**Technion**

*Electrical Engineering Department*

High Speed Digital System Lab

Symbol Generator

Project

- Part B -

**Version**: 1

**Last update**: 12.04.2013

**Written By**: Olga Liberman and Yoav Shvartz

**Supervised By**: Moshe Porian

Table of Changes

|  |  |  |
| --- | --- | --- |
| Version | Date of Change | Description |
| 1 | 12.04.2013 | Creation |

Table of content

[1 Introduction 8](#_Toc353554984)

[1.1 Abstract 8](#_Toc353554985)

[1.2 Project Goal 8](#_Toc353554986)

[1.3 Project Requirements 8](#_Toc353554987)

[1.4 Displayed Image 9](#_Toc353554988)

[1.5 Engineering Tools 9](#_Toc353554989)

[1.6 Applicable Documents 9](#_Toc353554990)

[2 General Description 11](#_Toc353554991)

[2.1 Detailed Project Scheme 11](#_Toc353554992)

[2.2 Data Flow 12](#_Toc353554993)

[3 HW Implementation and Integration 16](#_Toc353554994)

[3.1 Top Architecture 16](#_Toc353554995)

[3.2 Reused Blocks Architecture 17](#_Toc353554996)

[3.2.1 UART Protocol 17](#_Toc353554997)

[3.2.2 VESA Generator Controller 19](#_Toc353554998)

[3.2.3 Memory Management 21](#_Toc353554999)

[3.2.4 SDRAM Controller 22](#_Toc353555000)

[3.2.5 Global Nets 23](#_Toc353555001)

[3.3 Display Controller 24](#_Toc353555002)

[3.3.1 General Description 24](#_Toc353555003)

[3.3.2 SG Register 24](#_Toc353555004)

[3.3.3 Symbol Generator 25](#_Toc353555005)

[3.3.4 Reused Video Blocks 33](#_Toc353555006)

[3.3.5 SG WBM IF 35](#_Toc353555007)

[4 SW IF 37](#_Toc353555008)

[4.1 GUI Preview 37](#_Toc353555009)

[4.2 GUI Functions 38](#_Toc353555010)

[4.2.1 New Test 38](#_Toc353555011)

[4.2.2 Locating and Choosing the Symbols 39](#_Toc353555012)

[4.2.3 Clear Screen 40](#_Toc353555013)

[4.2.4 Display Preview 40](#_Toc353555014)

[4.2.5 Create Frame 41](#_Toc353555015)

[4.2.6 Registers Panel 41](#_Toc353555016)

[5 Testing 42](#_Toc353555017)

[5.1 Simulation 42](#_Toc353555018)

[5.1.1 Automatic Simulating 42](#_Toc353555019)

[5.1.2 Simulation-GUI 44](#_Toc353555020)

[5.1.3 TCL script 44](#_Toc353555021)

[5.1.4 Image Comparison 44](#_Toc353555022)

[5.1.5 Simulation Waveforms 45](#_Toc353555023)

[5.2 Lab Examination 48](#_Toc353555024)

[5.2.1 The System 48](#_Toc353555025)

[5.2.2 Life Indications 49](#_Toc353555026)

[5.2.3 Signal Tap 51](#_Toc353555027)

[5.2.4 Working System 53](#_Toc353555028)

[6 Implementation Tools 54](#_Toc353555029)

[6.1 Synplify 54](#_Toc353555030)

[6.2 Quartus 55](#_Toc353555031)

[6.2.1 Resources Usage 56](#_Toc353555032)

[7 Working Methods 57](#_Toc353555033)

[7.1 SVN 57](#_Toc353555034)

[8 Summary 59](#_Toc353555035)

[8.1 Project Usage 59](#_Toc353555036)

[8.2 Problems during the project 59](#_Toc353555037)

[8.3 Conclusions 60](#_Toc353555038)

[9 Abbreviations 60](#_Toc353555039)

[10 Appendix A: Register Map 61](#_Toc353555040)

[10.1 Registers Table 61](#_Toc353555041)

[10.2 Type Register Values 62](#_Toc353555042)

[11 Appendix B: Writing, Reading from and to SDRAM and Registers 63](#_Toc353555043)

[11.1 Register Access 63](#_Toc353555044)

[11.1.1 Limitations 63](#_Toc353555045)

[11.1.2 Write to Registers 63](#_Toc353555046)

[11.1.3 Read Registers 63](#_Toc353555047)

[11.2 SDRAM Access at Debug Mode 65](#_Toc353555048)

[11.2.1 Write to SDRAM in Debug Mode 65](#_Toc353555049)

[11.2.2 Read SDRAM in Debug Mode 65](#_Toc353555050)

Tables of Figures

[Figure 1 - Displayed Image Example 9](#_Toc353555277)

[Figure 2 - Top Platform 11](#_Toc353555278)

[Figure 3 - Detailed Scheme 12](#_Toc353555279)

[Figure 4 - RAM initialization 13](#_Toc353555280)

[Figure 5 - SDRAM initialization 13](#_Toc353555281)

[Figure 6 - Detailed stages – (1), (2), (3) 14](#_Toc353555282)

[Figure 7 - Detailed stage (4) 15](#_Toc353555283)

[Figure 8 - Detailed stages – (5), (6) 15](#_Toc353555284)

[Figure 9 - Top Architecture 16](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555285)

[Figure 10 - UART Message Pack Structure 18](#_Toc353555286)

[Figure 11 - UART transaction 19](#_Toc353555287)

[Figure 12 - VESA standart timings 20](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555288)

[Figure 13 - Video Frame 21](#_Toc353555289)

[Figure 14 - Memory Management block 22](#_Toc353555290)

[Figure 15 - Display Controller Top diagram 24](#_Toc353555291)

[Figure 16 - Symbol Generator Top diagram 26](#_Toc353555292)

[Figure 17 - Divided video frame 27](#_Toc353555293)

[Figure 18 - Symbol image 28](#_Toc353555294)

[Figure 19 - Symbols stored in SDRAM 28](#_Toc353555295)

[Figure 20 - Opcode structure 28](#_Toc353555296)

[Figure 21 - The symbol blocks in the frame 29](#_Toc353555297)

[Figure 22 - RAM inside Symbol Generator block 30](#_Toc353555298)

[Figure 23 - Data stored in FIFO 31](#_Toc353555299)

[Figure 24 - Data as represented on the display 31](#_Toc353555300)

[Figure 25 - Toggling between FIFOs 32](#_Toc353555301)

[Figure 26 - Clock Domain Crossing 34](#_Toc353555302)

[Figure 27 - 2.3.5 SG WBM IF FSM 35](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555303)

[Figure 28 - GUI preview 37](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555304)

[Figure 29 - New Test button 38](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555305)

[Figure 30 – RAM initialization message 38](#_Toc353555306)

[Figure 31 - Pool of Symbols 39](#_Toc353555307)

[Figure 32 - Locating and Choosing the Symbols 39](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555308)

[Figure 33 - Clear Screen button 40](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555309)

[Figure 34 - Example of a Preview 40](#_Toc353555310)

[Figure 35 - Create Frame button 41](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555311)

[Figure 36 - Register Panel 41](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555312)

[Figure 37 - Test Directory structure 43](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555313)

[Figure 38 - Simulation Panel 44](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555314)

[Figure 39 - Example for tests.do fie 44](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555315)

[Figure 40 - Initialization Process 45](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555316)

[Figure 41 - Arrival of New Opcodes 45](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555317)

[Figure 42 - Manager Block function 46](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555318)

[Figure 43 - SG reads from SDRAM 46](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555319)

[Figure 44 - VESA Output 46](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555320)

[Figure 45 - Output Image 47](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555321)

[Figure 46 - Full System image 48](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555322)

[Figure 47 - Blinking LED on DE2 Board 49](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555323)

[Figure 48 -Register's values on 7-Segment Dsiplay 50](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555324)

[Figure 49 - Signal Tap example 51](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555325)

[Figure 50 - Working System's image 53](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555326)

[Figure 51 - Synplify Project 54](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555327)

[Figure 52 - Synplify Clock constraints 54](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555328)

[Figure 53 - Quartus Project 55](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555329)

[Figure 54 - Synplify to Quartus 55](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555330)

[Figure 55 - Place&Route Report 56](file:///H:\Project\docs\Part_B\Symbol_Generator_Project_Document_Part_B.docx#_Toc353555331)

[Figure 56 - SVN snapshot 57](#_Toc353555332)

[Figure 57 - SVN project structure 58](#_Toc353555333)

[Figure 58 - optional project's usage 59](#_Toc353555334)

# 

# Introduction

## Abstract

Generating symbols on display screens is a common and essential function in everyday lives technology. The need of doing so efficiently by minimizing frame transmission time and resources becomes more and more crucial with the progress of technology.

## Project Goal

The main goal: Implement a software symbol generator (using Matlab) with a HW extractor (FPGA) using VHDL.

There is a set of known symbols (such as letters, digits, icons, etc), saved as 32X32 pixels, which are generated onto the screen through Graphical User Interface.

The user chooses the appearance of the frame on the screen: which symbols to place on the screen and where to place them, or which symbols to delete.

The main goal is to save time, resources and bandwidth. Therefore, the program transfers the HW only the changes of the next desired frame from the current one, by a set of UART commands, instead of sending whole frame.

## Project Requirements

1. Transmit 640x480 BMP grayscale picture, using Matlab, to the DE2 board.
2. The frame is divided into 20x15 (row x column) aligned blocks.
3. Each symbol size 32x32 pixels (1 pixel = 8 bits).
4. A black screen will be displayed, until the first frame will be generated and transmitted.
5. Data will be sent from host to the DE2 board using UART 115,200 bits/sec.
6. Internal communication via Wishbone protocol
7. SDRAM
   1. Reading / Writing in up to 256 words burst.
   2. Required size in SDRAM: (num of symbols) x 32 x 32 bytes
   3. SDRAM Arbiter priority: Writing, then reading, in order to prevent data loss.
8. Clocks in systems:
   1. 40 MHz VESA clock
   2. 100MHz System clock
   3. 133MHz SDRAM clock

## Displayed Image

Figure - Displayed Image Example

## Engineering Tools

The tools which were used in this project are:

1. Altera DE2 Development and Education board with Altera Cyclone II 2C35 FPGA
2. Snplify Pro for synthesis
3. Quartus II 12.1 for Place and Route.
4. Matlab (including Image Toolbox)
5. TCL scripts

## Applicable Documents

Following is the list of the applicable documents for additional reading:

1. "Symbol Generator" Project Document – Part A  
   <https://symbol-generator.googlecode.com/svn/docs/Project_Document/Symbol_Generator_Project_Document.docx>
2. "Modular Decompression System" Project Document  
   <https://runlen.googlecode.com/svn/tags/Version_1.0_June_2012/Doc/Project/Project%20DOC.docx>
3. Clocks and Reset Document  
   <https://runlen.googlecode.com/svn/tags/Version_1.0_June_2012/Doc/Clock_and_Reset/Clock_and_Reset.docx>
4. UART Document  
   <https://runlen.googlecode.com/svn/tags/Version_1.0_June_2012/Doc/UART/UART_doc.docx>
5. VESA Generator Document  
   <https://runlen.googlecode.com/svn/tags/Version_1.0_June_2012/Doc/VESA/VESA_Generator.docx>
6. Memory Management Document  
   <https://runlen.googlecode.com/svn/tags/Version_1.0_June_2012/Doc/Memory_Management/Memory_Management.docx>
7. SDRAM Controller  
   <https://runlen.googlecode.com/svn/tags/Version_1.0_June_2012/Doc/SDRAM/SDRAM_doc.docx>
8. Register Map Document  
   <https://runlen.googlecode.com/svn/tags/Version_1.0_June_2012/Doc/HSID/RunLen_register_map.docx>

## 

# General Description

SG project uses the runlen project as a platform environment. The following blocks were reused: Rx Path, Tx Path, Memory Management, Wishbone Intercon, Display Controller and SDRAM controller. See "Reused Blocks Architecture" chapter for more information.

Top platform:

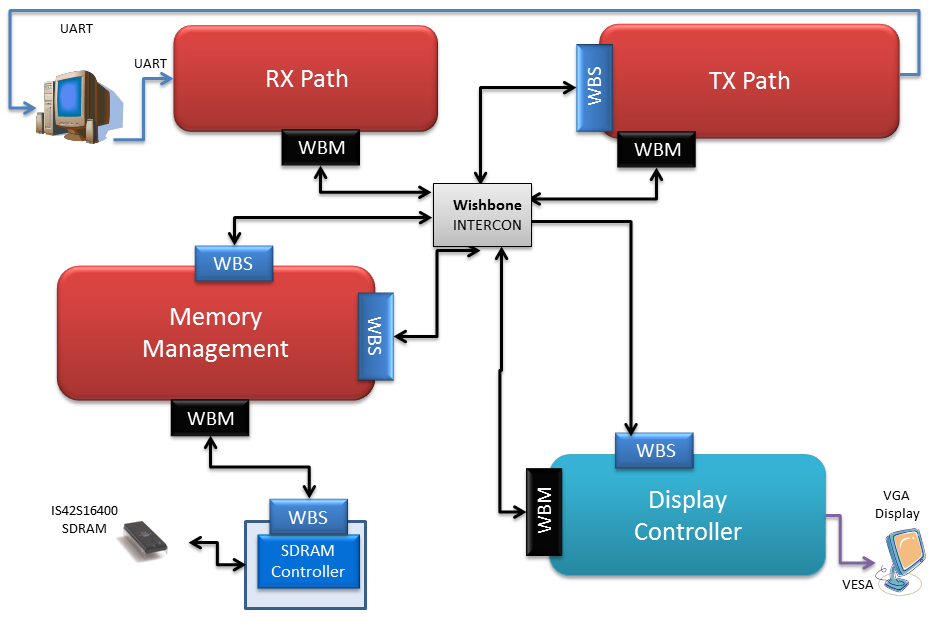


Figure - Top Platform

## Detailed Project Scheme

Project's top block scheme, including Symbol Generator in the Display Controller block and detailed Wishbone Intercons' interactions:

Detailed Scheme:

Figure - Detailed Scheme

## Data Flow

**Initialization:**

**Host** transmits wrapped initialization message to the **RX Path**. The message composed of 2 parts:

1. Initialization of the RAM in the SG TOP block.
2. Loading bank of symbols which will be used during the continuous use.

The Message is decoded (from the SOF, CRC and EOF) by **Message Pack Decoder** and validation **Checksum** and correctness of message length are being held. The First part of the message is transmitted to the **Display Controller** block and the second part to the SDRAM through **Memory Management** block.

An example of initialization message can be seen in uart\_txt\_1.txt file:

<http://symbol-generator.googlecode.com/svn/Matlab/GUI/SG_GUI_2/uart_tx_1.txt>

RAM initialization:

Figure - RAM initialization

SDRAM initialization:

Figure - SDRAM initialization

**Continuous use:**

**User** chooses the desired frame display, symbols & locations, using GUI platform (1). **Host** transmits wrapped message of data packets composed of the changes in the current display (relatively to the previous one) to the **RX Path**. Message is decoded by **Message Pack Decoder** and validation **Checksum** and correctness of message length are being held (2). Unwrapped message is transmitted to **SG TOP**, which decode the message (3).

The address from which line to read the symbol in **SDRAM** is updated in **DBG Addr Reg** in the **Memory Management** block using **SG WBM** (4). Through **Mem Ctrl Rd**, relevant symbols are read from **SDRAM** and sent **SG TOP** (5), which relay them to **VESA Ctrl** and to the display (6).

Detailed stages – (1), (2), (3):

Figure - Detailed stages – (1), (2), (3)

Detailed stage (4):

Figure - Detailed stage (4)

Detailed stages – (5), (6):

Figure - Detailed stages – (5), (6)

# HW Implementation and Integration

## Top Architecture

Following is the top architecture of the Symbol Generator project:

Figure - Top Architecture

The Symbol Generator project is based on the system that was developed in the "Modular Decompression System" project. This system uses standard interfaces:

* UART protocol to communicate with the host (through RS-232 connector)
* SDRAM IF
* VESA protocol to display video

These standard interfaces are implemented in separated building blocks, generics IPs, and described shortly in the following sub-chapters.

Some of these reused blocks were used in our project "as is". This case was for UART Receiver and Transmitter and SDRAM Controller blocks, which were used as a "black box", and only the understanding of their usage was needed.

Other blocks were reused with changes for our own need, and a deeper understanding was needed for them. This is the case of VESA Generator Controller and Memory Management blocks, which were adapted with changes.

The necessary details of understanding these blocks are found in the following chapter of "Reused Blocks Architecture", including the changes that were made in this project.

Part A of the Symbol Generator Project dealt with the original blocks that were built in order to solve the purpose of our project. Here, in part B of the project, we're dealing with integration of these blocks in the bigger system of the FPGA and with the standard interfaces.

In chapter "Display Controller" is found a short review of the algorithm and implementation that were done in part A of the project, and description of the Display Controller's Wishbone Master IF. The purpose of this special Wishbone Master IF is part of the integration with the Memory Management block, in order to execute SDRAM read transactions in debug mode.

## Reused Blocks Architecture

The Symbol Generator project reused important IPs from a previous "Modular Decompression System" project.

The following description of the reused block includes the original implementation details and also the changes that were made in our project.

### UART Protocol

#### UART Receiver and Transmitter

This component is used for asynchronous serial data channel.

The receiver (UART RX) converts serial start bit, data, parity and stop bit to parallel data.

The transmitter (UART TX) converts parallel data into serial form and automatically adds start bit, parity and stop bit.

Both of the components are based on the same characteristics:

1. Start bit
2. Data length: 8 data bits (between Start and stop bit).
3. Parity bit
4. Stop bit
5. Reset polarity is active low
6. Baud Rate (Transmission rate)
7. System Clock: 100MHz

#### UART Message Pack Description

Figure - UART Message Pack Structure

The UART message has the following structure:

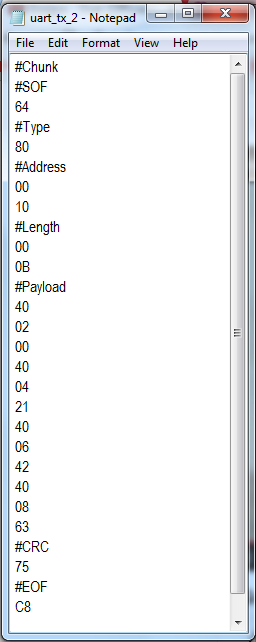
1. **SOF** – Start of Frame
2. **Type** – Message type
3. **Address** – Address for the data
4. **Length** – Data length. Data length CANNOT be less than 1.
5. **Data** **(Payload)** – The wrapped data
6. **CRC** – CRC of Type, Address, Length and Data blocks
7. **EOF** – End of Frame

For examples of reading/writing data from registers/memory, refer to Appendix B in this document.

#### Changes

The Symbol Generator project made use of the UART Receiver block with the *Address Depth* generic value of 2, which means the address is 2 bytes (16 bits).

Example of a UART data transaction from SW with this new address's depth:



Address Depth = 2 bytes

Figure - UART transaction

### VESA Generator Controller

The VESA Generator Controller generates video in the VESA standard:

#### VESA Standard

The VESA standard constitutes of the following control signals:

* Clock – Pixel Clock  
  The refresh rate is defined as the period that a video signal must redraw the entire screen to provide motion in the image and to reduce flicker. In 800X600 resolution mode, with a 60Hz refresh rate, this is approximately 25nsec per pixel, which requires a pixel clock with frequency of 40MHz.
* VSync – New Frame (image) signal  
  The VSync signal commands the monitor to start displaying a new frame, and the monitor starts in the upper left corner.
* HSync – New Line signal  
  The HSync signal commands the monitor to refresh another row of 800 pixels.

After 600 rows of pixels are refreshed with 600 HSync signals, a VSync signal resets the monitor to the upper left corner and the process continues.

* Blank – Blanking signal to the screen (ignore the RGB inputs)  
  During the time when pixel data is not being displayed (Blanking time) and the beam is returning to the left column to start another horizontal scan, the RGB signals should all be set to black color (all zero).

Colors are produced by changing the analog levels of the three RGB signals:

* R (RED) – Analog signal, used to control the color
* G (GREEN) – Analog signal, used to control the color
* B (BLUE) – Analog signal, used to control the color

#### VESA Generator Controller Characteristics

The VESA Generator Controller block has the following characteristics:

* The (R, G, B) Pixels should be supplied to the VESA generator by an outside component.
* Back Porch, Front Proch, Left - Right - Upper - Lower borders, Sync Time and Active pixels / lines should be defined, according to the VESA standard, using the generic parameters.

Definition of the VESA standart timings:

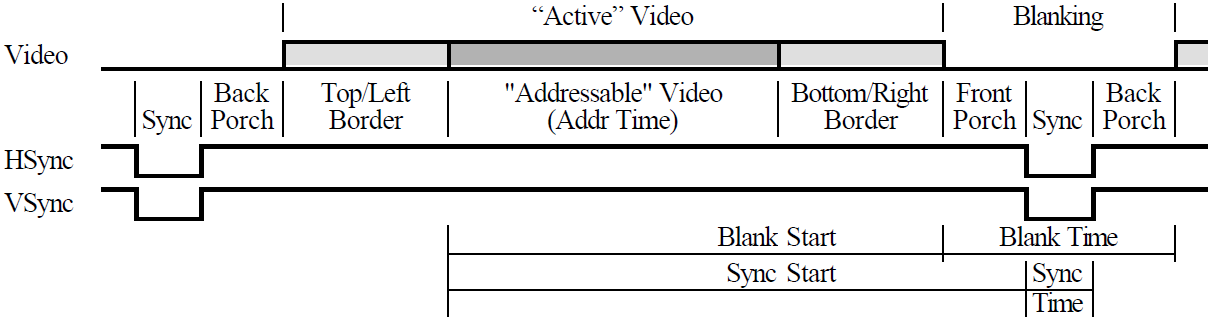


Figure - VESA standart timings

* The R, G and B port sizes may be changed individually, using a generic parameter.
* The R, G and B values will be transmitted, one by one, in the each pixel-clock.

#### Video Frame

The received video frame of resolution 640X480 pixels, is wrapped by this block to the output resolution of 800X600 pixels. The outside black frame is generated using the left, right, lower and upper input registers.

Figure - Video Frame

Note: ROI = Region of Interest (the requested image to display).

More details about the usage and integration with the VESA Generator Controller block is found in the "Display Controller Block" chapter in this document.

### Memory Management

Accessing to the SDRAM is implemented through the Memory Management block, which is built from two main building blocks:

1. Memory Control Write – writing to SDRAM through this IF, using the compatible Wishbone slave target.
2. Memory Control Read – reading from SDRAM through this IF, using the compatible Wishbone slave target.

The arbitration between write and read requests is done using the SDRAM Arbiter block, which is connected to both Memory Control Write and Memory Control Read.

Figure - Memory Management block

Memory Management block includes two important registers:

1. **Type Register (address 0xD)**

The relevant bits in this register are:

* Bit 0: '0' for Normal Mode, '1' for Debug Mode
* Bit 7: '0' for Data Transmission, '1' for Registers Transmission

1. **Debug Address Register (address 0x2-0x4)**

These 3 registers have the address to write/read from in SDRAM at Debug Mode (that is controlled with the Type Register).

The usage of these two registers for Symbol Generator is explained in the "Display Controller" chapter in this document.

#### Changes

In the integration process of our project with the reused system, the need to interface with write requests through the read IF was encountered. Therefore, the Wishbone control signals IF was expanded, and includes now also the Data and Write-Enable control signals, according to the Wishbone protocol.

Adding these changes to the read IF, enabled the accessing to the "Debug Address" and "Type" registers through the Wishbone salve read IF. For details, refer to the Display Controller chapter in this document.

These changes in the IF of the Memory Management block, led also to a few implementation changes in the Memory Control Read block.

### SDRAM Controller

The SDRAM controller implements the IS42S16400 SDRAM Controller, Wishbone compatible, with the following characteristics:

1. Row width: 12 bits
2. Column width: 8 bits
3. Bank width: 2 bits
4. Address structure:
   1. Bank (21 downto 20)
   2. Row (19 downto 8)
   3. Column (7 downto 0)

Address structure illustration:

|  |  |  |
| --- | --- | --- |
| 21-20 | 19-8 | 7-0 |
| Bank | Row | Column |

1. CAS Delay = 3 (required for 133MHz clock)
2. Burst Length = Full Page (256 words - cyclic)
3. 4096 refreshes cycles will be automatically execute per each 64msec.
4. Maximum read/write burst length is 256.

### Global Nets

The global nets in the FPGA system consumed of:

1. **Clock Block**  
   This block receives the FPGA clock of 50MHz, and creates the required clocks in the system, using PLL:
   1. SDRAM clock – 133MHz
   2. System clock – 100MHz
   3. VESA clock – 40MHz
2. **Reset Block**  
   This block is responsible to filter the FPGA asynchronous reset, and create synchronized reset de-assertion to each clock, with PLL-locked dependency.

## Display Controller

### General Description

Display Controller Top diagram:

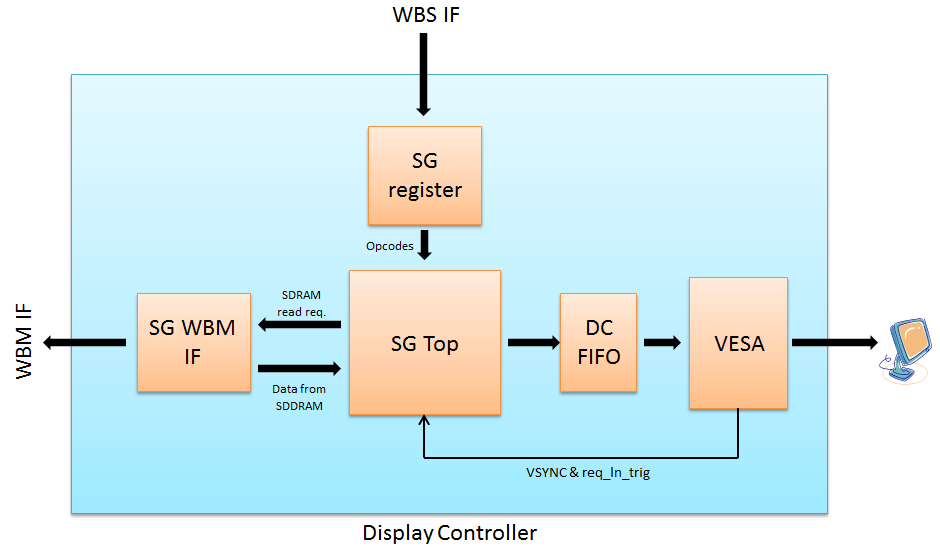


Figure - Display Controller Top diagram

The Display Controller block consists of the following sub-blocks:

1. **SG WBM IF**

This block is the IF of the Wishbone Master to intercon Y in the top architecture

1. **SG register**

This register is responsible to receive opcodes from the host.

1. **Symbol Generator Top**

The Symbol Generator main block, following is a detailed description.

1. **DC FIFO**Stores the incoming data from the SG block, until it is read by the VESA Controller.
2. **VESA Controller**Generates VESA control signals to the VGA output, using the input RGB values from the DC FIFO.

### 

### SG Register

The SG Register is a general register, and is a Wishbone target from outside the Display Controller block.

The SG register uses the concept of base address and address space, which means that every address inside the allowed region is accepted and data is stored:



The maximum number of possible changes in the display is 15x20 symbols = 300 opcodes.

Each opcode consists of 3 bytes transmitted via the Wishbone.

Therefore, the maximum number of bytes in the change of the display is:

(300 opcode) x (3 bytes) = 900 bytes

The SG Register, particularly, has the base address 0x10, and address space of 900 available addresses, with the reason explained above. In total, the legal decimal addresses to the register are:



### Symbol Generator

The Symbol Generator block was described in details in the document of part A of the project.

Here is a brief reminder of the main building blocks.

#### Symbol Generator Top Diagram

The following diagram describes the top architecture of the Symbol Generator block, the

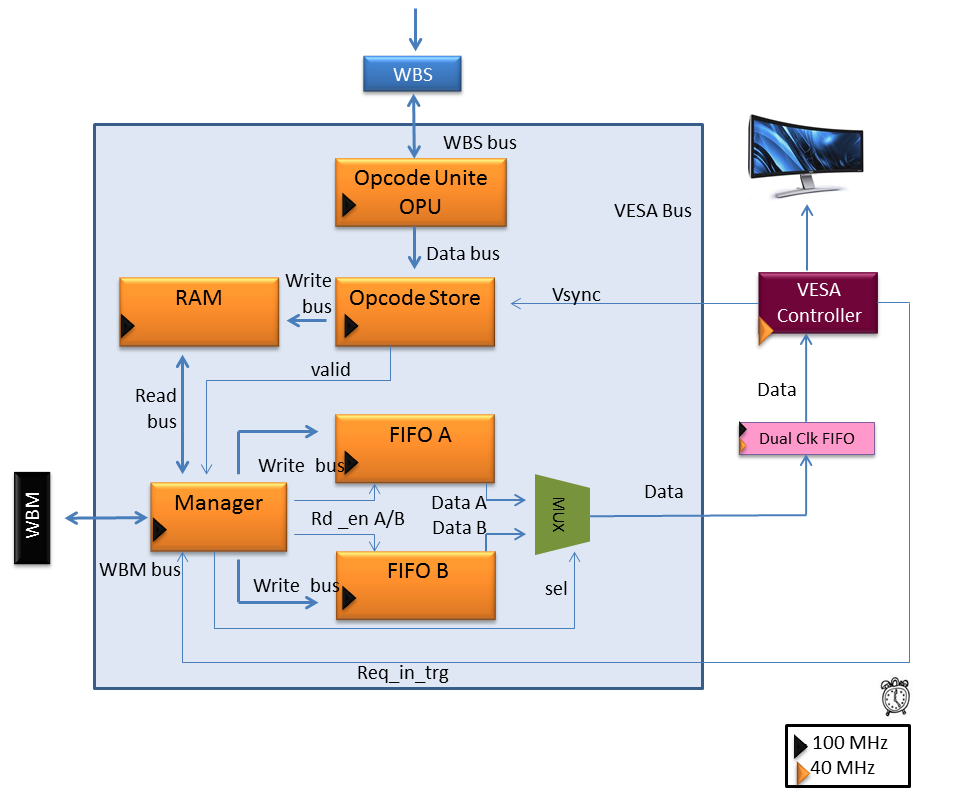
interfaces to the Display Controller block and its building blocks:

Figure - Symbol Generator Top diagram

#### Symbol Generator Algorithm

##### Overview

The main goal of this project is to have a control over changes in the display through the SW, using the **minimum bandwidth** in the HW. To achieve the goal of the minimum bandwidth, there is a need to send only the changes in the display from the current display to the next wanted.

Another important aspect there was a need to deal with is the **limited HW resources in the FPGA**. A simple calculation showed that the required amount of memory for saving a video frame inside the FPGA is not available. Therefore, all the data is saved in an external memory (SDRAM), and accessed when it is needed for the display.

The algorithm of the Symbol Generator gives the answer for these two main issues explained above: bandwidth and limited resources.

##### Video Frame and Symbols

The video frame resolution is 640x480 (row x column) greyscale pixel (each pixel is 8 bits).

The frame is divided into 20x15 blocks (row x column). The blocks are aligned and do not run over each other. Each block size is 32x32 pixels.

The term for a block described above is **symbol**.

Visualization of divided video frame into blocks:

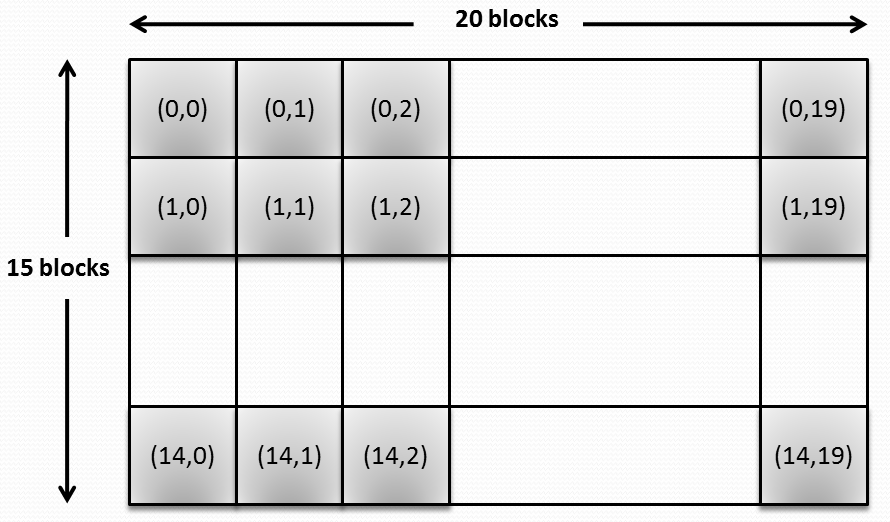


Figure - Divided video frame

The coordinates inside each symbol in the figure above describe its location on the screen, in terms of symbols: (row, column).

The row and column parameters have the following ranges: 

After dividing the display into blocks, the lowest resolution for changes in the display is now in terms of blocks.

##### Interaction with SDRAM

The SDRAM has the following sizes:

* Depth: 2^12 rows
* Width: 2^8 words (1 word = 16 bits) = 256 x 16 = 4096 bits
* Total of 4 banks

As described above, each symbol size is:

32x32 pixels = 32 x 32 x 8 bits = 8192 bits = 2 x (4096 bits)

This calculation shows that each symbol can be stored in the SDRAM using 2 sequential rows of it. It means that 1 row in the SDRAM stores 16 out of 32 rows of the symbol.

Visualization of the relation between the SDRAM and the symbols:

The symbol:

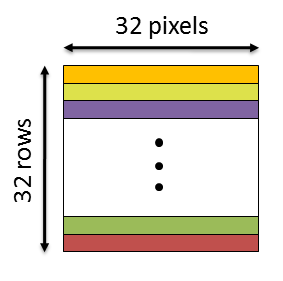


Figure - Symbol image

The symbols stored in the SDRAM:

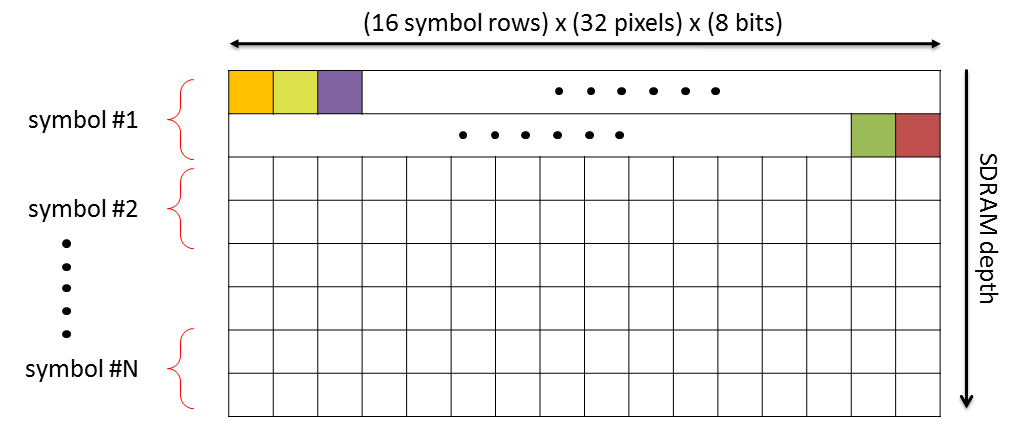


Figure - Symbols stored in SDRAM

##### Opcode Structure

Transferring only the changes of the display is done using **opcodes**, which are data packets.

These opcodes have only the necessary information to imply the required change in the display:

* The location of the change in the frame.
* The character of the change: adding data or removing.
* If adding data: where this data is stored in the memory?

After understanding the necessary fields, the structure of the opcode is set as following:

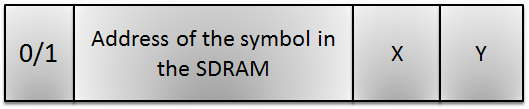


Figure - Opcode structure

The opcode is total 23 bits:

* **0/1 – command type (1 bit)**0 = remove the symbol from frame, instead insert black pixels and ignore all the next fields in this opcode.  
  1 = add the symbol to the frame, with the information from the next fields in this opcode.
* **Address of the symbol in the SDRAM (13 bits)**
  + SDRAM row (11 bits)  
    SDRAM depth is 2^12 rows, which means that 12 bits are needed to represent the row number in the SDRAM.  
    Each symbol is saved in 2 sequential rows in the SDRAM (as described in section 4.3).  
    Therefore, it's enough to use the 11 MS bits in the SDRAM row representation. The lowest 12th bit is calculated in the Symbol Generator block according to the required data.
  + SDRAM bank (2 bits)  
    SDRAM has 4 banks, therefore, 2 bits to represent the bank.
* **X – column coordinate in the frame (5 bits)**The column coordinate in the frame has the range: 
* **Y – row coordinate in the frame (4 bits)**The row coordinate in the frame has the range: 

##### Inside Symbol Generator

Description of the main functions of the building blocks of Symbol Generator:

1. **RAM**

The Symbol Generator block receives the opcodes that represent the changes in the display, and it needs to apply these changes.

The implementation contains a RAM of size 300 rows; each row matches a symbol block on the display (total of 15 x 20 blocks in the frame).

The Symbol Generator block extracts from the opcode the information of the character of the change and its location in the frame. This data is contained in the row of the SDRAM and in the data stored there.

Visualization of the relation between the RAM and the symbols in the frame:

The symbol blocks in the frame:

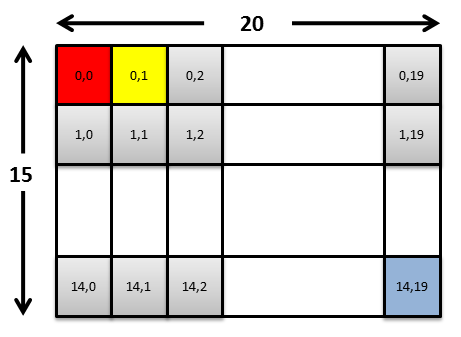


Figure - The symbol blocks in the frame

The RAM inside Symbol Generator block:

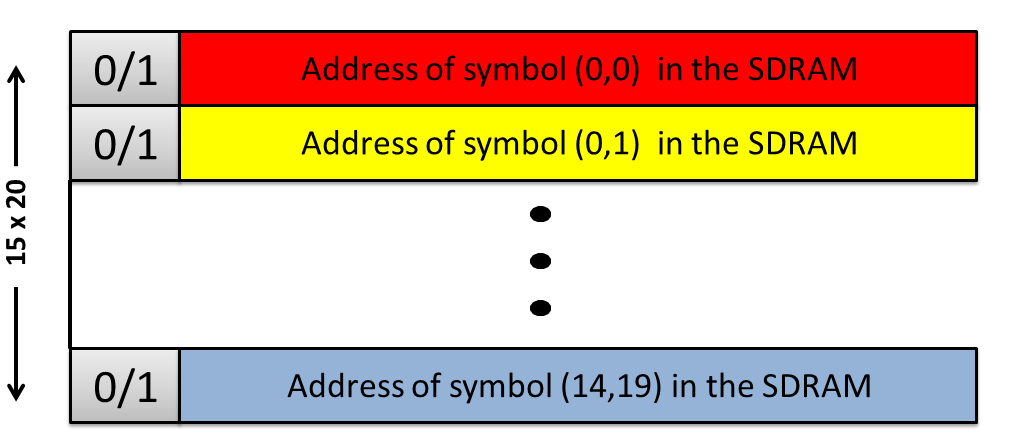


Figure - RAM inside Symbol Generator block

The visualization above describes the unique match between a block in the frame and a row in the RAM.

Calculating the row in the RAM from the frame coordinates:



x – symbol column in the frame , 

y – symbol row in the frame , 

The data that is stored for each symbol in the RAM is the address of the symbol data in the SDRAM, exactly as it appeared in the opcode.

1. **FIFOs**

As described in the overview of the algorithm, the required amount of memory for saving a video frame inside the FPGA is not available. Therefore, all the data is saved in an external memory (SDRAM), and accessed when it is needed for the display. This means that the Symbol Generator is responsible for managing the reading of data from the SDRAM and passing is to the VESA display.

Each FIFO size is one video frame row, means: (640 rows in the FIFO) x (8 bits per pixel).

Visualization the relation between the data stored in a FIFO and the video frame:

First video row is stored in the FIFO:

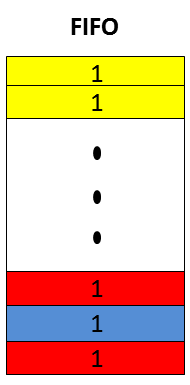


Figure - Data stored in FIFO

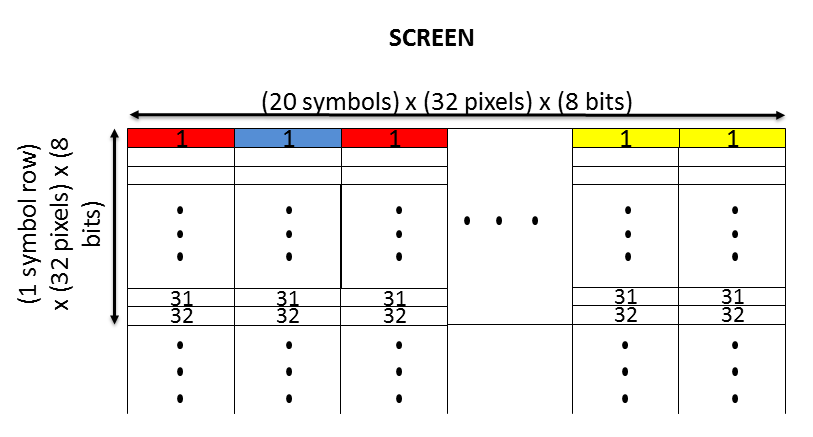
This data as represented on the display:

Figure - Data as represented on the display

The described process, of managing the reading of data from the SDRAM and passing is to the VESA display, is implemented by using two FIFOs; each FIFO contains data for one video row. The core of the algorithm is in the toggling between the FIFOs: while reading data from one FIFO to the current row in the display, the second FIFO prepares data for the next row by reading the relevant data from the SDRAM.

The toggling between the two FIFOs, FIFO A & FIFO B:

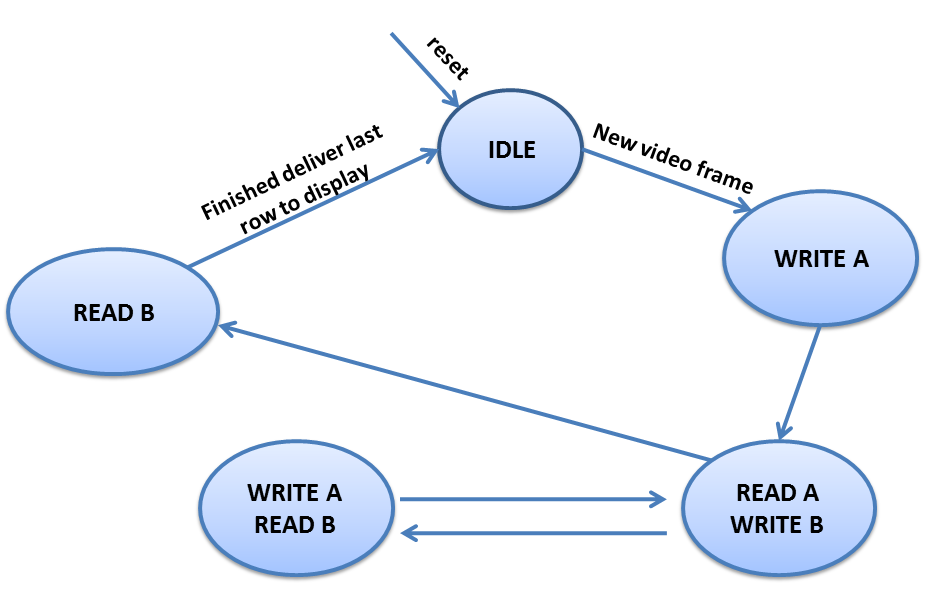


Figure - Toggling between FIFOs

#### Symbol Generator Details

Following is a brief description of each sub-block in the Symbol Generator architecture.

##### Opcode Unite

Opcode Unite block, unites every 3 packets from WBS into 1 opcode and writes data to Opcode Store block.

##### Opcode Store

The main purpose of the Opcode Store block is to store commands from the Opcode Unite block. The reason that we want to store the commands before transferring them to the RAM is because we don't want to override the current state of the display in the RAM, in case it is the middle of the frame. We will write to the RAM, only at the beginning of a video frame (When VSYNC of the VESA is active). Therefore, a FIFO is embedded in Opcode Store, used as a buffer between current frame and next one.

##### RAM

RAM size: 300 x 13 (rows x bits).

The RAM stores the current display ready to be transferred to VESA. It is filled by Opcode Store, so each row in RAM matches to a symbol on the display respectively. In each row the address of the first pixel (of the symbol) in SDRAM is saved as 13 bits. Manager manipulates the RAM, extracting in each clock the relevant address from RAM.

##### Manager

The Manager block is the controlling "brain" of the Symbol Generator block. It executes the main algorithm explained in section 4. Manager is being operated by Opcode Store, through a control signal, and by VESA Controller, through the req\_in\_trg signal.

In this block, 3 functions are being fulfilled:

1. Calculating the relevant row in the RAM, from which it manages the read. This includes asserting the RAM\_rd\_en signal for the RAM.
2. Receiving the address of the symbol in the SDRAM from the RAM, and calculating the desired row and column in the SDRAM for read, according to the current video row in the display frame.
3. Managing the toggling between the two FIFOs, using a final state machine. This FSM controls to which FIFO to write to and from which FIFO to send the data to VESA. This involves assertion of the control signals of both FIFOs (read and write enables).

##### FIFO A\B

The toggled FIFOs, FIFO A and FIFO B, are each the size of 640X8 [rows x bits]. While one is being read by VESA Controller, the other is filled with the next video row. They are both being controlled through Manager.

##### MUX

The MUX is controlled by the Manager. It is used as a selector between FIFO A and B. Whenever data is ready to be sent by one of the FIFOs, it is transferred through the MUX to the DC FIFO.

### Reused Video Blocks

The Display Controller block contains original blocks, like the Symbol Generator block and its internal sub-blocks, and also re-used blocks.

The Re-use of blocks from a previous project that was done by somebody else, requires a deep understanding of the architecture and detailed implementation. And also gives the challenge of integrating these re-used blocks with the original blocks.

In the Display Controller, the re-used blocks are: DC FIFO and VESA Generator Controller.

#### DC FIFO

The purpose of the DC (=Dual Clock) FIFO is for clock domain crossing: from the clock domain of the system clock (100MHz) to the domain of the VESA pixel clock (40MHz).

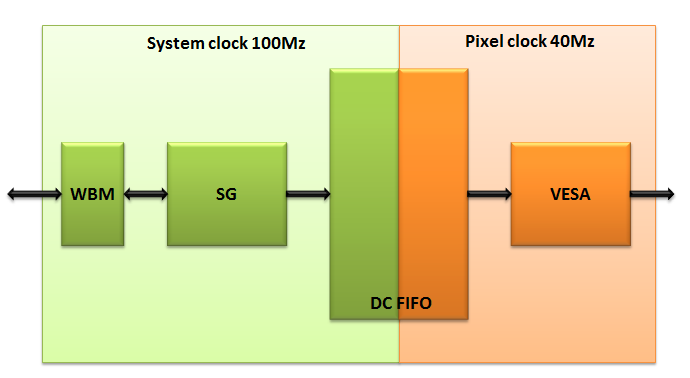


Figure - Clock Domain Crossing

#### VESA Generator Controller

The VESA Generator Controller was described in the "Reused Blocks Architecture" chapter.

### SG WBM IF

The SG WBM IF block is an interface between the SG block and the Wishbone communication net in the FPGA architecture.

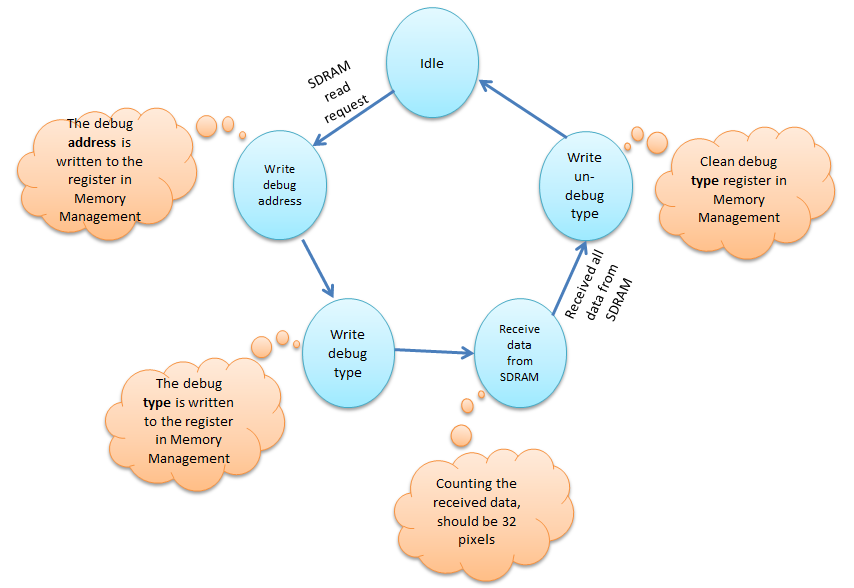
The SG WBM IF block receives the read requests to the SDRAM, translates these requests to Wishbone protocol and executes them according to the following FSM:

Figure - 2.3.5 SG WBM IF FSM

* The Debug Address Register is a register in the Memory Management block. And the debug address is the address that the Manager component requests the read from the SDRAM. This address is 24-bits, and is written to the 3 registers of the Debug Address Register.
* The Type Register is a register in the Memory Management block. In order to be in debug mode, the value '1' is written to the LSB of this register, as explained in the "Reused Blocks Architecture" chapter.
* After writing to these two registers, the SG WBM IF block initiates a read transaction from the SDRAM. Because it is a debug mode, the Memory Management uses the read address as the address which is stored in the Debug Address Register.  
  The read data from SDRAM is sent to the Symbol Generator, stored in one of the FIFOs (FIFO A or FIFO B), and then transferred to the DC FIFO to be shown on the VESA display.  
  The SG WBM IF block counts the received data, which should be 32 pixels as requested.
* After receiving the 32 requested pixels, the debug mode is cleaned from the Type Register, so that normal functioning could be continued, and idle state is reached until another SDRAM read request.

This process of the FSM is executed for each symbol row in the video row. This means, it is executed 20 times in one video row.

The mentioned Memory Management registers, Debug Address and Debug Type, are described in details in "Reused Blocks Architecture" chapter in this document.

# SW IF

The interface with the SW is done by MATLAB GUI, which sends serial words through the RS-232 connector to the FPGA.

## GUI Preview

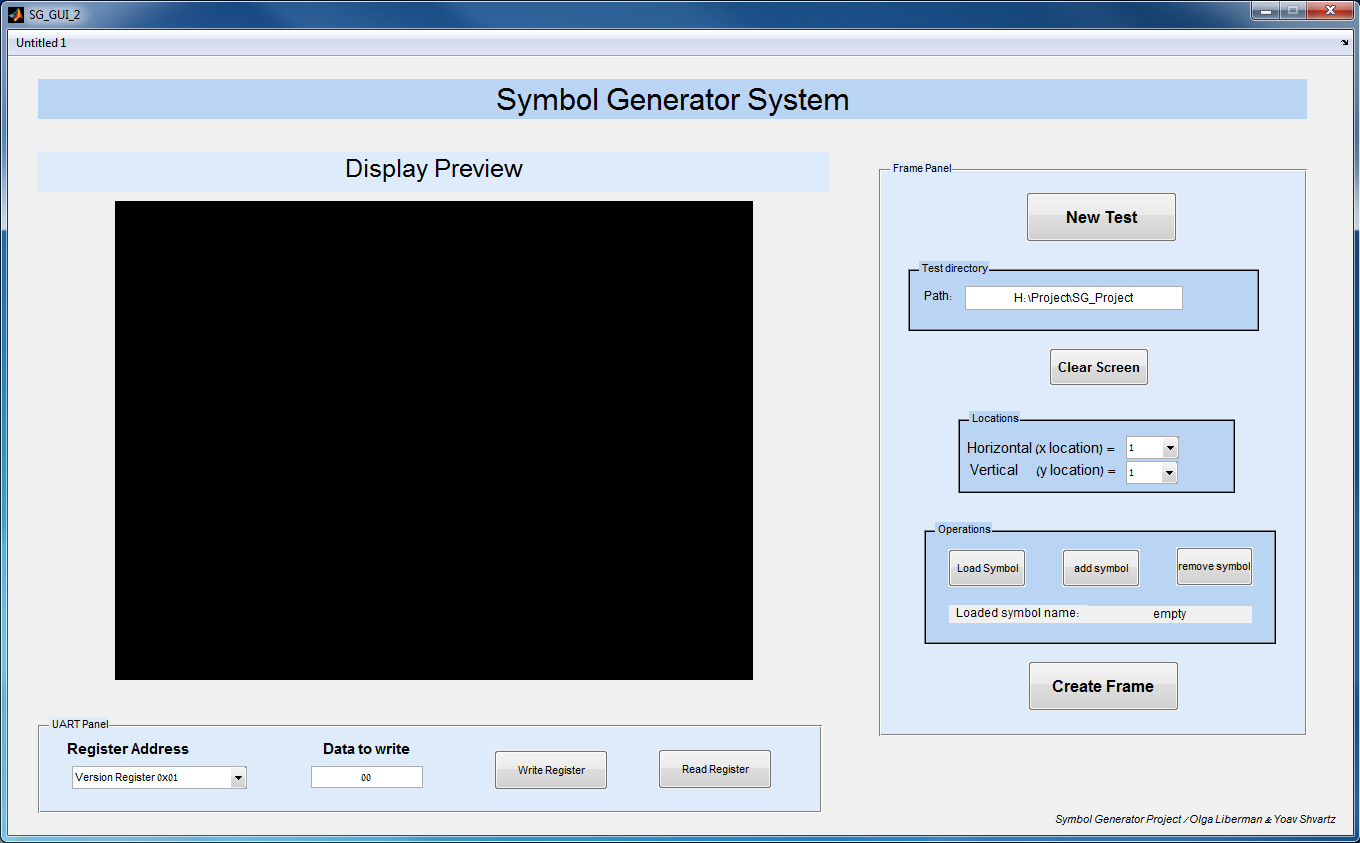
Following is the GUI for the Symbol Generator project:

Figure - GUI preview

## GUI Functions

### New Test



Figure - New Test button

The "New Test" button sends a chunk of commands to the FPGA:

1. Initializing the RAM inside the Symbol Generator block with zeros. Because the RAM reflects the 300-symbols on the screen, the initial value should be zeros, which are black symbols.

Example for an initialization message:

Symbol Generator register address is 0x10

Length is 0x0383, 899 in decimal

It means 900 bytes in the UART chunk

Type 0x80 means writing to a register

Payload that sets the Symbol Generator RAM with zeros

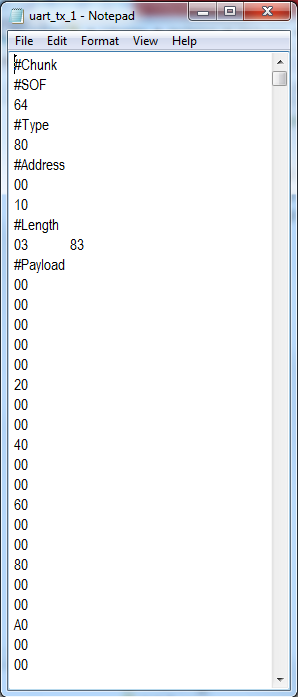


Figure – RAM initialization message

1. Loading the SDRAM with the pool of symbols.

Pool of symbols for example:

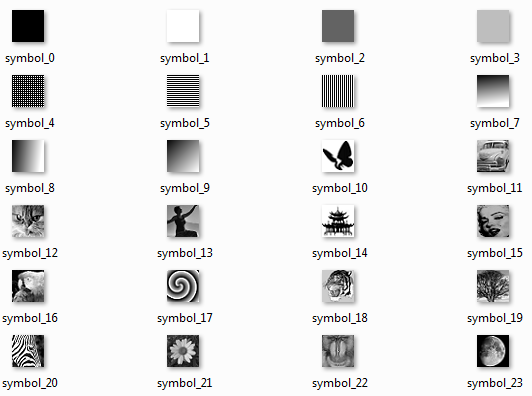


Figure - Pool of Symbols

After loading the symbols to the SDRAM, these symbols will be available for display on the screen.

### Locating and Choosing the Symbols

Figure - Locating and Choosing the Symbols

This panel gives the possibility to choose the wanted symbol through the "Load Symbol" button.

Choosing the location for the symbol on the screen is done by choosing the horizontal and vertical coordinates on the screen.

### Clear Screen

The "Clear Screen" button clears all the symbols from the preview screen.

Figure - Clear Screen button

When the "Create Frame" button will be pushed, then a chunk of black symbols will be sent to the FPGA.

### Display Preview

Through the whole work in the GUI, the "Display Preview" panel gives an authentic preview of the expected view on the screen.

Example of a preview:

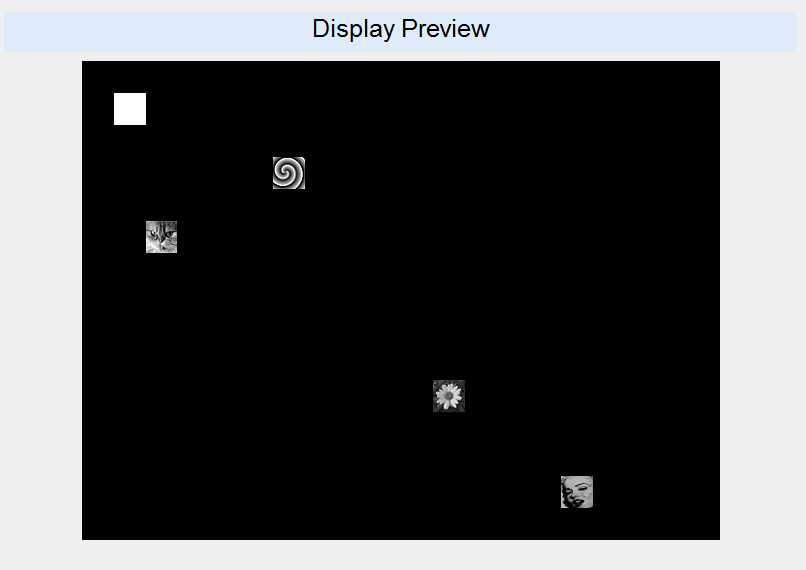


Figure - Example of a Preview

### Create Frame

Figure - Create Frame button

After choosing the symbol and its location and adding it to the display, pushing "Create Frame" transmits the relevant opcodes to the FPGA with the display change.

This action leads to the same change on the display which is connected to the board.

### Registers Panel

Figure - Register Panel

The GUI also gives an option to read and write to registers in the FPGA.

When writing to a register: choose the register from the list, fill the data in Hexadecimal in the appropriate field and push "Write Register" button.

When reading to a register: choose the register from the list and push "Read Register" button. The data from the register will appear in a new window on the screen.

# Testing

The testing part of the project can be divided into 2 categories:

1. Simulation tests
2. Lab tests

## Simulation

### Automatic Simulating

The main goal of creating the Automatic Test Manager is the ability to run a series of tests in click of a mouse.

The idea is the **"Golden Model"**: test environment that runs the tests, compares the output with the expected output and gives the result of the test – pass or fail.

Making automation in the simulation environment, involved the following:

1. Preparing test plan
2. Preparing test-directory's structure
3. Adjusting the GUI for the simulation purposes
4. Running the tests with a TCL script
5. MATLAB algorithm for comparing the "expected" to the "output"

#### Test Plan

The purpose of a good test plan is to bring the simulated system to its edges, and find "bugs" if exist.

These test cases we used have the following categories:

* Data scenarios  
  different data amounts that arrive to the Symbol Generator block.
* Timing Scenarios  
  data packets that arrive on different timings relatively to the VSYNC signal.

For example, to test the different data scenarios, we used **random tests** that were generated using MATLAB (random locations on the screen and random symbols).

#### Test Directory

The test directory has the following structure:

Figure - Test Directory structure

The top directory is "test files", and it includes all the tests.

Beneath each test directory (for example: "test\_0" in the image), the daughter directories are:

1. TX files: Directory consists of all the "transmitted" UART data to the FPGA.
2. Expected: Directory consists of all the expected images, which are the output of the Simulation-GUI.
3. Output: Directory consists of all the result images, which are the output of the ModelSim simulation.

### Simulation-GUI

For simulation purposes, we used a MATLAB GUI that was adjusted for simulations.

It means there's no connection to Serial Port in the GUI. Instead of transmitting the UART data to the "COM", the data is written in text TX files in directory "uart\_tx" (as presented in the image above).

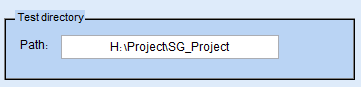
The special panel in the GUI for simulations:

Figure - Simulation Panel

This path shows under which directory the "test\_files" directory will be built, with all its' tests.

The "New Test" button in the GUI is responsible for creating a new TX file for the current test.

### TCL script

Using a TCL script, we could control the tests we're running.

The script ran a simulation for each test in the list of the file "tests.do", with the unique generics for each test.

Example for tests.do file:

Figure - Example for tests.do fie

### Image Comparison

A MATLAB algorithm was used to compare the directories of "expected" and "output" images.

"PASS" was the result when the directories had the same images.

### Simulation Waveforms

#### Initialization Process

Following is the waveform of the initialization process, which includes:

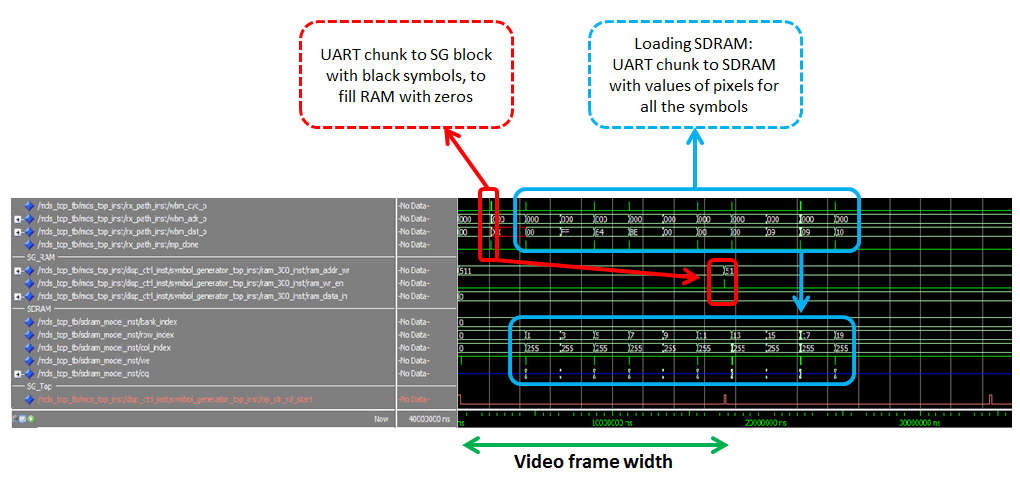
1. Initializing the Symbol Generator's RAM with zeros (=black symbols).
2. Loading the SDRAM with symbols.

Figure - Initialization Process

#### Arrival of New Opcodes

The following waveform shows the arrival of new opcodes from the UART to the Symbol Generator block, and its influence on the Symbol Generator's RAM:

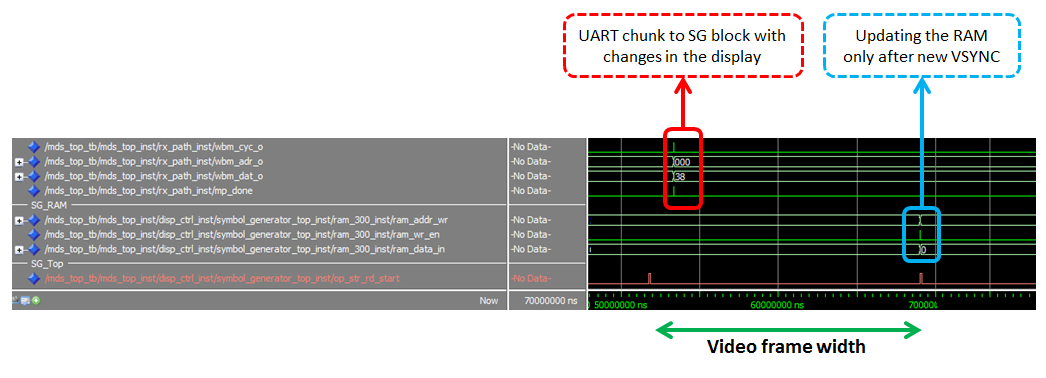


Figure - Arrival of New Opcodes

#### Function of Symbol Generator

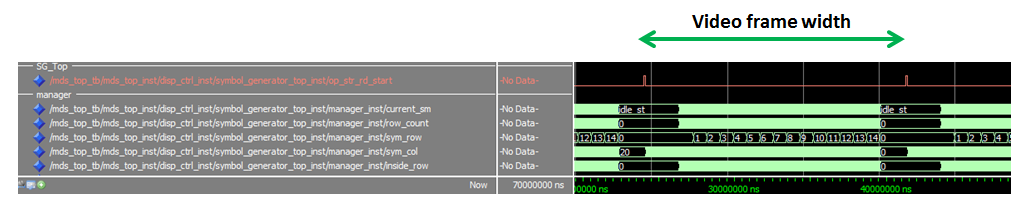
The following waveform shows the functioning of the Manager block inside the Symbol Generator. The counters represent the state in the FSM, row inside the frame, row inside symbol and column inside symbol.

Figure - Manager Block function

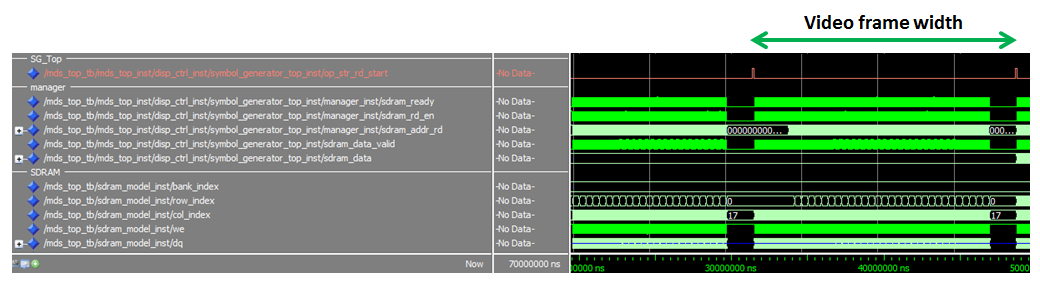
The following waveform shows read requests from the SDRAM that the Manager block initiates and the SDRAM's response with data:

Figure 43 - SG reads from SDRAM

#### VESA Output

The Following waveform shows the VESA control signals and data to the display:

Figure - VESA Output

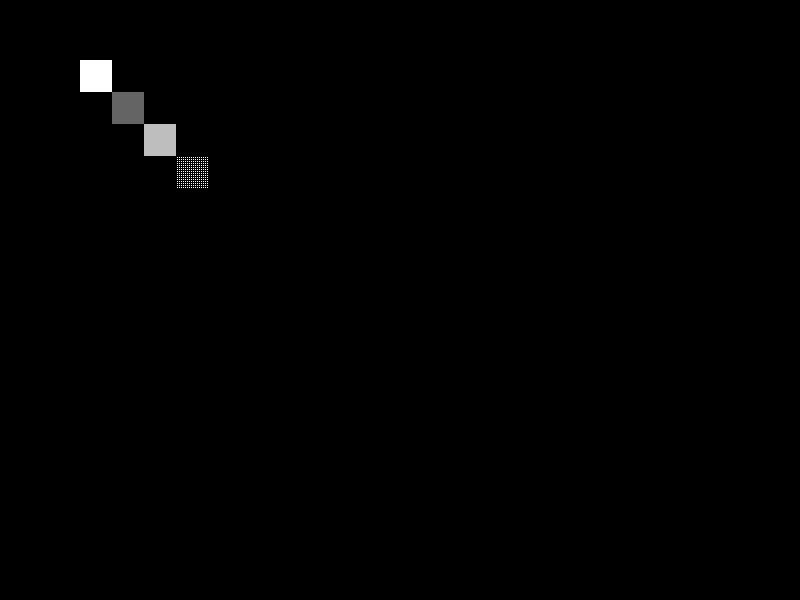
The VESA pattern in the waveform above gives the following result image:

Figure - Output Image

## Lab Examination

### The System

The full system includes:

1. Altera DE2 board with Cyclone II FPGA and SDRAM
2. Computer
3. MATLAB GUI (as our SW IF)
4. JTAG connector to the computer, to burn the FPGA
5. RS-232 connector to serial port in the computer
6. VGA connector between the board and the screen for display
7. Standard computer display

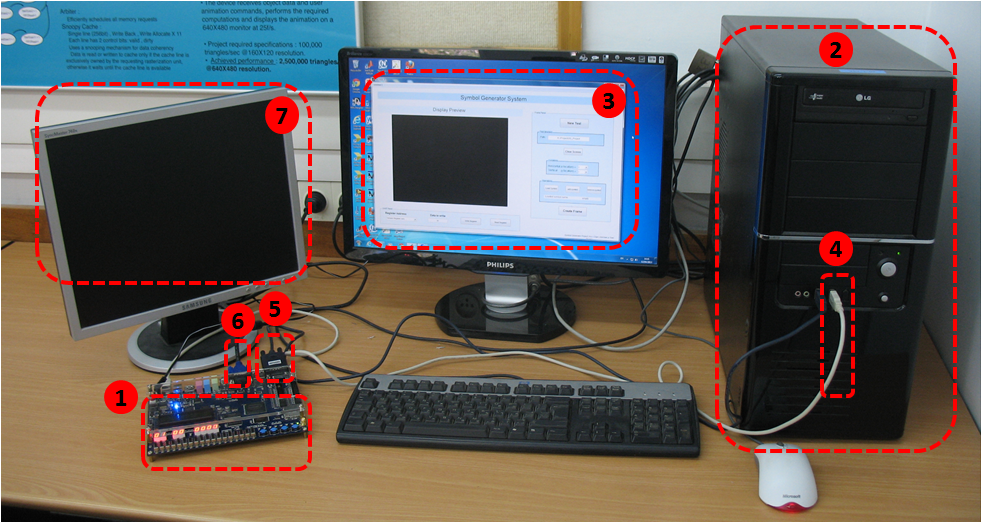
Following is an image of our connected system with all the listed components above:

Figure - Full System image

### Life Indications

Here is a list of useful methods to check if the board and FPGA are "alive".

#### LED

The very first "life indication" from the board and FPGA is using a blinking LED (1Hz) on the board to indicate that the PLL in the FPGA is locked:

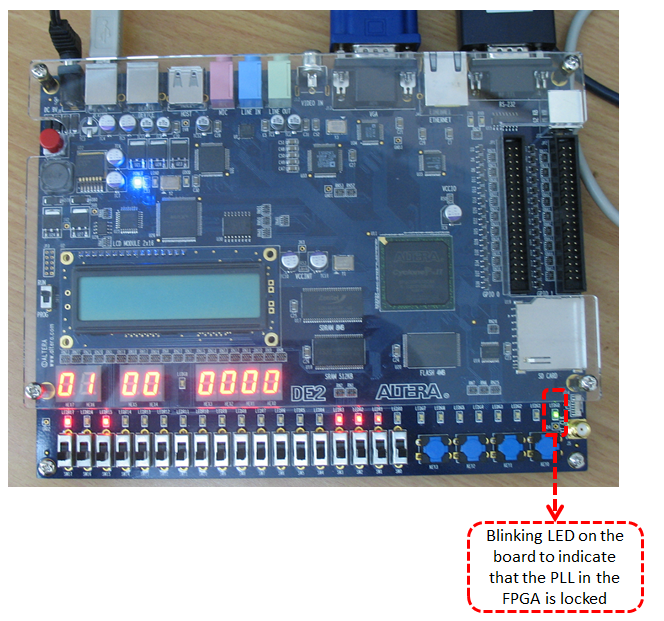


Figure - Blinking LED on DE2 Board

#### Registers

The next "life indications" are reflecting the values of important registers to the 7-segment-display on the DE2 board:



Figure -Register's values on 7-Segment Dsiplay

### Signal Tap

After connecting the full system for the first time and burning the first FPGA version, of course nothing worked. And a lab debug was needed.

The first problem we encountered was with the port and interfacing it from the MATLAB. Therefore, a Signal Tap was used to see the exact data that arrives from the MATLAB to the FPGA:

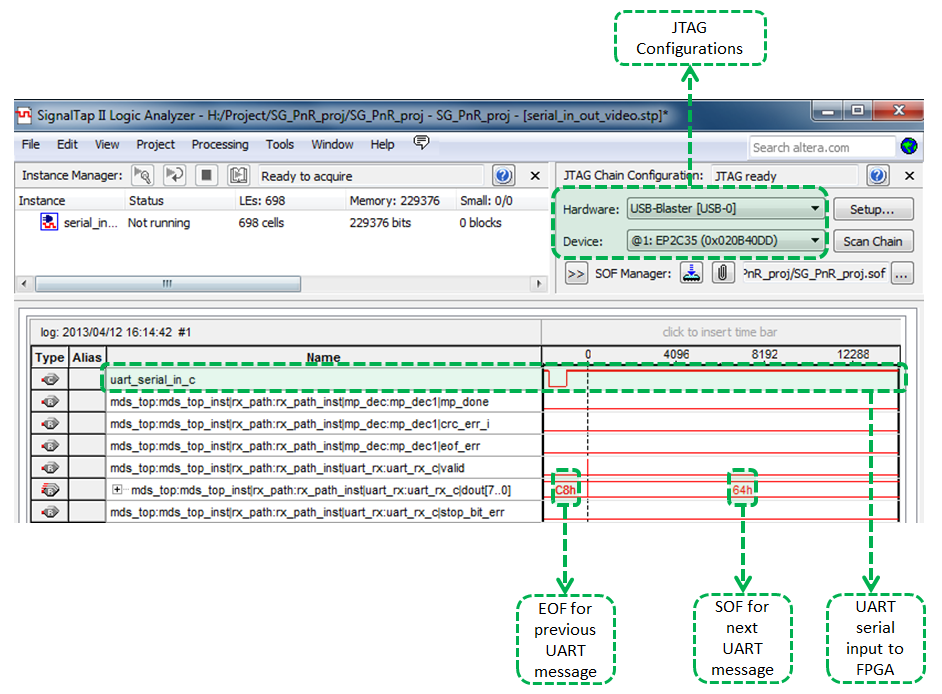
After fixing the serial RS-232 IF, the next problem was a white screen on the display. And there was a need to figure out what part of the system in the FPGA didn't work.

Figure - Signal Tap example

The Signal Tap was used here to look at all the important parts of the FPGA, for example: SDRAM IF, VESA IF and inner blocks of the Symbol Generator.

The result on the Signal Tap signals was that no data was written to SDRAM when it should have been loaded to it. This fact led us to look deeply in the Synthesis and Place&Route reports, and to see that a mistake in the SDRAM pin allocation was made.

Fixing the pin allocation in the FPGA solved the problem, and the Signal Tap signals showed the values of pixels being written to the SDRAM.

These two examples show the usage we did with the Siganl Tap tool, which helped us to solve real-time problems that occurred only in the lab and not in the simulation world.

### Working System

After using all the possible engineering and debug tools we had, the full system started to work as we planned.

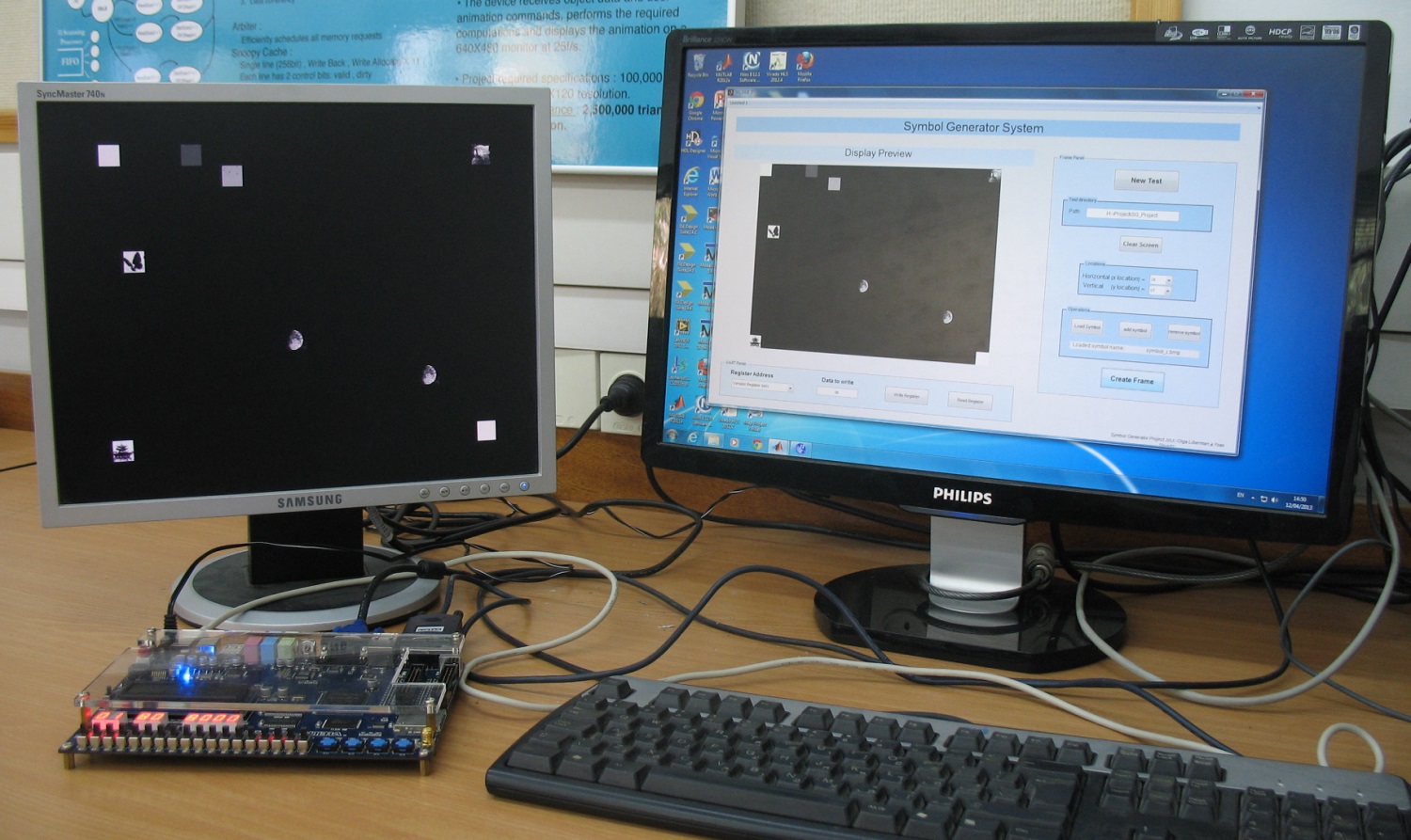
Following is an image of the working system:

Figure - Working System's image

Note the display preview on the GUI, and exactly the same display on the screen to the left of the GUI.

# Implementation Tools

## Synplify

In this project we used the Synplify tool for FPGA synthesis.

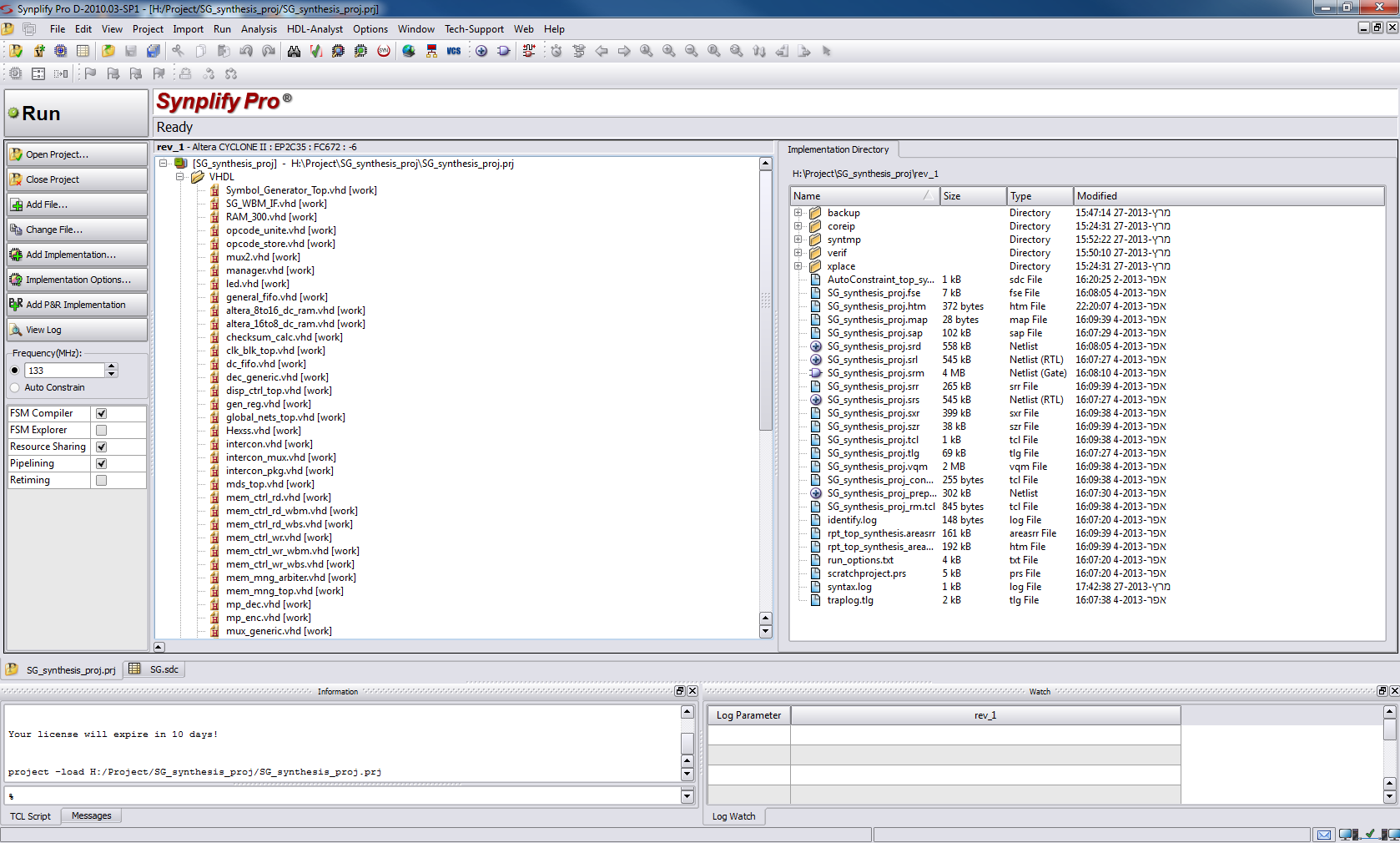
A screen shot of the project in Synplify:

Figure - Synplify Project

The constraints on the clocks in the design:

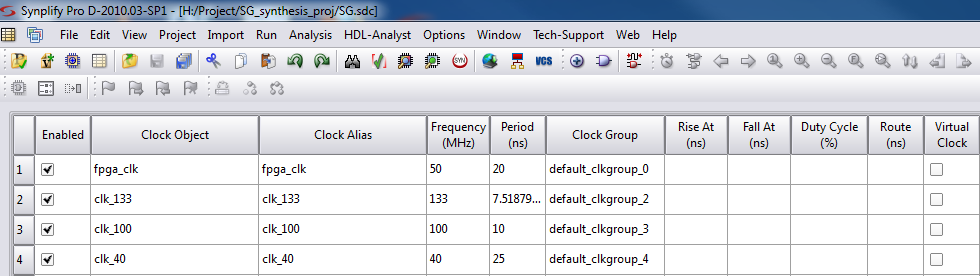


Figure - Synplify Clock constraints

## Quartus

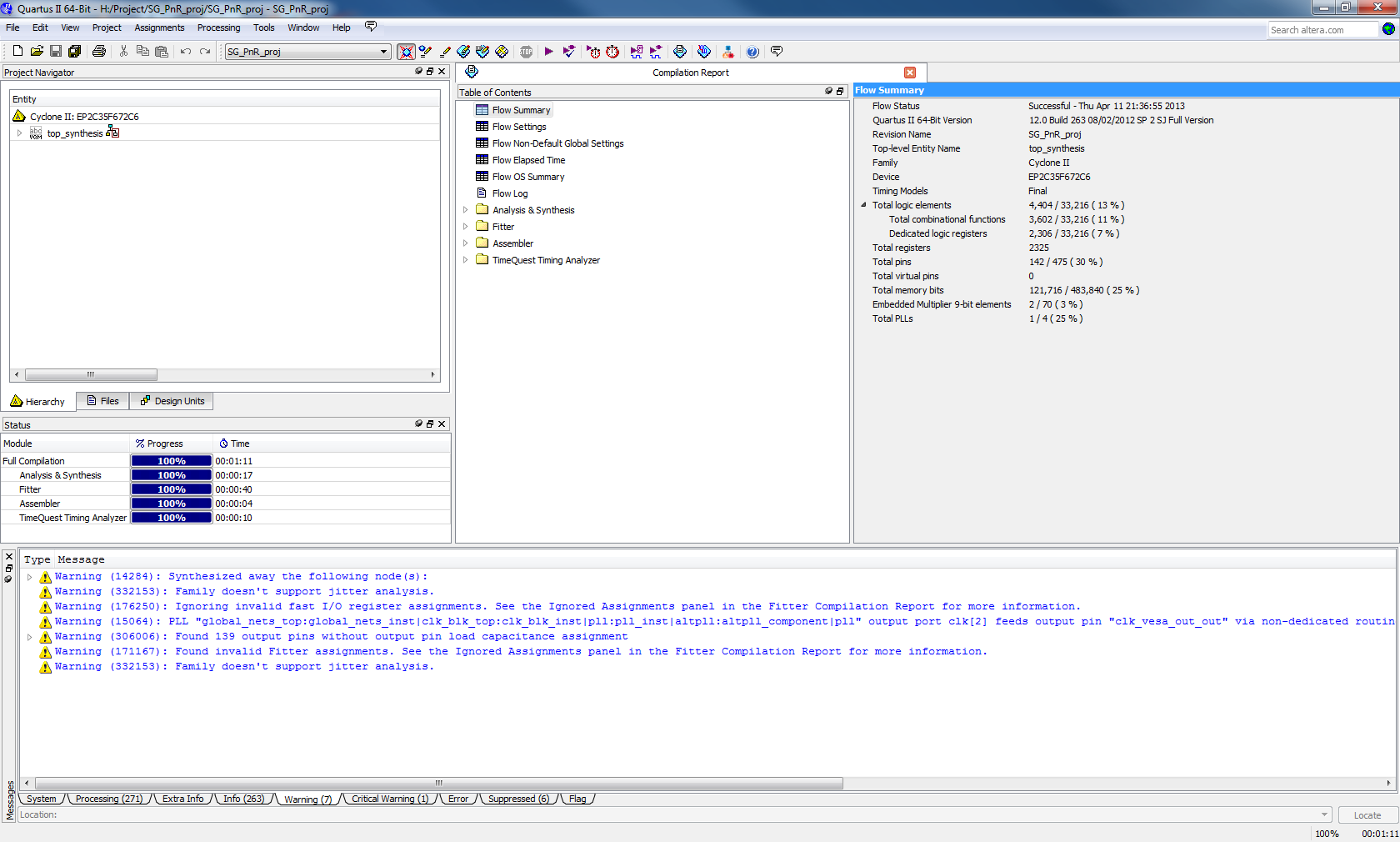
The Quartus tool was used for Place&Route process:

Figure - Quartus Project

The Quartus used as an input the result of the synthesis in the Synplify (\*.vqm file):

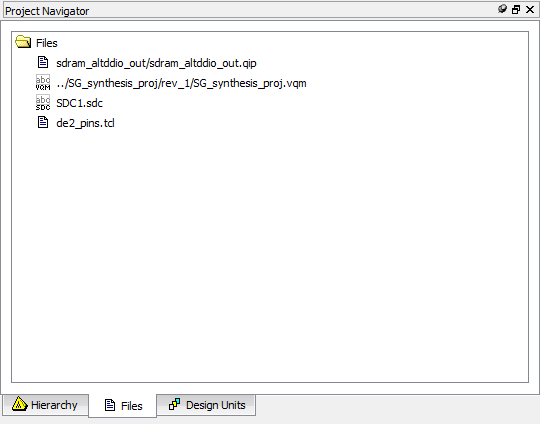


Figure - Synplify to Quartus

### Resources Usage

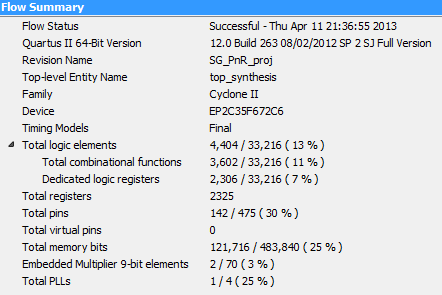
Following is the report of the Place&Route process in Quartus:

Figure - Place&Route Report

# Working Methods

## SVN

The SVN proved to be a useful tool in the documentation process. The tool helps synchronizing the project designers and supervisor, document the changes with the project progress and backup each version. An SVN snapshot example can be shown below:

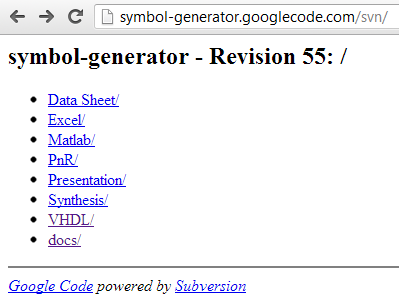


Figure - SVN snapshot

The structure of the project in the SVN library:

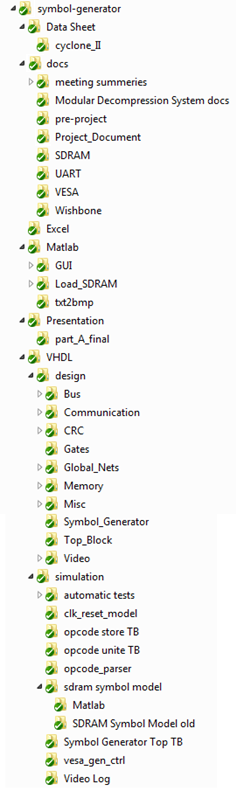


Figure - SVN project structure

# Summary

## Project Usage

Generating symbols on display screens can be found in any system that uses them. It can be mobile phones, TV screens and even military applications. The main issue, which our project deals with, is how to display the images fast and in real time.

|  |  |  |
| --- | --- | --- |
| C:\Users\Olga\Documents\technion\project\presentation\iphone-apple.png | C:\Users\Olga\Documents\technion\project\presentation\tv-simpsons2.jpg | C:\Users\Olga\Documents\technion\project\presentation\ELEC_ANVIS-HUD_Elbit_Day-Night_lg2.jpg |
| mobile phones | TV screens | military applications |

Figure - optional project's usage

## Problems during the project

We have encountered some obstacles during the project. The major obstacles are detailed here:

1. **Problem:** Interfacing the Memory Management block from the Display Controller block. This was crucial because the addresses from where to read from the SDRAM had to be updated before the extraction of the bits.

**Solution:** Connecting Intercon Y to Memory Management block so a write operation can be done as well as a read operation. See "Reused Blocks Architecture" chapter for more details.

1. **Problem:** Converting the SG Top read and write requests to the WB protocol. This had to be done since the platform communication between different blocks is based on this protocol.

**Solution:** Creating SG WBM IF block for the conversions. For this, a full understanding of the WB protocol had to be done first. See "SG WBM IF" chapter for more details.

1. **Problem:** UART transmission from Matlab to the board could not be implemented properly.

**Solution:** Signal Tap was used to track the source of this problem. After examination of the signals, the problem was solved. See "Signal Tap" chapter for more details.

## Conclusions

The main goals of this project, building SG blocks, integrating to an outside platform, design the system while considering constrains, simulating and debugging VHDL code, syntheses and P&R had been achieved. In addition, an automatic test GUI tool and operational GUI tool for the user's usage were created.

Moreover, goals had been achieved by:

1. Organized working methods that saved time including SVN use.
2. Proper use of Quartus and Synplify capabilities.
3. Documentations – helps for better understanding the system.

# Abbreviations

Video row – the pixels of a specific row in a video frame

Symbol row – the pixels of a specific row in a symbol

SDRAM – Synchronous Dynamic Random Access Memory

RAM – Random Access Memory

TX – Transmission

RX – Receive

FIFO – First in First out

PLL – Phased Locked Loop

TB – Test bench

SOF – Start of frame

EOF – End of frame

MP – Message Pack

TB – Test Bench

UART – Universal Asynchronous Receiver Transmitter

VESA - Video Electronics Standards Association

VGA - Video Graphics Array

DVI - Digital Visual Interface

IP – Intellectual Property

LSB – Least Significant Bit

MSB – Most Significant Bit

MHz – Mega Hz

DUT – Design under test

SG – Symbol Generator

WB – Wishbone

WBS – Wishbone slave

WBM – Wishbone master

P&R – Place and Route

# Appendix A: Register Map

## Registers Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Register's name | Address | Size (bytes) | Purpose | Place |
| Symbol Generator Version | 0x0001 | 1 | Symbol Generator Version | Disp\_ctrl\_top |
| Dbg\_reg | 0x0002🡪0x0004 | 3 | Write to / read from this address in SDRAM at Debug mode | Mem\_mng\_top |
| Top\_frame\_reg | 0x0007 | 1 | Upper frame of image | Disp\_ctrl\_top |
| Buttom\_frame\_reg | 0x0008 | 1 | Lower frame of image | Disp\_ctrl\_top |
| Rd\_burst\_len\_reg | 0x0009🡪0x000A | 2 | Read burst length from SDRAM / Register in Debug mode | tx\_path |
| Dbg\_cmd\_reg | 0x000B | 1 | (Clear on Read)  When its value is 0x1, then a Wishbone Read transaction is executed from register / SDRAM, and transmitted through UART. | tx\_path |
| Reg\_addr | 0x000C | 1 | Address of register to be read at Debug Mode | tx\_path |
| Type\_reg | 0x000D | 1 | Type of message | Mem\_mng\_top, |
| Type\_reg | 0x000E | 1 | Type of message | disp\_ctrl\_top, |
| Type\_reg | 0x000F | 1 | Type of message | tx\_path |
| Symbol Generator Opcodes  See note (1) | 0x0010 to 0x0393 | 1 | Transmit opcodes to the Symbol Generator block |  |

Notes:

* + 1. The "Symbol Generator Opcodes" registers are located in address space: 0x0010 to 0x0393, total of 900 available addresses. Each register in this address space is of 1-byte width.

When writing to Symbol Generator block, start at address 0x0010 in a burst mode. The addresses will increase automatically with the burst mode in UART RX.

## Type Register Values

Type Register addresses: 0xD, 0xE, 0xF.

* Type\_register [0]: '0' for Normal Mode, '1' for Debug Mode.
* Type\_register [1]: '0' for Image Transaction, '1' for Summary Transaction.
* Type\_register [2]: Displayed image from VESA generator: '0' for Image Transaction (From SDRAM), '1' for Synthetic Pattern Generator.
* Type\_Register [7]: '0' for Data Transmission, '1' for Registers Transmission.

Type register (all the 3, from address 0xD to 0xF) will be automatically updated at each message transmission from the RX\_PATH.

# Appendix B: Writing, Reading from and to SDRAM and Registers

## Register Access

### Limitations

**Important Note:** Two groups of registers cannot be written / read at the same transaction.

For example:

1. DBG\_REG (Address: 0x2; Burst length: 0x2 = 3 bytes) transaction is OK.
2. DBG\_REG & Left\_Frame\_Reg (Address: 0x2; Burst length: 0x3 = 4 bytes) cannot be executed, and will cause the system to stop function until Reset is executed.

### Write to Registers

***Note***: *Type\_reg should not be written by the software. It is done automatically by the design at each UART message transmission (message = SOF, TYPE … EOF). Type\_reg* *value will be set to the TYPE of the UART message*.

In order to write to register, the following UART message should be sent:

* SOF (=0x64)
* TYPE, where the MSB is '1'. (i.e: 0x80 = write to registers)
* ADDRESS, which is the register's address. (i.e: 0x0002 for dbg\_reg)
* LENGTH, which is the burst length minus 1. (i.e: 0x0002 for dbg\_reg, which is 3 bytes wide)
* PAYLOAD, which is the register's value to be written. (i.e: for dbg\_reg, PAYLOAD might be 0x[01 FA 00] )
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

This will generate Wishbone Write transaction to the relevant register.

### Read Registers

In order to read value from the SDRAM, a UART write transmission to *reg\_addr* (#12 = 0xC in the table above) register should be written, with the register's address to be read, according to the table above.

The following UART messages should be sent:

Message #1 – Register's address

* SOF (=0x64)
* TYPE, where the MSB is '1'. (i.e: 0x80 = write to registers)
* ADDRESS of the *reg\_addr* register (0x000C)
* LENGTH, which is the burst length minus 1. In this case – 0 (burst of 1 byte)
* PAYLOAD, which is the register's address to be read. (i.e: 0x07 – top frame register)
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

Message #2 – Burst length

* SOF (=0x64)
* TYPE, where the MSB is '1'. (i.e: 0x80 = write to registers)
* ADDRESS of the *rd\_burst\_len\_reg* register (0x0009).
* LENGTH, which is the burst length minus 1. In this case – 0 (burst of 1 byte).
* PAYLOAD, which is the register's length to be read. (i.e: for left frame register, length is 0, which represents 1 byte)
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

Message #3 – Execute read command

* SOF (=0x64)
* TYPE, where the MSB is '1'. (i.e: 0x80 = write to registers)
* ADDRESS of the *dbg\_cmd\_reg* register (0x000B).
* LENGTH, which is the burst length minus 1. In this case – 0 (burst of 1 byte).
* PAYLOAD, which should command to start the read sequence. In this case: 0x1.
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

## SDRAM Access at Debug Mode

### Write to SDRAM in Debug Mode

In order to write to SDRAM to specific address, a UART write transmission to *dbg\_reg* (#2🡪4 in the table above) register should be written, with the SDRAM's address to be read.

The following UART messages should be sent:

Message #1 – Debug Register address

* SOF (=0x64)
* TYPE, where the MSB is '1'. (i.e: 0x80 = write to registers)
* ADDRESS of the *dbg\_reg* register (0x0002)
* LENGTH, which is the burst length minus 1. In this case – 2 (burst of 3 bytes = 24 bit address)
* PAYLOAD, which is the register's address to be read. (i.e: 0x[00 00 00])
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

Message #2 – Data to SDRAM

* SOF (=0x64)
* TYPE, where the MSB is '0' and LSB is '1'. (i.e: 0x1 = write to SDRAM in DBG Mode)
* ADDRESS should be 0x0000.
* LENGTH, which is the burst length minus 1. For example, for 512 bytes burst, 0x1FF (511 decimal) should be used..
* PAYLOAD, which is the SDRAM data. Payload size should match the LENGTH.
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

### Read SDRAM in Debug Mode

In order to read SDRAM value, a UART write transmission to *dbg\_reg* (#2🡪4 in the table above) register should be written, with the SDRAM's address to be read.

The following UART messages should be sent:

Message #1 – Debug Register address

* SOF (=0x64)
* TYPE, where the MSB is '1'. (i.e: 0x80 = write to registers)
* ADDRESS of the *dbg\_reg* register (0x0002)
* LENGTH, which is the burst length minus 1. In this case – 2 (burst of 3 bytes = 24 bit address)
* PAYLOAD, which is the register's address to be read. (i.e: 0x[00 00 00])
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

Message #2 – Burst length

* SOF (=0x64)
* TYPE, where the MSB is '1'. (i.e: 0x80 = write to registers)
* ADDRESS of the *rd\_burst\_len\_reg* register (0x0009).
* LENGTH, which is the burst length minus 1. In this case – 0 (burst of 1 byte).
* PAYLOAD, which is the SDRAM bytes to be read. (i.e: 0x10)
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)

Message #3 – Execute read command

* SOF (=0x64)
* TYPE, where the MSB and LSB are '1'. (i.e: 0x81 = Write to Registers + Debug Mode)
* ADDRESS of the *dbg\_cmd\_reg* register (0x000B).
* LENGTH, which is the burst length minus 1. In this case – 0 (burst of 1 byte).
* PAYLOAD, which should command to start the read sequence. In this case: 0x1.
* CRC, which is the checksum for TYPE🡪PAYLOAD (inclusive)
* EOF (=0xC8)