CorSight2

OpenCamera

Reference Manual

Rev1.03-180323

Commercial-in-Confidence

23 March 2018

NET New Electronic Technology GmbH
Lerchenberg 7
86923 Finning, Germany

Tel: +49 8806 9234 0
Fax: +49 8806 9234 77

info@net-gmbh.com
www.net-gmbh.com

Revision History

Revision	Date	Author	Modifications
Rev1.00	06/02/18	MK	Initial Release
Rev1.01	23/02/18	MJ	Added chapter 12: Custom Module SW Interface
Rev1.02	01/03/18	MJ	Modified chapter 9.3: CorSight2 FPGA Flash Programming
Rev1.03	23/03/18	MK	Added Chapter 6.4: Custom Module Configuration Fixed wrong bit numbering for 8-bit RGB in Figure 7

Table of Contents

Index of Tables

Index of Figures

1 CorSight2 Camera Overview

This reference manual introduces the third-party OpenCamera design methodology for the CorSight2 camera system. New user functions can be included in the image pipeline of the CorSight2 FPGA by adding appropriately designed processing modules, which are referred to as “Custom Modules”. Custom Modules can be designed and added by the customer without requiring direct support from NET GmbH.

The CorSight2 vision system is an intelligent camera based on the Intel Atom E3845 Processor operating under a Windows or Linux OS. Housed in a sealed IP67 cabinet, the camera is suitable to be used in rough environments. CorSight2 supports a wide range of CCD and CMOS sensors and uses a flexible C-Mount based lens system which incorporates a Flash Strobe Lighting Module to provide optimal lighting conditions for image capture. To control image acquisition a versatile trigger system is implemented based on TTL and Optocoupler inputs/outputs.

Image acquisition is handled inside the CorSight2 Artix-7 FPGA which serves as the interface between the sensor and the Atom Processor. Image processing tasks performed in the CorSight2 camera system can be divided between the FPGA and the Atom Processor (Host). The FPGA is best suited to perform data intensive tasks, such as:

- Image pre-processing
- Image encryption / compression
- Extraction, object detection / feature analysis
- 2D algorithms, point-to-point or neighbourhood operations
- Laser scan line analysis

Whereas the Processor is best suited to perform high-level, decision-based algorithms, the results of which are communicated to the outside world (i.e. application).

For a detailed specification of the CorSight2 camera please refer to the camera data sheet.

1.1 CorSight2 OpenCamera Concept

The OpenCamera concept introduces a simulation and compilation flow for the CorSight2 FPGA which can be performed by the customer. The OpenCamera Development Kit (ODK) includes a complete set of source files for all modules which make up the FPGA Framework in which the Custom Module is embedded. Image input and output ports connect the Custom Module to the FPGA Framework. A SystemBus interface is also provided to allow host access to the control and status registers of the Custom Module. In addition, the Custom Module has direct access to a dedicated 1-Gbit DDR3-RAM as well as to the Opto/TTL GPIO pins of the CorSight2 camera.

Having written a suitable Custom Module, the customer can proceed with compiling the entire FPGA, load the resulting firmware file into the FPGA Flash device on CorSight2 and run image acquisitions through the Custom Module function block.

This manual describes the OpenCamera development environment in detail and provides a step-by-step guide to integrate a custom application into the CorSight2 camera.

2 OpenCamera Development Kit (ODK)

2.1 OpenCamera Pre-requisites

The following pre-requisites have to be met before attempting to simulate or compile a CorSight2 OpenCamera project:

1. CorSight2 OpenCamera Development Kit (ODK), FPGA firmware revision 1.03 or later.
2. X86 compatible PC running on an MS-Windows (7.1 or 10.0 Pro, 64-bit) or Linux (Red Hat, SUSE, CentOS, Ubuntu) operating system. Details of supported operating systems can be found at https://www.xilinx.com/support/answers/54242.html
3. Xilinx **Vivado 2017.3**. The Webpack edition supports all Artix-7 FPGAs and is therefore suitable to be used for CorSight2. The **Vivado 2017.3** Webpack Edition can be downloaded from the Xilinx website at: https://www.xilinx.com/support/download.html

2.2 OpenCamera Development Kit Installation

The CorSight2 ODK is contained in a single archive file called:

CS2_OpenCam_Project_VerXX-YY.zip

"XX" indicates the major revision number and "YY" indicates the minor revision number of the FPGA firmware release. This manual refers to FPGA firmware Ver01-03 (or later).

It is recommended to extract the entire archive to a directory on a computer where the Xilinx Vivado FPGA synthesis and implementation software is installed. In this document it is assumed that the ODK development system is extracted to a directory called:

<OpenCamDir>/...

All location references to the OpenCamera source files and directories made in this document are defined relative to this path.

2.2.1 OpenCamera Environment Variable Setup

Once the ODK is installed, the user must first update required environment variables used by various OpenCamera compilation script files. The variables are defined in the following file:

<OpenCamDir>/CompileTools/Set_Customer_EnvVar.tcl

Table 1 shows the default content of **Set_Customer_EnvVar.tcl**. Please note that this file is an executable TCL file, which therefore needs to adhere to correct TCL syntax.

Please Note: The path separators used in **Set_Customer_EnvVar.tcl** must adhere to TCL syntax. MS-Windows paths definitions must therefore be changed using the forward slash **'/'**.

The required user information is defined in an environment variable called **Cs2OpenCamCfg()**. It is recommended not to use spaces in the path names to avoid potential misinterpretation of the intended directory targets.

```
#-------------------------------------------------------------------------------
# CorSight2 OpenCamera local environment variables
# Copyright (C) 2017 NET GmbH. All Rights Reserved
#-------------------------------------------------------------------------------

# Set paths to OpenCamera source, compile and release directories.

# IMPORTANT: Use the TCL path separator '/' in all OS environments!
# Example: set Cs2OpenCamCfg(OpenCamDir) "C:/NET_GmbH/CorSight2/OpenCamera"

puts "-- \[Set_Customer_EnvVar.tcl\] Retrieving customer-defined local environment variables..."

set Cs2OpenCamCfg(OpenCamDir)  "Customer_Path_Definition"
set Cs2OpenCamCfg(CompileDir)  "Customer_Path_Definition"
set Cs2OpenCamCfg(ReleaseDir)  "Customer_Path_Definition"

return -code ok Cs2OpenCamCfg
```

Table 1: Set_Customer_EnvVar.tcl Environment Setup Script

- **Cs2OpenCamCfg(OpenCamDir)**: Path to the OpenCamera source directory into which the archive has been extracted, i.e. **<OpenCamDir>**.
- **Cs2OpenCamCfg(CompileDir)**: Path to the OpenCamera compile directory which will be used when running FPGA synthesis/implementation. It is recommended to keep the length of this path as short as possible since the ODK and Vivado add numerous levels of sub-directories below the specified compile directory and the possibility exists that the maximum path length of 260 characters under MS-Windows is exceeded during compilation.
- **Cs2OpenCamCfg(ReleaseDir)**: Path to the OpenCamera firmware release directory. After successfully running an FPGA compilation, the OpenCamera scripts produce a firmware file which will be copied from the compile directory to the release directory. The name of the firmware file adheres to the required CorSight2 firmware naming convention.

2.2.2 Vivado Installation Path and Version Setting

Various OpenCamera batch files running under MS-Windows require the Vivado installation path information in order to launch the Vivado tool. ODK version 01.03 only works in conjunction with **Vivado 2017.3.** By default Vivado 2017.3 installs to a directory "**C:/Xilinx/Vivado/2017.3**" which is the default path setting for the OpenCamera batch files.

Please note: It is highly recommended to install Vivado to the default location!

If Vivado has been installed to a different location the relevant ODK environment variables need to change. The Vivado path variables are defined in:

<OpenCamDir>/CompileTools/Set_Vivado_Version.bat

Table 2 shows the top portion of the Vivado version file which the user must update if a non-default Vivado installation path has been chosen. Once these variables have been defined the user is free to simulate and compile the OpenCamera example designs.

```
::-------------------------------------------------------------------------
:: Project   : CorSight2 - Custom Module
:: Title     : Vivado Installation Path and Version Settings
:: Copyright (C) 2017 NET GmbH. All Rights Reserved
::-------------------------------------------------------------------------
@echo off

:: Set Vivado Installation Path and Version
set VivadoPath=C:\Xilinx\Vivado
set VivadoVer=2017.3

.
.
.
```

Table 2: Set_Vivado_Version.bat Environment Setup Script

3 OpenCamera Directory Structure

The CorSight2 OpenCamera Development Kit is partitioned into six subdirectories, as listed below:

- **CompileTools**: Contains the OpenCamera FPGA compile scripts.
- **doc**: Custom Module documentation (including this document).
- **IP**: Contains all Vivado IP cores used in the FPGA Framework logic. It also contains encrypted design source files for all FPGA function blocks which (in combination with the IP cores) make up the FPGA Framework logic.
- **sim**: Custom Module simulation test bench files.
- **src**: Custom Module HDL source code files and example designs.
- **syn**: FPGA synthesis and implementation script files, as well as Vivado project and constraint files.

The complete directory structure of the CorSight2 ODK is shown in Table 3 and further explained in the following paragraphs.

```
+----------------------+
| CorSight2 OpenCamera |
|   Development Kit    |
+----------------------+
          |
          |-- CompileTools
          |
          |-- doc
          |
          |-- IP
          |   |-- DmaCtrl
          |   |-- DualMCB
          |   |-- LogicAnalyzer
          |   |-- PcieCtrl
          |   |-- SecureIp
          |   |-- TechLib
          |   |-- XadcMonitor
          |
          |-- src
          |   |-- BlockDemo
          |   |-- CannyFilter
          |   |-- Default
          |   |-- DiffPic
          |   |-- LutDesign
          |   |-- MemTest
          |   |-- ScaleD
          |   |-- UserDesign
          |   |-- Wrapper
          |
          |-- syn
          |   |-- InitFiles
          |   |-- ScriptFiles
          |   |
```

continues next page...

```
|    |
|    |-- ConstraintFiles
|    |    |-- CS2_Rev1
|    |    |-- CS2_Rev2
|    |    |-- CS2_Rev3
|    |
|    |-- ProjectFiles
|    |    |-- CS2_Rev1
|    |    |-- CS2_Rev2
|    |    |-- CS2_Rev3
|
|-- sim
     |
     |-- Simulation
     |    |-- Custom_Canny
     |    |    |-- Xsim
     |    |-- Custom_DiffPic
     |    |    |-- Xsim
     |    |-- Custom_LutDesign
     |    |    |-- Xsim
     |    |-- Custom_MemTest
     |    |    |-- Xsim
     |    |-- Custom_ScaleD
     |    |    |-- Xsim
     |    |-- Custom_UserDesign
     |    |    |-- Xsim
     |    |
     |    |-- Images
     |
     |-- TestBench
          |-- Chipscope
          |-- LogicAnalyzer
          |-- DDR3-RAM_1Gbit
          |-- MCB_SIM
               |-- MT41K64M16XX107
                    |-- clocking
                    |-- controller
                    |-- ecc
                    |-- ip_top
                    |-- phy
                    |-- ui
```

Table 3: OpenCamera Directory Structure

3.1 OpenCamera Source Files

Custom Module example design files are located in the **<OpenCamDir>/src** subdirectory. This directory is subdivided into 9 sub-sections:

- **Default**: Default Custom Module design files. Contains an empty Custom Module used to compile a bare-bone Framework FPGA without any custom logic. There is no simulation testbench setup for the default Custom Module. See Chapter 7.1 for more information.
- **UserDesign**: Custom Module user design files. Contains a recommended Framework for new custom designs. Customers are encouraged to use and modify the UserDesign to implement their own custom logic. See Chapter 7.2 for more information.
- **LutDesign**: Example Lookup-Table design. Contains a reference design of a Colour and Monochrome Lookup-Table. Features demonstrated in this design include the use of the ImageBus Receive and Transmit Ports and the usage of memory space within the SystemBus address space (i.e. BlockRAMs). See Chapter 7.3 for more information.

- **MemTest**: Example DDR3-RAM memory test design. Runs read and write operations to the DDR3-RAM and verifies the integrity of the DDR3-RAM interface. Can also be used as a test design to gauge maximum achievable read/write data bandwidth to the DDR3-RAM. MemTest does not involve ImageBus transfers, i.e. the ImageBus Transmit Port is directly connected to the ImageBus Receive Port. See Chapter 7.4 for more information.
- **DiffPic**: Example design which calculates the difference between successive monochrome images on a pixel-by-pixel basis. Demonstrates the use of the ImageBus and MemoryBus Interfaces. See Chapter 7.5 for more information.
- **CannyFilter**: Example Canny Filter design. Contains a reference design of a Canny Filter implementation. See Chapter 7.6 for more information.
- **BlockDemo**: This design generates an RGB block pattern overlay. A customized XML description for control registers is provided. Image data can be either monochrome or RGB. See Chapter 7.7 for more information.
- **ScaleD**: This design allows 2-dimensional downscaling of the ImageBus input data and demonstrates how to implement image processing with different input and output image sizes. Image data can be either monochrome or RGB. See Chapter 7.8 for more information.
- **Wrapper:** OpenCamera Wrapper configuration files. Contains VHDL constant definitions for all Custom Module designs. Except for the CustomPkg.vhd file (see Chapter 6.4), Wrapper files must not be modified.

3.2 OpenCamera Simulation Files

Custom Module testbench and simulation files are located in the **<OpenCamDir>/sim** subdirectory. All Custom Module example designs run off the same testbench which can be configured to suit each example design.

3.2.1 OpenCamera Testbench

The OpenCamera VHDL testbench files are located in **<OpenCamDir>/sim/TestBench** which make up the simulation environment in which a Custom Module can be embedded. The testbench does not include FPGA Framework logic, instead ImageBus and SystemBus stimuli modules are used which work in conjunction with simulation scripts provided for each Custom Design.

The testbench has a separate configuration file for each custom design. Testbench configuration files are named: **CustomCfg_<CustomDesignName>.vhd** and provide a reference to the TestBench initialization file, located in the Simulation environment. Also, the type of DDR3-RAM used during simulation is defined here.

The user can select to simulate the MCB/DDR3-RAM interface of the Custom Module with either a fully functional 1 Gbit DDR3-RAM model provided by Micron Technology, Inc or use a scaled down DDR3-RAM substitute model which has been specifically designed for the OpenCamera simulation testbench. The advantage of the fully functional 1 Gbit DDR3-RAM model is that it provides cycle-accurate simulations but at a reduced simulation speed.

The substitute model provides a small, functionally correct MCB/DDR3-RAM interface but does not produce the exact same timing sequence as the fully functional model. Using the substitute MCB/DDR3-RAM model significantly improves simulation times.

3.2.2 OpenCamera Simulation Environment

Pre-configured simulation scripts and simulation environments are provided for each Custom Module example design. The relevant files are located in **<OpenCamDir>/sim/Simulation**. The CorSight2 ODK supports simulations with the Xsim simulator which is part of the Vivado design suite. Other simulation engines can be used but may require the customer to adapt the simulation scripts to the new platform.

The OpenCamera example designs can be easily simulated in a few steps using the provided simulation scripts. For more details on how to run a simulation please refer to Chapter 8.

3.3 OpenCamera Synthesis Files

When starting an OpenCamera FPGA compilation the compile scripts require a list of Custom Module source files. These files are provided for existing example designs and are located in the **<OpenCamDir>/syn** directory. If new Custom Module designs are added by the user corresponding **InitFiles/CS2_<CustomName>_Init.tcl** and **ProjectFiles/CS2_Revx/CS2_Revx_<Custom Name>.prj** files must be added. "**CS2_Revx**" denotes the applicable CorSight2 board revision number, x = [1..3].

3.4 Synthesis Initialization Files

The **<OpenCamDir>/syn/InitFiles** directory contains files which are used to control the FPGA synthesis/implementation flow. The files contain a number of user-definable parameters which the user can set to the required values before running an FPGA compilation. There is one file for each of the eight Custom Module example designs, referred to as **CS2_xxx_Init.tcl** (with "**xxx**" being a reference to the design name). The files are TCL scripts, therefore TCL syntax must be adhered to. Table 4 shows the initial content of the **UserDesign** initialization TCL script.

```
#-------------------------------------------------------------------------------
# Project    : CorSight2 - Custom Module
# Title      : User-defined FPGA compile variables for UserDesign
#-------------------------------------------------------------------------------
# Copyright (c) 2017 NET New Electronic Technology GmbH
#-------------------------------------------------------------------------------

# Set the Custom Module revision number (two digits):
set CUSTOM_REV_NUM  "01"

# Set the Vivado exit mode at end of compilation [OPEN, CLOSE]
set VIVADO_MODE  "CLOSE"

# Select the required debug module [NONE, CHIPSCOPE, ANALYZER] (DEBUG mode only)
set CUSTOM_DEBUG_MODULE  "CHIPSCOPE"

# Select the required debug device [ILA1x128x1024, ILA1x256x1024] (DEBUG mode only)
set CUSTOM_DEBUG_DEVICE  "ILA1x256x1024"

return -code ok
```

Table 4: Custom Module Revision Number Setup

- **CUSTOM_REV_NUM** defines the revision number for the Custom Module design. The module designer can freely assign any 2-digit integer number. The chosen revision number

is added to the firmware file name after a successful FPGA compilation has been performed. In the **UserDesign**, **LutDesign** and **MemTest** examples the **CUSTOM_REV_NUM** parameter is also added to the HDL source files via a Version package file which is automatically generated by the ODK compile scripts. In these examples the revision number can be read from the SystemBus as a status register.

- **VIVADO_MODE** determines if the Vivado GUI remains open after the FPGA implementation is complete **[OPEN]** or if Vivado automatically closes **[CLOSE]**. The **OPEN** option is useful if the user wants to inspect the Vivado implementation results at the end. The **CLOSE** option should be used when running multiple FPGA compilations in a batch file setup.
- **CUSTOM_DEBUG_MODULE** is used to add the ChipScopePro or Vivado Logic Analyzer IP to the FPGA compilation project, if so required. This parameter is available only for the **UserDesign**, **LutDesign** and **MemTest** examples.

  If **CUSTOM_DEBUG_ MODULE** is set to either **"CHIPSCOPE"** or **"ANALYZER"** a corresponding IP instantiation (which must be present in the Custom Module HDL code) is enabled. It is only used in Debug Mode which is invoked via a compilation command line argument, described in Chapters 9.1 and 9.2.
- **CUSTOM_DEBUG_DEVICE** selects the ILA device which either ChipScopePro or the Vivado Logic Analyzer uses. This parameter is available only for the **UserDesign**, **LutDesign** and **MemTest** examples.

  The **"ILA1x128x1024"** option selects a 128-bit wide and 1024 deep ILA device, whereas the **"ILA1x256x1024"** selects a 256-bit wide and 1024 deep ILA device. Due to it's wider port width the **"ILA1x256x1024"** option requires more FPGA resources (BlockRAMs). However, unless severe resource limitations apply for a given design the recommended device is the 256-bit ILA option.

Please note: A ChipScope debug option only exists on specially prepared CorSight2 cameras. An external JTAG connector to the CorSight2 camera is required to connect a Xilinx Platform Cable USB II or another Xilinx compatible JTAG adapter. Please contact NET GmbH if a JTAG connection is required.

3.5 OpenCamera IP and FPGA Framework Source Files

The **<OpenCamDir>/IP** directory contains all Vivado IP cores used in the FPGA Framework logic. It also contains encrypted CorSight2 design source files (in **<OpenCamDir>/IP/SecureIp)** which in combination with the IP cores make up the FPGA Framework logic. The encrypted source files can not be viewed but are decrypted by Vivado during the synthesis process. No other device or tool is permitted to decrypt the OpenCamera source files.

This directory does not contain any user-modifiable files.

3.6 OpenCamera Documentation Files

The **<OpenCamDir>/doc** directory contains information on the current FPGA firmware release and the OpenCamera Reference Manual (this file). Also, control/status register map definition files for the six Custom Module example designs are provided.

4 CorSight2 FPGA Architecture

The CorSight2 Artix-7 FPGA contains all functions/modules to control the image acquisition from the sensor, process the pixel stream in an Image Pipeline and sending the image data to the Atom processor via the DMA/PCI-Express Interface. The image stream also passes through a Frame Buffer Controller which is used to temporarily store the incoming images.

The FPGA Framework consists of a number of core FPGA modules to handle the external FPGA interfaces and to perform system level functions. The Framework modules are required in every design implementation and include the **Sensor Interface**, **PCIe Interface**, **DMA Controller**, **Frame Buffer**, **SystemBus Master**, **Timers** and other I/O modules. The FPGA block diagram is shown in Figure 1.

The Sensor Interface receives sensor specific image and control data and presents the raw image data on the ImageBus. The image stream passes through an (optional) image processing pipeline before reaching the Custom Module. The ImageBus Pipeline contains the following functions:

- Defect Pixel Correction (DPC)
- Offset/Gain Control (OGC)
- Flat Field Correction (FFC)
- Bayer Decoder (BAY)
- Colour Conversion Matrix (CCM)
- Gamma Correction (GCOR)

The Custom Module has the option of storing images in it's own, dedicated DDR3-RAM. A suitable interface exists in the Custom Module to read and write data to the DDR3-RAM.

Having processed the incoming pixel stream, the Custom Module presents the modified image stream on the ImageBus Transmit Port. This port is directly connected to the CorSight2 Frame Buffer Controller to prepare the image to be send to the Atom Processor for further analysis and display purposes.

A second optional image processing module is the Geometric Correction Module (GEO). As with the ImageBus Pipeline the customer has the option of removing these modules from the FPGA Framework by selecting a suitable Design option during the FPGA compilation process (see Chapter 4.1). The removal of these modules frees up resources in the FPGA fabric which can be utilized by the Custom Module design, if so required.

To allow for a flexible trigger based image acquisition a large number of Timer as well as TTL- and/or Opto-based GPIO signals can be used. Timers are used to process incoming GPIO and Sensor signals which in turn generate trigger and exposure pulses to the sensor. Timers can also detect possible trigger mismatches, by detecting trigger signals while the sensor is busy.

Tied in with the trigger generation is the control of the inbuilt LED Flash Strobe Lighting device. To guarantee proper image illumination at the time an image acquisition is made, the Flash Strobe needs to be triggered a certain amount of time before acquisition is started. This process is also handled by the Timer Module.

FPGA modules can be initialised by the Host via the SystemBus infrastructure. This bus system is based on the Wishbone-like bus specification, which features separate Address, DataIn and DataOut bus structures.

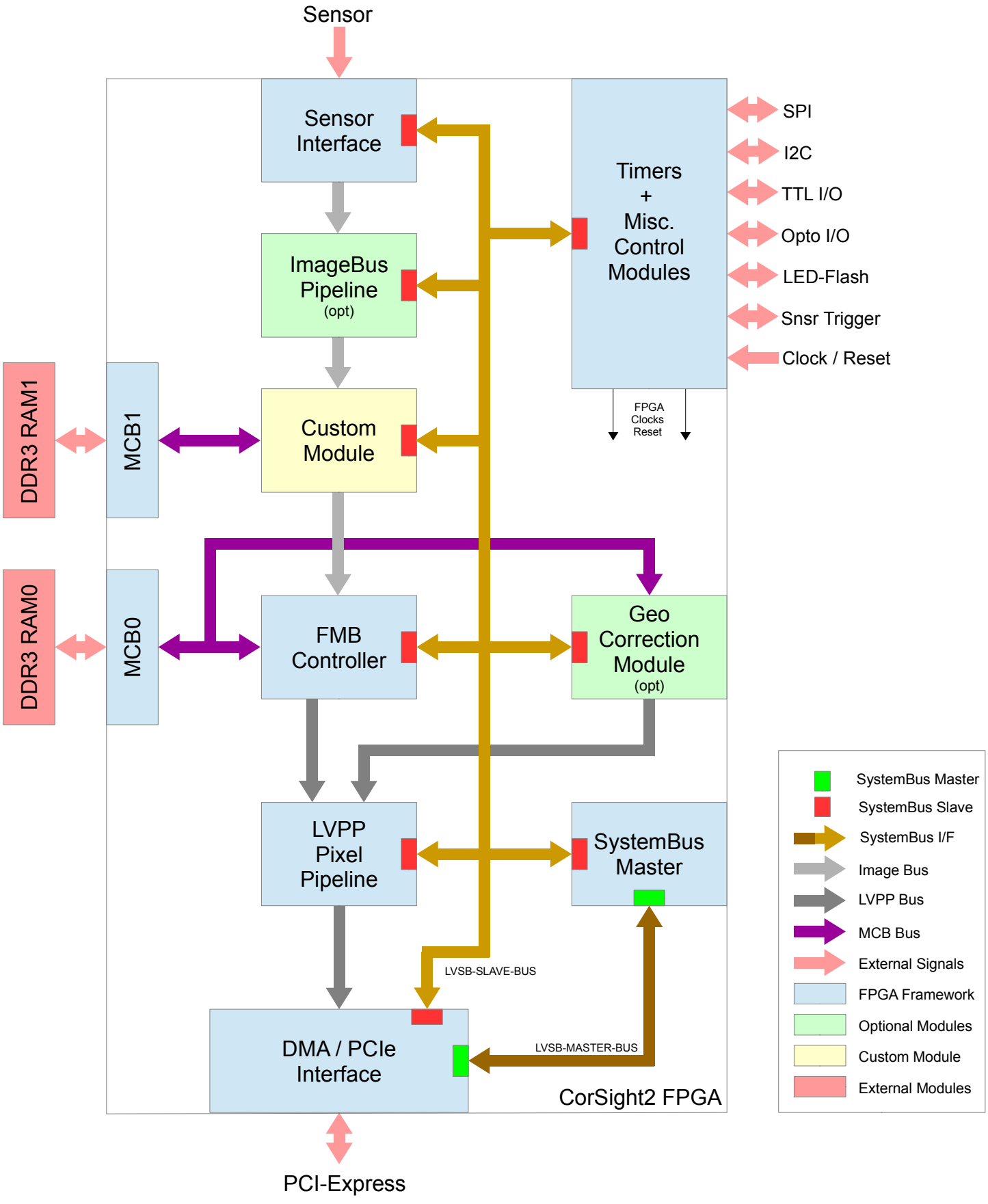

Figure 1: CorSight2 FPGA Block Diagram

4.1 FPGA Framework Design Options

As shown in the CorSight2 FPGA Block Diagram (Figure 1) the internal logic blocks of FPGA can be divided into 3 categories:

- FPGA Framework Modules (blue)
- Optional Image Processing Modules (green)
- Custom Module (yellow)

To provide a high degree of flexibility when compiling the CorSight2 FPGA the OpenCamera Development Kit provides the option of eliminating certain modules from the compilation process in order to gain more free FPGA resources for the Custom Module design. The following three design options exist:

- **Design-00 (DSG00)**: Includes all FPGA Framework modules as well as the ImageBus Pipeline and the Geo-Correction Module.
- **Design-01 (DSG01)**: Includes all FPGA Framework modules as well as the ImageBus Pipeline. The Geo-Correction Module has been removed.
- **Design-02 (DSG02)**: Includes all FPGA Framework modules. The ImageBus Pipeline and the Geo-Correction Module have both been removed.

CorSight2 uses a Xilinx Artix-7 FPGA device (**XC7A75T-FGG484-2**) which can comfortably accommodate the FPGA Framework logic as well as the optional image processing modules. **DESIGN-00** is therefore the standard FPGA configuration. However, if the Custom Module logic requires large amounts of internal FPGA resources **DESIGN-01** or **DESIGN-02** can be used instead. The customer can select the desired OpenCamera FPGA Design-ID at compile time.

The used and available FPGA resources for each of the three design options are shown in Table 5.

		DESIGN 00		DESIGN 01		DESIGN 02	
Resource	Available	Utilization	Utilization %	Utilization	Utilization %	Utilization	Utilization %
LUT	47200	34521	73.14	30670	64.98	25000	52.97
LUTRAM	19000	1489	7.84	1262	6.64	1201	6.32
FF	94400	36762	38.94	31511	33.38	23957	25.38
BRAM	105	88	83.81	50.50	48.10	31	29.52
DSP	180	37	20.56	25	13.89		
IO	285	155	54.39	155	54.39	155	54.39
GT	4	1	25.00	1	25.00	1	25.00
BUFG	32	12	37.50	12	37.50	12	37.50
MMCM	6	4	66.67	4	66.67	4	66.67
PLL	6	2	33.33	2	33.33	2	33.33
PCIe	1	1	100.00	1	100.00	1	100.00

Table 5: CorSight2 FPGA Framework Resource Utilization

5 Custom Module Architecture

The OpenCamera Custom Module can be designed without reference to the overall workings of the CorSight2 FPGA. All information required for a successful completion of a custom design is contained within the Custom Module itself. The design only needs to adhere to the required bus protocols connected to the module.

The recommended Custom Module layout includes a SystemBus interface and a Custom Design Block which contains the user application, as shown in Figure 2. Custom Module configuration parameters are provided in a separate package file which define important module constants and type definitions.

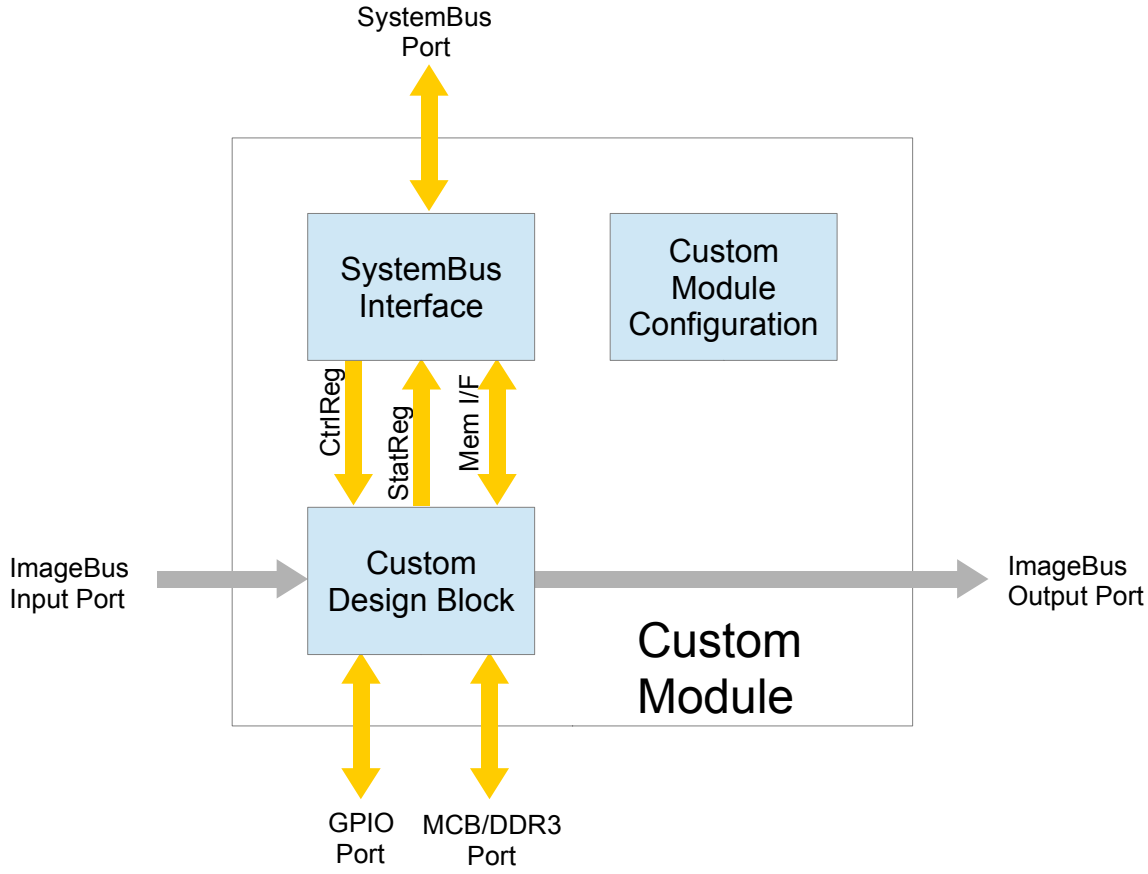

Figure 2: Custom Module Architecture

The OpenCamera Development Kit contains several example designs intended to demonstrate how to design, simulate and implement a custom design. The **UserDesign**, **LutDesign** or **MemTest** adhere to the recommended Custom Module architecture shown in Figure 2.

The **UserDesign** is intended to be used by the application designer to expand the provided **UserDesign** source files to implement a desired custom function. This approach has the advantage of utilising the pre-configured implementation and simulation scripts without having to write the entire design environment from scratch.

For a detailed description of the **UserDesign** and other example designs provided with the ODK see Chapter 7.

5.1.1 Custom Design Block

To implement a clean and well structured Custom Module design, it is suggested to include the application-specific design files in a separate Custom Design Block and to instantiate the SystemBus Interface and the Custom Design Block in the top-level Custom Module file. This approach separates the CorSight2-specific design elements from the application and encourages an environment-independent design practice.

5.1.2 Custom Module Configuration

Important Custom Module definitions are specified in a central VHDL configuration package. This may include (but is not limited to) SystemBus and ImageBus related parameters. Template definitions are provided in the **UserDesign** files to define the structure of control and status registers for SystemBus interactions with the Module. Furthermore, the GPIO output pin configuration is defined in this file.

Another package file is used to define the current version number of the Custom Module to be implemented in the FPGA. This file is automatically generated by the FPGA compile script and should not be modified by the user. For information on how to set the version number please refer to Chapter 3.4.

5.1.3 SystemBus Interface

The SystemBus Interface is used by the Host to communicate with the Custom Module and to initialise the Module for the upcoming image processing task. Control and status registers (defined in the Custom Module Configuration package) are allocated to the required address and data locations in this design unit. The SystemBus Interface also includes the mandatory System ID status registers at address location 0 which identifies the Module to the Host S/W.

The SystemBus Interface also provides an optional System Memory Interface, which directly connects to the Custom Design Block in case a Host-configurable memory is present in the custom design.

For a complete description of the SystemBus Interface please refer to Chapter 6.2.

5.1.4 ImageBus Interface

The Custom Design Block connects to the ImageBus Receive and Transmit Ports to facilitate image transfers in and out of the Custom Module. When generating image transfers on the ImageBus Transmit port the Custom Design Block must produce a stream of fixed-sized images. The size of transmitted images can differ from the size of received images.

For a complete description of the ImageBus Interface please refer to Chapter 6.3.

5.1.5 GPIO Interface

The Custom Module can connect to the external GPIO pins of the CorSight2 camera, which consists of 12 signals: 4x Opto outputs, 4x Opto inputs, 2x TTL output and 2 TTL input. GPIO signals can be shared between the Custom Module and the System GPIO Handler. Each GPIO output can be configured from within the Custom Module to determine if it is driven by the Module or the

Handler. GPIO inputs are always available to both sections.

Please note: The TTL(1) pin is currently not connected to an external GPIO pin in a standard CorSight2 camera configuration.

For a complete description of the GPIO Interface please refer to Chapter 6.5.

5.1.6 MemoryBus Interface

The Custom Module has exclusive access to a 128MByte DDR3-RAM device which can be used for image storage/retrieval functions or for other large-scale data applications.

The Memory Interface interacts with a Xilinx Memory Controller Block (MCB) IP which is located in the FPGA Framework and which defines the protocol the interface must adhere to. The interface is divided into 3 separate bus units, which facilitate the reading/writing processes to/from the DDR3-RAM"

- The MemoryBus Command Port is used to issue read or write commands to the MCB Controller in the FPGA Framework. Commands support read/write burst transfers of up 64 DWords in length.
- The MemoryBus Read Port is used to receive read data from the MCB which has previously requested via the MemoryBus Command Port.
- The MemoryBus Write Port is used to transfer write data from the Custom Module to the MCB. Write data must be transferred to the MCB before a corresponding Write Command is issued.

For a complete description of the MemoryBus Interface please refer to Chapter 6.6

6 Custom Module Bus Interface Descriptions

The top level entity of the Custom Module has a predefined structure. The port/signal definitions which connect the Custom Module to the FPGA Framework are shown in the VHDL entity declaration in Table 6. To ensure a successful Custom Module integration, it is highly recommended not to modify the entity declaration in any way. If a particular output port/signal is not used, the Custom Module should tie the signal to it's inactive state but it should never be removed. Unused input signals can be ignored in the Custom Module, but should not be removed from the entity declaration. The following chapters provide a detailed description for all Custom Module interfaces.

```
entity CustomModule is
    port (
        -- Module Infrastructure Signals
        SysClkxCI         : in  std_logic;
        SysRstxRI         : in  std_logic;
        ImgRstxRI         : in  std_logic;

        -- SystemBus Control Interface
        SysReqxMI         : in  std_logic;
        SysAckxMO         : out std_logic;
        SysErrxMO         : out std_logic;
        SysWrEnxMI        : in  std_logic;
        SysAddrxAI        : in  std_logic_vector(16 downto 0);
        SysWrDataxDI      : in  std_logic_vector(31 downto 0);
        SysRdDataxDO      : out std_logic_vector(31 downto 0);

        -- Custom Module GPIO
        CustIoCtrlxMO     : out std_logic_vector(5 downto 0);
        CustIoDataOutxDO  : out std_logic_vector(5 downto 0);
        CustIoDataInxDI   : in  std_logic_vector(5 downto 0);

        -- ImageBus Receive Port
        ImgRxPixValxMI    : in  std_logic;
        ImgRxPixRdyxMO    : out std_logic;
        ImgRxPixStatxSI   : in  std_logic_vector(2 downto 0);
        ImgRxPixDataxDI   : in  std_logic_vector(35 downto 0);

        -- ImageBus Transmit Port
        ImgTxPixValxMO    : out std_logic;
        ImgTxPixRdyxMI    : in  std_logic;
        ImgTxPixStatxSO   : out std_logic_vector(2 downto 0);
        ImgTxPixDataxDO   : out std_logic_vector(35 downto 0);

        -- MemoryBus Command Port
        px_cmd_en_o       : out std_logic;
        px_cmd_full_i     : in  std_logic;
        px_cmd_instr_o    : out std_logic_vector(2 downto 0);
        px_cmd_bl_o       : out std_logic_vector(5 downto 0);
        px_cmd_addr_o     : out std_logic_vector(29 downto 0);

        -- MemoryBus Read Port
        px_rd_en_o        : out std_logic;
        px_rd_empty_i     : in  std_logic;
        px_rd_data_i      : in  std_logic_vector(31 downto 0);

        -- MemoryBus Write Port
        px_wr_en_o        : out std_logic;
        px_wr_full_i      : in  std_logic;
        px_wr_data_o      : out std_logic_vector(31 downto 0));
end CustomModule;
```

Table 6: VHDL Custom Module entity declaration

6.1 Module Infrastructure

The Custom Module uses 3 general purpose infrastructure signals, as listed below:

- **SysClkxCI**: SystemBus/ImageBus Clock (150 MHz). Driven by the System Clock/Reset Generator in the FPGA Framework. Used to clock the entire Custom Module logic, including all SystemBus and ImageBus logic.
- **SysRstxRI**: SystemBus/ImageBus Reset. Driven by the System Clock/Reset Generator in the FPGA Framework. SysRstxRI is active after power-up to ensure FPGA logic is initialized to its default state. Used to reset the entire Custom Module logic, including all SystemBus and ImageBus logic. The SysRstxRI is asserted (low-to-high transition) asynchronously to **SysClkxCI**. Deasserting SysRstxRI (high-to-low transition) however is performed synchronously to **SysClkxCI**.
- **ImgRstxRI:** ImageBus Reset. Driven by a control register in the FPGA Framework. ImgRstxRI is used to reset the internal ImageBus data pipeline. It is asserted by the Host (i.e. SynView) before an image transfer is started for the first time. ImgRstxRI should be used to flush pixel data from the internal Custom Module pixel pipeline which may still be present from a previous image transfer. ImgRstxRI is also asserted after power-up. Once asserted, ImgRstxRI stays active for 4 **SysClkxCI** cycles. The Custom Module must reset its internal pixel pipeline and control logic within this reset period.

  Please note: ImgRstxRI is used regularly during operation, it must not be used to reset SystemBus control registers.

6.2 SystemBus Interface

The SystemBus is used by the Host Processor to interact with the Custom Module, to initialise control registers and on-chip memory or to read status registers/memory. The SystemBus Master is part of the FPGA Framework and interacts with the SystemBus Slave in the Custom Module via read or write access cycles. All Master and Slave signals are synchronous to the SystemBus clock signal **SysClkxCI**.

The SystemBus Interface only supports single cycle 32-bit data transfers, burst transfers or 8/16-bit data transfers are not supported. The SystemBus address bus contains 17 address lines, hence the maximal Custom Module SystemBus address space is 128 kByte.

6.2.1 SystemBus Signal Description and Protocol

The SystemBus Interface consists of 7 signal groups:

- **SysReqxMI**: Driven by the SystemBus Master in the FPGA Framework to request a new SystemBus read or write cycle.
- **SysAckxMO**: Driven by the SystemBus Slave (Custom Module) to terminate a pending read or write cycle, if no error condition is present.
- **SysErrxMO**: Driven by the SystemBus Slave (Custom Module) to terminate a pending read or write cycle, if an error condition has been detected. The error response from the Custom Module is an optional feature, if it is not implemented, the Custom Module must terminate a pending SystemBus cycle by asserting SysAckxMO.
- **SysWrEnxMI:** Driven by the SystemBus Master in the FPGA Framework to flag if a SystemBus read cycle (SysWrEnxMI = ‘0’) or write cycle (SysWrEnxMI = ‘1’) is pending.

- **SysAddr(16..0)**: Driven by the SystemBus Master in the FPGA Framework SysAddr(16..0) is a byte address which identifies a register or memory location within the Custom Module address space involved in the current SystemBus cycle. The SystemBus only supports 32-bit read or write cycles, hence SysAddr(1..0) are always “00”.
- **SysWrData(31..0)**: Driven by the SystemBus Master in the FPGA Framework with a 32-bit data word which is written to a register or memory location addressed by SysAddr(16..0).
- **SysRdData(31..0)**: Driven by the SystemBus Slave (Custom Module) with a 32-bit data word read from a register or memory location addressed by SysAddr(16..0).

6.2.2 SystemBus Read/Write Cycles

Figure 3 and Figure 4 show the basic read and write cycles, as described below:

In order to initiate a SystemBus cycle the Master asserts the **SysReqxMI** signal at the rising edge of **SysClkxCI** together with a valid register/memory address on **SysAddrxAI(16..0)**. If a write cycle is performed **SysWrEnxMI** is driven high and a valid data word is placed on **SysWrDataxDI(31..0)**. If the Master wants to perform a read operation **SysWrEnxMI** is driven low.

Upon detecting a high level on **SysReqxMI** the Slave device (Custom Module) performs the requested cycle. During write operations **SysWrDataxDI(31..0)** is stored at the requested register/memory location. For read operations, the Slave retrieves the requested data and places it on the **SysRdDataxDO(31..0)** bus.

When the Slave has completed the access cycle and no error condition has been detected, it asserts **SysAckxMO** to inform the Host to end the request. In response to **SysAckxMO** = ‘1’ the Master deasserts **SysReqxMI**. The Slave can at this point deassert **SysAckxMO** to allow the next transfer cycle to start on the following clock cycle or alternatively it can wait until it detects **SysReqxMI** = ‘0’.

If an error condition (such as an address to a non-existing register/memory location, or similar) has been detected, the SystemBus Slave has the option of terminating the access by asserting **SysErrxMO**. This informs the Host that the requested cycle has not been completed successfully. The assertion/de-assertion of **SysErrxMO** follows the same protocol as **SysAckxMO**, described above. The error response signal **SysErrxMO** is an optional feature and it is up to the designer of the SystemBus Slave Interface to implement it or alternatively ignore the error and terminate every cycle with **SysAckxMO**.

At the start of a new access cycle the SystemBus Master is permitted to assert **SysReqxMI** only if **SysAckxMO**/**SysErrxMO** has been driven to a low level by the Slave. **SysReqxMI** is always driven low for at least one clock cycle between two successive SystemBus cycles.

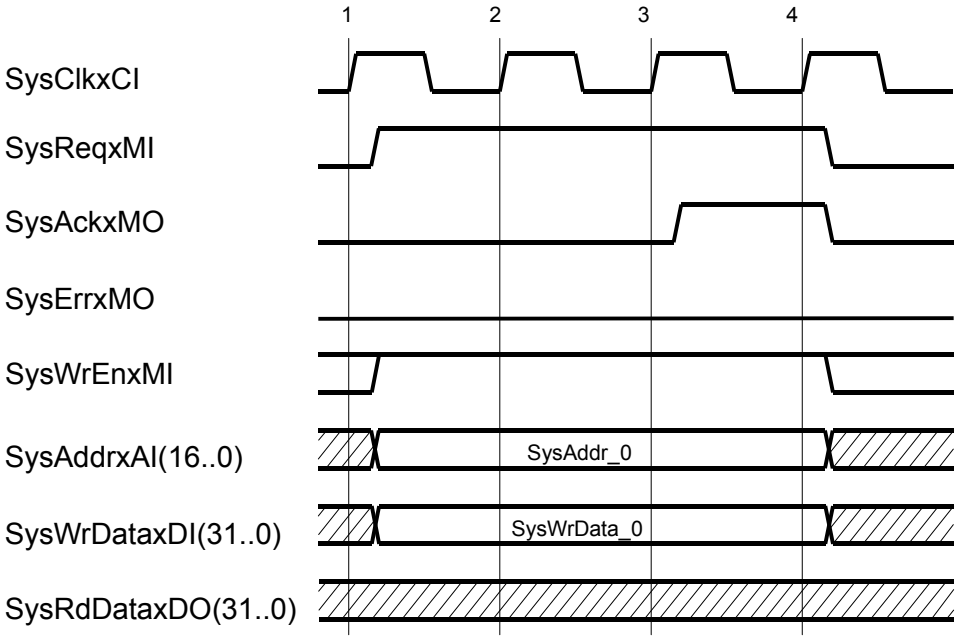

Figure 3: SystemBus Write Cycle

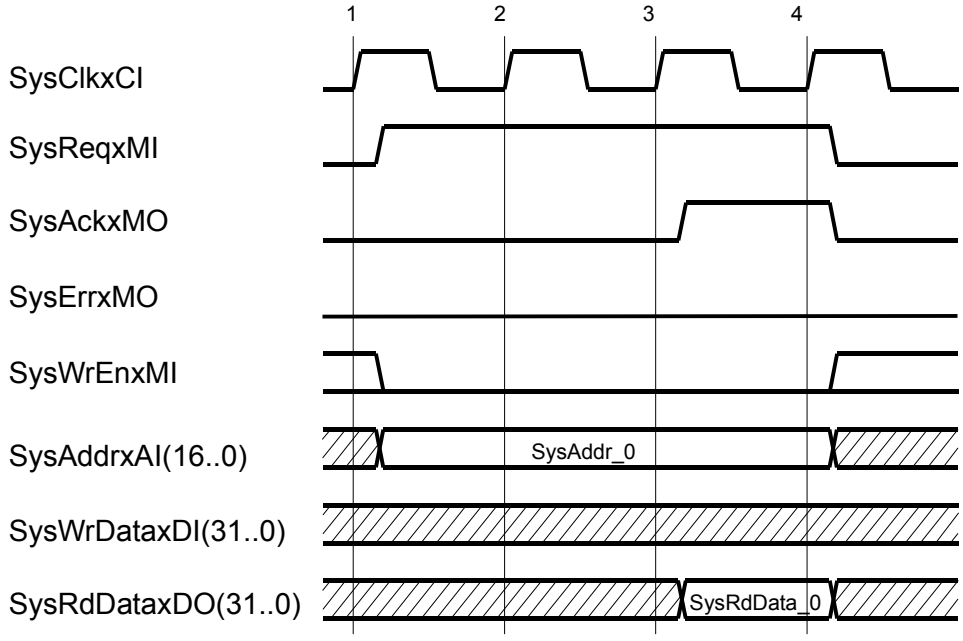

Figure 4: SystemBus Read Cycle

6.2.3 SystemBus Address Space

The internal SystemBus address space of the Custom Module can be freely allocated to module register or memory space. The **SysAddrxAI** bus is limited to 17 address lines which allows for a maximum address space of 128kByte. It is recommended (but not mandatory) to place control and status registers into the lower address range. If memory space is defined in the Module, it is recommended to place this in the upper portion of the available address space, i.e. at address 0x10000+

While the internal Module address space if free to be used by the application, one system requirement has to be implemented in every Module. A Custom Module ID status register must be located at Address 0 of the Module address space. This register must be present and must use a fixed bit format as shown below:

- Bit 0..9: **Custom-Module-ID**: A pre-determined system identifier is used to determine that the associated module within the larger CorSight2 environment belongs to a Custom Module. The **Custom-Module-ID** opcode is fixed at 47 (0x2F). Do not modify.
- Bit 10..20: **Custom-Version-ID**: The version number of the Custom Module design can be freely chosen by the User.
- Bit 21..31: **Custom-Design-ID**: Identifies the Custom Module design. The **Custom-Design-ID** can be defined by the User in the CustomPkg.vhd package file (see Chapter 6.4)

6.2.3.1 SystemBus Base Address

The SystemBus Interface of the Custom Module is controlled by the SystemBus Master, which is located within the FPGA Framework. The global address mapping for all CorSight2 SystemBus Slave interfaces is performed by the Master module which places the 128kByte address space for the Custom Module at address **0x300000 to 0x31FFFF**.

6.2.3.2 SystemBus DDR3-RAM Access

The DDR3-RAM which is attached to the Memory Bus of the Custom Module can be directly accessed by the Host via a dedicated SystemBus Slave interface. The DDR3-RAM Slave interface is part of the FPGA Framework. Therefore the Custom Module does not need to implement a SystemBus-to-MemoryBus Bridge function if Host access to the RAM is required. The DDR3-RAM is defined within the global CorSight2 address space at address **0x60000000 to 0x07FFFFFF**.

Please Note: CorSight2 has an internal address range of 2 GByte which is mapped onto the external 64 MByte Host address space via special SystemBus address map registers. All accesses to the internal address space above 0x1000000 (16 Mbyte) require setting a SystemBus Address Map Register followed by the read/write access to the mapped address space.

6.3 ImageBus Interface

The ImageBus is a synchronous, point-to-point connection between one ImageBus source device (Master) and one ImageBus destination device (Slave). The Custom Module receives pixel data from the Sensor Interface or the ImageBus Pipeline on the ImageBus Receive Interface. Pixel data generated in the Custom Module is transferred back to the FPGA Framework via the ImageBus Transmit Interface.

6.3.1 ImageBus Receive Port

The ImageBus Receive Interface consists of 4 signal groups:

- **ImgRxPixValxMI**: ImageBus Receive Pixel Valid. Driven by the ImageBus Master in the FPGA Framework to indicate a valid pixel transfer.
- **ImgRxPixRdyxMO**: ImageBus Receive Pixel Ready. Driven by the Custom Module to indicate that the Module is ready to receive pixel data. ImgRxPixRdyxMO is not used on CorSight2, tie permanently to '1'.
- **ImgRxPixStatxSI(2..0)**: ImageBus Receive Pixel Status. Driven by the ImageBus Master in FPGA Framework to indicate the status of the currently transferred pixel data.
- **ImgRxPixDataxDI(35..0)**: ImageBus Receive Pixel Data. Driven by the ImageBus Master in FPGA Framework to transfer pixel data from the ImageBus Master to the Custom Module.

6.3.2 ImageBus Transmit Port

The ImageBus Transmit Interface also consists of 4 signal groups:

- **ImgTxPixValxMO**: ImageBus Transmit Pixel Valid. Driven by the Custom Module to indicate a valid pixel transfer.
- **ImgTxPixRdyxMI**: ImageBus Transmit Pixel Ready. Driven by the FPGA Framework to indicate that the ImageBus Slave is ready to receive pixel data. ImgTxPixRdyxMI is not used on CorSight2, it is permanently tied to '1'.
- **ImgTxPixStatxSO(2..0)**: ImageBus Transmit Pixel Status. Driven by the Custom Module to indicate the status of the currently transferred pixel data.
- **ImgTxPixDataxDO(35..0)**: ImageBus Transmit Pixel Data. Driven by the Custom Module to transfer pixel data from the Custom Module to the ImageBus Receiver in the FPGA Framework.

Note: Since the ImageBus Receive and Transmit Interfaces are functionally identical, further references to their respective signal names are made in generic terms, i.e. **ImgPixVal**, **ImgPixRdy**, **ImgPixStat** and **ImgPixData**.

6.3.3 ImageBus Pixel Transfer

When an ImageBus Master wants to transfer pixel data, it asserts the pixel status (**ImgPixStat**) and data (**ImgPixData**) onto the ImageBus with the rising edge of **SysClkxCI**. At the same time the pixel valid signal (**ImgPixVal**) is asserted high. The Slave samples all signals and upon detecting **ImgPixVal** high, latches the **ImgPixStat** and **ImgPixData**. Since there is no flow control implemented in the ImageBus architecture (the **ImgPixRdy** signal is not used) the Slave device must be able to accept the pixel data unconditionally at any time. A pixel transfer is deemed to have taken place at the rising edge of **SysClkxCI** when the **ImgPixVal** signal is asserted. Please refer to Figure 5 as an illustration of valid pixel transfers. In this example the Master transfers 3 pixels in successive clock cycles.

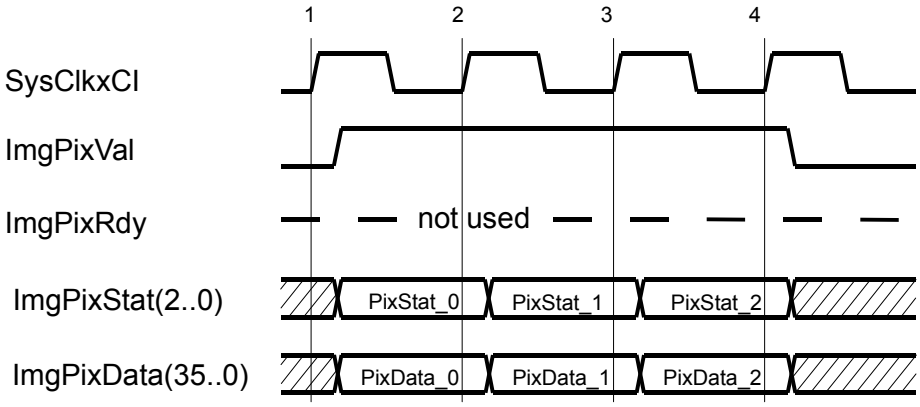

Figure 5: ImageBus Pixel Transfer

6.3.4 ImageBus Status Bus

The ImageBus Master must accompany every pixel data with a relevant status information to indicate the relative position of the data with a frame. **ImgPixStat** is a 3-bit bus and the decoding is shown in Table 7.

ImgPixStat(2..0)	Status Decoding
0	None (inside frame)
1	Start of Frame (SOF)
2	Start of Single Line Frame (SOSF) (LineScan)
3	(not used)
4	Start of Line (SOL)
5	Start of Last Line (SOLL)
6	End of Line (EOL)
7	Error (ERR)

Table 7: ImageBus Pixel Status Decoding

Figure 6 shows the positions of the pixel status as it appears in an image. The Start-of-Frame (SOF) status indicates the first pixel of an area scan image. If a linescan sensor is connected to the CorSight2 camera, SOF is replaced with the Start-of-Single-Line-Frame (SOSF) status. The first pixel of every other line is labelled as Start-of-Line until the last line is reached. The first pixel of the last line is referred to as Start-of-Last-Line (SOLL). The last pixel of every line is referred to as End-of-Line (EOL). The smallest possible area scan image is a 2x2 image.

In normal operating mode the ImgStatus sequence is repeated for each frame transferred over the ImageBus. However, if a transfer error occurs in the image pipeline the Master which is detecting the error must assert the Error (ERR) status as soon as the error occurs regardless of the current pixel position. Once an error status has been transmitted, the transfer of the remaining image is immediately aborted and the Master must then wait until the beginning of the next frame before commencing pixel transfers. In other words: After transmitting an Error status, the next valid pixel transfer must be either an SOF or SOSF status.

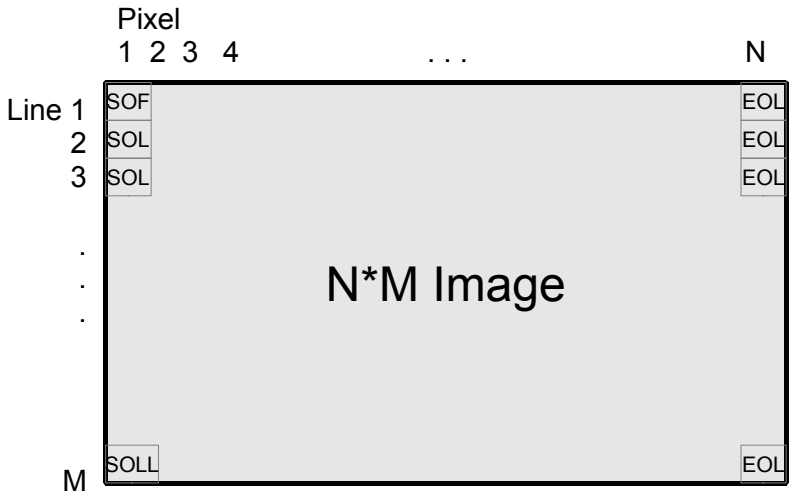

Figure 6: ImageBus Status Frame Sequence

6.3.5 ImageBus Data Bus

Pixel data is transferred over the 36-bit **ImgPixData** bus. For each valid pixel transfer (which is indicated by **ImgPixVal** = '1') **ImgPixData(35..0)** transports a single pixel from the ImageBus Master to the Slave device. The pixel format is application dependant and can either be 8-bit or 12-bit monochrome or 8-bit or 12-bit RGB. Pixel data is always MSB-aligned, i.e. monochrome pixel data is aligned to **ImgPixData(35)**. Figure 7 illustrates the bit allocation of pixel data on **ImgPixData(35..0)** for each pixel format.

Figure 7: ImageBus Pixel Data Formats

6.4 Custom Module Configuration

A couple of user configurable Custom Module design options are available in a VHDL package file:

<OpenCamDir>/src/Wrapper/CustomPkg.vhd

The user configurable options are:

- **Custom-Design-ID**: A mechanism to uniquely identify a new Custom Module design is provided via the Custom-Design-ID. Each distinct Custom design is allocated a unique ID number. Once a Module-ID has been allocated to a specific Custom Design it must not be re-allocated to any other Custom Module design.
  To facilitate this mechanism a list of already defined Design-ID constants is presented in CustomPkg.vhd. The User can add a new Design-ID to this list (which must be an integer number with a value > 1023) and reference the ID in the SystemBus Interface design. The Design-ID is part of the Custom Module ID status register, located at Address 0 of the Module address space (see Chapter 6.2.3).

- **Custom-Module-Bypass**: For debug purposes, the FPGA Framework contains a mechanism to include or exclude the Custom Module in the CorSight2 image flow. This is implemented

via an ImageBus multiplexer which can be dynamically controlled in Synview via the "Custom Module Bypass" feature [ON/OFF] (see Chapter 12.3).

Custom designs which do not require this bypass feature can remove the multiplexer at FPGA compile time by setting the cCUSTOM_BYPASS_EN constant in CustomPkg.vhd to "False".

6.5 GPIO Interface

The Custom Module is capable of controlling the GPIO output pins of the FPGA as well as reading the status of the GPIO input pins. While the GPIO input pins are always available to the Module, the GPIO outputs require a special control port to configure the pins. The Custom Module GPIO Interface consists of three ports, each being 6 bit wide. Bits(0..3) control GPIO OptoOut(0..3) and Bits(4..5) control the GPIO TTLOut(0..1) pins.

Please Note: In a standard CorSight2 camera the TTL(1) pin is not made available to an external GPIO pin.

- **CustIoCtrlxMO(5..0)**: Driven by the Custom Module and determines if a GPIO output pin is controlled by the GPIO Handler in the FPGA Framework (CustIoCtrlxMO(x) = 0) or by the Custom Module (CustIoCtrlxMO(x) = 1). CustIoCtrlxMO is usually driven by a control register defined in the SystemBus Interface of the Custom Module.
- **CustIoDataOutxDO(5..0)**: If CustIoCtrlxMO(x) is set to '1' CustIoDataOutxDO(x) determines the level of the associated Opto/TTL GPIO output pin of the FPGA. For the OptoOut pins a low level on CustIoDataOutxDO(x) means the photo-diode in the Opto driver is turned OFF, a high level turns the photo-diode ON.
- **CustIoDataInxDO(5..0):** Shows the current level of the Opto/TTL GPIO input pin.

For the control of the GPIO output pins it may be advisable to implement a firmware-based, fail-safe mechanism to allow Custom Module access to only those GPIO pins which the module designer intends of using but disables access to all others. Such a mechanism is implemented in the **LutDesign** example project. **LutPkg.vhd** (in **<OpenCamDir>/src/LutDesign)** contains a constant **cCUSTOM_GPIO_ENABLE** which permanently enables or disables access to selected GPIO outputs. However, implementing such a mechanism is entirely at the discretion of the designer.

6.6 MemoryBus Interface

The MemoryBus Interface consists of three separate interfaces which are functionally inter-connected but perform their assigned tasks independently of each other. It interfaces to the Memory Control Block (MCB) in the FPGA Framework which in turn interacts with the DDR3-RAM device. The MCB is a Xilinx IP block which handles the intricate timing and sequencing requirements of the DDR3-RAM interface.

6.6.1 MemoryBus Command Port

The MemoryBus Command Port is used to launch read or write requests to the MCB. A Write Command can be issued on the Command Port after write data has been transferred on the MemoryBus Write Port. For read operations a Read Command is issued followed by the transfer of the read data from the MCB on the MemoryBus Read Port.

The MemoryBus Command Port consists of the following signals:

- **px_cmd_en_o:** The MemoryBus Command Valid output signal is used to indicate to the MCB that a new command is valid and available on the **px_cmd_instr_o(2..0)**, **px_cmd_bl_o(5..0)** and **px_cmd_addr_o(29..0)** ports. A command is deemed to have been transferred to the MCB when **px_cmd_en_o** = '1' and **px_cmd_full_i** = '0'.
- **px_cmd_full_i:** When the MemoryBus Command Buffer Full input is asserted (i.e. '1') the MCB is not read to accept a new command. If the Custom Module has a command pending at that time (**px_cmd_en_o** = '1') it must hold this command until the MCB de-asserts **px_cmd_full_i**.
- **px_cmd_instr_o(2..0):** The MemoryBus Command Instruction output signal determines which command is to be processed in the current cycle. There are two possible commands:

  - Read Command: **px_cmd_instr_o(2..0)** = "001"

  - Write Command: **px_cmd_instr_o(2..0)** = "000"
- **px_cmd_bl_o(5..0):** The MemoryBus Command Burst Length output signal indicates how many (read or write) data transfers are involved in the current command cycle. **px_cmd_bl_o** is a zero-based signal which must be programmed with a value of:

  px_cmd_bl_o = Number-of-actual-data-transfers – 1

  A maximum of 64 32-bit data transfers (i.e. **px_cmd_bl_o(5..0)** = 63) can be processed per command.
- **px_cmd_addr_o(29..0):** The MemoryBus Command Address identifies the memory location of the first DWord involved in the (read or write) data burst transfer. **px_cmd_addr_o** is a byte address, however only 32-bit data transfers are allowed on the Memory Read and Write ports. As a consequence the two least significant address bits **px_cmd_addr_o(1..0)** must be "00" in all command cycles.

  The Custom Module on CorSight2 is connected to a 128MByte DDR3-RAM. The uppermost valid DDR3-RAM address is therefore **0x07FFFFFC**. Care must be taken by the Custom Module control logic when issuing burst commands accessing the upper RAM address ranges. No automatic wrap around is performed in the MCB if a burst transfer exceeds the upper address boundary.

Figure 8 shows a timing diagram of valid MemoryBus command cycles. At the rising clock edge of **SysClkxCI** (1) the Custom Module issues a Write Command by asserting **px_cmd_en_o** = '1' and by setting **px_cmd_instr_o(2..0)** = "000". At the same time **px_cmd_bl_o(5..0)** and **px_cmd_addr_o(29..0)** are set to appropriate values.

The MCB indicates at clock edge (2) that it is not ready to accept new commands (**px_cmd_full_i** = '1') which forces the Custom Module to hold the command for another clock cycle. At clock edge (3) the write command transfer succeeds since **px_cmd_en_o** = '1' and **px_cmd_full_i** = '0'. The Custom Module is now free to remove the command from the MemoryBus Command Port or to issue a new command. In Figure 8 the Write Command is followed by a Read Command at clock edge (3). Since **px_cmd_full_i** is still '0' at clock edge (4) the read command succeeds immediately and **px_cmd_en_o** is de-asserted by the Custom Module as there are no further commands pending at the time.

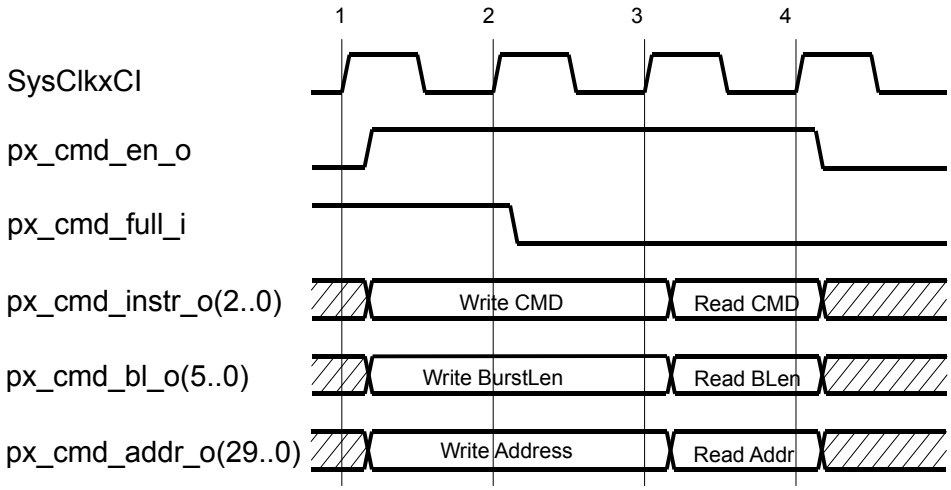

Figure 8: MemoryBus Command Cycle

6.6.2 MemoryBus Read Port

- **px_rd_en_o:** The MemoryBus Read Data Valid output signal is used by the Custom Module to indicate to the MCB that it is ready to receive data on the **px_rd_data_i(31..0)** port. When the Custom Module samples **px_rd_empty_i** = '0' and **px_rd_en_o** is set to '1' the read data transfer is deemed to have taken place and the Custom Module must capture the read data at this point.
- **px_rd_empty_i:** When the MemoryBus Read Data Buffer Empty input signal from the MCB is '0' it indicates that valid read data is available on **px_rd_data_i(31..0)**. If **px_rd_empty_i** = '1' the Read Data Buffer in the MCB is empty, i.e. no valid data is available.
- **px_rd_data_i(31..0):** The MemoryBus Read Data Bus input is used to transfer read data from the MCB to the Custom Module. Only 32-bit data transfers are supported.

Figure 9 shows two successive data read cycles being performed on the MemoryBus Read Port. At the rising edge of **SysClkxCI** (1) the Custom Module asserts **px_rd_en_o** to '1' to indicate that it is ready to receive read data from the MCB. The data had previously been requested by a Read Command on the MemoryBus Command Port. Note that the Read Command is not shown in Figure 9. Read data becomes available in the MCB at clock edge (2) when **px_rd_empty_i** is de-asserted to '0' and at the same time the data is presented on **px_rd_data_i(31..0)**. The Custom Module must capture the data whenever it samples **px_rd_en_o** = '1' and **px_rd_empty_i** = '0' which is the case at clock edge (3) and (4).

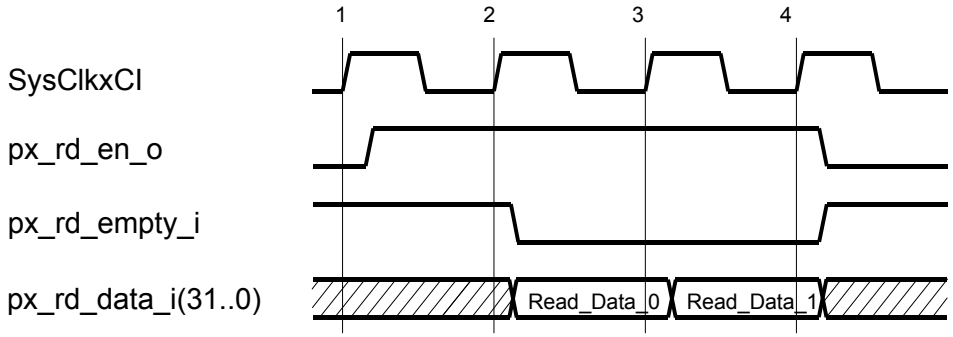

Figure 9: MemoryBus Read Cycle

6.6.3 MemoryBus Write Port

- **px_wr_en_o:** The MemoryBus Write Data Valid output signal is used by the Custom Module to indicate to the MCB that it has placed valid write data on the **px_wr_data_i(31..0)** port. When the Custom Module samples **px_wr_full_i** = '0' and **px_wr_en_o** is set to '1' the write data transfer is deemed to have taken place and the Custom Module is now free to replace the write data with a new data word. Please note that all relevant write data within a write burst cycle must be transferred to the MCB before the Custom Module is allowed to issue the corresponding Write Command on the MemoryBus Command Port.
- **px_wr_full_i:** When the MemoryBus Write Data Buffer Full input signal from the MCB is '0' it indicates that it is ready to accept write data on **px_wr_data_o(31..0)** port. If **px_wr_full_i** = '1' the Write Data Buffer in the MCB is full and the Custom Module must hold valid write data (if any) for at least one more clock cycle.
- **px_wr_data_o(31..0):** The MemoryBus Write Data Bus output is used to transfer write data from the Custom Module to the MCB. Only 32-bit data transfers are supported.

Figure 10 shows two successive data write cycles being performed on the MemoryBus Write Port. At the rising edge of **SysClkxCI** (1) the Custom Module asserts **px_wr_en_o** to '1' and at the same time presents 32-bit of write data on **px_wr_data_i(31..0)**. At clock edge (2) the Custom Module samples **px_wr_full_i** = '1' which means that the Write Data Buffer in the MCB is currently full and no write data transfers can be made at this time. The Custom Module must therefore hold **px_wr_en_o** = '1' and keep the write data on **px_wr_data_i(31..0)** valid. At clock edge (2) the MCB de-asserts **px_wr_full_i** to '0' which indicates that the MCB is now ready to capture the write data on the next rising clock edge (3). At clock edge (3) the Custom Module indicates that it wants to perform a second write data transfer by keeping the **px_wr_en_o** signal high and presenting new write data on **px_wr_data_i(31..0)**. Since **px_wr_full_i** is still '0' at clock edge (4) the second write data transfer is taking place. Once all write data transfers have for a given write burst cycle have been completed the Custom Module must issue a corresponding Write Command on the

MemoryBus Command Port. Note that the Write Command is not shown in Figure 10.

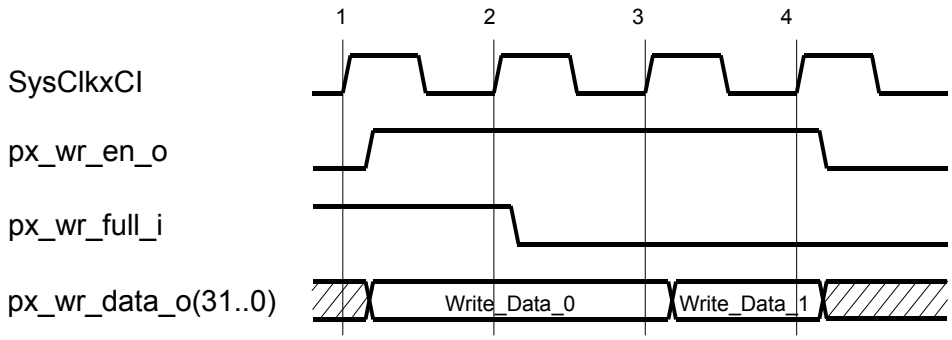

Figure 10: MemoryBus Write Cycle

7 Custom Module Example Designs

The CorSight2 OpenCamera Development Kit includes eight example designs which can be synthesized/implemented with Vivado 2017.3 as well as simulated using the Vivado Xsim simulator.

7.1 OpenCamera Default Design

The OpenCamera Default design contains an empty Custom Module used to compile a bare-bone Framework FPGA without any custom logic. There is no simulation testbench setup for the default design.

7.2 OpenCamera UserDesign

A new user application design for the Custom Module can be made with the aid of the User Design files, located in **<OpenCamDir>/src/UserDesign**. These files are provided to enable the user to quickly come up with an embedded custom design by copying and modifying the source files as required.

There are five VHDL files in this directory, which make up the bare bone User Design:

- **UserModule.vhd:** The top level module file, instantiates the UserDesign and UserSysBusIf components and includes the ImageBus output port FIFO/Buffer stage. A ChipScopePro Logic Analyzer can optionally be instantiated in this file to be used as a debug aid for the user application.
- **UserCtrl.vhd**: The entity and architecture body of the actual user application is a place holder for the user design logic. Instantiate the top-level user application design in this file.
- **UserSysBusIf.vhd:** SystemBus interface for the UserDesign. Implements the control/status register address allocation and register bit mapping. The control/status registers for the user application (defined in UserPkg.vhd) can be added in the SysBusProc process.
- **UserPkg.vhd:** Contains default constant and type definitions for the Custom Module. The user should modify this file to suit the requirements of the new user application design.
- **UserVerPkg.vhd:** Contains the version number of the user design. Do not manually alter this file as it is automatically generated during the FPGA compile process as per the command line options and the initialization parameters provided by the user..

7.3 OpenCamera LutDesign

To demonstrate a completed Custom Module design a Colour/Monochrome LUT example design is provided in **<OpenCamDir>/src/LutDesign**. This reference design can be synthesised with Xilinx Vivado and simulated with Xsim.

A SystemBus register map/description for the LUT design can be found in **<OpenCamDir>/doc/RegisterDescr_LutDesign.txt** and a sample control register setup procedure can be found in **<OpenCamDir>/sim/Simulation/Custom_LutDesign/SimCmd_Lut.ini**.

- **LutModule.vhd:** The top level LUT design file is equivalent to the UserModule.vhd file in the UserDesign directory. It instantiates the SystemBus interface (LutSysBusIf.vhd), the

actual LUT control entity (LutCtrl.vhd) and optionally a ChipScopePro Logic Analyzer.

- **LutCtrl.vhd:** Contains the LUT Controller, including the LUT BlockRAM memory instantiations. This file is the equivalent to the UserCtrl.vhd file in the UserDesign subdirectory.
- **LutSysBusIf.vhd:** Implements the SystemBus interface for the LUT design. It allocates system addresses to control and status registers defined in LutPkg.vhd. It also provides an interface to the SystemBus Memory Port, i.e. the LUT BlockRAM read/write port.
- **LutRAM.vhd:** Instantiates the actual LUT BlockRAM devices.
- **LutPkg.vhd:** Type and constant definition package for the LUT design. It defines the control and status registers for the LUT.
- **LutVerPkg.vhd:** Contains the version number of the LUT design. Do not manually alter this file as it is automatically generated during the FPGA compile process as per the command line options and the initialization parameters provided by the user.

7.4 OpenCamera MemTest Design

The MemTest design is located in **<OpenCamDir>/src/MemTest** which runs read and write operations to the DDR3-RAM and verifies the integrity of the DDR3-RAM interface. Can also be used as a test design to gauge maximum achievable read/write data bandwidth to the DDR3-RAM. MemTest does not involve ImageBus transfers, i.e. the ImageBus Transmit Port is directly connected to the ImageBus Receive Port.

A SystemBus register map/description for the Memory Test design can be found in **<OpenCamDir>/doc/RegisterDescr_MemTest.txt** and a sample control register setup procedure can be found in **<OpenCamDir>/sim/Simulation/Custom_MemTest/SimCmd_MemTest.ini**.

- **MemTestModule.vhd:** The top level MemTest design file is equivalent to the UserModule.vhd file in the UserDesign directory. It instantiates the SystemBus interface (MemTestSysBusIf.vhd), the actual MemTest control entity (MemTestCtrl.vhd) and optionally a ChipScopePro Logic Analyzer.
- **MemTestCtrl.vhd:** Contains the MemTest Controller. This file is the equivalent to the UserCtrl.vhd file in the UserDesign subdirectory.
- **MemTestSysBusIf.vhd:** Implements the SystemBus interface for the MemTest design. It allocates system addresses to control and status registers defined in MemTestPkg.vhd.
- **MemTestPkg.vhd:** Type and constant definition package for the MemTest design. It defines the control and status registers for MemTest.
- **MemTestVerPkg.vhd:** Contains the version number of the MemTest design. Do not manually alter this file as it is automatically generated during the FPGA compile process as per the command line options and the initialization parameters provided by the user..

7.5 OpenCamera DiffPic Design

Located in **<OpenCamDir>/src/DiffPic**, the DiffPic example design calculates the pixel difference between successive monochrome images. When running image transfers a frame is written to the DDR3-RAM and retrieved at the start of the next frame. The pixel data of the incoming frame is compared with the pixel data of the stored frame and the difference value is transmitted via the ImageBus Transmit Port. The first frame of a frame sequence (i.e. when image transfers have first

been enabled) produced on the ImageBus Transmit port is the same as the input frame.

A SystemBus register map/description for the DiffPic design can be found in **<OpenCamDir>/doc/ RegisterDescr_DiffPic.txt** and a sample control register setup procedure can be found in **<OpenCamDir>/sim/Simulation/Custom_DiffPic/SimCmd_DiffPic.ini**.

7.6 OpenCamera Canny Filter Design

The Canny Filter design is located in **<OpenCamDir>/src/CannyFilter**. This is a Verilog design which can be synthesized with Vivado and simulated with Xsim. **CustomModule_cs2.v** is the top level design file for the Canny Filter design.

A SystemBus register map/description for the Canny Filter design can be found in **<OpenCamDir>/doc/ RegisterDescr_Canny.txt** and a sample control register setup procedure can be found in **<OpenCamDir>/sim/Simulation/Custom_Canny/SimCmd_Canny.ini**.

7.7 OpenCamera BlockDemo Design

The RGB block pattern overlay design is located in **<OpenCamDir>/src/BlockDemo**. This Verilog design can be synthesized with Vivado. No simulation testbench is provided. **blockDemo_cm.v** is the top level design file for this design.

A SystemBus register map/description for the Canny Filter design can be found in **<OpenCamDir>/doc/ RegisterDescr_BlockDemo.txt**.

7.8 OpenCamera ScaleD Design

The image scaler design is located in **<OpenCamDir>/src/ScaleD**. This design can be synthesized with Vivado and simulated with Xsim. **scale_d_cm.vhd** is the top level design file for the scaler design.

A SystemBus register map/description for the Image Scaler design can be found in **<OpenCamDir>/doc/ RegisterDescr_ScaleD.txt** and a sample control register setup procedure can be found in **<OpenCamDir>/sim/Simulation/Custom_ScaleD/SimCmd_ScaleD.ini**.

8 Custom Module Simulation

The OpenCamera simulation environment provides system-level simulation functions by performing Module initialisation via the SystemBus followed by image transfers in and out of the Module via the ImageBus. These functions are user-programmable by modifying configuration and command script files.

8.1 Simulation File Structure

The **<OpenCamDir>/sim** directory contains files for simulating the OpenCamera example designs and provides a generic simulation test bench for all designs. It is subdivided into 2 categories:

- **TestBench:** Contains the test bench source files for simulation. The test bench configures the UUT (Unit-Under-Test) via the SystemBus to setup internal control registers and transfers test images to the UUT via the ImageBus. The ImageBus output stream can be compared on a pixel-by-pixel basis if a known "good" image is available.
- **Simulation:** Contains files for the simulation runtime environment. There are 6 designs available for simulation:
  - **Custom_UserDesign**
  - **Custom_LutDesign**
  - **Custom_MemTest**
  - **Custom_DiffPic**
  - **Custom_Canny**
  - **Custom_ScaleD**

  Each design has their own test bench configuration files **TestBench_xxx.ini** and **SimCmd_xxx.ini** (with "**xxx**" being a reference to the design name). Simulations can be run with the Vivado Xsim simulator. The relevant simulation runtime files are stored in the Xsim sub-directories.

  Please Note: No simulation testbench is provided for the **BlockDemo** example design.

8.1.1 Simulation TestBench Files

The test bench controls the overall simulation flow and provides stimulus functions to the Unit-Under-Test. The **<OpenCamDir>/sim/TestBench** directory contains the simulation test bench source files. At the start of simulation the test bench reads the configuration files **TestBench_xxx.ini** and **SimCmd_xxx.ini** to determine the simulation setup and control flow. External image files (in pgm ASCII format) can be read into the test bench and be used as stimulus for the UUT. The following files make up the OpenCamera test bench:

8.1.1.1 Test Bench Source Files

- **CustomModuleTB.vhd:** Top level test bench file. Instantiates the UUT and the SystemBus and ImageBus Stimulus Modules as well as the back-end ImageBus Analyzer. The Analyzer can automatically compare the image output from the UUT with known image files.
- **ImgBusStimulus.vhd:** Reads a sequence of pgm image files and streams the images via the ImageBus to the UUT.

- **SysBusStimulus.vhd:** Reads the **SimCmd_xxx.ini** file to initialize the UUT control registers and to coordinate the overall simulation flow.
- **SimParaInit.vhd:** Reads the **TestBench_xxx.ini** file and sets internal test bench registers.
- **ImgBusAnalyzer.vhd:** Reads a sequence of image comparison files if comparison mode is enabled (selectable via the **TestBench_xxx.ini** file). It also checks the consistency (i.e. line and frame length and status sequence) of the UUT image output.

8.1.1.2 Test Bench Configuration Files

The Custom Test Bench is highly parameterised to allow for a large variety of simulation scenarios.

- **CustomFctLib.vhd:** Contains a collection of VHDL functions used in various test bench modules.
- **CustomSimLib.vhd:** Defines internal data structures and type/constant definitions used throughout the test bench.
- **CustomCfgLib.vhd:** Contains test bench configuration constants, such as the test bench log file name. It also specifies log settings to determine the extend of test bench progress and error messages being printed to the screen and to a log file. CustomCfgLib.vhd references the CfgModule package **CustomCfg_xxx.vhd** (with "**xxx**" being a reference to the design name) which determines which design is being simulated.
- **CustomCfg_xxx.vhd** determine the TestBench configuration for each OpenCamera example design. It defines the name of the **TestBench_xxx.ini** file read by the TestBench at the start of simulation. A simulation compile script (i.e. project file) must only include one of these files. If a new user design is added to the test bench, copy, rename and modify a **CustomCfg_xxx.vhd** file and reference the new file in the simulation project file.

8.2 Simulation Run-time Files

The **<OpenCamDir>/sim/Simulation** directory contains the following files and subdirectories:

- **Images:** Contains sample stimulus and comparison image files for simulation which are read by the ImageBus Stimulus and Analyzer Modules. The user can add their own image files to this directory and specify the file names in **TestBench_xxx.ini**.
- **Custom_xxx directories:** Contain the simulation files for each OpenCamera example design. "**xxx**" is a reference to the design name.

The **<OpenCamDir>/sim/Simulation/Custom_xxx** directories contain a **Xsim** subdirectory which contain simulation files required to setup and run a simulation using one of the two supported simulation engines. It is recommended to copy the **TestBench_xxx.ini** and **SimCmd_xxx.ini** files as well as the relevant XSim files into a separate directory and to create a simulation project at that location.

- **SimCmd_xxx.ini:** The Simulation Command file is read by the SystemBus Stimulus Module to initialize control registers in the UUT and to interactively control the simulation flow. The SystemBus Stimulus Module initiates frame transfers by communicating with the ImageBus Stimulus Module based on commands found in **SimCmd_xxx.ini**. It can also adjust the simulation flow by querying dedicated signals in the UUT or delay execution of simulation commands by inserting wait states.

- **TestBench_xxx.ini:** Contains test bench setup information which is read by the SimParaInit Module at the start of simulation. Parameters such pixel transfer rates, image size, image output and comparison information are contained in this file.

8.2.1.1 Xsim Files

Located in **<OpenCamDir>/sim/Simulation/Custom_xxx/Xsim** (with "**xxx**" being a reference to the design name), the following files are used to setup an Xsim simulation run. "xxx" denotes the design name:

- **Custom_xxx_Simulation.bat:** Executed from a Command Window under MS Windows this batch file calls the TCL file **Custom_xxx_Simulation.tcl** located in the same directory.
- **Custom_xxx_Simulation.tcl:** Invokes the Vivado Xsim simulator and runs a simulation of the associated example design.

8.3 Test Bench Initialisation File

Each custom design requires a test bench parameter configuration file which is named **TestBench_xxx.ini**, where "**xxx**" stands for the name of the design being simulated.

The Test Bench is initialised at the beginning of the simulation to suit the requirements of the application. The relevant parameters are read from the TestBench initialisation file and set by the SimParaInit Module. To ensure the test bench finds and opens the correct file, the **<OpenCamDir>/sim/TestBench/CustomCfg_xxx.vhd** file contains a link to this file.

TestBench_xxx.ini is subdivided into up to 5 sections which handle different aspects of the simulation run. Each section contains a defined set of parameters. Not all sections or parameters have to be listed in the initialisation file. If a section/parameter is not included the associated parameter(s) are set to default values. Section and parameter names (i.e. keywords) are case-insensitive. Sections are defined and limited by a section name which is entered in square brackets:

[SectionName]

The defined section names are:

- **[Simulation]:** Defines global simulation run parameters.
- **[ImageMode]:** Defines simulation parameters associated with the image transfers in and out of the UUT.
- **[ImageInput]:** Defines a Region-of-Interest into the input images as well as a list of image file names. All image files are located in the **<OpenCamDir>/sim/Simulation/Images** directory.
- **[ImageCompare]:** If the test bench is used to validate the pixel output stream from the UUT reference images must be provided and stored in **<OpenCamDir>/sim/Simulation/Images**. For each UUT input image a corresponding comparison image must be made available.
- **[ImageOutput]**: To (optionally) capture the image output of the UUT to a file, the name(s) and format of the image file are defined in this parameter section.

Within each section, parameters are entered in a standard initialisation file notation format with the following syntax:

<ParameterKeyword>=<Value>

The sections and associated parameters are described in the following paragraphs. Please refer to an

existing **TestBench_xxx.ini** file to illustrate the exact syntax and usage for each parameter.

8.3.1.1 Simulation Section Parameters

- **CommandFile:** Provides an alternative way of defining the name of a Simulation Command File. The **<OpenCamDir>/sim/TestBench/CustomCfgLib.vhd** file provides a (static) method of defining a reference to the **SimCmd_xxx.ini** file. This definition can be overridden by including the CommandFile parameter in the TestBench Parameter Initialisation file. The advantage of using a dynamically defined command file is that the test bench does not have to be recompiled if a new file is to be used for the next simulation run.
- **ResetPeriod:** At simulation start-up the Reset input to the UUT and test bench is automatically asserted. The default ResetPeriod value, which applies when this parameter is not specified is: 25 us. The duration of the reset period can be set by the ResetPeriod parameter. It is specified as:

  ResetPeriod=<Value><Time Unit>

  <Value>: is a decimal or hexadecimal (Prefix: 0x) integer number which defines the length of the Reset period.

  <Time Unit>: Allowed time units are: “ns”, “us”, “ms” and “sec”.
- **MaxSimTime:** The maximum simulation run time can be set with the MaxSimTime parameter. Once the simulation reaches the specified simulation time it is automatically aborted. The default MaxSimTime value, which applies when this parameter is not specified is: 0 ms. In this case the simulation run time must be set in the simulator itself, or alternatively the simulation runs for an indefinite period of time.MaxSimTime is specified as:

  MaxSimTime=<Value><Time Unit>

  <Value>: is a decimal or hexadecimal (Prefix: 0x) integer number which defines the maximal length of the simulation period.

  <Time Unit>: Allowed time units are: “ns”, “us”, “ms” and “sec”.

8.3.1.2 ImageMode Section Parameters

- **SingleFrm:** Enables single frame transfer mode in the ImageBus Stimulus Module. Permissible values: [0, 1]. When a **StartImgBus** command in the Simulation Script is executed the SingleFrm parameter determines if a single frame (SingleFrm = 1) or a frame sequence (SingleFrm = 0) is transferred to the UUT. If SingleFrm = 0 frames are being transferred for the duration of the simulation run or until a **StopImgBus** command is executed or until the last image in the **SrcImgFile** image file list has been transferred and **RepeatImgSeq** = 0.
- **RepeatImgSeq:** If RepeatImgSeq = 0 the ImageBus Stimulus Module stops frame transfers when the last frame of the **SrcImgFile** image file list has been transferred. If RepeatImgSeq = 1 the frame sequence is restarted at the beginning of the image file list.
- **PixInFreq:** Defines the pixel frequency with which pixels are transferred from the ImageBus Stimulus Module into the UUT. PixInFreq is specified as an integer in the range [0..150].

- **PixOutFreq:** Defines the maximum pixel frequency with which the ImageBus Analyser Module accepts image transfers from the UUT. PixOutFreq is specified as an integer in the range [0..150]. Please note that PixOutFreq must be set to 150 if the UserDesign does not handle the ImageBus flow control mechanism, i.e. if the constant cIMG_HANDSHAKE_EN is set to “False”.
- **PixWaitSeq:** To insert periodic pixel transfer wait states by the ImageBus Analyzer Module PixWaitSeq can be specified as a quotient of **<NumWaitStates>/<NumPixels>**, which means that the ImageBus Analyzer inserts <NumWaitStates> every <NumPixels> pixel transfers. PixWaitSeq provides an alternative to **PixOutFreq** to throttle the pixel transfer rate of the UUT to better simulate wait state behaviour of the UUT.
- **ScanLineMode:** When the simulation runs in scanline mode (ScanLineMode = 1) each line of an image transfer is considered a frame. The ImageBus Stimulus Module issues a “Start Of Single-Line Frame” opcode at the beginning of each line, otherwise a “Start-of-Frame”/”Start-of-Line”/”Start-of-Last-Line” sequence is generated.
- **HorSync:** Specifies the number of blank pixel transfers which are inserted by the ImageBus Stimulus Module at the end of every line. During the horizontal blanking period the “Pixel-Valid” output signal from Stimulus Module (ImgRxPixValxMO) is remains deasserted.
- **VerSync:** Specifies how many blank lines are inserted by the ImageBus Stimulus Module at the end of every frame. During the vertical blanking period the “Pixel-Valid” output signal from Stimulus Module (ImgRxPixValxMO) is remains deasserted.

8.3.1.3 ImageInput Section Parameters

- **StartPixel:** Defines the left edge of a Region-of-Interest imposed on the image source file, specified in **SrcImgFile**. The ImageBus Stimulus Module starts line transfers at the pixel position specified by StartPixel. To start with the first pixel in a line, set StartPixel = 0.
- **StartLine:** Defines the top edge of a Region-of-Interest imposed on the image source file, specified in **SrcImgFile**. The ImageBus Stimulus Module starts frame transfers at the line specified by StartLine. To start with the first line in a frame, set StartLine = 0.
- **NumPixel:** Defines the number of pixels per line to be transferred within the Region-of-Interest of the image source file. Please note that every image listed in SrcImgFile must have a minimum line length of (StartPixel + NumPixel) pixels.
- **NumLine:** Defines the number of lines per frame to be transferred within the Region-of-Interest of the image source file. Please note that every image listed in **SrcImgFile** must have a minimum frame length of (StartLine + NumLine) lines.
- **SrcImgFile:** Defines a list of image source files used by the ImageBus Stimulus to generate the pixel stimulus to the UUT. The file list is executed from top to bottom during the simulation. If the transfer of the last image is completed, image transfers either stop at this point (if **RepeatImgSeq** = 0) or restart from the top (if **RepeatImgSeq** = 1). Image source files must be provided in pgm ASCII format. Each image source file name can be up to 256 characters long (including the path name). The file path is rooted in the current simulation directory.

8.3.1.4 ImageCompare Section Parameters

- **CmpImgPixel:** Enables the comparison of the pixel output stream from the UUT with known good images. Permissible values: [0, 1]. When CmpImgPixel = 1 the ImageBus Analyser reads comparison image files listed in **CmpImgFile** and compares the ImageBus output of the UUT with the pixel sequence found in the files. Discrepancies are reported as errors on the console and in the test bench log file.
- **CmpImgFormat:** Enables Image Format Checking by the ImageBus Analyser. Permissible values: [0, 1]. Image Format Checking uses the ImageBus status signals of the UUT to verify that a correct pixel status sequence is generated. For example, the first pixel of a frame must be either a SOF (Start-of-Frame) or SOSF (Start-of-Single-Line-Frame) opcode followed by 0 or more NOP (No Opcode), followed by EOL (End-of-Line) etc. If an illegal sequence is detected an error message is printed on the console and in the log file. Image Format Checking also checks for the correct line length of an image transfer by counting the number of pixels of the first line in the first frame of the simulation and comparing it to all subsequent line transfers. Similarly, the frame length is determined by counting the number of lines in the first frame and comparing it to all subsequent frame transfers. Any mismatches between the actual and expected line/frame length are flagged as errors.
- **CmpImgPixErr**: Determines the maximum number of comparison error messages printed to the console or to the test bench log file during the simulation run. Once <CmpImgPixErr> errors have been detected no further messages are displayed.
- **CmpImgAbort:** Works in conjunction with **CmpImgPixErr** to abort the simulation when the maximum number of pixel comparison errors have been detected.
- **CmpImgFile:** Defines a list of image comparison files used by the ImageBus Analyser. CmpImgFile must contain a list of file names which correspond to the image source file list defined in **SrcImgFile** . The comparison file list is executed from top to bottom. If the transfer of the last image is completed the list is restarted from the top. Image source files must be provided in pgm ASCII. Each comparison file name can be up to 256 characters long (including the path name). The file path is rooted in the current simulation directory.

8.3.1.5 ImageOutput Section Parameters

- **OutImgEnable:** The test bench can save the entire pixel transfer sequence from the UUT ImageBus output to a file. If OutImgEnable is set to 1 files are generated to log the UUT pixel stream.
- **OutImgType:** Defines the UUT pixel output file type and format. OutImgType is a string parameter to select between binary or ASCII pgm/ppm files. Permissible values are:

  Xsim: “ascii_pgm”, “ascii_ppm”.
- **OutImgFile:** Defines the UUT image output file name. OutImgFile is a string parameter of up to 256 characters long (including the path name). Each output frame generated by the test bench is stored in a different file, the name of which is specified as:

  <OutImFile>_<Format>_<Num>.<Type>

  <Format>: either “bin” or “ascii”, as specified in **OutImgType**.

  <Num>: Sequential image number.

  <Type>: either “pgm” or “ppm”, as specified in **OutImgType**.

8.4 Simulation Command Script File

The Simulation Command Script file (**SimCmd_xxx.ini**) contains initialisation and simulation flow commands which are executed during the simulation run. After the test bench has completed the power-up reset cycle, the simulation commands are executed in sequential fashion. The following commands are available to customize the UUT simulation:

- **SysRead:** Performs a SystemBus read operation of a single register/memory location within the UUT. When the read operation is completed the returned data value and address are displayed on the console/log file. The command is specified as:

  SysRead <Address>

  <Address>: is a 32-bit address pointing to the required UUT location in either decimal or hexadecimal (Prefix: 0x) format. Leading zeros may be omitted. The base address of the Custom Module is fixed at address **0x300000**.

- **SysWrite:** Performs a SystemBus write operation to a single register/memory location within the UUT. Write operations can be monitored on the console or the log file if the appropriate **LogRegInit** / **LogMemInit** log flags are set. The command is specified as:

  SysWrite <Address> <WrData>

  <Address>: is a 32-bit address pointing to the required UUT location in either decimal or hexadecimal (Prefix: 0x) format. Leading zeros may be omitted. The base address of the Custom Module is fixed at address **0x300000**.

  <WrData>: is a 32-bit data word containing the required write data. Defined in either decimal or hexadecimal (Prefix: 0x) format. Leading zeros may be omitted.

- **SysVerify:** Performs a data verify operation on a single register/memory location within the UUT. SysVerify can be used to check if the register or memory location contains an expected data bit pattern. It performs a read operation from the UUT and compares the returned data word with an expected data word. The comparison is performed on a bit-by-bit basis using a mask to exclude data bits which are of no interest in the operation. The command is specified as:

  SysVerify <Address> <ExpData> <Mask>

  <Address>: is a 32-bit address pointing to the required UUT location in either decimal or hexadecimal (Prefix: 0x) format. Leading zeros may be omitted. The base address of the Custom Module is fixed at address **0x300000**.

  <ExpData>: is a 32-bit data word containing the expected comparison data. Defined in either decimal or hexadecimal (Prefix: 0x) format. Leading zeros may be omitted.

  <Mask>: is a 32-bit data word containing the verify data mask. Mask bits which are set to '1' enable the corresponding data bit for comparison. Defined in either decimal or hexadecimal (Prefix: 0x) format. Leading zeros may be omitted.

- **Wait:** The Wait command instructs the test bench to suspend Simulation Script command processing for a defined period of time. The command is specified as:

  Wait <Value><Time Unit>

  <Value>: is a decimal or hexadecimal (Prefix: 0x) integer number which defines the length of the Wait period.

  <Time Unit>: defines the time unit of the Wait command. Permissible values: “ns”,

"us", "ms" and "sec".

- **ImgBusReset:** Performs an ImageBus reset. When executing ImgBusReset the ImgRstxRI inputs to the UUT, ImageBus Stimulus and Analyser Modules are asserted for 2 clock cycles. Active frame transfers are immediately aborted and the entire ImageBus structure returns to idle mode.

  Please Note: The **ImgRstxRI** input of the UUT **must not** be used to reset UUT-internal SystemBus control or configuration registers. SystemBus registers can only be reset during the power-up reset phase.

- **StartImgBus:** Instructs the ImageBus Stimulus Module to start a new frame transfer using the next available image defined in **SrcImgFile**. If the **SingleFrm SingleFrm** parameter is set only a single frame is transferred, otherwise frame transfers continue until one of the following conditions are met:

  1) The **StopImgBus** command is executed.

  2) The **AbortImgBus** command is executed.

  3) The end of the **SrcImgFile** list is reached and **RepeatImgSeq** is reset (0).

  4) The end of the simulation run is reached.

- **StopImgBus:** Instructs the ImageBus Stimulus Module to stop frame transfers at the end of the currently active frame. StopImgBus does not need to be executed if the **SingleFrm** parameter is set.

- **AbortImgBus:** Aborts the currently active ImageBus frame transfer by sending an Abort signal to the ImageBus Stimulus Module. When detecting an active Abort signal, the Stimulus Module immediately stops transferring pixels to the UUT and returns to the Idle state, awaiting new commands. AbortImgBus can be useful to simulate the recovery behaviour of the UUT if an unexpected image transfer condition arises.

- **EventSignal:** This command is used to synchronise the execution of the Simulation Script file with the internal processing status of the UserDesign. During simulation the test bench executes the Simulation Script commands and at the same time monitors the EventPort signals, which monitor the progress of the simulation. When the "EventSignal" command is encountered the test bench checks the status of the specified EventPort signal and stops script execution until the required signal status is detected.

  The EventSignal command is specified as:

  EventSignal(<Index>) <Event>

  <Index> defines which EventPort signal is being queried. Must be an integer value within the range of [0..31].

  <Event> defines the status of the EventPort signal which must be detected by the test bench before the Simulation Script Command processing resumes. <Event> is a string variable which can take on the following values:

  - **Rising**: Wait until the next rising edge of EventPort(Index).

  - **Falling**: Wait until the next falling edge of EventPort(Index).

  - **High**: Wait until EventPort(Index) is high.

  - **Low**: Wait until EventPort(Index) is low.

- **EventBus:** works similar to the **EventSignal** command. It is used to synchronise the execution of the Simulation Script File with the internal processing status of the UserDesign. The EventBus command allows a consecutive range of EventPort signals to be compared with a defined value. When the comparison condition is met execution of Simulation Script commands resumes.

  The EventBus command is specified as:

  EventBus(<HighIndex>..<LowIndex>) <Operation> <Value>

  <HighIndex> defines the upper EventPort(31..0) signal included in the EventBus command. Must be an integer value [0..31], with HighIndex > LowIndex.

  <LowIndex> defines the lower EventPort(31..0) signal included in the EventBus command. Must be an integer value with a range of [0..31], with HighIndex > LowIndex.

  <Operation> defines the comparison operation which is performed by the EventBus command. <Operation> is a string variable which can take on the following values:

  - **EQ**: Wait until EventPort(HighIndex .. LowIndex) is equal to <Value>.
  - **NE**: Wait until EventPort(HighIndex .. LowIndex) is not equal to <Value>.

  <Value> defines the comparison value used in the EventBus command.. It can be a decimal or hexadecimal (prefix 0x) number.

8.5 Test Bench Log Messages

When running a simulation, the test bench can generate a large amount of log messages to indicate simulation progress or errors. To limit the amount of messages it is possible to configure the log mechanism to suit individual requirements.

The test bench configuration file **<OpenCamDir>/sim/TestBench/CustomCfgLib.vhd** contains the enumeration constant **cLOG_ITEM_ARRAY** which can be modified by the user. The type definition **tLOG_ITEM_LIST** lists all available log message categories. By including (or excluding) log items the amount of messages can be reduced or expanded. The following log message flags are available:

- **LogError**: Generates messages related to simulation setup errors, such incompatible image file types etc.
- **LogExec**: Generates messages when main test bench modules start execution. Useful for tracking test bench errors resulting in simulation crashes.
- **LogReturn**: Generates messages when main test bench modules finish execution. Useful for tracking test bench errors resulting in simulation crashes.
- **LogImgHeader**: Displays image file header information when read by the test bench.
- **LogSimInit**: Displays parameters contained in the test bench parameter configuration file **TestBench_xxx.ini** as they are read by the test bench.
- **LogRegInit**: Displays messages when performing SystemBus write operations to the UUT register space.
- **LogMemInit**: Displays messages when performing SystemBus write operations to the UUT memory space.
- **LogWait**: Displays status messages when the test bench performs Wait commands listed in the Simulation Command Script file.
- **LogFlowCtl**: Displays messages when certain events occur during the simulation run, such as reading image files, starting or stopping frame transfers etc.
- **LogEvtWait**: Displays messages when the test bench enters or exits wait states associated with the EventSignal/EventBus commands.
- **LogAnalyzer**: Displays image comparison progress messages associated with the ImageBus Analyser.

8.6 Xsim Simulator

To simulate the OpenCamera example designs with Xsim, TCL scripts as well as MS-Windows batch files are provided for each example design. These scripts/batch files generate a new Vivado project, compile the requested example design and automatically run the simulation.

8.6.1 Xsim TCL Simulation

To run an OpenCamera Xsim simulation within a Vivado TCL shell window, open a Vivado TCL shell session and cd to:

<OpenCamDir>/sim/Simulation/Custom_xxx/Xsim

On the TCL command line issue the following command to run the simulation:

source Custom_xxx_Simulation.tcl

8.6.2 Xsim Windows Simulation

To run an OpenCamera Xsim simulation on MS-Windows open a Command Window and cd to:

<OpenCamDir>\sim\Simulation\Custom_xxx\Xsim

Execute the **Custom_xxx_Simulation.bat** batch file to run the simulation.

9 OpenCamera FPGA Compilation

The entire CorSight2 FPGA, including the Framework and Custom Module design is compiled when running the OpenCamera FPGA compilation scripts. Two compile options are provided with the OpenCamera Development Kit:

- **TCL Compile Script**
- **Windows Batch Script**

9.1 OpenCamera TCL Compile Script

The OpenCamera FPGA compilation scripts are fully TCL based, which can be executed using the Vivado TCL shell environment. The TCL compile option runs independent of the underlying operating system. This option must be selected when using a Linux OS. To run an FPGA compilation follow the steps described below:

1. Update the **<OpenCamDir>/syn/InitFiles/CS2_xxx_Init.tcl** file ("**xxx**" being a reference to the design name) with the required Custom Module synthesis and implementation parameters (see Chapter 3.4).

2. Open the **Vivado 2017.3 TCL Shell** and cd to **<OpenCamDir>/CompileTools**.

3. Run the following command to invoke Vivado. All required command arguments must be provided in the order indicated below.

   **exec vivado -source Start_OpenCam_FPGA.tcl -tclargs **
   <CustomID> <DesignID> <RevisionID> <CompMode>

   <CustomID> (required) = **[Default, UserDesign, LutDesign, MemTest, DiffPic, Canny, BlockDemo, ScaleD]**
   <DesignID> (required) = **[DSG00, DSG01, DSG02]**
   <RevisionID> (required) = **[REV01, REV02, REV03]**
   <CompMode> (optional) = **[Release, Debug] - Default Value: Release**

   The **<CustomID>** argument selects which Custom Module is to be compiled and **<DesignID>** selects the FPGA Framework Design within which the Custom Module is embedded (see Chapter 3.1 and Chapter 4.1, respectively).

   <RevisionID> selects which CorSight2 board revision the compilation is targeting. If the board revision of the actual CorSight2 target board does not match **<RevisionID>** the resulting FPGA firmware file can not be loaded into the CorSight2 Flash device (see Item 6 below).

   The compile mode argument **<CompMode>** is optional and determines if the FPGA compilation is performed in **"Release"** or **"Debug"** mode. The default mode is **"Release"**. The **"Debug"** option is ignored for the **"Default"**, **"DiffPic", "Canny", "BlockDemo"** and **"ScaleD"** modules since there is no debug logic present in these designs.

   When Debug Mode is enabled (**UserDesign**, **LutDesign** and **MemTest** only) any debug logic (i.e. ChipScopePro or Vivado Logic Analyzer or other discrete debug logic) which the

designer may have included in the Custom Module is instantiated in the FPGA compilation process.

Debug logic is enabled or disabled in the HDL source files via the **cCUSTOM_DEBUG_ENABLE** constant defined in the Version package file **XXX_VerPkg.vhd**. The Version package file is automatically generated by the OpenCamera FPGA compile scripts which uses the **<CompMode>** parameter to set the constant value.

4. The Vivado GUI opens and compiles the specified OpenCamera design.

5. If the compilation is successful a binary firmware file is created in the OpenCamera Release directory. The location of the release directory is specified by the **Cs2OpenCamCfg (ReleaseDir)** environment variable which is defined in the file **<OpenCamDir>/Compile Tools/Set_Customer_EnvVar.tcl**.

   Each distinct OpenCamera FPGA compilation will be allocated a dedicated subdirectory within the release directory. There are two levels of subdirectories:
   - The CorSight2 Board Revision and Firmware version numbers are included in the upper subdirectory level, for example: **CS2_REV02_OPC_VER01-03**.
   - This directory in turn accommodates other subdirectories which include the **<CustomID>**, **<DesignID>** and **<CompMode>** arguments, for example: **CANNY_DSG02_REL.** The compiled firmware file is contained in this subdirectory.

6. The firmware file is named by the OpenCamera script with the correct naming syntax, e.g.:

   F114Rx_A7-75-2_CFG00_DSG00_USERDESIGN_OPC_REL_VER01-03-yy.bin

   The "**F114Rx"** component in the file name denotes the CorSight2 board type and revision number.

   "A7-75-2_CFG00" identifies the FPGA type, size and speed grade as well as the FPGA configuration option, which is hard-wired on the CorSight2 PCB. For currently available CorSight2 cameras this a static component of the firmware file name.

   "DSG00" and **"USERDESIGN"** define the selected FPGA Framework and Custom Module option used to compile the FPGA.

   The **"OPC"** component denotes that the firmware has been compiled as an OpenCamera project.

   "REL" denotes that the firmware contains a **"Release"** Custom Module version and is alternatively be replaced by **"DBG"** if the firmware has been compiled in **"Debug"** mode.

   "VER01-03" defines the Major and Minor FPGA release version number and **"-yy"** denotes the Custom Module revision number which has been defined in the **CS2_xxx_Init.tcl** file (see Item 1 above).

7. Check the OpenCamera compilation log file **Cs2CompileLog.txt** located in the compilation directory (see **Cs2OpenCamCfg(CompileDir)** in Chapter 2.2.1). If timing errors or setup errors have been reported correct the errors and re-run the FPGA compilation, see Item 8

below.

Please Note: The compile directory uses the same structure and naming convention as the release directory, as described in Item 5.

8. If a design exhibits timing errors open the Vivado project file **CS2_Revx.xpr** located in the compile directory and analyse the offending signal paths. This is best accomplished by opening the "Implemented Timing Report" within Vivado and examine the "Timing" tab/window. If signals exhibit high fan-outs or show a large number of logic levels modify the source code to streamline the offending construct.

   As an alternative, it may be possible to eliminate timing errors which are due to a high FPGA device utilisation and/or due to routing congestion by recompiling the FPGA with a different **<DesignID>**. As shown in Chapter 4.1, Table 5 the Framework logic for **DSG00** requires the highest device utilisation. Try compiling the new Custom Module with either **DSG01** or **DSG02** to see if timing errors persist. Otherwise change the selected synthesis or implementation strategies in Vivado.

   Please Note: All FPGA Framework / Custom Module design combinations supplied by NET GmbH compile without timing errors.

9.2 OpenCamera Windows Batch Script

As an alternative to using the Vivado TCL Shell script directly, MS-Windows users can run the FPGA compilation from a Windows Command Window. The procedure to follow is the same as running the compilation from the TCL script as described in Chapter 9.1, except Item 2 and 3 are replaced as follows:

2. Open an MS-Windows Command Window and cd to **<OpenCamDir>\CompileTools**.

3. Run the following command to invoke Vivado. All required command arguments must be provided in the order indicated below.

Run_CS2_OpenCam.bat <CustomID> <DesignID> <RevisionID><CompMode>

with:

<CustomID>	(required)	= **[Default, UserDesign, LutDesign, MemTest, DiffPic, Canny, BlockDemo, ScaleD]**
<DesignID>	(required)	= **[DSG00, DSG01, DSG02]**
<RevisionID>	(required)	= **[REV01, REV02, REV03]**
<CompMode>	(optional)	= **[Release, Debug] - Default Value: Release**

The Windows batch file simply invokes the TCL compile script **Start_OpenCam_FPGA.tcl**, described in Chapter 9.1.

9.3 CorSight2 FPGA Flash Programming

After a successful OpenCamera FPGA compilation the resulting firmware file can be programmed into the CorSight2 Flash device using the command:

cbcmaload.x64.exe 4 <Firmware_File_Name.bin>

The **cbcmaload.x64.exe** tool is part of the SynView SDK which has to be installed on the CorSight2 camera. This tool is located in the SynView **./bin** directory which is included in the system path. The update process may take a few minutes for programming the flash device. When the update process is completed the FPGA still runs with the old configuration – you have to power cycle the camera to load the new firmware into the CorSight2 FPGA.

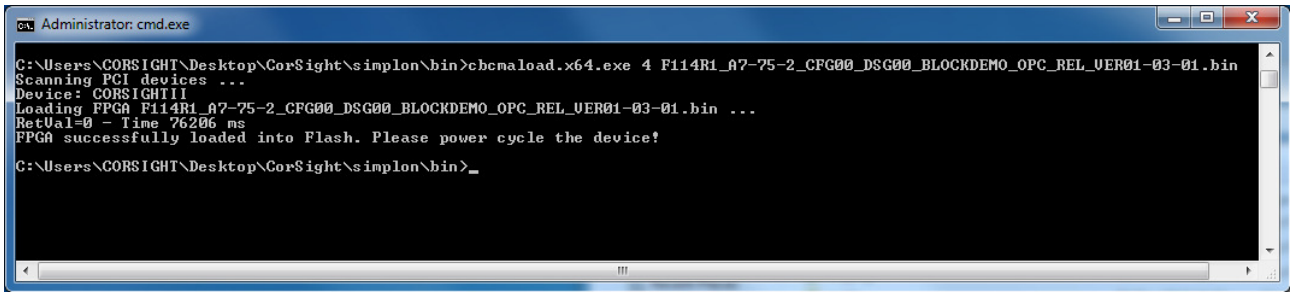

Figure 11: FPGA Flash Programming

The naming of **<Firmware_File_Name.bin>** determines if the file can be loaded into the Flash memory of a given CorSight2 camera. If a mismatch is detected (i.e. wrong CorSight Board Revision Number) the **cbcmaload.x64.exe** tool aborts the load process. If in doubt, run **cbcmaload.x64.exe** without providing any arguments to display an "expected" firmware name example.

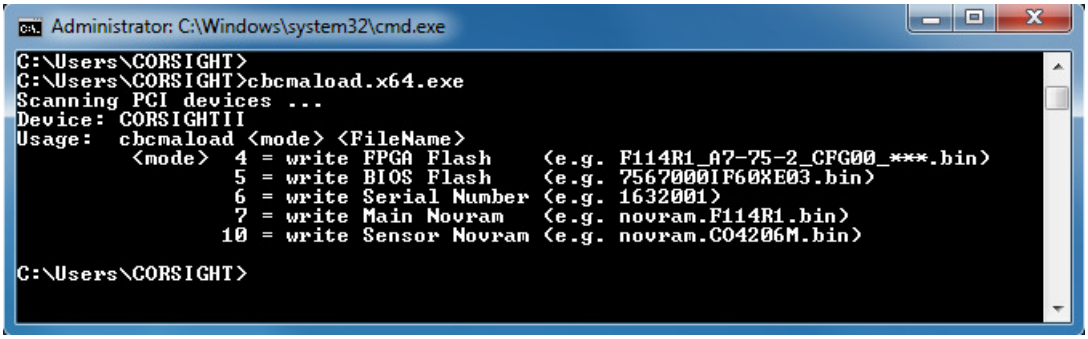

Figure 12: cbcmaload options

<u>IMPORTANT</u>: Do not manually change the firmware file name if a mismatch is detected. Instead, re-compile the correct FPGA version and repeat the Flash programming step.

<u>IMPORTANT</u>: The Flash Update procedure must not be interrupted in any way, otherwise the CorSight2 camera may become inoperable after the next power cycle!

10 Custom Module Design Implementation

When given the task of creating a new CorSight2 Custom Module design the customer has a choice between two implementation options:

- Remain within the OpenCamera Development Kit environment and re-use/modify existing compile and simulation scripts etc.
- Create a new design from scratch outside the ODK environment and use own project tools to implement/simulate the design.

Both alternatives have advantages and disadvantages and the designer must choose which approach best suit their needs. The following chapters explain both options and highlights their pros and cons.

10.1 Modifying the existing UserDesign Example

When staying within the ODK environment the proposed method of creating a new Custom Module design is to modify and re-use the **UserDesign** example design.

10.1.1 UserDesign Source Files

The existing UserDesign VHDL source files provide a recommended Custom Module design structure which has been described in Chapter 7.2. It is at the discretion of the module designer to keep the proposed structure or to replace the existing files with new ones.

10.1.1.1 Updating UserDesign Source Files

When using the existing UserDesign source files the designer can update the following files:

- **UserCtrl.vhd**: Instantiate the new top-level Custom Module entity and connect all ports/signals to the upper UserDesign level, as required.
- **UserPkg.vhd:** Define the SystemBus control/status register structure in the **tUSR_CTRL_REG** and **tUSR_STAT_REG** type declarations.
- **UserSysBusIf.vhd**: Implement the control/status register address allocation and register bit mapping, as per the defined control/status register structure.

10.1.1.2 Replacing UserDesign Source Files

When using a new set of source files the following restrictions have to be observed:

- The declaration for the top-level Custom Module entity (as defined in UserModule.vhd) must remain unchanged.
- The SystemBus register at address location 0x00 must have a layout as described in Chapter 6.2.3

It is likely that new or additional files are required for a Custom Module design. The location of the source files is not important as that can be changed in the associated **UserDesign** project file (see Chapter 10.1.2). Therefore it is possible to keep the Custom Module source files at a location outside the ODK.

10.1.2 UserDesign Project File

The **UserDesign** can be compiled within either one of the three available FPGA Framework designs **DSG00**, **DSG01** or **DSG02** (see Chapter 4.1), depending on the required Framework functionality and resource requirements. **DSG00** is the standard and recommended design option.

When adding new HDL design files the designer has to update the **UserDesign** project file called **CS2_Revx_UserDesign_FpgaProject.tcl**, which is located in:

<OpenCamDir>/syn/ProjectFiles/CS2_Revx

"**CS2_Revx**" denotes the applicable CorSight2 board revision number, x = [1..3]. The required UserDesign source files are listed in the project file. The paths and file names of the new design files have to be included and unused example design files have be removed. All other entries in the project file should remain unchanged.

Once the project file has been updated and new source files have been debugged the design can be synthesized/implemented by following the compilation procedure described in Chapter 9.

10.1.3 UserDesign Simulation

The new UserDesign can be simulated with the existing simulation Testbench by modifying the relevant simulation project file and the Testbench and SystemBus initialization files, as described below:

10.1.3.1 Xsim Simulation

Xsim simulations use the project file **Custom_UserDesign_Simulation.tcl**, located in:

<OpenCamDir>/sim/Simulation/Custom_UserDesign/Xsim

The UserDesign files are declared at line 60+ which must be changed as per the requirements of the new Custom Module design.

10.1.3.2 Simulation Initialization

A new Custom Module may require a different module initialization procedure at the start of the simulation run. The simulation scripts

<OpenCamDir>/sim/Simulation/Custom_UserDesign/TestBench_User.ini

<OpenCamDir>/sim/Simulation/Custom_UserDesign/SimCmd_User.ini

must be modified to change the Testbench setup and to initialize the SystemBus control registers of the new Custom Module. For detailed descriptions of available simulation commands see Chapters 8.3 and 8.4.

Once all simulation scripts have been updated simulations can be executed as described in Chapter 8.6 (Xsim).

10.1.4 UserDesign Limitations

An important consideration when using the **UserDesign** as the basis for a new Custom Module design is that user-updated ODK files in an existing ODK revision can not easily be transferred when upgrading to a new ODK revision.

Updates which the module designer has made to existing ODK files, for example the Vivado project files (described in Chapter 10.1.2) or simulation files (described in Chapter 10.1.3), have to be merged with the corresponding files included in the new ODK revision.

Also, if the UserDesign sources directory files (**<OpenCamDir>/src/UserDesign**) have been changed by the designer the changes also have to be merged with files in the new revision ODK.

At the same time, the advantage of the UserDesign approach is that the FPGA synthesis/implementation scripts as well as the simulation scripts and Testbench which are part of the ODK can be reused and do not need to be developed by the customer.

10.2 Creating a new Custom Module Design

If the module designer decides to create a new Custom Module design from scratch and not rely on the ODK environment a number of steps need to be performed to extract required FPGA Framework compilation files from the ODK.

10.2.1 Custom Module Design Guidelines

In the following description it is assumed that the new Custom Module design resides entirely outside of the ODK environment. The location of the new top-level Custom Module directory is referred to as: **<CustomModDir>**.

Perform the following steps (in no particular order):

- Create a new Custom Module top-level directory **<CustomModDir>.**
- Copy the entire OpenCamera IP directory **<OpenCamDir>/IP** to **<CustomModDir>**.
- Copy the FPGA timing and pinout constraint files located in **<OpenCamDir>/syn/ ConstraintFiles** to a suitable location within **<CustomModDir>.**
- Select an existing Vivado project file with the required Design-ID component **DSG00**, **DSG01** or **DSG02** (in **<OpenCamDir>/syn/ProjectFiles**) and cut-and-paste the **SecureIP**, **FPGA Constraints** and the **Design-IP** entries to a new Custom Module project file, as required.
  - Edit the path names of the copied project file entries to suit the new environment.
  - Add entries for the new Custom Module design files to the project file.
  - The commands to copy the **Design-IP** files to the compile directory can be omitted.
- Create a new Vivado project for an **Artix7 XC7A75T-FGG484-2** FPGA and source the newly created project file.
- Synthesize the design using the **Flow_PerfOptimized_high** strategy.
- Implement the design using the **Performance_ExplorePostRoutePhysOpt** strategy.
- If timing errors occur analyze the offending paths and fix any long delay paths. Repeat the FPGA synthesis and implementation process. If necessary change the synthesis / implementation strategies.
- Generate a bitstream / Flash programming file. The resulting bin file will be located in the Vivado compile directory under **<DesignName>.runs\impl_1**. The bin file has to be renamed to adhere to the CorSight2 firmware naming convention if the file is loaded into the

FPGA Flash device using the **cbcmaload** tool (see Chapter 9.3). All CorSight2 firmware file names have to start with a string "**F114Rx_A7-75-2_CFG00"**.

10.2.2 Custom Module Simulation

A new externally located Custom Module design is not supported in the CorSight2 ODK simulation environment. The ODK Testbench is an integral part of the Development Kit and can not be exported. Therefore, it is the responsibility of the module designer to create a suitable Custom Module simulation testbench and to verify correct behaviour of the design. Once the new design has been verified via simulation a FPGA implementation can be attempted.

11 CorSight2 ODK Release Upgrade

From time to time NET GmbH may release a new FPGA firmware revision for the CorSight2 camera. A new firmware release includes an updated OpenCamera Development Kit which the user may want to incorporate into their existing user application design.

When merging user-updated files in the old CorSight2 ODK with corresponding files in the new ODK the user must update all new ODK files in the same way as the old files have been updated previously. This applies in particular to modified project files (located in **<OpenCamDir>/syn/ProjectFiles**) which may contain references to new files present in the new ODK. To simplify this process it is recommended to use an appropriate file comparison/merging tool to make the updates.

To ensure that all new ODK files have been updated, the user is encouraged to check the guidelines described in Chapters 2.2, 3.4 as well as Chapter 10.

In addition, the customer environment files:

<OpenCamDir>/CompileTools/Set_Customer_EnvVar.tcl

<OpenCamDir>/CompileTools/Set_Vivado_Version.bat

must be updated as described in Chapters 2.2.1 and 2.2.2.

IMPORTANT: A new CorSight2 firmware release may require a new Xilinx Vivado installation. The user should check the pre-requisite requirements of the new ODK listed in Chapter 2.1.

12 Custom Module SW Interface

The host CPU interacts with the Custom Module through the SystemBus Master in the FPGA Framework. This communication is hidden in the PCIe driver of SynView and is not available to the host application. Instead the host application can access all Custom Module registers through the GenICam interface of the SynView API. The required SynView version for CorSight2 Custom Module is SynView-1.03.004.

Because the GenICam interface is generic it does not need information about the actual Custom Module design implementation. The only exception is the Custom Module ID register (@addr 0), which has to be different from 0 and 0xFFFFFFFF to enable the GenICam interface.

12.1 GenICam Interface

Being GenICam compliant, the CorSight2 camera contains an XML file with a detailed description of the camera features. The standard CorSight2 XML file already contains a category for accessing control registers of the Custom Module. As soon as the camera detects a valid Custom Module ID, the corresponding XML category “Custom Control” will be visible (see Figure 13). The camera XML file provides basic 32-bit Custom Module register access with address and data parameters.

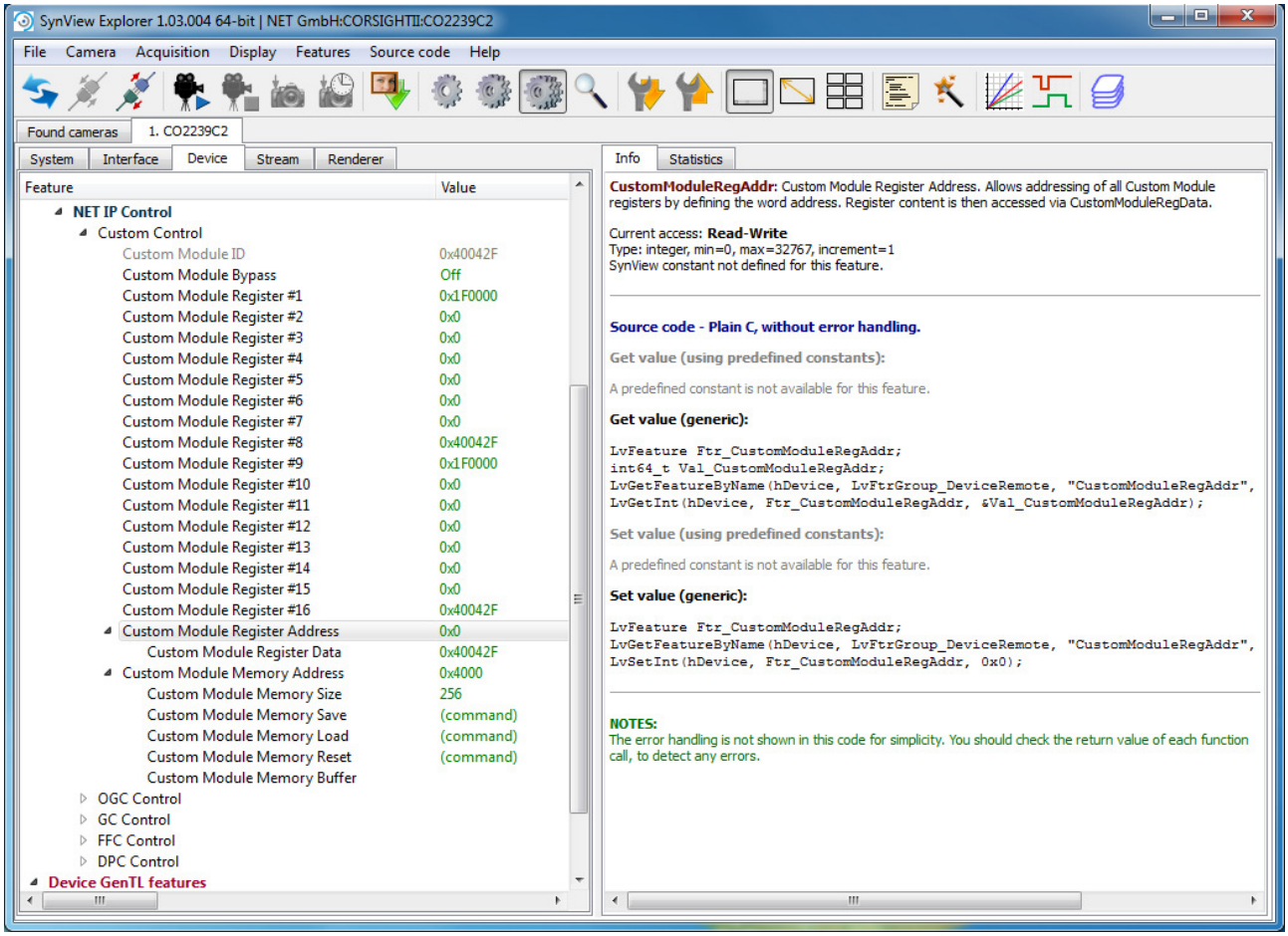

Figure 13: Custom Module GenICam Interface

12.2 Address Space

The Custom Module internal address space is 128KByte, allowing implementation of max. 32K word register (see Table 8). The GenICam interface addresses these registers indirectly so that the SystemBus base address is hidden from the host application. In contrast to the SystemBus the GenICam interface uses word addresses because byte addressing is not possible.

Word Address	CPU Access	Description	Update
Module ID (mandatory)			
0x0000	R	Custom Module ID Range 0x00000001...0xFFFFFFFE	boot
User Registers (optional)			
0x0001...0x0010	RW	Custom Module register space which can be accessed by host application. It can optionally be stored in the non-volatile camera UserSet1-4 and is then automatically reloaded during reset.	Any time
0x0011...0x7FFF	RW	Custom Module register space which can be accessed by host application.	Any time
User Memory (optional)			
0x0000...0x7FFF	RW	Custom Module memory space which can be accessed by host application.	Any time
Special Function Registers (for communication between CPU and Custom Module)			
0x????	R or W	tbd	start

Table 8: Custom Module Register Space

The GenICam interface does not know how many registers are really implemented in the Custom Module design. Therefore it is possible to address non-existing registers!

12.3 XML Features

A number of pure XML features belong to the GenICam interface which are not implemented in the Custom Module design (see Table 9).

Type	CPU Access	Description	Update
Custom Module Bypass			
bool	RW	Completely bypasses the Custom Module. This function is implemented in the FPGA Framework and is not part of the Custom Module design!	Locked
Custom Module Register Address			
15-bit int 0x0000...0x7FFF	RW	Allows addressing of all Custom Module registers by defining the word address. Register content is then accessed via Custom Module Register Data.	Any time
Custom Module Register Data			
32-bit int	R	Read data value from addressed register.	Any time
32-bit int	W	Write data value into addressed register.	Any time
Custom Module Memory Address			
15-bit int 0x0000...0x7FFF	RW	Allows addressing of all Custom Module memory by defining the word address. Memory content is then accessed via Custom Module Memory Buffer.	Any time
Custom Module Memory Size			
15-bit int 0x0001...0x8000	RW	Defines number of words accessed in Custom Module memory. Memory content is then accessed via Custom Module Memory Buffer.	Any time
Custom Module Memory Buffer			
char[Size*4]	RW	Accesses all the Custom Module memory in a single access without using individual addressing.	Any time

Table 9: Custom Module GenICam Features

12.4 How to use the SW Interface

The following source code snippets are copied out of SynView Explorer and show how to access Custom Module registers through the SynView API.

Read Custom Module ID:

```
LvFeature Ftr_CustomModuleId;
int64_t Val_CustomModuleId;
LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleId",
&Ftr_CustomModuleId);
LvGetInt(hDevice, Ftr_CustomModuleId, &Val_CustomModuleId);
```

Write Custom Module register 0x100:

```
LvFeature Ftr_CustomModuleRegAddr;
LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleRegAddr",
&Ftr_CustomModuleRegAddr);
LvSetInt(hDevice, Ftr_CustomModuleRegAddr, 0x100);

LvFeature Ftr_CustomModuleRegData;
LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleRegData",
&Ftr_CustomModuleRegData);
LvSetInt(hDevice, Ftr_CustomModuleRegData, 0x12345678);
```

Read LUT from Custom Module memory space:

```
LvFeature Ftr_CustomModuleMemAddr;
LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleMemAddr",
&Ftr_CustomModuleMemAddr);
LvSetInt(hDevice, Ftr_CustomModuleMemAddr, 0x4000);

LvFeature Ftr_CustomModuleMemSize;
LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleMemSize",
&Ftr_CustomModuleMemSize);
LvSetInt(hDevice, Ftr_CustomModuleMemSize, 256);

LvFeature Ftr_CustomModuleMemBuffer;
char Val_CustomModuleMemBuffer[1024];
LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleMemBuffer",
&Ftr_CustomModuleMemBuffer);
LvGetBuffer(hDevice, Ftr_CustomModuleMemBuffer, Val_CustomModuleMemBuffer,
sizeof(Val_CustomModuleMemBuffer));
```

Write LUT to Custom Module memory space:

```
LvFeature Ftr_CustomModuleMemAddr;

LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleMemAddr",
&Ftr_CustomModuleMemAddr);

LvSetInt(hDevice, Ftr_CustomModuleMemAddr, 0x4000);


LvFeature Ftr_CustomModuleMemSize;

LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleMemSize",
&Ftr_CustomModuleMemSize);

LvSetInt(hDevice, Ftr_CustomModuleMemSize, 256);


LvFeature Ftr_CustomModuleMemBuffer;

char Val_CustomModuleMemBuffer[1024];

memset(Val_CustomModuleMemBuffer, 0, sizeof(Val_CustomModuleMemBuffer)); /*
replace this line by real buffer initialization */

LvGetFeatureByName(hDevice, LvFtrGroup_DeviceRemote, "CustomModuleMemBuffer",
&Ftr_CustomModuleMemBuffer);

LvSetBuffer(hDevice, Ftr_CustomModuleMemBuffer, Val_CustomModuleMemBuffer,
sizeof(Val_CustomModuleMemBuffer));
```

13 Imprint

NET New Electronic Technology GmbH

Address:
Lerchenberg 7
D-86923 Finning
Germany

Contact:
Phone: +49-88 06-92 34-0
Fax: +49-88 06-92 34-77
www.net-gmbh.com
E-mail: info@net-gmbh.com

VAT- ID: DE 811948278
Register Court: Augsburg HRB 18494