**Technion**

*Electrical Engineering Department*

High Speed Digital System Lab

Symbol Generator

Project

- Part A -

**Version**: 1

**Last update**: 01.12.2012

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

**Supervised By**: Moshe Porian

Table of Changes

|  |  |  |
| --- | --- | --- |
| Version | Date of Change | Description |
| 1.0 |  | Creation of document |

Table of content

[1 Introduction 7](#_Toc342151644)

[1.1 Abstract 7](#_Toc342151645)

[1.2 Applications 7](#_Toc342151646)

[2 Goals 7](#_Toc342151647)

[3 Requirements 8](#_Toc342151648)

[4 Algorithm 9](#_Toc342151649)

[4.1 Overview 9](#_Toc342151650)

[4.2 Video Frame and Symbols 9](#_Toc342151651)

[4.3 Interaction with SDRAM 10](#_Toc342151652)

[4.4 Opcode Structure 11](#_Toc342151653)

[4.5 Inside Symbol Generator 11](#_Toc342151654)

[4.5.1 RAM 11](#_Toc342151655)

[4.5.2 Toggle-FIFOs 12](#_Toc342151656)

[4.6 Combining All Together 15](#_Toc342151657)

[5 General Description 16](#_Toc342151658)

[5.1 Project's Top Blocks Scheme 16](#_Toc342151659)

[5.2 Block Descriptions 17](#_Toc342151660)

[5.2.1 RX Path 17](#_Toc342151661)

[5.2.2 TX Path 18](#_Toc342151662)

[5.2.3 Memory Management 18](#_Toc342151663)

[5.2.4 Display Controller 19](#_Toc342151664)

[5.2.5 Wishbone Interconnector 19](#_Toc342151665)

[5.3 Data flow 20](#_Toc342151666)

[6 Implementation 21](#_Toc342151667)

[6.1 General Top Architecture 21](#_Toc342151668)

[6.1.1 Top Architecture Diagram 21](#_Toc342151669)

[6.1.2 Detailed Data flow Description 22](#_Toc342151670)

[6.2 Detailed Top Architecture 23](#_Toc342151671)

[6.2.1 Opcode Unite 23](#_Toc342151672)

[6.2.2 Opcode Store 25](#_Toc342151673)

[6.2.3 RAM 27](#_Toc342151674)

[6.2.4 Manager 28](#_Toc342151675)

[6.2.5 FIFO A\B 32](#_Toc342151676)

[6.2.6 MUX 34](#_Toc342151677)

[7 Testing 36](#_Toc342151678)

[7.1 Testing components separately 36](#_Toc342151679)

[7.2 Top TB for the Symbol Generator block 37](#_Toc342151680)

[7.3 GUI 38](#_Toc342151681)

[7.3.1 Goal 38](#_Toc342151682)

[7.3.2 Snapshot 38](#_Toc342151683)

[7.3.3 Description 38](#_Toc342151684)

[7.4 External environment for Testing 39](#_Toc342151685)

[7.5 Simulation Models 41](#_Toc342151686)

[7.5.1 SDRAM Model 41](#_Toc342151687)

[7.5.2 Opcode Parser 43](#_Toc342151688)

[7.5.3 Log File 45](#_Toc342151689)

[7.6 Test Plan 47](#_Toc342151690)

[7.7 Automatic Test Manager 49](#_Toc342151691)

[7.7.1 Overview 49](#_Toc342151692)

[7.7.2 Implementation 49](#_Toc342151693)

[7.7.3 Example: Running Test 51](#_Toc342151694)

[8 Working Methods 52](#_Toc342151695)

[8.1 Top down design 52](#_Toc342151696)

[8.2 Testing each component separately 52](#_Toc342151697)

[8.3 SVN 52](#_Toc342151698)

[8.4 Documentation (DOC) 54](#_Toc342151699)

[8.5 Coding Guidelines 54](#_Toc342151700)

[9 Summary 55](#_Toc342151701)

[9.1 Problems and Solutions 55](#_Toc342151702)

[9.2 Coding conclusions 55](#_Toc342151703)

[9.3 Project Conclusions 56](#_Toc342151704)

[10 Abbreviations 57](#_Toc342151705)

Tables of Figures

[Figure 1 - Application example 7](#_Toc342151706)

[Figure 2 - Divided video frame 9](#_Toc342151707)

[Figure 3 - Symbol image 10](#_Toc342151708)

[Figure 4 - Symbols stored in SDRAM 10](#_Toc342151709)

[Figure 5 - Opcode structure 11](#_Toc342151710)

[Figure 6 - The symbol blocks in the frame 12](#_Toc342151711)

[Figure 7 - RAM inside Symbol Generator block 12](#_Toc342151712)

[Figure 8 - Data stored in FIFO 13](#_Toc342151713)

[Figure 9 - Data as represented on the display 13](#_Toc342151714)

[Figure 10 - Toggling between FIFOs 14](#_Toc342151715)

[Figure 11 - Relation of all the components together 15](file:///C:\Users\Olga\Dropbox\Yoav%20-%20Olga\project\Symbol_Generator_Project_Document_olga.docx#_Toc342151716)

[Figure 12 - Top Blocks Scheme 16](#_Toc342151717)

[Figure 13 - Top Blocks Scheme with Symbol Generator 17](file:///C:\Users\Olga\Dropbox\Yoav%20-%20Olga\project\Symbol_Generator_Project_Document_olga.docx#_Toc342151718)

[Figure 14 - RX Path block 17](#_Toc342151719)

[Figure 15 - TX path block 18](#_Toc342151720)

[Figure 16 - Memory Management block 18](#_Toc342151721)

[Figure 17 - Display Controller block 19](#_Toc342151722)

[Figure 18: Symbol Generator Top 21](#_Toc342151723)

[Figure 19 - Opcode Unite block diagram 23](#_Toc342151724)

[Figure 20 - Opcode Unite FSM 23](#_Toc342151725)

[Figure 21 - Opcode Unite simulation 24](file:///C:\Users\Olga\Dropbox\Yoav%20-%20Olga\project\Symbol_Generator_Project_Document_olga.docx#_Toc342151726)

[Figure 22 - Opcode Store block diagram 25](#_Toc342151727)

[Figure 23 - Opcode Store block diagram 26](#_Toc342151728)

[Figure 24 - RAM block diagram 27](#_Toc342151729)

[Figure 25 – Manager block diagram 28](#_Toc342151730)

[Figure 26 - Manager FSM 30](#_Toc342151731)

[Figure 27 - Manager simulation 32](#_Toc342151732)

[Figure 28 - FIFO block diagram 32](#_Toc342151733)

[Figure 29 - FIFOs simulation 34](#_Toc342151734)

[Figure 30 - MUX block diagram 34](#_Toc342151735)

[Figure 31 - Testing components diagram 36](file:///C:\Users\Olga\Dropbox\Yoav%20-%20Olga\project\Symbol_Generator_Project_Document_olga.docx#_Toc342151736)

[Figure 32 - Top TB for Symbol Generator 37](#_Toc342151737)

[Figure 33 - GUI snapshot 38](#_Toc342151738)

[Figure 34 – Full testing environment 39](#_Toc342151739)

[Figure 35 - SDRAM Model interfaces 41](#_Toc342151740)

[Figure 36 - Example of the memory text file 42](#_Toc342151741)

[Figure 37 - Opcode Parser interfaces 43](#_Toc342151742)

[Figure 38 - Example of the simulation text file 44](file:///C:\Users\Olga\Dropbox\Yoav%20-%20Olga\project\Symbol_Generator_Project_Document_olga.docx#_Toc342151743)

[Figure 39 - Log File block in the simulation process 45](file:///C:\Users\Olga\Dropbox\Yoav%20-%20Olga\project\Symbol_Generator_Project_Document_olga.docx#_Toc342151744)

[Figure 40 - Log File interfaces 45](#_Toc342151745)

[Figure 41 - Example of log files 46](#_Toc342151746)

[Figure 42 - Test plan example 47](#_Toc342151747)

[Figure 43 - output frames from the simulation process 48](#_Toc342151748)

[Figure 44 - Test directory structure 49](#_Toc342151749)

[Figure 45 - Example for a test log 50](#_Toc342151750)

[Figure 46 - running run\_tests.do script 50](#_Toc342151751)

[Figure 47 - Example of tests.do file 50](#_Toc342151752)

[Figure 48 - Example for a log file 51](file:///C:\Users\Olga\Dropbox\Yoav%20-%20Olga\project\Symbol_Generator_Project_Document_olga.docx#_Toc342151753)

[Figure 49 - SVN snapshot 52](#_Toc342151754)

[Figure 50 - SVN project structure 53](#_Toc342151755)

[Figure 51 - Clock domain crossing 55](#_Toc342151756)

# 

# Introduction

## Abstract

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

## Applications

Sending symbols on display is an integrated operation which is used in various applications where:

1. The communication bandwidthis limited.
2. The hardware size and space is limited.
3. The memory capacityis limited.

Systems who deal with these kinds of limitations, such as mobile phones, televisions and computer screens, need to take this into account and provide sufficient solutions.



Figure 1 - Application example

# Goals

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), all having in common the same dimensions, and are generated on the screen.

The user chooses the appearance of the frame on the screen: where to place symbols, from where to delete them or both. For this purpose, he uses a graphic interface implemented in Matlab.

The purpose is to save time, resources and bandwidth. Therefore, the program transfers the HW only the change of the wanted frame from the current one, using a command for the HW, instead of sending the whole frame.

# Requirements

1. Transmit 640x480 BMP grayscale picture, using Matlab, to the DE2 board.
2. The frame is divided into 20x15 blocks (row x column), the blocks are aligned and do not run over each other.
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. Two FIFOs shall be implemented (FIFO\_A and FIFO\_B), each FIFO size: 640 x 8 bits (row x column). One FIFO will load a video row from the SDRAM, while the other FIFO will transmit the previous video row to the VESA.
6. Data will be sent from host to the DE2 board using UART 115,200 bits/sec.
7. Internal communication via Wishbone protocol
8. UART protocol:
   1. Line not active = '1'
   2. Parity bit will not be used, since checksum is being calculated
   3. 8 bits will be wrapped by *start bit*, represented by '0', and *stop bit*, represented by '1'.

Message Pack Structure:

* 1. SOF – Start of Frame
  2. Type – Message Type (Data, Debug…)
  3. Address – registers address
  4. Data (payload) Length – number of bytes – 1 in data block
  5. Data (payload)
  6. CRC (Checksum will be used instead CRC)
  7. EOF – End of Frame

Main types ('b' in the Message Pack Structure):

* 1. Last Packet Summary (End of packet) – contains the number of bytes, which had been transmitted for this picture (Size of compressed image in bytes)
  2. Reading / Writing registers
  3. Reading and writing from SDRAM for debug
  4. More types will be added later, in case such are required

1. 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.
2. Clocks in systems:
   1. 40 MHz for VESA
   2. 100MHz Main clock

# 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.

The solution to these issues includes:

* Working with the video display and VESA protocol.
* Transferring only the changes in the display to the FPGA.
* Understanding the relations between the external memory (SDRAM) and Symbol Generator block.
* Understanding the relations between Symbol Generator block and the display with VESA protocol.

Detailed description is followed in the next sections of this chapter.

## 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:

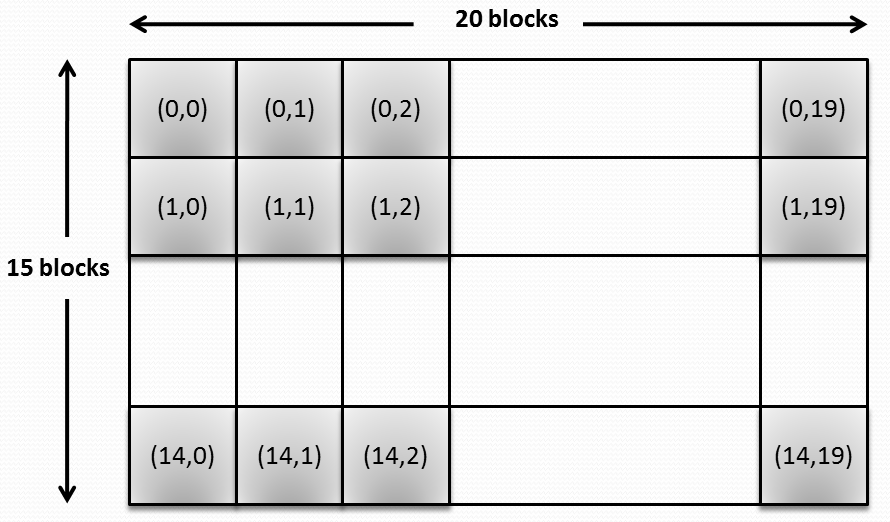


Figure 2 - Divided video frame

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 following work will be in terms blocks in the frame. It means that the lowest resolution for changes in the display is now in terms of blocks. For example: data inside the blocks can be changed, blocks can be removed and etc.

## 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:

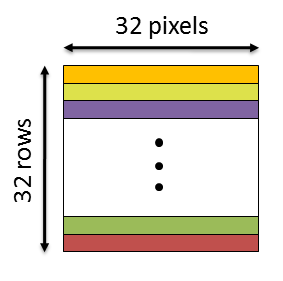


Figure 3 - Symbol image

The symbols stored in the SDRAM:

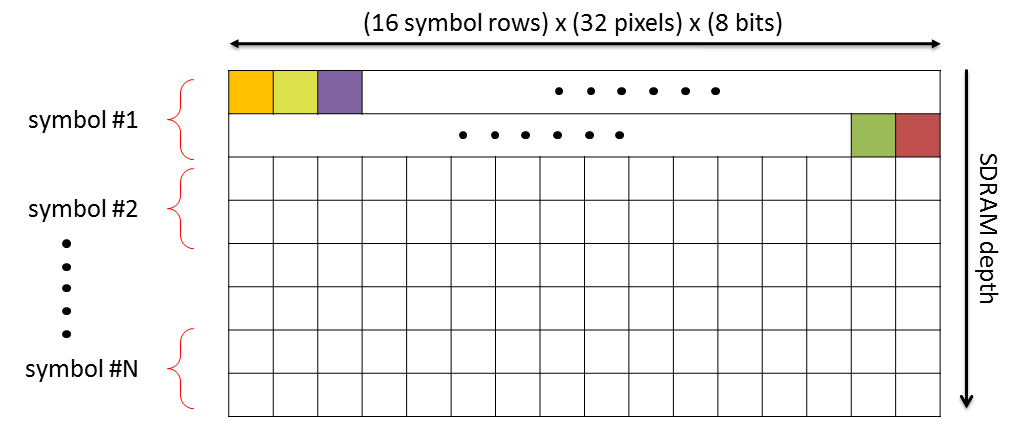


Figure 4 - Symbols stored in SDRAM

## 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:

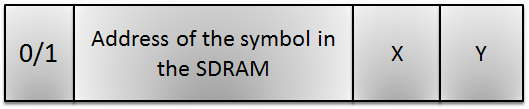


Figure 5 - Opcode structure

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

### 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, as described in section 4.2).

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:

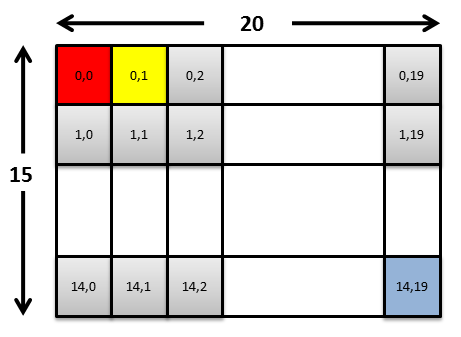


Figure 6 - The symbol blocks in the frame

The RAM inside Symbol Generator block:

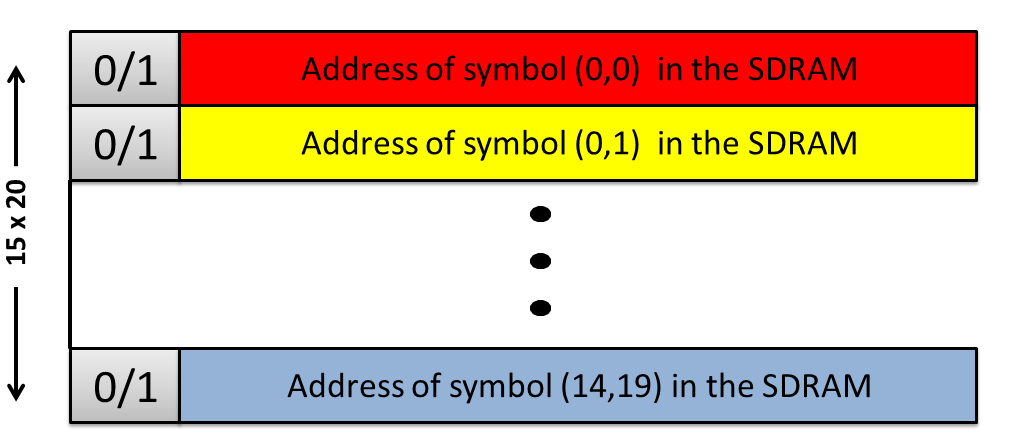


Figure 7 - RAM inside Symbol Generator block

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.

### Toggle-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:

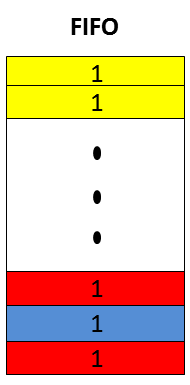


Figure 8 - Data stored in FIFO

This data as represented on the display:

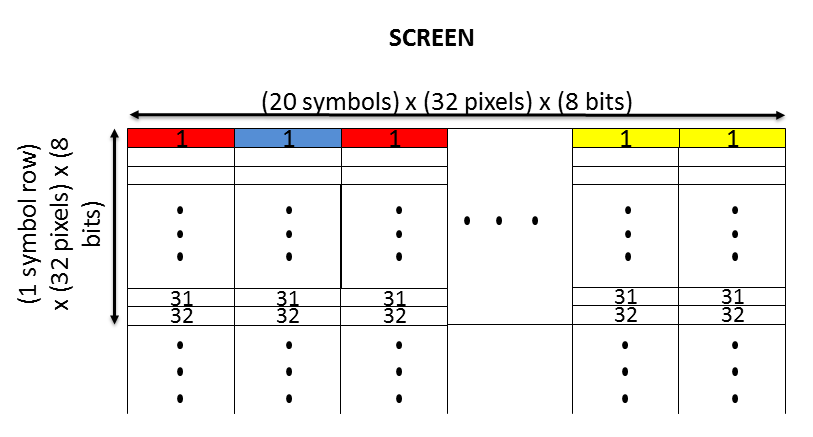


Figure 9 - Data as represented on the display

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:

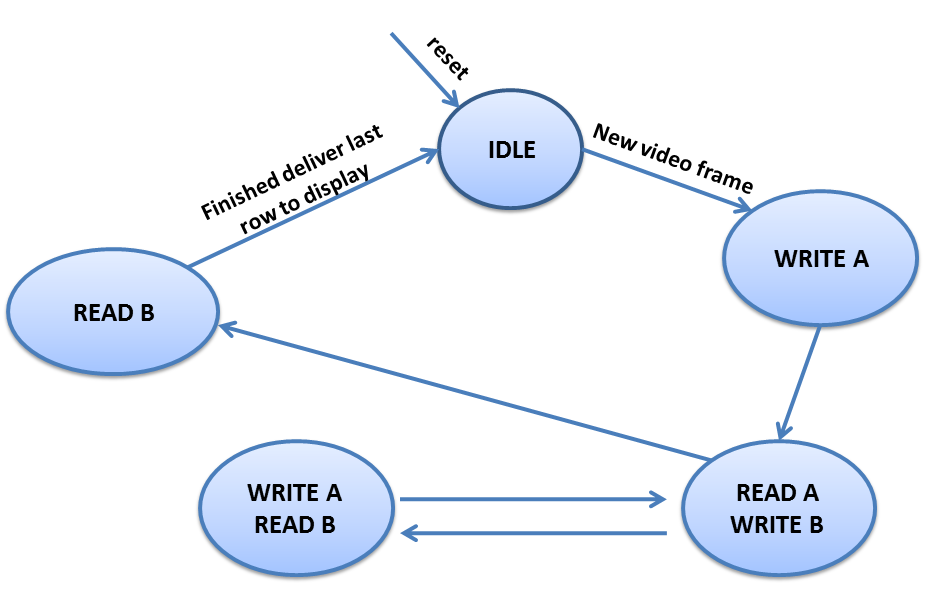


Figure 10 - Toggling between FIFOs

## Combining All Together

Visualization of the relation of all the components together – SDRAM, RAM, FIFOs and display:

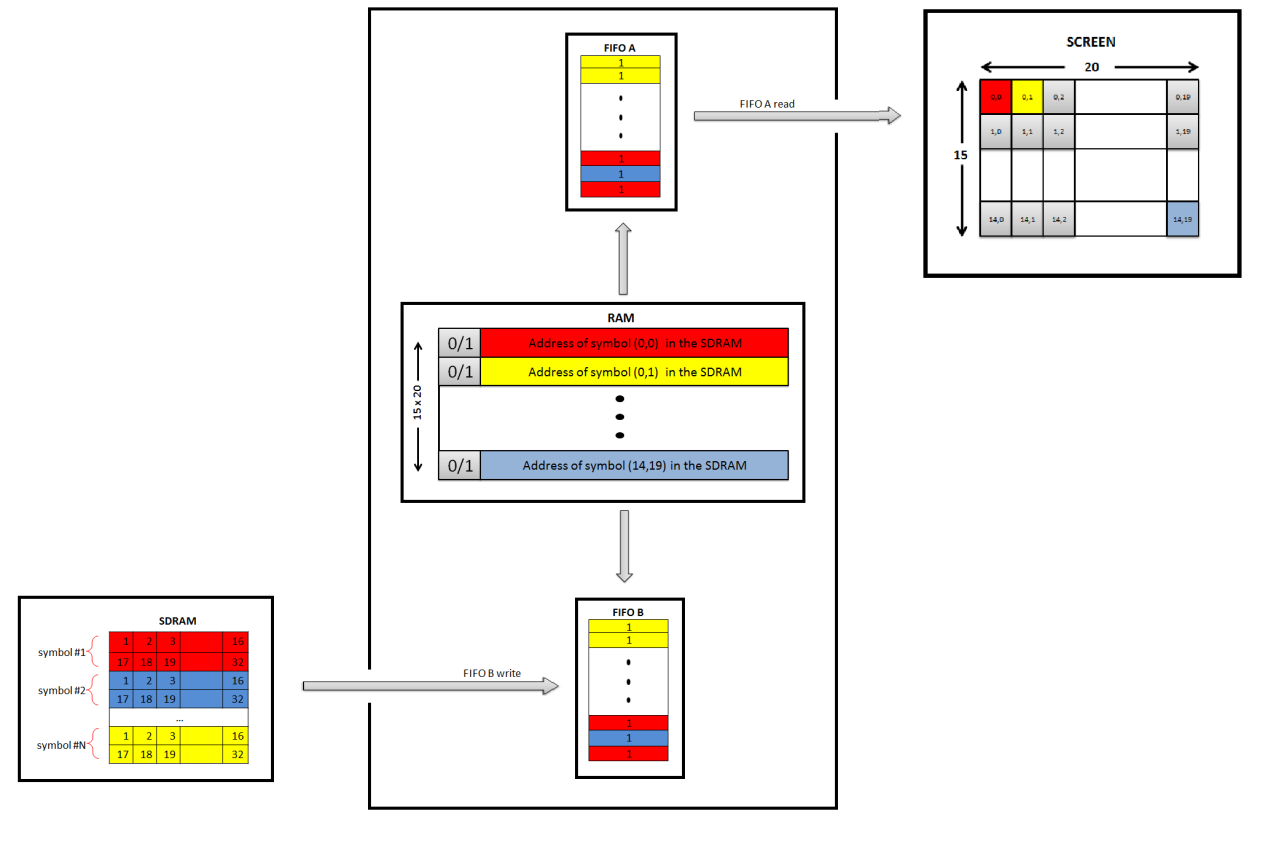


Figure 11 - Relation of all the components together

In the figure above, FIFO A stores the data for the current row in the display and FIFO B prepares the data for the next row.

# General Description

## Project's Top Blocks Scheme

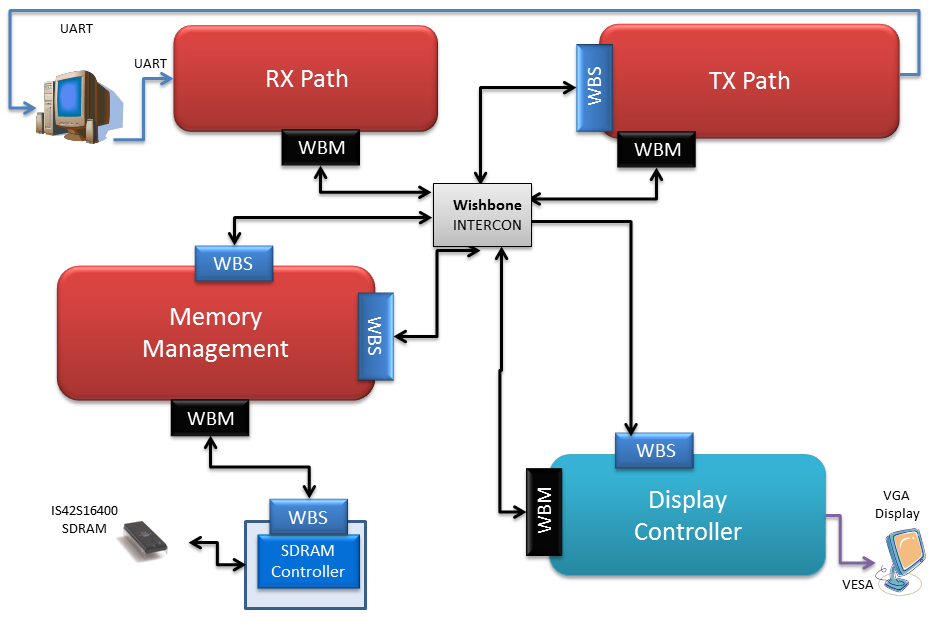
****

Figure 12 - Top Blocks Scheme

SG project uses a previous one as a platform environment. The following blocks were reused: Rx Path, Tx Path, Memory Management, Wishbone Intercon, Display Controller and SDRAM controller.

SG project uses bitmap images, while the platform supported Run-Length coding. Hence, the following changes in the platform had to be made:

* Removing the Run-Length extractor
* Creating a new block – Symbol Generator

Project's top block scheme, including Symbol Generator in the Display Controller block:

Figure 13 - Top Blocks Scheme with Symbol Generator

## 

## Block Descriptions

### RX Path

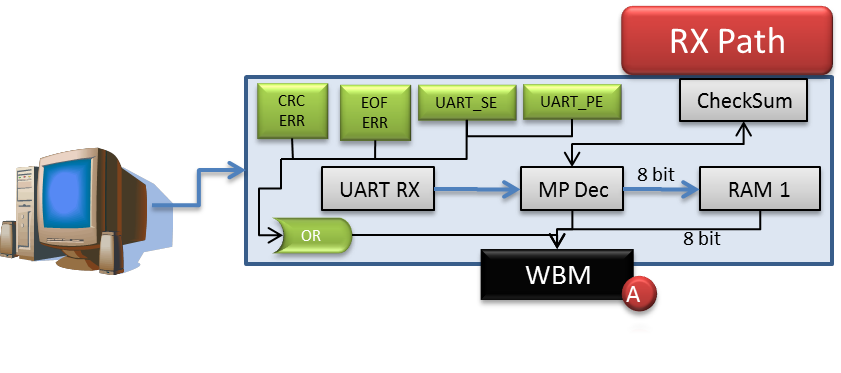


Figure 14 - RX Path block

This component is used for asynchronous serial data channel.

When a transmission is being received to the RX block, it is being unwrapped: the receiver converts serial start bit, data, parity and stop bit to parallel data.

Write transaction is being initialized to the required blocks when specific terms are satisfied, detailed in the "Modular Decompression System" document.

The data word length can be 5-8 bits, according to generic parameter. Parity bit can be, using appropriate generic parameters odd, even or can inhibited.

All inputs and outputs are synchronized with the positive edge of the clock.

Any system clock and any baud rate are supported, according to generic parameter.

These generics are specified in "Modular Decompression System" documentation.

### TX Path

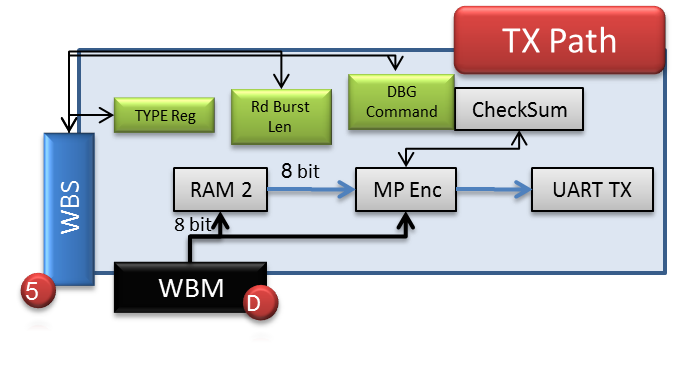


Figure 15 - TX path block

The TX Block transmits data for debug purposes, using UART protocol, to the host (Matlab, which is running on a PC). It converts parallel data into serial form and automatically adds start bit, parity and stop bit.

See detailed information in "Modular Decompression System" document.

### Memory Management

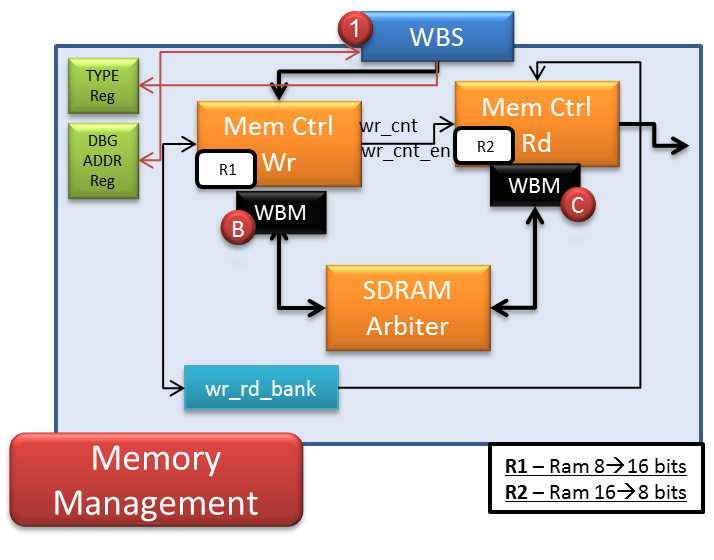


Figure 16 - Memory Management block

The Memory Management block manages the write and read transaction to and from the SDRAM, through the SDRAM Controller. See detailed information in "Memory Management" documentation.

### Display Controller

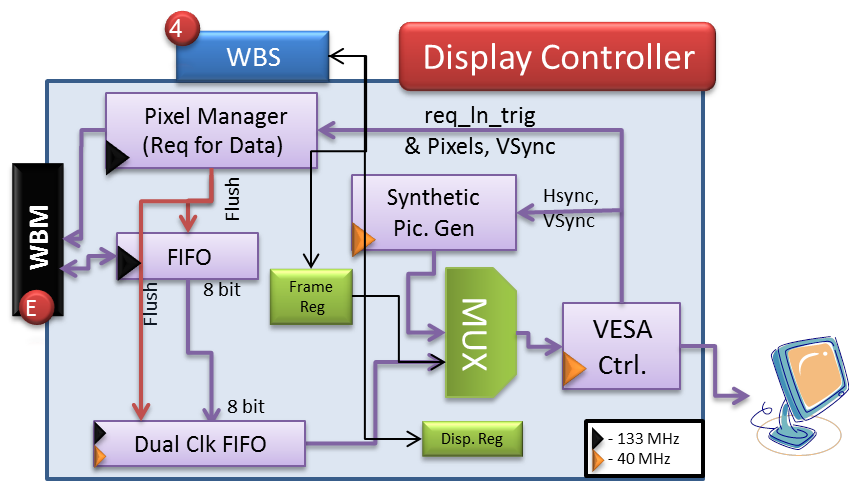


Figure 17 - Display Controller block

The Display Controller block transmits the data from the SDRAM, through the DAC on the DE2, to the VGA.

The FIFO's depth is 256X2X3 = 1536 (X8 bits), for 3 SDRAM read burst.

The Dual Clock FIFO's depth is 640X6 = 3,840 (X8 bits), for 6 lines.

The Symbol Generator block is implemented inside the Display Controller.

### Wishbone Interconnector

Three wishbone interconnectors are being operated in the system:

1. **Intercon X -** According to WBM (D) command (TGC\_I), the interconnector routes the path toward reading from SDRAM in the Memory Block or from the registers, through INTERCON (Z).
2. **Intercon Y -** Receives the TYPE register, which is transmitted from the WBM (A). In case of debug mode, WBM (D) will be connected to the path. In case of normal mode, WBM (E) will be connected to the path.

* **Intercon Z** - Two Wishbone Masters are connected to this interconnection:

1. WBM (A) from RX Path, which responsible to write data (SDRAM / Registers) to Wishbone Slaves.
2. WBM (D) from TX Path, which responsible to read data (SDRAM / Registers) from Wishbone Slaves.

For more information use "Modular Decompression System" document.

## Data flow

**Initialization:**

**Host** transmits wrapped message of data packets composed of the symbol images to the **RX Path**. Message is decoded (from the SOF, CRC and EOF) by **Message Pack Decoder** and validation **Checksum** and correctness of message length are being held. Message is transmitted to the **Memory Management** block, and stored into **Storage Device**. From there data is written to the SDRAM.

**Consecutive use:**

**User** chooses the desired frame display, symbols & locations, using GUI platform. **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. Unwrapped message is transmitted to **Symbol Generator**, which decode the message. Relevant symbols are read from **SDRAM** and sent to **VESA Generator** and to the display.

Detailed information can be found in paragraph 6.1.2.

**Remark:** Status and debug signals can be sent through the **TX Path** to the **Host**.

# Implementation

This chapter contains general and detailed information about the implementation of the components inside the Symbol Generator block, which is a component in the Display Controller block.

The Symbol Generator block receives from the user, the changes of the display, and implies them on the next showed display on the screen.

## General Top Architecture

The Symbol Generator block is responsible for the following functions:

1. Processing the received opcodes
2. Relevant calculations for showing the video on the display
3. Reading data from external memory
4. Passing the data to the VESA Controller block

### Top Architecture Diagram

The following diagram describes the top architecture of the Symbol Generator block, the interfaces to the Display Controller block and its building blocks:

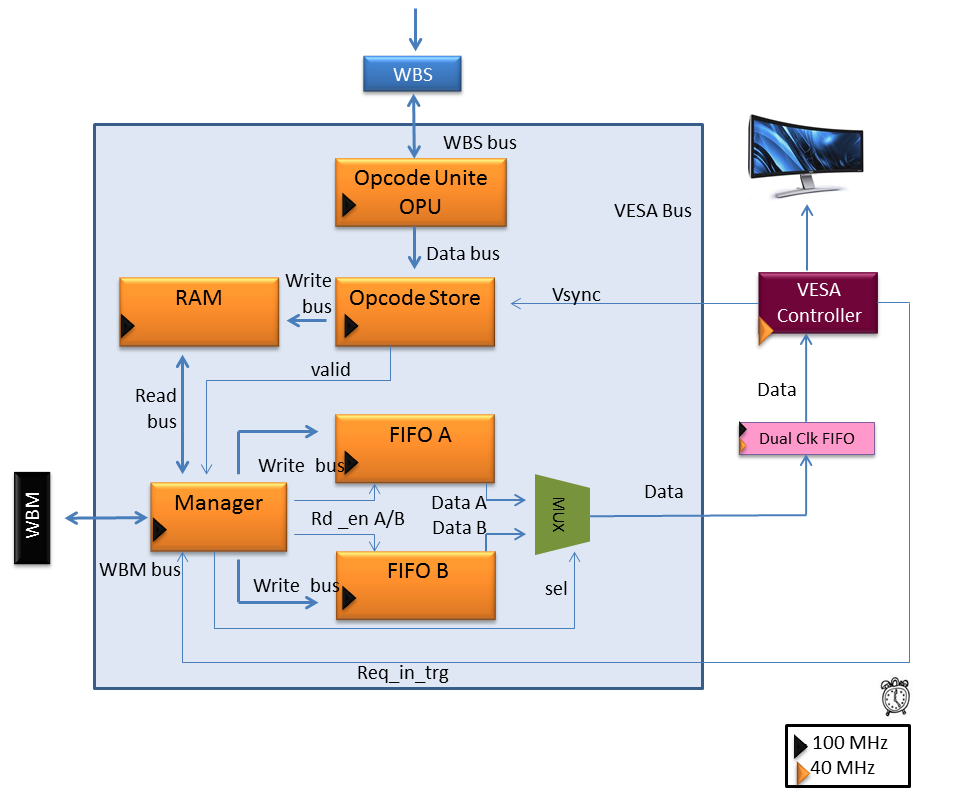


Figure 18: Symbol Generator Top

The mentioned functions are achieved using the IFs with the "outside world":

1. WBS – Wishbone Slave IF  
   This interface is responsible to receive the opcodes from the user, and transfer them to the Symbol Generator block.
2. WBM – Wishbone Master IF  
   Through this interface, the Symbol Generator block initiates read transactions to the external memory (SDRAM).
3. VESA IF  
   The Symbol Generator block interfaces with the Dual Clock FIFO and the VESA Controller blocks, in order to pass the video data to the display.

### Detailed Data flow Description

The encoded changes in the display, which are the opcodes, are sent to the WBS (Wishbone slave) interface. Symbol Generator block receives the opcodes in packets, and needs to unite 3 packets of opcode to get a full opcode, which consists of 3 bytes. This process happens in the **Opcode Unite** block. Then, the opcodes are stored in the **Opcode Store** block until the next video frame. At the beginning of each video frame, these opcodes are transferred to the **RAM** block.

The **Manager** block is responsible for the calculations of the coordinates on the display. It counts the rows, columns and the symbols. The information about what symbol should be presented on which coordinate on the screen, it retrieves from the **RAM** that stores this information.

The **Manager** also implements the toggling between the **FIFOs** for read and write data. On each video row, data is read from one FIFO, while the other one prepares data for the next row. The data that is read from one of the FIFOs, is passed to a **MUX**, and then to the VESA Controller block, which is responsible for the video that is displayed.

## Detailed Top Architecture

All the implemented blocks detailed blow are packed in SG\_TOP block. The block is used as the interface to the Display Controller entity.

### Opcode Unite

#### Block Diagram

Figure 19 - Opcode Unite block diagram

#### Description

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

The implemented FSM to unite opcodes:

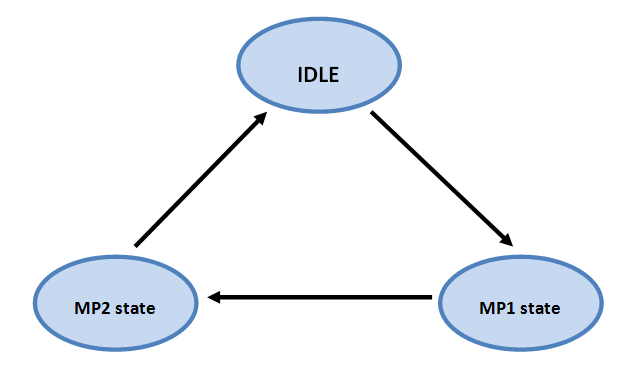


Figure 20 - Opcode Unite FSM

The states of the presented FSM also appear in the following wave diagram in paragraph 6.1.

#### Inputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Received from** | **description** |
| Clk | std\_logic | SG\_TOP | the main clock to which all the internal logic of the Symbol Generator block is synchronized |
| Reset\_n | std\_logic | SG\_TOP | asynchronous reset |
| opu\_data\_in | std\_logic\_vector(7:0) | SG\_TOP | Packets of data from WBS |
| opu\_data\_in\_valid | std\_logic | SG\_TOP | Signal that indicates when data is valid from WBS |

#### Outputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Linked to** | **description** |
| opu\_wr\_en | std\_logic | Opcode Store | This signal is an enable signal to write opcodes to the Opcode Store block. It is active when a new opcode was united and is ready to be stored in the FIFO. |
| opu\_data\_out | std\_logic\_vector(23:0) | Opcode Store | The data signal is the united opcode to be stored in the Opcode store block. |

#### Simulations

Data packets arriving with valid signal

United opcode is ready when valid signal is high

Uniting data packets with FSM and counter

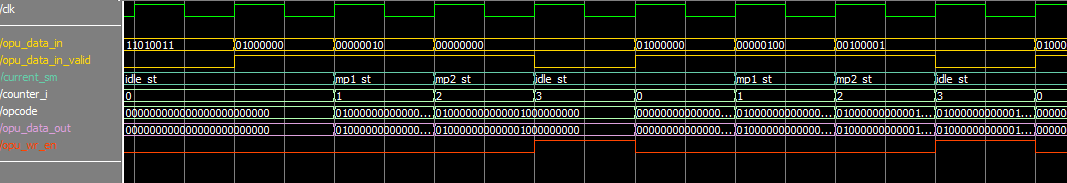


Figure 21 - Opcode Unite simulation

### Opcode Store

#### Block Diagram

Figure 22 - Opcode Store block diagram

#### Description

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.

#### Inputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Received from** | **description** |
| clk | std\_logic | SG\_TOP | Wishbone Clock (100MHz) |
| reset\_n | std\_logic | SG\_TOP | asynchronous reset |
| op\_cnt | std\_logic\_vector(9:0) | Opcode\_Unit | number of total changes X 3:  1 change (24 bits) = add/remove 1 symbol ( 24 bits are being sent in 3 packs of 8 bits) |
| op\_str\_valid | std\_logic | opcode\_unite | Enable writing to opcode\_store FIFO |
| op\_str\_data\_in | std\_logic\_vector(23:0) | opcode\_unite | Data input |
| op\_str\_rd\_start | std\_logic | SG\_TOP | Vsync signals starting of opcode\_store operation |

#### Outputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Size(bit)** | **Linked to** | **description** |
| ram\_addr\_wr | std\_logic\_vector(8:0) | RAM | ram's address to be written into |
| ram\_wr\_en | std\_logic | RAM | ram write enable |
| ram\_data | std\_logic\_vector(12:0) | RAM | data sent to ram |
| mng\_en | std\_logic | Manager | activating Manager |
| op\_str\_empty | std\_logic | SG\_TOP | FIFO is empty (for debug) |
| op\_str\_full | std\_logic | SG\_TOP | FIFO is full (for debug) |
| op\_str\_used | std\_logic\_vector(9:0) | SG\_TOP | current number of elements in FIFO (for debug) |
| vsync\_out | std\_logic | SG\_TOP | VSYNC sampled in 100MHz CLK domain |

#### Simulations

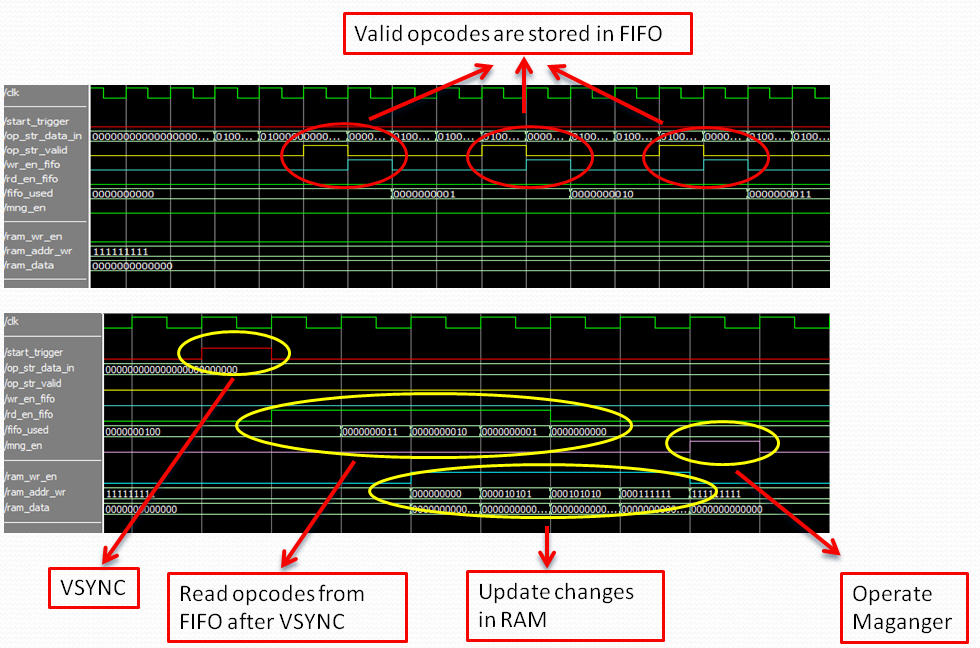


Figure 23 - Opcode Store block diagram

### RAM

#### Block Diagram

Figure 24 - RAM block diagram

#### Description

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. Since the screen is divided to 20X15 blocks referred to their (x,y) location, it was easy to determine that, row 0 in RAM matches symbol in location(0,0), row 1 matches symbol (0,1), …,row 299 matches symbol (14,19). 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.

Details about the functionality of the RAM in the algorithm are found in paragraph 4.5.1.

#### Inputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Received from** | **description** |
| clk | std\_logic | SG\_TOP | Wishbone Clock (100MHz) |
| reset\_n | std\_logic | SG\_TOP | asynchronous reset |
| ram\_addr\_wr | std\_logic\_vector(8:0) | Opcode\_Store | Input address for writing |
| ram\_wr\_en | std\_logic | Opcode\_Store | Enable write to RAM |
| ram\_data\_in | std\_logic\_vector(12:0) | Opcode\_Store | Input data for write |
| ram\_rd\_en | std\_logic | Manager | Enable read from RAM |
| ram\_addr\_rd | std\_logic\_vector(8:0) | Manager | Input address for reading |

#### Outputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Size(bit)** | **Linked to** | **description** |
| ram\_data\_out | std\_logic\_vector(12:0) | Manager | Data output |

### Manager

#### Block Diagram

Figure 25 – Manager block diagram

#### Description

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 mng\_en signal and by VESA controller through 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).

The implementation of the algorithm described in chapter 4 is done using internal counters:

**Sym\_row** = the row in terms of symbols, values: 0,…,14

**Sym\_col** = the col in terms of symbols, values: 0,…,19

**Row** = the row inside the current symbol, values: 0,…,31

The pseudo-code that describes the HW implementation:

For **sym\_row** = 0 to 14

{

For **row** = 0 to 31

{

For **sym\_col** = 0 to 19

{

…

}

}

}

Calculating the relevant row in the RAM:

For a given **sym\_row** and **sym\_col**:

RAM\_ROW <= 20\***sym\_row** + **sym\_col**;

Calculating the read address in the SDRAM:

* If **row**=0…15 then we will use the first row of the symbol in the SDRAM.

Row\_SDRAM<= RAM\_DATA\_out & '0';

Col\_SDRAM <= 16\***row**;

* If **row**=16…31 then we will use the second row of the symbol in the SDRAM.

Row\_SDRAM<= RAM\_DATA\_o&'1';

Col\_SDRAM<= 16\*(**row**-16);

Length of the burst is 16 words (each 16 bits) in both cases.

**Illustration of FSM:**

Figure 26 - Manager FSM

#### Inputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Received from** | **description** |
| clk | std\_logic | SG\_TOP | Wishbone Clock (100MHz) |
| reset\_n | std\_logic | SG\_TOP | asynchronous reset |
| ram\_data\_out | std\_logic\_vector(12:0) | RAM | Data from RAM |
| req\_in\_trg | std\_logic | SG\_TOP | Originally from VESA Generator block. It indicates when to start preparing valid data in the Dual Clk FIFO for a req\_lines\_g lines in advance. (In our case it is 1 line in advance). |
| mng\_en | std\_logic | opcode\_store | Start trigger |
| sdram\_data | std\_logic\_vector (7:0) | SG\_TOP | data from sdram |
| sdram\_data\_valid | std\_logic | SG\_TOP | data from sdram validation |
| fifo\_a\_empty | std\_logic | FIFO\_A | fifo a empty indication |
| fifo\_b\_empty | std\_logic | FIFO\_B | fifo b empty indication |

#### Outputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Size(bit)** | **Linked to** | **description** |
| ram\_rd\_en\_out | std\_logic | RAM | Enable reading RAM. |
| ram\_addr\_rd | std\_logic\_vector(8:0) | RAM | Input address for reading |
| fifo\_a\_rd\_en | std\_logic | FIFO\_A | Enable reading fifo a |
| fifo\_a\_wr\_en | std\_logic | FIFO\_A | Enable writing fifo a |
| fifo\_a\_data\_in | std\_logic\_vector(7:0) | FIFO\_A | data input to fifo a |
| fifo\_b\_rd\_en | std\_logic | FIFO\_B | Enable reading fifo b |
| fifo\_b\_wr\_en | std\_logic | FIFO\_B | Enable writing fifo b |
| fifo\_b\_data\_in | std\_logic\_vector(7:0) | FIFO\_B | data input to fifo b |
| sdram\_addr\_rd | std\_logic\_vector (23:0) | SG\_TOP | this signal is the full address in SDRAM: "00--bank(2)--row(12)--col(8)" |
| sdram\_rd\_en\_out | std\_logic |  | sdram\_addr\_rd validation |

#### Generics

| Generic Parameter | default Value | Description |
| --- | --- | --- |
| sym\_row\_g | 15 | Total symbol row count for the frame |
| sym\_col\_g | 20 | Total symbol column count for the frame |
| inside\_row\_g | 32 | Total row count inside the symbol |
| sdram\_burst\_g | 16 | -- The length of the burst from SDRAM |
| hor\_active\_pixels\_g | 640 | 640 active pixels per line |
| ver\_active\_lines\_g | 480 | 480 active lines |
| hor\_left\_border\_g | 0 | Horizontal Left Border (Pixels) |
| hor\_right\_border\_g | 0 | Horizontal Right Border (Pixels) |
| hor\_back\_porch\_g | 48 | Horizontal Back Porch (Pixels) |
| hor\_front\_porch\_g | 16 | Horizontal Front Porch (Pixels) |
| hor\_sync\_time\_g | 96 | Horizontal Sync Time (Pixels) |
| ver\_top\_border\_g | 0 | Vertical Top Border (Lines) |
| ver\_buttom\_border\_g | 0 | Vertical Bottom Border (Lines) |
| ver\_back\_porch\_g | 31 | Vertical Back Porch (Lines) |
| ver\_front\_porch\_g | 11 | Vertical Front Porch (Lines) |
| ver\_sync\_time\_g | 2 | Vertical Sync Time (Lines) |

#### Simulations

Figure 27 - Manager simulation

### FIFO A\B

#### Block Diagram

Figure 28 - FIFO block diagram

#### Description

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

Details about the functionality of the FIFOs in the algorithm are found in paragraph 4.5.2.

#### Inputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Received from** | **description** |
| clk | std\_logic | SG\_TOP | Wishbone Clock (100MHz) |
| reset\_n | std\_logic | SG\_TOP | asynchronous reset |
| din | std\_logic\_vector (width\_g-1 : 0) | Manager | Input Data |
| wr\_en | std\_logic | Manager | Write Enable |
| rd\_en | std\_logic | MUX | Read Enable (request for data) |

#### Outputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Size(bit)** | **Linked to** | **description** |
| dout | std\_logic\_vector (width\_g-1 : 0) | MUX | Output Data |
| dout\_valid | std\_logic | MUX | Output data is valid |
| empty | std\_logic | Manager | FIFO is empty |
| used | std\_logic\_vector (log\_depth\_g: 0) | Manager | Current number of elements is FIFO of total 2^log\_depth |

#### Generics

| Generic Parameter | default Value | Description |
| --- | --- | --- |
| width\_g | 8 | Width of data |
| log\_depth\_g | 10 | Logarithm of depth\_g (Number of bits to represent depth\_g. 2^10=1024) |

#### Simulations

Figure 29 - FIFOs simulation

### MUX

#### Block Diagram

Figure 30 - MUX block diagram

#### Description

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.

#### Inputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Received from** | **description** |
| clk | std\_logic | SG\_TOP | Wishbone Clock (100MHz) |
| reset\_n | std\_logic | SG\_TOP | asynchronous reset |
| mux\_din\_a | std\_logic\_vector  (width\_g - 1 : 0) | FIFO\_A | data from fifo\_a |
| mux\_din\_b | std\_logic\_vector  (width\_g - 1 : 0) | FIFO\_B | data from fifo\_b |
| rd\_en\_a | std\_logic | Manager | read enable fifo a |
| rd\_en\_b | std\_logic | Manager | read enable fifo b |
| fifo\_a\_dout\_valid | std\_logic | FIFO\_A | FIFO\_A data validation |
| fifo\_b\_dout\_valid | std\_logic | FIFO\_B | FIFO\_B data validation |

#### Outputs

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Size(bit)** | **Linked to** | **description** |
| mux\_dout\_valid | std\_logic | SG\_TOP | MUX data output validation |
| mux\_dout | std\_logic\_vector  (width\_g-1 downto 0) | SG\_TOP | data out to DC FIFO |

#### Generics

| Generic Parameter | default Value | Description |
| --- | --- | --- |
| width\_g | 8 | Width of data |

# Testing

The testing part of the project wasn't a separate stage in the work, but an essential part of the progressing work upon the design of Symbol Generator block. While developing the design of the algorithm, the unit blocks and the whole system, we needed to pass through the testing steps.

The main testing techniques we used during our work:

* Separate testing environment for each component separately
* Top testing environment for the Symbol Generator block with all its unit blocks
* Using GUI as a test-builder
* Models for the external interfaces in the FPGA and outside
* External environment for overall simulation
* Test plan for common and unique test cases
* Automatic test manager for running the tests

Detailed description of these techniques is found in the following chapters in this part of the document.

## Testing components separately

The work upon the unit blocks of Symbol Generator has been done in a sequence of the data flow in the block. After building each unit block, we used a separate TB to check its correctness.

The following diagram describes the steps:

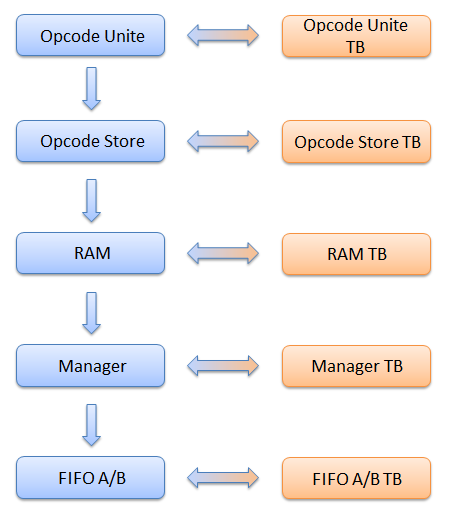


Figure 31 - Testing components diagram

The advantage of testing each block separately is elimination of dependent mistakes in other blocks. This way, we could test and verify each unit, and only when it worked as we wanted it, we could continue to the design of the next block in the data flow sequence.

## Top TB for the Symbol Generator block

After testing each unit block separately, we could build the TB for the whole block together. This environment's main purpose is an integration of all the unit blocks, to see how they work together.

Top TB for the Symbol Generator diagram:

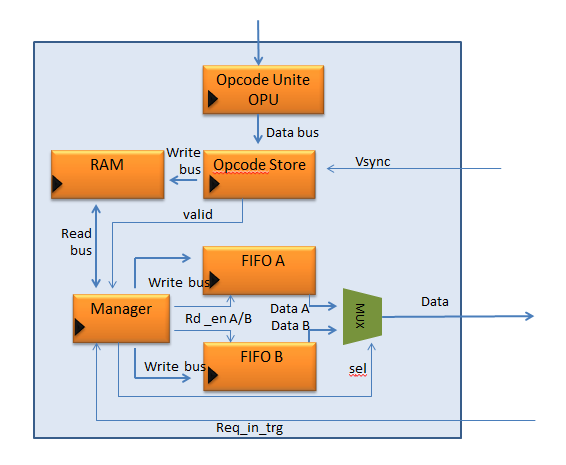


Figure 32 - Top TB for Symbol Generator

This Top TB helped us to get integration between the unit blocks in terms of correctness of the data and timing between the separate blocks working together.

## GUI

### Goal

For examining overall system operation carefully, we have built a GUI platform, used as a test-builder. With it, one can examine Symbol Generator block functionality more easily and efficiently; by creating test files for the use of the External Test Environment (see paragraph 7.4).

### Snapshot

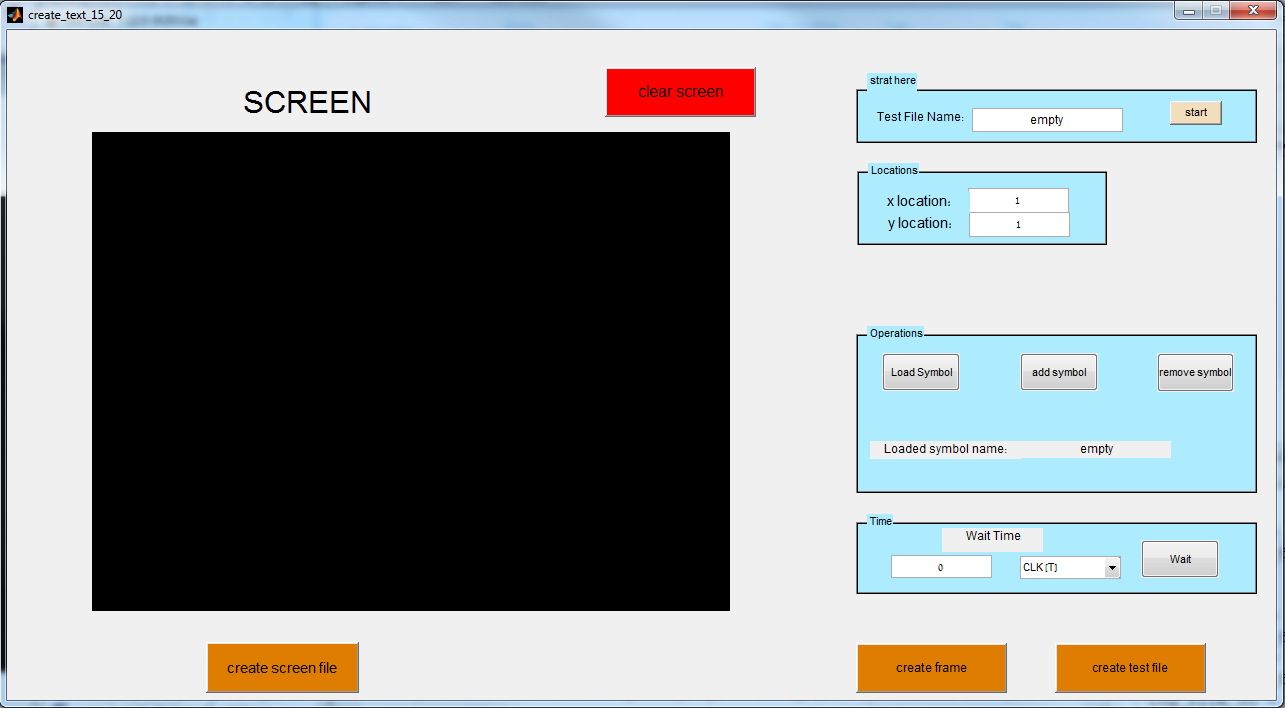


Figure 33 - GUI snapshot

### Description

GUI\_Test is a Graphical User Interface with which the user can build frames with added or removed symbols which are being loaded by the GUI. The user can choose waiting time between opcodes and frames with the 'wait' panel. This may help in examine CLK issues.

The frames with their relevant opcodes and waiting times are written into a test file with a chosen headline.

In addition, the user can print the screen display into another file in any given time, and use the DIFF functionality in the External Test Environment between the desired and received frame display.

Detailed information on how to use GUI\_Test can be found within the GUI Matlab (.m) file.

## External environment for Testing

The next step in the test plan was to create a full test environment that includes our Symbol Generator block, but also the external interfaces to it.

Top Diagram for the full testing environment:

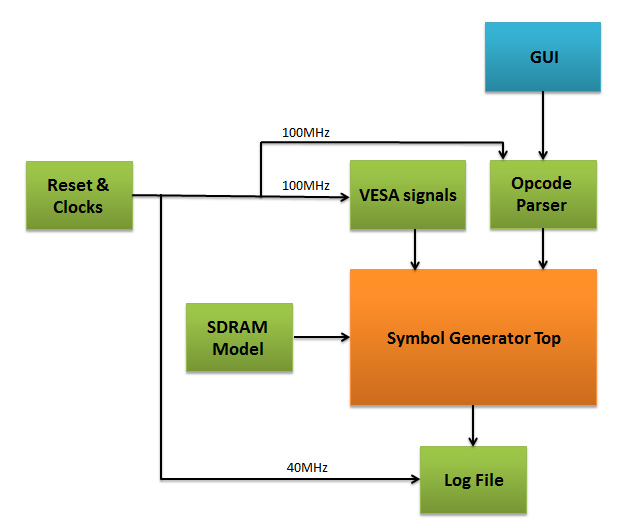


Figure 34 – Full testing environment

Overview of the building blocks in the diagram:

* **Symbol Generator Top**  
  Top block for the design of Symbol Generator. It includes inside all the contents of the block. This is the DUT (=Design Under Test) for the simulation environment.
* **GUI**  
  The SW interface with the HW environment.
* **Reset & Clocks**Simulation block that creates the reset and the required clocks for the simulation blocks and the design block.
* **VESA Signals**This block can be found in the design of the Display Controller of the system we use in this project. Its purpose in the simulation is to create the VESA control signals for the video frame.
* **SDRAM Model**

SDRAM Model is a simulation block that models the functionality of the SDRAM.

* **Opcode Parser**The Opcode Parser is a simulation block for managing the simulation.
* **Log File**

This is a simulation block that saves in a log file the video frames using the VESA control signals.

Detailed description of the simulation building blocks is found in the next chapter.

## Simulation Models

### SDRAM Model

#### Overview

The SDRAM Model is a simulation block. Its purpose is to model the functionality of the SDRAM in the testing process of the Symbol Generator.

#### Interfaces

The interfaces of the SDRAM Model block with the Symbol Generator Top block:

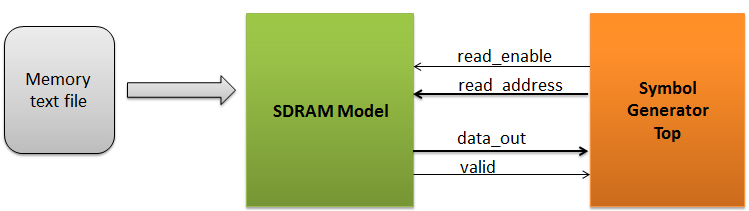


Figure 35 - SDRAM Model interfaces

The communication between the SDRAM Model and Symbol Generator Top is implemented with a request-acknowledge protocol:

* The Symbol Generator Top blocks requests data from the SDRAM Model with an address for the read.
* The SDRAM Model answers the request with 32 bytes of data and a signal that indicates the validity of the data.

#### Implementation

The SDRAM Model uses a text file that models the memory.

The memory text file has the following block structure:

address=<address>

<pixel 1: 8 bits>

<pixel 2: 8 bits>

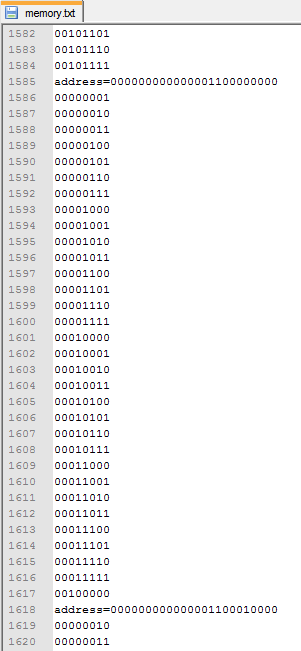
...

...

...

<pixel 32: 8 bits>

Example for the memory text file:



Address in SDRAM

Stored data for this address

Figure 36 - Example of the memory text file

### Opcode Parser

#### Overview

The Opcode Parser is a simulation block for managing the simulation. It uses a text file for controlling the simulation, and transfers the opcodes to be displayed to the Symbol Generator Top with the required timings.

#### Interfaces

The interfaces of the Opcode Parser block with the Symbol Generator Top block:

#### 

Figure 37 - Opcode Parser interfaces

The Opcode Parser transfers to the Symbol Generator Top the opcodes with validity indication.

#### Implementation

The implementation is based on a "language" that we created for controlling the simulation.

The Opcode Parser supports the following commands:

* -f opcode count iiii

-f means a start of a new frame

i = integer that indicates number of following opcodes in this frame (the number is in bytes, means num. of opcodes X 3)

* -p bbbbbbbb

-p means opcode field

b = binary (0/1), this field is an opcode byte (start with MSB part of opcode)

* -t wait T iiii

-t means time field

i = integer that indicates the num. of clock cycles to wait

* -e

Flag at the end of the text file, to show that the text file is finished correctly

* comment

Each line that starts with "--" is skipped from the parser model

Example of a text file:

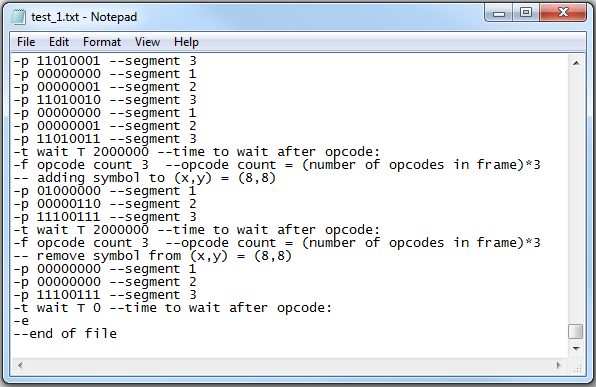


Figure 38 - Example of the simulation text file

Create new frame

Wait

opcodes

Finish test

### Log File

#### Overview

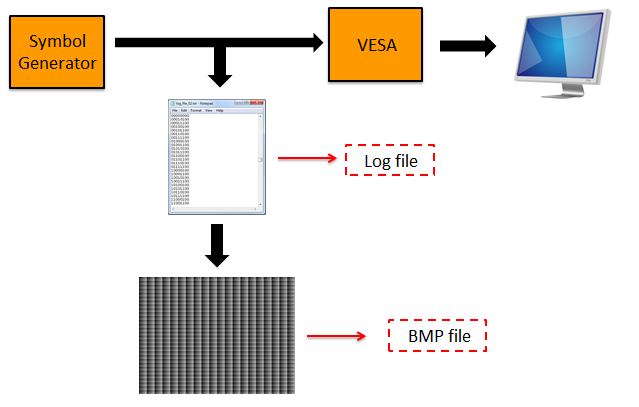
The purpose of the Log File block is to create a log of the data traffic from Symbol Generator to the VESA component. It upgrades the simulation process by visualizing the results: the log file, which represents the video frame, is converted to Bitmap file and shows the actual data.

Figure 39 - Log File block in the simulation process

#### Interfaces

#### The interfaces of the Log File block with the Symbol Generator Top block:

#### 

Figure 40 - Log File interfaces

The Log File block receives the data and validity indication of it from Symbol Generator Top.

The VSYNC arrives from the VESA Signals model block in the simulation environment. It informs the Log File block that a new video frame start, in order to create a new log file.

#### Implementation

The Log File block creates a new text log file for each transferred video frame.

When the block receives the VSYNC control signal for a start of a new frame it saves the previous text file and creates a new text file for the current video frame. The name of each created log file has an index of its order.

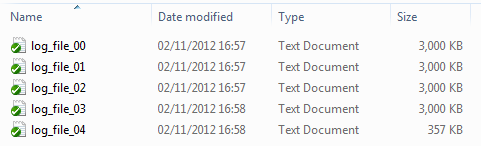
For example:

Figure 41 - Example of log files

## Test Plan

With the ready simulation environment, there was a need to think of all the special test cases in order to check the design. The main goal of the tests is to bring the DUT to

These test cases 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.

Test plan example:

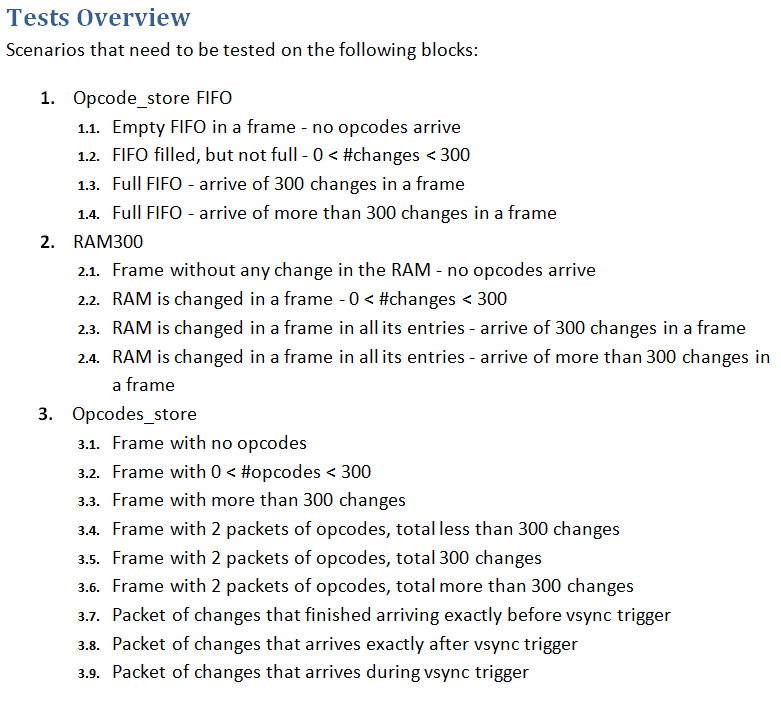


Figure 42 - Test plan example

Examples of output frames from the simulation process:



Figure 43 - output frames from the simulation process

## Automatic Test Manager

### Overview

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.

### Implementation

#### TCL Scripts

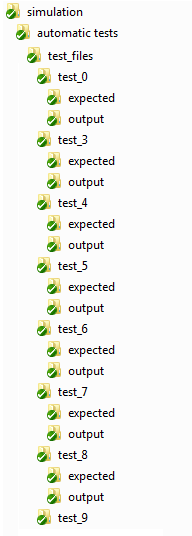
The implementation of the Automatic Test Manager was using the TCL script language.

The TCL language has the following advantages:

* Good interface with ModelSim program
* Strong abilities of working with text files

#### Test Directory Structure

The test directory structure:



test\_5\_log.txt

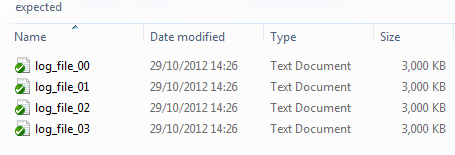
run\_tests.do

check\_test.do

tests.do

log.txt

test\_5.txt



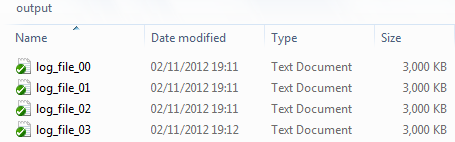


Figure 44 - Test directory structure

#### Detailed Description

The test environment was built using two scripts:

1. **check\_test.do**

This script runs from the test directory (for example: "**test\_5**" directory in the figure above), and uses the "expected" and "output" directories.  
The script compares the output vs. the expected, and returns the log for the test (for example: "**test\_5\_log.txt**" directory in the figure above) with the result of the test.

Example for a test log:

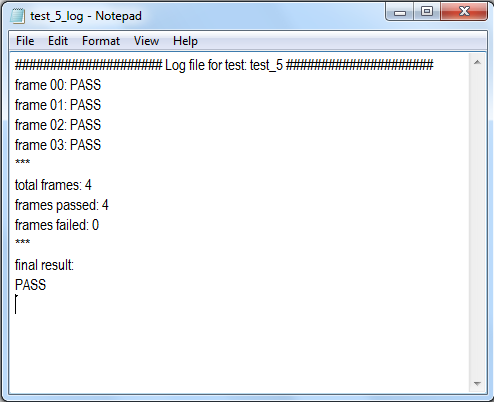


Figure 45 - Example for a test log

1. **run\_tests.do**  
   Main script to run simulation from the **test\_files** directory.

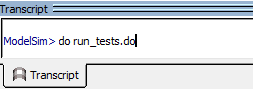
****

Figure 46 - running run\_tests.do script

It runs the list of tests that are written in the **tests.do** file.

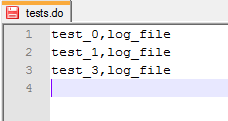


Figure 47 - Example of tests.do file

For each test, the script performs the following actions:

* + setting the generic values
  + running ModelSim simulation
  + running the **check\_test.do** script on the test
  + collecting the result of the test from its log file: pass or fail

The overall result of the tests is saved in the "log.txt" file.

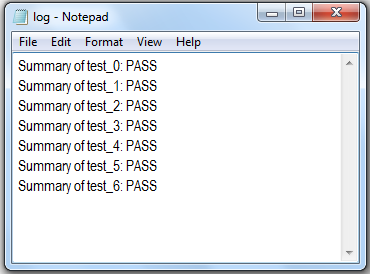
Example of a log file:

Figure 48 - Example for a log file

### Example: Running Test

An example for running simulation test using the automatic environment can be found in the link:

[http://www.youtube.com/watch?v=4gEru0Onms0&feature=youtu.be](http://www.youtube.com/watch?v=4gEru0Onms0&feature=youtu.be%20)

# Working Methods

The SG project was planned and implemented using design tools and methodology methods delivered by project supervisor and lab staff. Those techniques were found to be very useful.

## Top down design

This method was implemented from the beginning of the design, as recommended. Detailed description of the overall system and integrated blocks was documented and examined carefully in a top down review.

This was a crucial part of the design, mainly because of the need of integrating with an outside platform, planned by other designers.

## Testing each component separately

As recommended, TBs were planned for each component separately. Each update of the design led to detailed testing, a useful and important part of the proof of design methodology.

## 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:

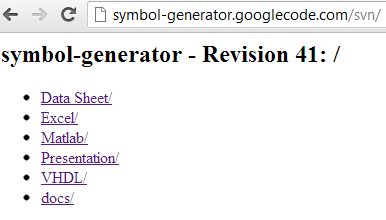
****

Figure 49 - SVN snapshot

The structure of the project in the SVN library:

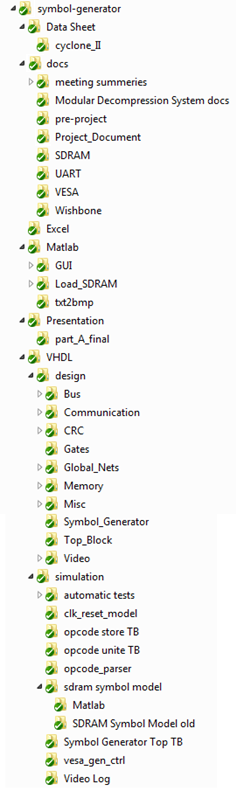


Figure 50 - SVN project structure

## Documentation (DOC)

Documentation of each VHDL block can be found within the VHDL code. Moreover, the project design, meeting summaries and updated versions can be found in the link: <http://symbol-generator.googlecode.com/svn/>

## Coding Guidelines

The project was accompanied by strict coding guidelines, for the benefit of the designers and future users. More details can be found in "Coding Guidelines" documentation.

# Summary

## Problems and Solutions

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

1. **Problem:** The SG project reuses another project as its platform environment. This may lead to potential integration problems.

**Solution:** Use a top block which is used as a buffer between the SG blocks and the external environment. The block amalgamates the different components of SG as a single unit.

1. **Problem:** Loading the RAM with the next frame changes may cause a collision in the Manager operation:The Manager uses the RAM contents for extracting the relevant symbols for the current frame out of the SDRAM. Loading the RAM with the next frame changes will interrupt the Manager operation.

**Solution:** Implementing a FIFO in Opcode Store component, used as a buffer between sequential frames. The FIFO contains the next frame changes so only when Manager asks for the current frame are fulfilled, the RAM can be loaded with the FIFO contents.

1. **Problem:** Different CLK domains – the SG operates in 100MHz while VESA operates in 40MHz. This may cause Meta Stabilization while sampling the inputs.

**Solution:** Sampling inputs which come from different CLK domain twice, so the Meta Stabilization option is minimized. Implementation of this solution can be found in Opcode Store code documentation:

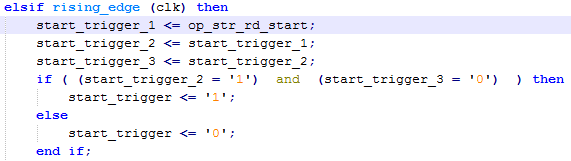
****

Figure 51 - Clock domain crossing

## Coding conclusions

* Divide the design into several processes makes the design more readable, organized, efficient and manageable.
* Synchronizing the reset negation to the clock’s rising edge is essential, to prevent Meta Stability from entering the system.
* Using top block as a wrapper for other blocks can be very useful in debug, and prevents fatal errors.
* Using automatic test tool, although required some time in building and debugging it, can save a lot of time and effort in the long run.

## Project Conclusions

The main goals of this project, building SG blocks with integrating to an outside platform, design the system while considering constrains and simulating and debugging VHDL code had been achieved. In addition, an automatic test GUI tool was created for easy debug purpose.

Moreover, goals had been achieved by:

1. Organized working methods that saved time including SVN use.
2. Code Review – educating and helpful.
3. Documentations – Causes to understand better what we are doing.

# 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