define HW_REGS_BASE          0xff200000
//#define HW_REGS_SPAN        0x00200000
#define HW_REGS_SPAN          0x00005000
// === now offsets from the BASE ===
#define LEDR_BASE             0x00000000
#define HEX3_HEX0_BASE        0x00000020
#define HEX5_HEX4_BASE        0x00000030
#define SW_BASE               0x00000040
#define KEY_BASE              0x00000050
#define JP1_BASE              0x00000060
#define JP2_BASE              0x00000070
#define PS2_BASE              0x00000100
#define PS2_DUAL_BASE         0x00000108
#define JTAG_UART_BASE        0x00001000
#define JTAG_UART_2_BASE      0x00001008
#define IrDA_BASE             0x00001020
#define TIMER_BASE            0x00002000
#define TIMER_2_BASE          0x00002020
#define AV_CONFIG_BASE        0x00003000
#define PIXEL_BUF_CTRL_BASE   0x00003020
#define CHAR_BUF_CTRL_BASE    0x00003030
#define AUDIO_BASE            0x00003040
#define VIDEO_IN_BASE         0x00003060
#define ADC_BASE              0x00004000
```

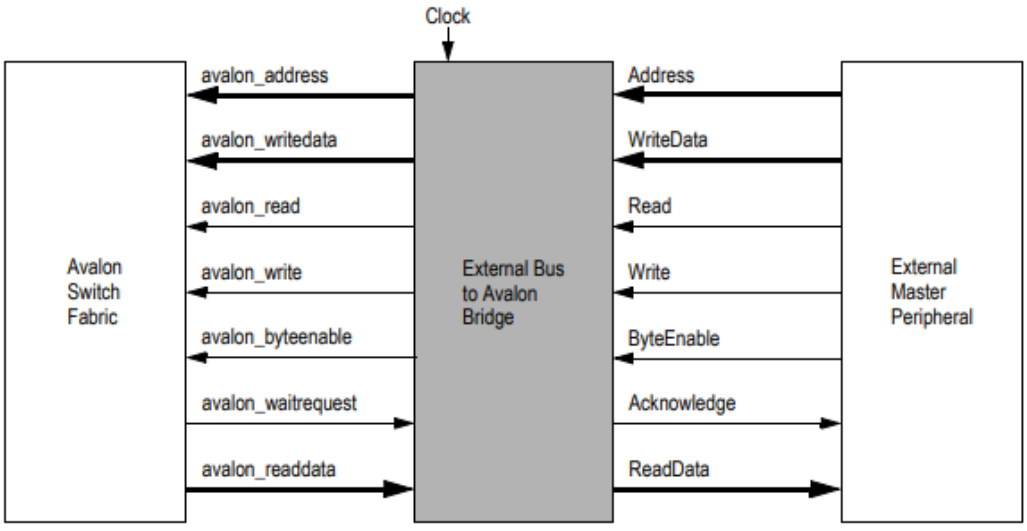
4. Implementation/Schematics:

i. Connecting the VGA using Bus master:

The bus master is a piece of IP that we can drag onto the Qsys bus but unlike the Pio port that is a bus slave the bus master allows you to manipulate the bus slave within the Verilog that we have on the avilon bus. Then when we export conduit into our Verilog it will export all of the signals shown on the right side of the diagram, so when we export it in the Qsys we will find seven new wires that we can tie to the Verilog to manipulate the bus master.

The bus master is going to be writing two specific addresses on the bus, one to the particular addresses associated with slaves on that bus. The bus master has as well the actual data that we are writing, two one-bit wide signal to either write or read, bit enable that does masking.

The only input to the bus master from the Verilog is an acknowledgment signal, that either tell that the completion of a read or that a write was successful.



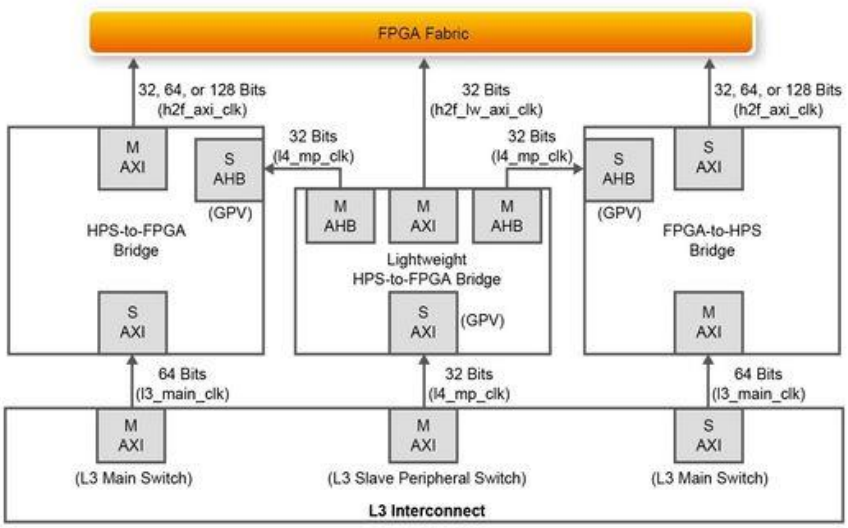
ii. Connecting the VGA using the HPS:

HPS (Hard Processor System) consists of processor, peripherals, and memory interface with the FPGA via high bandwidth interconnection. In order to get the HPS running, we need to install an operating system, Linux in our case, on the SD-card. After installing Linux, we should install GCC in order to compile the C code that we wrote using VI editor. One thing to note about Linux, is that it doesn't allow direct memory access, hence we need to map memories via mmap Linux utility using the address base provided for each component that we setup in the Qsys software, this requires coordination between Qsys and C program. The HPS contains the following HPS-FPGA AXI bridges:

1) FPGA-to-HPS Bridge

2) HPS-to-FPGA Bridge

3) Lightweight HPS-to-FPGA Bridge.



AXI Bridge Block Diagram

iii. QSYS:

The figure below shows the Qsys graphical implementation and component connections of the HPS connection we used:

-Sysrem_Pll: Generates an output clock by synchronizing itself to an input clock.

-ARM_A9_HPS: Cyclone 5 Hard Processor System

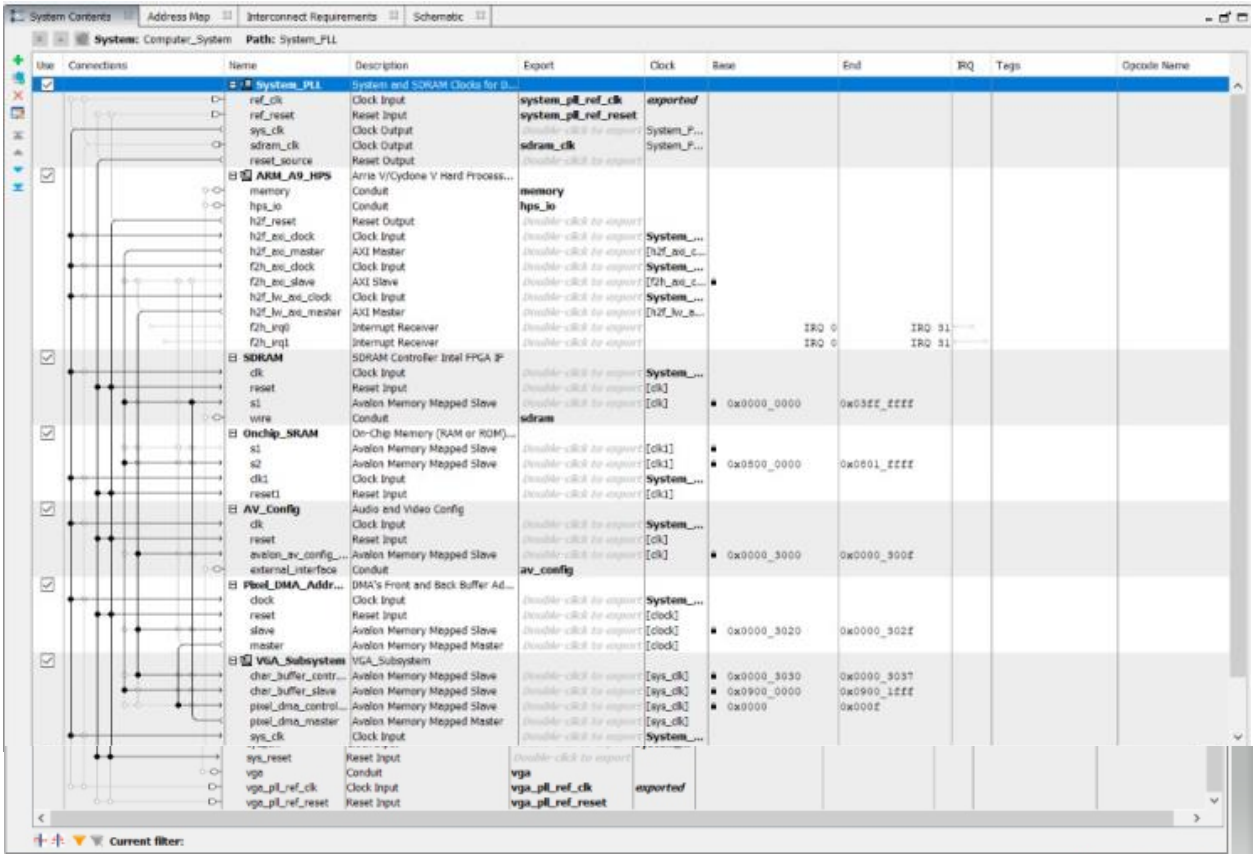
-SDRAM: SDRAM Controller

-OnChip_SRAM: On Chip memory

-AV_Config: Audio and video configuration

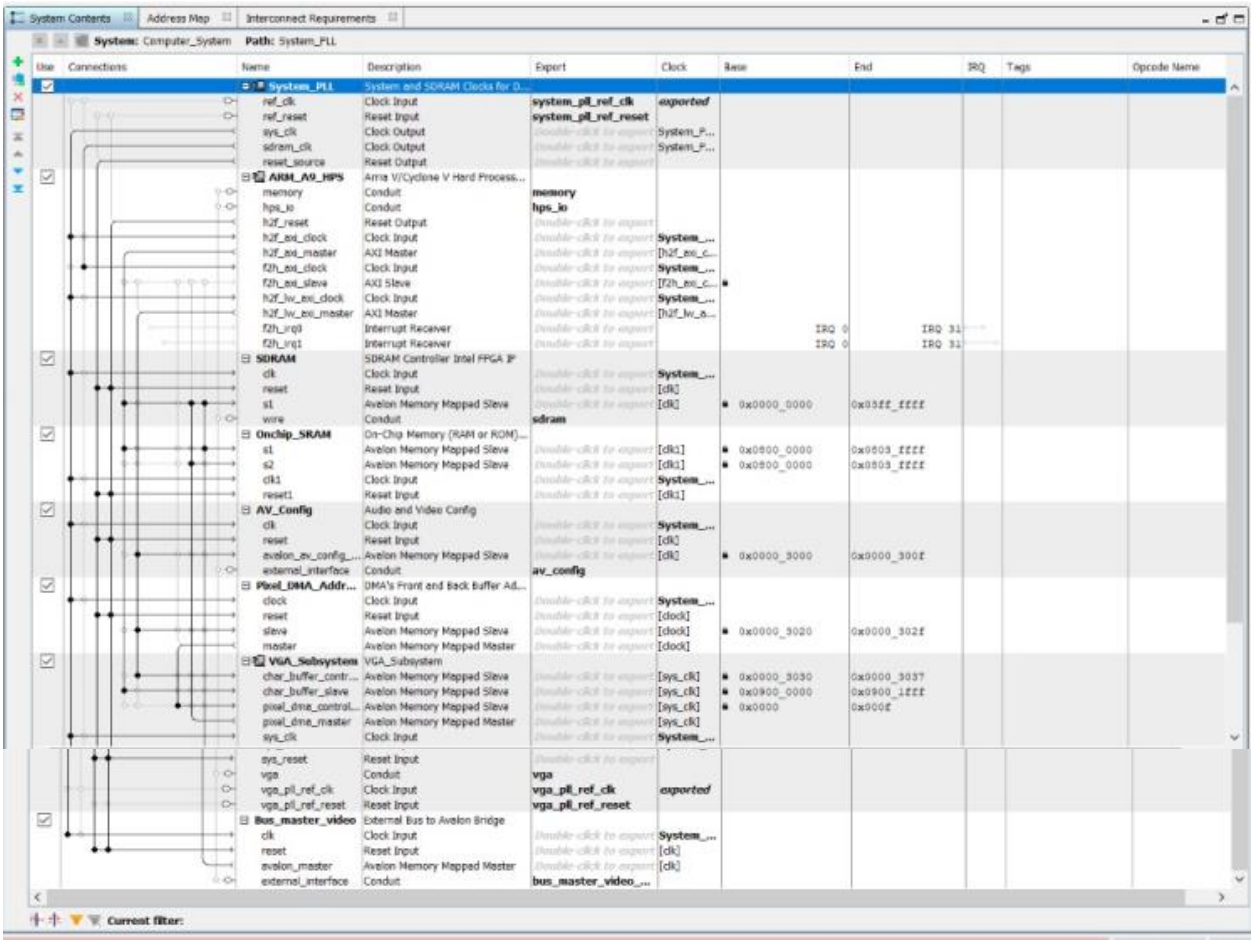
-Pixel_DMA_Address: Buffer address

-VGA_Subsytem



The figure below shows the Qsys graphical implementation and component connections of the Bus master connection we used:

-Bus_master_video: External Bus to Avalon bridge



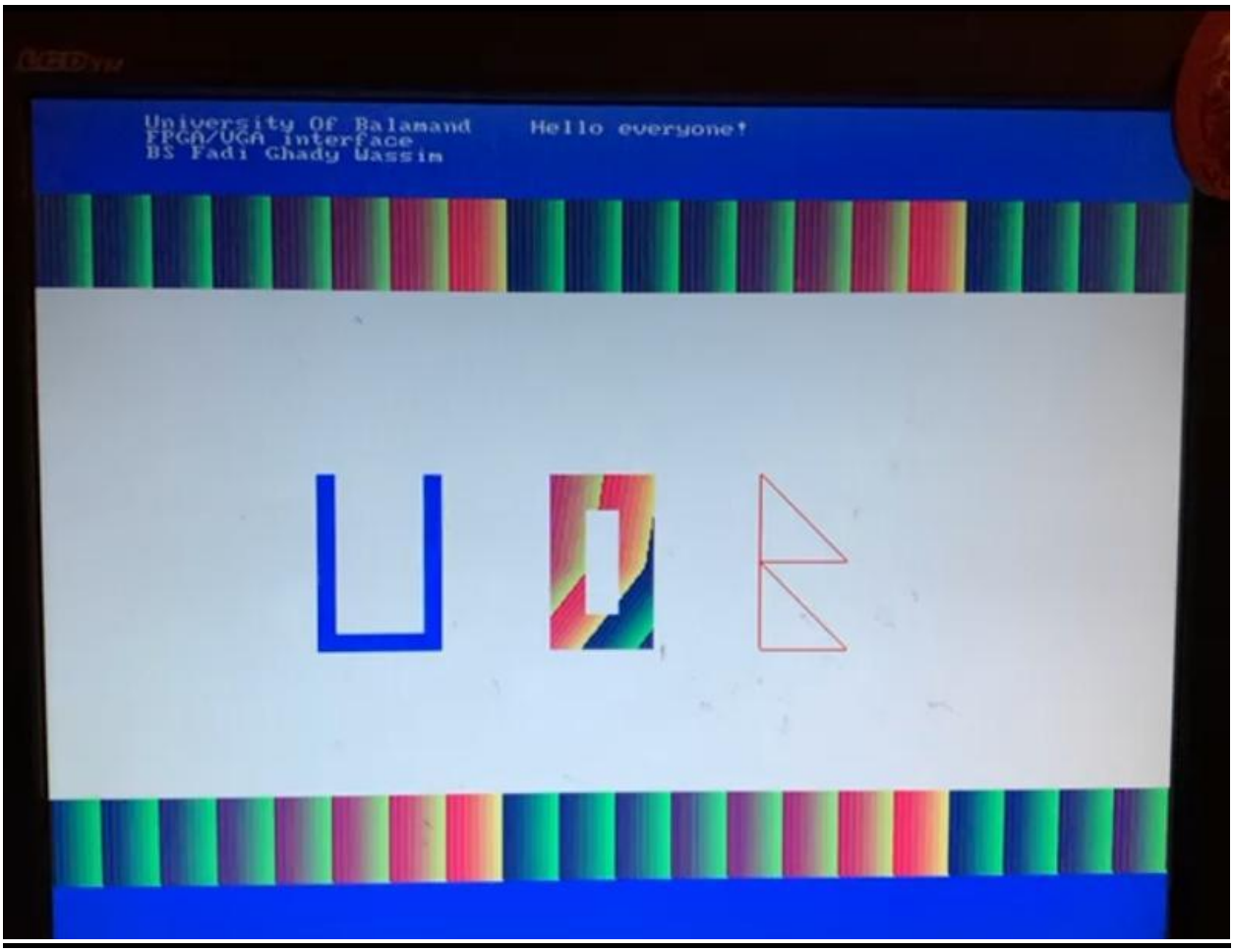
iv. The Verilog code for the Bus Master:

This is the code on the Verilog to be able to display pixels on the Bus Master

```
wire [31:0] bus_addr ; // Avalon address
wire [31:0] video_base_address = 32'h800_0000 ;  // Avalon address
wire [3:0] bus_byte_enable ; // four bit byte read/write mask
reg bus_read  ;       // high when requesting data
reg bus_write ;      //  high when writing data
reg [31:0] bus_write_data ; //  data to send to Avalog bus
wire bus_ack  ;       //  Avalon bus raises this when done
wire [31:0] bus_read_data ; // data from Avalon bus
reg [30:0] timer ;
reg [3:0] state ;
wire state_clock ;

// pixel address is
// from C: pixel_ptr = 0h0800_0000 + (y_cood<<10) + x_cood ;
reg [9:0] x_cood, y_cood ;
assign bus_addr = video_base_address + {22'b0,x_cood} + ({22'b0,y_cood}<<10) ;
// use byte-wide bus-master
assign bus_byte_enable = 4'b0001;

always @(posedge CLOCK2_50) begin //CLOCK_50

        // reset state machine and read/write controls
        if (~KEY[0]) begin
                state <= 0 ;
                bus_read <= 0 ; // set to one if a read opeation from bus
                bus_write <= 0 ; // set to on if a write operation to bus
                // base address of upper-left corner of the screen
                x_cood <= 0 ;
                y_cood <= 0 ;
                timer <= 0;
        end
        else begin
                timer <= timer + 1;
        end

        // write to the bus-master
        // but wait for when VGA is not reading
        if (state==0 && ~VGA_BLANK_N) begin // && timer==0 // && ((~VGA_VS | ~VGA_HS) || y_cood<256)
                state <= 2;

                // write all the pixels
                x_cood <= x_cood + 10'd1 ;
                if (x_cood > 10'd639) begin
                        x_cood <= 0 ;
                        y_cood <= y_cood + 10'd1 ;
                        if (y_cood > 10'd479) begin
                                y_cood <= 0 ;
                        end
                end
```

```
		// Make some patterns
		// set up the write data = white_red_green_blue
		// white = ff; red = e0; green = 1c; blue = 03;
		// AND signal the write request to the Avalon bus
		if (y_cood<50) begin
			bus_write_data <= 8'h03 ;
		end
		else if (y_cood<100) begin
			bus_write_data <= x_cood[7:0] + timer[29:22]  ;
		end



		else if (y_cood>200 && y_cood<300 && x_cood>280 && x_cood<340) begin
			if (y_cood>220 && y_cood<280 && x_cood>300 && x_cood<320)begin
				bus_write_data <= 8'hff ;
		   end
		else    bus_write_data <=  x_cood[7:0] + timer[20:13] ;
		end






		else if (y_cood>430) begin
			bus_write_data <= 8'h03 ;
		end
		else if (y_cood>380 && y_cood<431) begin
			bus_write_data <= x_cood[7:0] - timer[29:22]  ;
		end
		else if (y_cood>200 && y_cood<300 && x_cood>150 && x_cood<220) begin
			if (y_cood>190 && y_cood<290 && x_cood>160 && x_cood<210) begin
			bus_write_data <=8'hff ;
		end
			else    bus_write_data <= 8'h03 ;
		end
		else if (y_cood>200 && y_cood<300 && x_cood>400 && x_cood<403) begin
			bus_write_data <= 8'he0 ;
		end
		else if (x_cood>400 && x_cood<450 && y_cood>200 && y_cood<250 && x_cood==(y_cood +10'd200)) begin
			bus_write_data <= 8'he0 ;
		end
		else if (x_cood>400 && x_cood<450 && y_cood>250 && y_cood<253) begin
			bus_write_data <= 8'he0 ;
		end
		else if (x_cood>400 && x_cood<450 && y_cood>250 && y_cood<300 && x_cood==(y_cood+10'd150)) begin
			bus_write_data <= 8'he0 ;
		end
		else if (x_cood>400 && x_cood<450 && y_cood>300 && y_cood<303) begin
			bus_write_data <= 8'he0 ;
		end
		else bus_write_data <= 8'hff;
		// signal the bus that a write is requested
		bus_write <= 1'b1 ;
	end

	// detect bus-transaction-complete ACK
	// You MUST do this check
	if (state==2 && bus_ack==1) begin
		state <= 0 ;
		bus_write <= 0;
	end

end // always @(posedge state_clock)
```

The Result of the Bus Master code:



v. The C code for the HPS

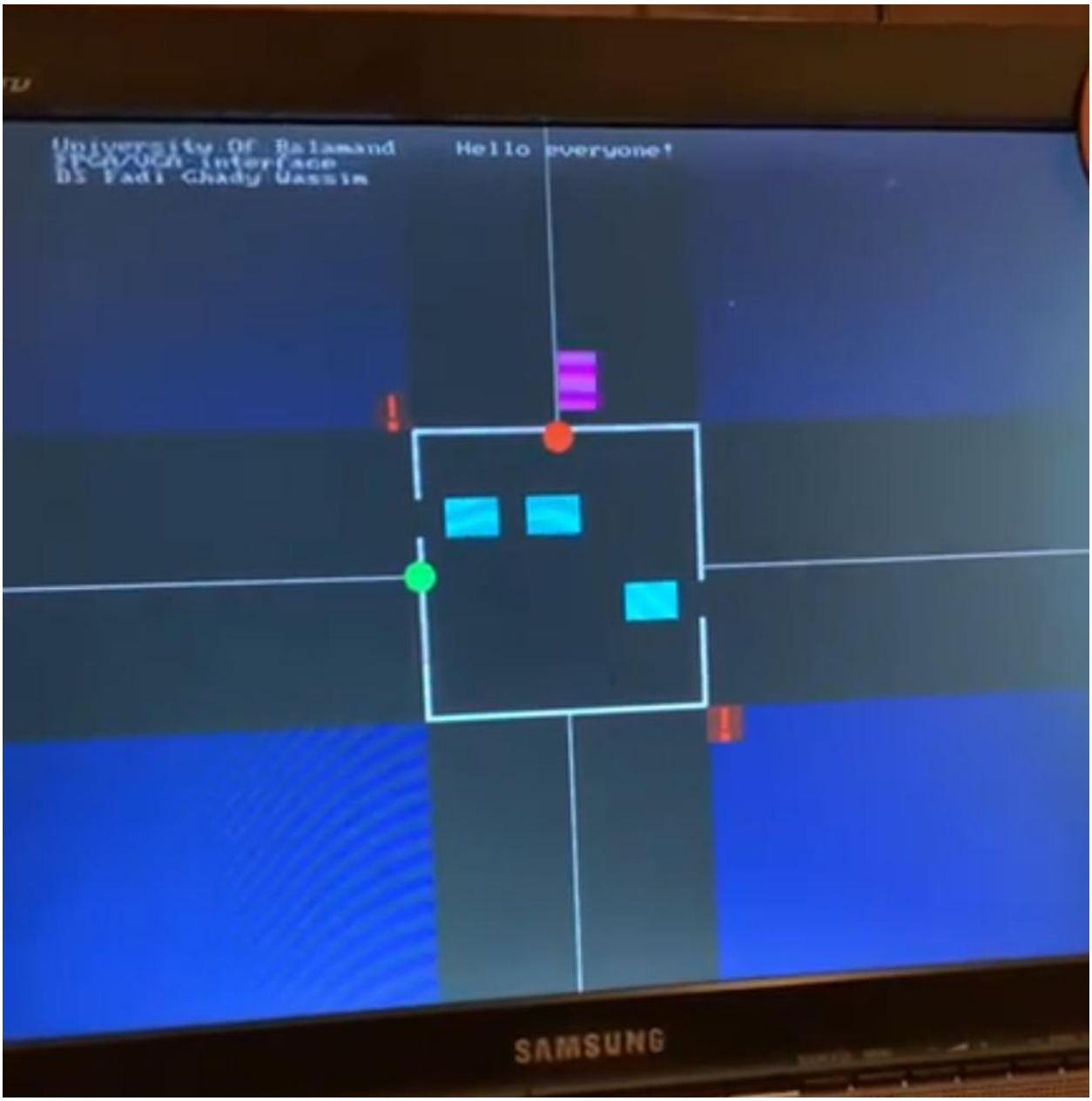
This is part of the main code where we used some pre-defined function to be able to display the pixels

```
/* create a message to be displayed on the VGA
          and LCD displays */
         char text_top_row[40] = "University Of Balamand\0";
         char text_bottom_row[40] = "FPGA/VGA interface\0";
         char text_next[40] = "BS Fadi Ghady Wassim\0";
         char num_string[20], time_string[20] ;
         char color_index = 0 ;
         int color_counter = 0 ;
         char text_fun[40] = "Hello everyone!\0";
         char text_warning_sign[40] = " ! \0";
         // position of disk primitive
         int disc_x = 0;
         // position of circle primitive
         int circle_x = 0 ;
         // position of box primitive
         int box_x = 5 ;
         int box_y = 35 ;
         int box_x1 = 50;
         int box_y1 = 80;
         int box_x2 = 635;
         int box_y2 = 605;
         int box_z = 5;
         int box_z1 = 35;
         int box_z2 =150;
         int box_z3 =180;

         // position of vertical line primitive
         int Vline_x = 350;
         // position of horizontal line primitive
         int Hline_y = 250;

         //VGA_text (32, 1, text_top_row);
         //VGA_text (32, 2, text_bottom_row);
         // clear the screen
         VGA_box (0, 0, 639, 479, 0x0000);
         // clear the text
         VGA_text_clear();
         // write text
         VGA_text (8, 1, text_top_row);
         VGA_text (8, 2, text_bottom_row);
         VGA_text (8, 3, text_next);
         VGA_text (34,1, text_fun);


         // R bits 11-15 mask 0xf800
         // G bits 5-10  mask 0x07e0
```

```
        // B bits 0-4   mask 0x001f
        // so color = B+(G<<5)+(R<<11);

        //VGA_box(int x1, int y1, int x2, int y2, short pixel_color)
                VGA_box(64, 0, 240, 50, blue); // blue box
                VGA_box(250, 0, 425, 50, red); // red box
                VGA_box(435, 0, 600, 50, green); // green box

                //draw background and street
                VGA_box (0, 0, 639, 479, dark_blue);
                VGA_box (0, 160, 639, 320, black);
                VGA_box (240, 0, 400, 479 , black);

                //draw street lines
                VGA_Hline(0, 240, 639, white);
                VGA_Vline(320, 0, 479, white);


                //draw the center box
                VGA_box(243, 163, 397, 317, black);
                VGA_rect(240, 160, 400, 320, white);
                VGA_rect(241, 161, 399, 319, white);
                VGA_rect(242, 162, 398, 318, white);




                //draw the traffic lights
                VGA_disc(240, 240, 8, green);
                VGA_disc(320, 165, 8, red);


                //draw a warning sign
                VGA_box(220, 140, 240, 160, dark_red);
                VGA_box(400, 320, 420, 340, dark_red);
                VGA_Vline(230, 142, 153, red);
                VGA_Vline(410, 322, 333, red);
                VGA_Vline(231, 142, 153, red);
                VGA_Vline(411, 322, 333, red);
                VGA_Vline(229, 142, 153, red);
                VGA_Vline(409, 322, 333, red);
                VGA_box(228, 156, 232, 160, red);
                VGA_box(408, 336, 412, 340, red);
```

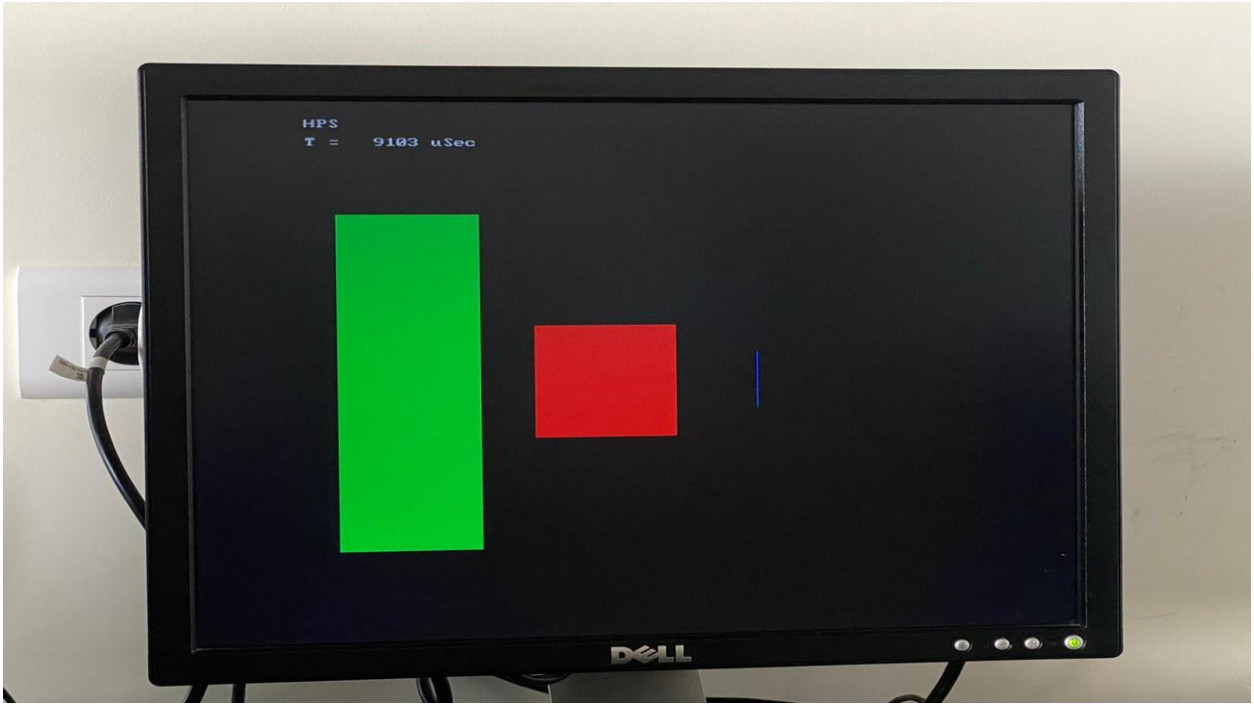
```
        while(1)
        {
                // start timer
                gettimeofday(&t1, NULL);
                // cycle thru the colors
                if (color_index++ == 11) color_index = 0;

                int i = 35;
                while(i<=635){
                //draw the moving car
                VGA_box(box_x, 218 ,box_y, 198, cyan);
                VGA_box(0, 218, box_x, 198, black);
                  VGA_box(box_x1, 218 ,box_y1, 198, cyan);
                VGA_box(box_y, 218, box_x1, 198, black);
                  VGA_box(box_x2, 272 ,box_y2, 252, cyan);
                VGA_box(640, 272, box_x2, 252, black);
        if(i<150 && i>35){
        VGA_box(370, box_z,390,box_z1,magenta);
        VGA_box(370,0,390,box_z,black);
        box_z+=10;
        box_z1+=10;
        }
                box_x2-=10;
                box_y2-=10;
                box_x1+=10;
                box_y1+=10;
                box_x+=10;
                box_y+=10;
                i+=10;
                usleep(200000);
        }
                  //draw the traffic lights
                VGA_disc(240, 240, 8, red);
                VGA_disc(320, 165, 8, green);

                // draw a car waiting at crossroad
          int j = 150;
                while(j<480){
        VGA_box(370, box_z2,390,box_z3,magenta);
        VGA_box(370,0,390,box_z2,black);
        box_z2+=10;
        box_z3+=10;
        j+=10;
        usleep(200000);
        if(j>450)
        break;


                 // stop timer
                 gettimeofday(&t2, NULL);
                 elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000000.0;      // sec to us
                 elapsedTime += (t2.tv_usec - t1.tv_usec) ;   // us
                 sprintf(time_string, "T = %6.0f uSec  ", elapsedTime);
                 VGA_text (10, 4, time_string);
                 // set frame rate
                 //usleep(17000);

         } // end while(1)
} // end main

/
```

The result of the HPS code:



vi. Timer difference:

Timer for the HPS:



The timer for the Bus Master

5 - Conclusion:

After working with the FPGA for a couple of month we have arrived to two conclusions:

- The first is that the bus master displays results much faster than the HPS, which can be used in sectors that relies on speed and accuracy.
- The second conclusion is that the HPS can be more flexible in terms of display and it is much more suitable for user interface as it uses C programming instead of Verilog with the Bus master.

To conclude each of the two methods can be used for different data and different sectors of work. They are both accurate and easily modified.

Our goal for this project is to set a base for students to use our design in future labs or projects, as the FPGA combined with the ARM processor is powerful board.