Design and test of an active memory interface module for an H.264 encoder

Olja Pehilj

Electronics System Design and Innovation
Submission date: June 2014
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Milica Orlandic, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Design and Test of an Active Memory Interface Module for an H.264 Encoder

~

Olja Pehilj

Department of Electronics and Telecommunications
Norwegian University of Science and Technology

June 19, 2014

Problem Description

An active memory interface module shall be designed and tested that will connect a DDR3 memory block to an existing H.264 video stream encoder. The module shall be able to support full speed operation of the encoder in two modes, for 8x8 and 16x16 macroblocks organized with resp. 8 and 16 pixel values on the input at each system clock tick.

The design will be in VHDL as the existing design is. It should be designed and tested for FPGA implementation and shown to work together with the specific encoder module.

Abstract

In this thesis, the author describes a propositional design for a DDR3 memory interface, for an existing H.264/AVC video transcoder. The design uses the Memory Interface Generator (MIG), a Xilinx IP, as an overlying memory controller interface. The different interfaces offered by the MIG are evaluated before the most fitting is chosen.

The interface is designed for use on the KC705 Kintex-7 development kit, with a XC7K325T FPGA. Initial tests show promising results for the design, which is able to both write and read data to and from an external DDR3 SDRAM memory. The design has only been tested through simulation, and more extensive verification is needed before it can be completely evaluated as an alternative. The simulations use a memory model to produce realistic behavior of the memory.

The interface uses two submodules, dedicated to writing and reading respectively. Both modules use data buffers, and the reading module has the ability of transferring data in different modes.

Some room for improvement has been discovered, and the proposed design is thoroughly discussed. It has been successfully implemented, reporting an area utilization of 8,123 slices, with a maximum clock frequency of 308 MHz.

Keywords: Memory Interface, DDR3 SDRAM, Xilinx, Memory Interface Generator, MIG.

Sammendrag

I denne rapporten presenter forfatteren er designforslag for et minnegrensesnitt for DDR3, som skal benyttes av en eksisterende H.264/AVC videotranskoder. Designet benytter seg av Xilinx sin Memory Interface Generator (MIG) IP, som et lag over DDR3-minnegrensesnittet. De forskjellige grensesnittene som tilbys av MIG-en er vurdert, før det mest passende ble valgt.

Grensesnittet er designet for bruk på KC705 Kintex-7-utviklingssettet, som har en XC7K325T FPGA. Innledende undersøkelser av designet viser lovende resultater. Grensesnittet kan både skrive og lese til og fra det eksterne DDR3 SDRAM-minnet. Designet har kun blitt testet gjennom simulering, så større og mer omfattende undersøkelser er nødvendig før det kan vurderes som et alternativ til transkoderens nåværende minnegrensesnitt. Simuleringene bruker en minnemodell utviklet av Micron Technology, for å skape realistisk minneoppførsel under simulering.

Grensesnittet har to undermoduler, som er dedikerte til henholdsvis skriving og lesing. Begge modulene har databuffere, og lesemodulen kan sende data i henhold til transkoderens modus.

Designet er møysommelig diskutert og evaluert, og noe forbedringspotensial er oppdaget. Implementering av designet har blitt gjennomført, noe som rapporterer et arealforbruk på 8.312 skiver, med en maksimal klokkefrekvens på 308 MHz.

Contents

List of Figures

List of Listings

List of Tables

List of Acronyms

AMBA	Advanced Memory Bus Architecture – first introduced by ARM in 2001, [1]
AVC	Advanced Video Coding – a video compression format. Also called H.264.
AXI4	Advanced eXtensible Interface 4 – for Advanced Memory Bus Architecture (AMBA) 4.0
BC4	Burst Length 4 (Burst Chop) – a DDR3 burst mode.
BL8	Burst Length 8 – a DDR3 burst mode.
CAS	Column Address Strobe
CLB	Configurable Logic Block – the basic logic unit in an FPGA
DDR3	Double Data Rate type 3
DQ	Data Queue
DQS	Data Queue Strobe
FIFO	First In, First Out module– a method for organizing and manipulating a data buffers where the oldest entry exits first
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FWFT	First-Word-Fall-Through
GUI	Graphical User Interface
HDL	Hardware Description Language
IOB	Input Output Block
IP	Intellectual Property Core – In this context: IP core for Xilinx

ISE	Integrated Software Environment – a design tool by Xilinx
IO	Input/Output
ITU-T	International Telecommunications Union - Telecommunication Standardization
JVT	Joint Video Team– a group of video coding experts from ITU-T Study Group 16 (VCEG) and ISO/IEC JTC 1 SC 29 / WG 11 (MPEG) [2]
LUT	Look-up-table
MIG	Memory Interface Generator – A Xilinx IP Generator Tool
PAR	Place and Route
QDRII+	High performance Quad Data Rate SRAM
RAS	Row Address Strobe
RLDRAM	Reduced-Latency Dynamic Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
SODIMM	Small Outline Dual In-line Memory Module
STD	State Transition Diagram – A figurative way of describe the behavior of a state machine
UI	User Interface – one of the available interfaces for the MIG Core
VHDL	Very High Speed Integrated Circuit (VHSIC) HDL
VHSIC	Very High Speed Integrated Circuit
WE	Write Enable

Chapter 1

Introduction

1.1 Motivation

These days, most embedded designs need external storage. Avnet estimated in 2012 that 80 % of Field Programmable Gate Array (FPGA) designers use memory in their designs. [3] The transcoder for which this memory interface is proposed, is currently using the MicroBlaze soft-core processor to handle the communication to the external Double Data Rate type 3 (DDR3) Synchronous Dynamic Random Access Memory (SDRAM) memory chip. It is desired to lighten the load of the processor, so its resources can be used on other tasks. The proposed memory interface design is developed to relieve it of some tasks, and at the same time improve the performance of the communication with the memory.

1.2 Problem Interpretation and Contributions

The focus of this thesis has been on developing a working memory interface module, which can become a part of an existing H.264/Advanced Video Coding (AVC) video transcoder design. The problem description for the thesis was fairly open with regards to how the memory interface should be designed. A memory controller Intellectual Property (IP) (the MIG) developed by Xilinx, is used as a basis to design a propositional DDR3 memory interface. A dedicated reading module is designed as well, to support transferring data according to the transcoder's selected mode. Through dialog with the co-supervisor, it was decided that the most pressing requirement would be to support the 8×8 and 4×4 modes. A goal to achieve a running frequency above 100 MHz was also added.

For simplicity in the design process, all signals are assumed to be factors of eight. Furthermore, it is assumed that a single pixel contains eight bits. This was done because the IPs used only had support for data lengths in factors of eight.

1.3 Thesis Organization

The chapters and appendices forming this thesis report, contain the following:

- Chapter 1 presents the motivation behind designing the proposed interface, and explains how the task was interpreted.
- Chapter 2 presents the necessary background information and the tools that have been used. It also describes how to set up the test environment, and configure the tools.
- Chapter 3 describes the architecture of the proposed design. First the Xilinx MIG IP, and the possibilities offered by it, are presented. It then goes on to describing the proposed architecture, and the modules forming the designed interface.
- Chapter 4 presents the results and verification obtained for the proposed interface.
- Chapter 5 discusses the main properties of the design, and points out some limitations. It also proposes some ideas for improving the design.
- Chapter 6 summarizes the most important results and contributions presented in the previous chapters.
- Appendix A shows the selected properties for the generated memory interface IP.
- Appendix B shows the selected properties for the generated FIFO IP.
- Appendix C presents the extensive reports from implementation of the different parts of the proposed interface.
- Appendix D shows the top level block diagram for the interface, after implementation.

Chapter 2

Background and Methodology

This chapter presents the relevant background theory for the proposed design. It then presents the used hardware and tools, as well as the possibilities offered by the Memory Interface Generator (MIG) tool. In the last part of this section, the set up and configuration of the test environment is described, before the validation and verification strategies used while developing the design are explained.

2.1 The MPEG-2 to H.264/AVC Transcoder

The design proposed in this thesis is intended to be used as a memory interface for an existing design of an MPEG-2 to H.264/AVC intra-frame transcoder, which is described in detail in [4]. In this context, transcoding means the process of converting video data from one encoding (MPEG-2) to another (H.264). MPEG-2 and H.264/AVC are two different video coding standards, where MPEG-2 is defined by the International Telecommunications Union - Telecommunication Standardization (ITU-T), and H.264/AVC is defined by the Joint Video Team (JVT). The H.264/AVC standard is more efficient and flexible than MPEG-2, but consequently requires more complex computations in the video processing. An illustration of the top level block diagram for the module, consisting of an MPEG-2 decoder and a H.264/AVC encoder, can be seen in Figure 2.1.

The demand for such a transcoder arises with the extensive desire of viewing video on several platforms. TV broadcasting widely uses MPEG-2, as opposed to mobile and networking platforms, who have scarcer bandwidth availability.

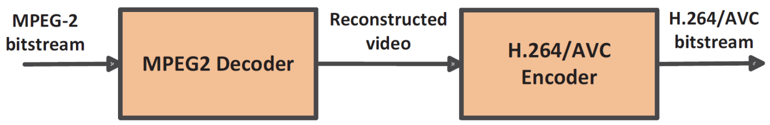


Figure 2.1: Block diagram of the existing transcoder. [4]

The encoding part of the transcoder supports processing of a 16×16 pixel macroblock with different granularities, depending on the currently used prediction mode. Granularity, in this context, means the further partitioning of a macroblock. The three types of intra prediction modes are Intra 4×4, 16×16 luminance and Intra 8×8 chrominance, in different profiles.

The memory interface proposed in this thesis should support data transferring in 4×4 and 8×8 mode. Because the intra prediction process introduces a dependency chain between blocks, the transcoder is fitted for using specific scanning order rearrangements. The transcoder supports reconfiguration, to accommodate different scenarios, depending on video requirements, among other properties. This is described further in [4, 5].

Extensive details about the H.264/AVC standard are beyond the scope of this thesis, and can be found in [6].

2.2 DDR3 SDRAM

Double Data Rate type 3 (DDR3) SDRAM is the memory standard following DDR2, and is described by JEDEC. It is a standard for external memory components, commonly chosen for many hardware designs. This is because it has the lowest cost per memory bit and largest density per chip. [3] The word “double” in the component name, comes from the fact that data is transferred on both rising and falling clock edges. A consequence of the dense dynamic nature of SDRAM memory, is that it needs to re-write data after reading, as well as performing periodic refreshes, to avoid data corruption and loss. [7] More detailed information about the DDR3 standard is available in [8].

As DDR3 is one generation after DDR2, it comes with some advantages over its predecessor. One is the higher bandwidth performance due to the eight bit prefetch buffer, instead of the four bit used by DDR2. This means that higher performance can be achieved through DDR3’s support for Burst Length 8 (BL8) in addition to the previous Burst Length 4 (Burst Chop) (BC4). DDR3 can also run at higher clock frequencies, as well as perform better at low power (1.5 V instead of 1.8 or 2.5 V). More information about the benefits of DDR3 is available in [9].

DRAMs are organized in a series of elements. They can contain one or more banks, and each of them consists of a series of rows. [10] The most significant signals used to interface with DDR3 SDRAM are listed below.

- Row Address Strobe (RAS) – Active low strobe for latching the row address
- Column Address Strobe (CAS) – Active low strobe for latching the column address
- Data Queue (DQ) – Bidirectional Input/Output (IO) data signal
- Data Queue Strobe (DQS) – Data strobe
- Write Enable (WE) – Low value: Write. High value: Read

An illustration of how writing is performed is shown in Figure 2.2. First a row is selected, by setting the `ras_n` signal low, while the corresponding address is set. This is denoted in the figure as ‘4’. If the memory has several banks, the `ba` signal is used to select the appropriate one. Then, the desired column address is set and the `cas_n` signal is set low, as denoted in the figure as '6'. Because this is a write command, the `we_n` signal is also set low, alongside the column address strobe. For a read operation, illustrated in Figure 2.3, the write enable signal is high throughout the interaction. At last, the data is transferred from and to the memory, respectively. It should be noted that, in addition to the illustrated signals, the figures do not include precharge commands. Such commands have to be issued when changing to a different row.

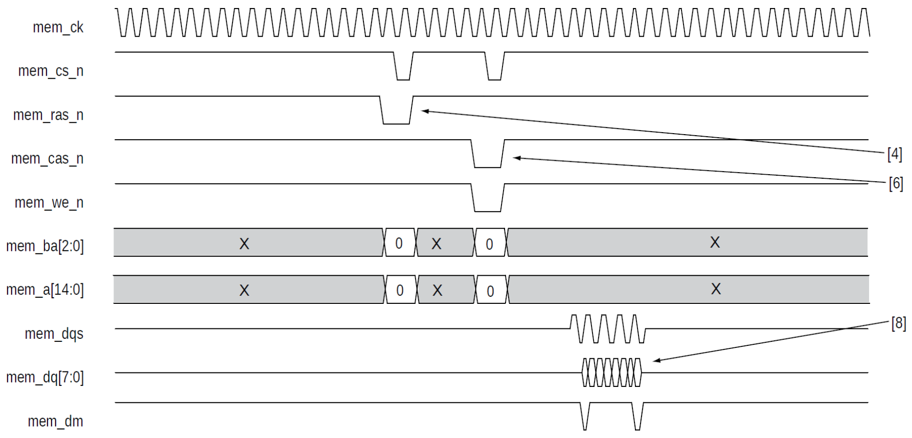


Figure 2.2: Timing diagram illustrating a single DDR3 writing command operation. [11]

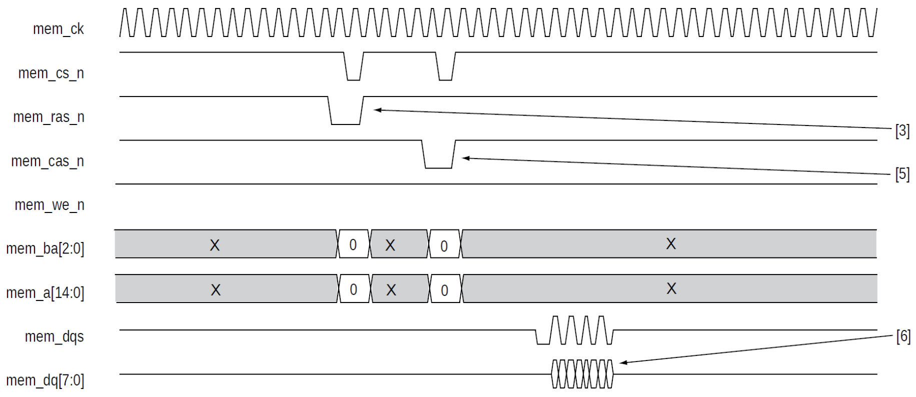


Figure 2.3: Timing diagram illustrating a single DDR3 reading command operation. [11]

Because a Xilinx IP is used as an abstraction layer, all the interaction with the DDR3 SDRAM is done by the generated memory controller. It also handles all calibration and refreshing operations. For this reason, only the necessary basic information has been presented.

2.3 Hardware - the KC705 Development Board

The development board, for which this design is targeted, is the KC705. The H.264/AVC transcoder is already implemented on the board, and thus the proposed memory interface is to be added. Some of the board's key features, as listed on Xilinx' website [12], are the following:

- The XC7K325T-2FFG900C FPGA
- 1GB DDR3 SODIMM 800MHz / 1600Mbps
- 128MB (1024Mb) Linear BPI Flash for PCIe Configuration
- 16MB (128Mb) Quad SPI Flash
- 8Kb IIC EEPROM
- SD Card Slot
- Fixed Oscillator with differential 200MHz output
- 5X Push Buttons
- 7 I/O pins available through LCD header

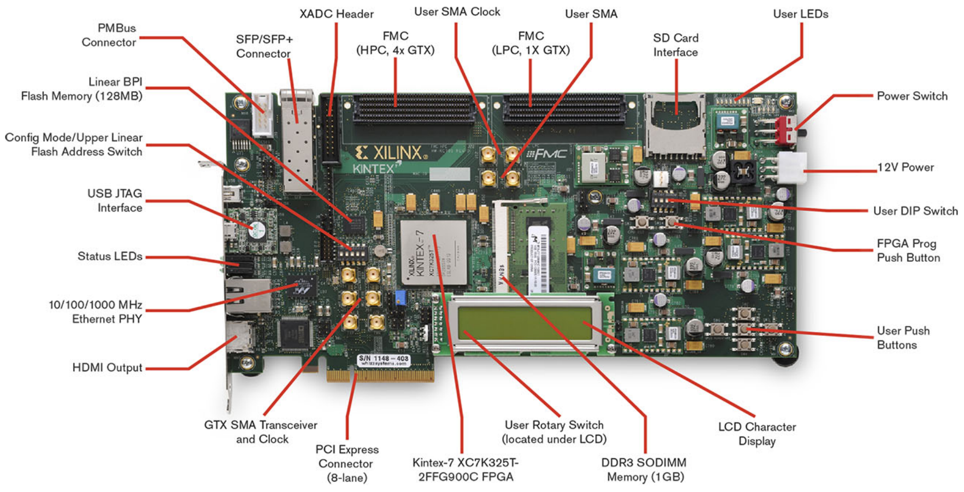


Figure 2.4: The KC705 Development Board. [12]

In their product brief, Xilinx state that the kit provides a flexible framework, for designing higher-level systems requiring DDR3 amongst other things. [13] With its fairly large sized FPGA, and on-board DDR3 memory, this board covers the needs for this design.

2.3.1 Kintex 7 FPGA

The KC705 is an evaluation board for the Kintex 7 FPGA (XC7K325T-2FFG900C). A sample from the feature summary [14] for this FPGA is

- 326,080 logic cells
- 50,950 slices (containing four LUTs and eight flip-flops)
- 4,000 Kb max distributed RAM
- 10 I/O banks in total

2.3.2 DDR3 Memory on the KC705 Board

The Xilinx KC705 board comes with on-board DDR3 memory, as listed in the previous section. The memory part is a Micron Technology MT8JTF12864HZ-1G6G1 [15, p. 10] It is a 1 GB 204-Pin Small Outline Dual In-line Memory Module (SODIMM) memory. The specified value for the module's bandwidth is 12.8 GB/s, meaning a transfer rate of 1600 MT/s on the eight bit wide channel. [16]

Because the correct memory part number was not found until late in the design process, a different memory part has been used. The default DDR3 SDRAM component, MT41J256M8XX-107 (also by Micron Technology) has been used during this design.

2.4 Tools

This section describes the tools used in this thesis, as well as some of their key functions. Below is a short list of all the tools, with corresponding version numbers.

- Xilinx Integrated Software Environment (ISE) Design Suite 14.7
  - Memory Interface Generator (MIG) 1.9
  - FIFO Generator 9.3
  - ISim 14.7 (P20131013)
- ModelTech ModelSim 10.2 (64 bit)
- Active-HDL Student Edition 9.3 (9.3.0.1)

2.4.1 Xilinx ISE Design Suite 14.7

The tool, in which the design for this thesis has been developed, is the System Edition of the XILINX ISE DESIGN SUITE, version 14.7. In addition to the hardware design tool with synthesis possibilities, the suite also contains a simulation tool, ISIM. Through the CHIPSCOPE software, debugging on the final result on the FPGA is also possible, by testing and capturing of the internal signals. This has not been done throughout this development process, due to time constraints.

The tool also contains the CORE GENERATOR Intellectual Property (IP) catalog, making it possible to use pre-developed IPs tailored for Xilinx FPGAs. The catalog contains several IPs readily available, ranging from First In, First Out modules (FIFOs) and the Memory Interface Generator (MIG) tool, to filters and more complex functions. [17] As Xilinx has made these freely available for use, it simplifies the process of designing a complete memory interface. Some of the possibilities offered are described in the following subsections.

Do note that the *WebPack* edition of the design suite does not support the Kintex 7 FPGA included on the KC705 board, as it only supports the XC7K70T and XC7K160T of the Kintex 7 series. [18]

2.4.2 Memory Interface Generator 1.9

The aforementioned CORE Generator contains several IPs, and one of these is the MIG. The MIG is an IP for generating a memory controller and physical layer (PHY) for interfacing with different types of memory, such as DDR2/DDR3 SDRAM, High performance Quad Data Rate SRAM (QDRII+) and Reduced-Latency Dynamic Random Access Memory (RLDRAM) II. Through the tool's Graphical User Interface (GUI) several features of the memory controller can be modified, and it can be customized according to ones needs. More information about the available features can be found in the core's user guide. [19]

Selection of hardware memory models is available, in addition to several options about the interface and target memory. The generated Verilog/VHDL files are not encrypted, and thus open for further modification, if it is desired. [20] An overview of all the selected properties for the MIG used in this thesis is included in Appendix A.

Differential clocks are selected as both system clock and reference clock, as a means to avoid potential clock skew and achieve more precise timing. [21] It might not be necessary for the low frequencies used, but this can be modified if it is no longer desired. The MIG and the designed interface use a single-ended clock, running at a quarter of the system clock frequency.

The generated files contain an example design, useful as a reference for developing a new design for interfacing with the MIG. The MIG also offers a simulation framework, which can be run in the ISIM tool[1], useful for seeing and verifying the behavior in simulations. The example design is synthesizable as well, as described in Section 2.6.4. The MIG offers the possibility of including signals for debugging of the memory controller, making it easily possible to verify the behavior on-chip, using the CHIPSCOPE tool.

The example design contains a traffic generator for generating read and write traffic to the memory. This is useful for initially verifying that the memory, and the interface, works correctly. Several properties of the traffic generator can be modified, to test different behavior.

2.4.3 Interfacing with the Memory Controller

There are three different interfaces that are supported by the generated memory controller. These are the Advanced eXtensible Interface 4 (AXI4) Slave Interface, the User Interface

[1]If the selected HDL is VHDL, ISIM does not work and MODELSIM must be used.

(UI) and the native interface. Some of the different properties of these are described in the following paragraphs, ending with the reasoning behind the chosen alternative.

Native Interface

The native interface is the most complex option, of the available interfaces. By using it, the designer has more control of a larger part of the interface itself. The data might be transferred out of order, and thus a design for handling such a behavior is needed. This interface is one level below the UI, meaning that it is necessary to design a complete interface to handle all communication to the PHY. According to Xilinx, the native interface offers higher performance in some situations. [19, p. 125]

User Interface

The UI is a more comprehensible memory interface, lying on top of the native interface. For one, it aggregates the address fields of the external DDR3 memory and presents a flat address space to interface with, as well as the ability of buffering both read and write data. [19, p. 64] This means that the data is returned in order, using a structure much like a FIFO, so extensive reordering control is not necessary.

AXI4 Slave Interface

AXI is a part of the ARM AMBA family of micro controller buses. AXI4 is the latest version of AXI, for AMBA 4.0. The MIG tool accommodates support for the AXI4 Slave Interface. It offers the possibility of having several masters and slave communicating over the same bus, and the interface is an attempt of making it easy to use. There are three types of AXI - the regular AXI4 for high-performance memory-mapped requirements, the AXI4-Lite for low-throughput applications, and the AXI4-Stream for high speed streaming data. [1] Xilinx also recommends AXI4 interface, over the other options, for communication between hardware and software partitions in co-design systems.

Please do note that the AXI4 slave interface for the MIG only is available in Verilog, and not VHSIC HDL (VHDL), at the time of writing. Additional information about the AXI standard for development with the Xilinx environment is available at [1].

Choosing an Interface

Of the three available interfaces, the UI has been chosen. The native interface could have been better, but would require continuous a more complex framework, as well as reordering of data during both reading and writing. The AXI4 interface also seemed fitting, with its possibility of using the AXI4-Stream type to meet the high data rates required by the transcoder. However, due to the restriction regarding the chosen hardware descriptive language being VHDL, it was discarded. The selected interface, the UI, is described in more detail in Section 3.1.

2.5 Verification Design

Among the files generated through the MIG tool, are two useful framework examples for verification - both for simulation and for synthesis. They both consist of several blocks, as is illustrated in Figure 2.5. The simulation design is the outer layer, containing the example design for synthesis. The simulation file has been used as a basis for verifying the behavior of the designed interface, as it instantiates a DDR3 memory model, developed by Micron Technology. The proposed interface has been implemented and tested by replacing the traffic generator (`traffic_gen_top`) module. Changes have been made in the example design (`example_top.vhd`), which is instantiated in the generated simulation file (`sim_tb_top.vhd`), as may be seen by the figure.

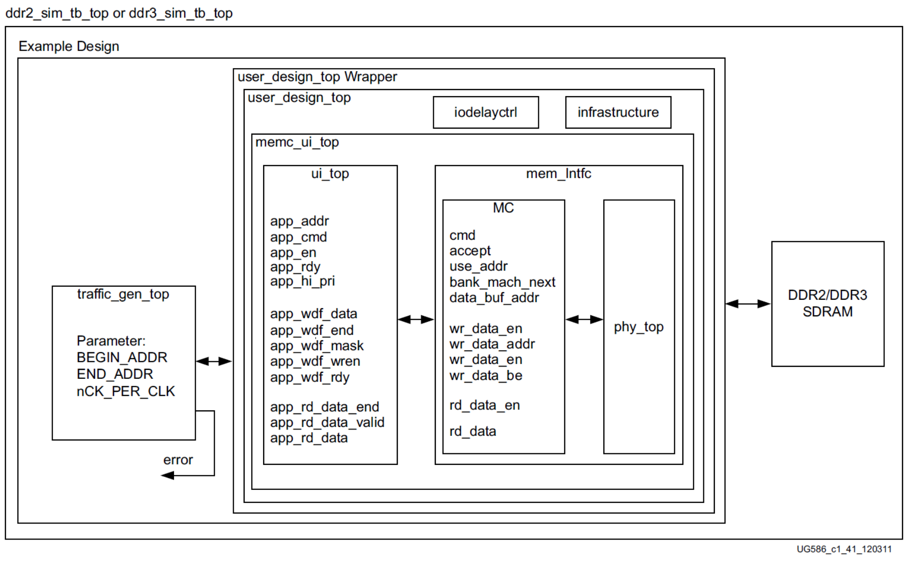


Figure 2.5: Example Design Block Diagram from the MIG. [22, p. 61]

Because the generated testbench only is available in Verilog, the top level test was made by extending this in its original Hardware Description Language (HDL). The behavior of the transcoder was simulated by applying different sequences and values to the interface signals.

Before the submodules were combined into one complete interface, they were tested and verified separately. The tests for the submodules were written in VHDL, and an attempt was made to simulate the behavior of the MIG's interface. For the reading module this was fairly simple, but due to the stochastic behavior of the memory, the test for the writing module was limited. Initial test were done on the submodule alone, but more extensive verification was conducted after it was combined and tested together with the communication top module.

The simulation and verification results are presented and described further in Section 4.1.

2.6 Test Environment Setup

This section contains descriptions of how the simulation environment is set up. Due to limitations in Xilinx' simulation tool, Mentor Graphics' MODELSIM has been used. The section explains how to set up MODELSIM and how to simulate Xilinx' IPs in simulators other than their own, as well as how to use ISIM. ISIM is mentioned as it has been used for testing of the submodules, before everything was combined into one complete system.

2.6.1 ModelSim Simulation Setup

To run simulations on the example design for the generated MIG, in VHDL, one has to use MODELSIM. This is because Xilinx' own simulator, ISIM, is not able to run the example design, unless Verilog is the chosen HDL. To be able to use Xilinx IPs, Xilinx' simulation libraries need to be compiled, as described in Section 2.6.2.

After this process is done, the report states where the libraries are compiled to. Then, one has to add the locations of these libraries in the generated *sim.do* file in the `sim` subdirectory, at the `vmap` lines, and uncomment these by removing the # characters. If one uses other design files than the ones in the example, these have to be added as well. Now the simulation design is ready to be run in MODELSIM. To set up MODELSIM to use a license located on a server, is done in Windows by adding the following two environment variables at "Start" → "Control Panel" → "System" → "Advanced" tab → "Environment Variables" → "User Variable".

```
Variable name:  MGLS_LICENSE_FILE = <port>@<server>
Variable value: LM_LICENCE_FILE   = <port>@<server>
```

Figure 2.6: Environment variable settings in Windows.

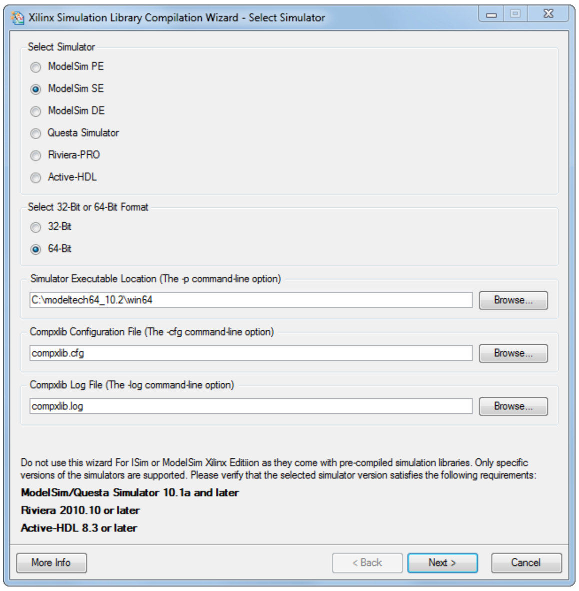
Do note that the example design for the MIG cannot be run in the Student Edition of MODELSIM PE due to the restriction on single-language designs. In addition, the student edition is unable to use encrypted files for simulation, making it impossible to use encrypted Xilinx modules.

2.6.2 Compiling Xilinx Libraries

To be able to simulate designs using Xilinx' IPs using other simulation software than ISIM, one has to compile the Xilinx' libraries for the chosen simulator tool. This is done using the XILINX SIMULATION LIBRARY COMPILATION WIZARD. It looks like shown in Figure 2.7, and is started by running the `compxlib` command in the XILINX ISE COMMAND PROMPT. One needs to select the simulator tool one has available, point to the executable's location and choose the desired HDLs.

2.6.3 Running simulation in ModelSim SE

To run the example design simulation in MODELSIM, one first needs to compile Xilinx' simulation libraries, as described in Section 2.6.2. To simulate the generated MIG example


Xilinx Simulation Library Compilation Wizard - Select Simulator

Select Simulator
- ModelSim PE
- [✓ ModelSim SE]
- ModelSim DE
- Questa Simulator
- Riviera-PRO
- Active-HDL

Select 32-Bit or 64-Bit Format
- 32-Bit
- [✓ 64-Bit]

Simulator Executable Location (The -p command-line option): C:\modeltech64_10.2\win64

Compxlib Configuration File (The -cfg command-line option): compxlib.cfg

Compxlib Log File (The -log command-line option): compxlib.log

Do not use this wizard For ISim or ModelSim Xilinx Editiion as they come with pre-compiled simulation libraries. Only specific versions of the simulators are supported. Please verify that the selected simulator version satisfies the following requirements:

ModelSim/Questa Simulator 10.1a and later

Riviera 2010.10 or later

Active-HDL 8.3 or later

Figure 2.7: Xilinx Simulation Library Compilation Tool window.

design, first, start the MODELSIM software from the ISE COMMAND PROMPT, to set the `$Xilinx` environment. In MICROSOFT WINDOWS, one can also add the path to the install location, i.e. `C:/Xilinx/14.7/ISE_DS/ISE` as the `XILINX` Environment Variable, as explained in Section 2.6.1. This way, MODELSIM is always able to find the Xilinx libraries.

After navigating to the `ipcore_lib` subdirectory of the design, the generated `do`-file can be run through the command `do sim.do`. This runs the simulation with the preferences specified in the do-file.

The simulated waveforms are stored in the `vsim.wlf` file after simulating. This file can be reopened in MODELSIM to view the static simulation data, equivalent to the one described in Section 2.6.5.

2.6.4 Synthesizable Example Design

Amongst the many files generated with the MIG, is a design which can be synthesized, and a design which can be simulated. These are located in the two subdirectories *user_design* and *example_design*, respectively. The design for simulation is useful for getting familiar with the behavior of the generated memory interface block. The synthesizable example design is a practical basis for developing a design for synthesis. To make a project with the synthesizable example design, the generated files contain a script file which should be executed, located in the `DESIGN_NAME/example_design/par` folder[2]. Run the ISE COMMAND PROMPT, move to the mentioned directory, and run the `create_ise.bat`

[2]For instance, `C:/<Project_name>/ipcore_dir/<MIG_component_name>/example_design/par`

script. This runs the `set_ise_prop.tcl` command, which is a script file that generates a project called `test.xise`. [19, p. 35] The generated project instantiates the `example_top.vhd` file, so any changes to the example design are maintained. The generated project also contains the pin locations in UCF format, but if new IOs have been added they have to be place manually. The project can be both synthesized, implemented (translation and Place and Route (PAR)) and a programming (`bit`) file can be generated, to be placed on the targeted FPGA.

2.6.5 Viewing Static Simulations in ISim

Simulating the behavior of communication has been found to be a time consuming process. It is often interesting to view a simulation which has been run (called *static*), either for comparison or for controlling behavior at a previous time. This section describes how this is achieved, using Xilinx' own simulator, ISIM. [23]

After a simulation has been run, the waveform configuration can be saved as a `wcfg` file. The simulation data is stored automatically while the simulation is run, in a waveform database (`wdb`) file. with the name being the same as the testbench module.

Assuming the files are available, start the ISE DESIGN SUITE 32/64 BIT COMMAND PROMPT, and run `ISimgui.exe`. This opens the ISIM GUI, and now one just needs to open the desired `wcfg` file. This shows the static simulation, based on the data in the `wdb` file. If no configuration file has been made, loading the `wdb` file alone is also possible.

Chapter 3

Architecture and Implementation

This chapter presents and describes the modules which form the memory interface. First the Xilinx IP, the MIG, is described, before the selected interface is explained. It then describes the the communication module itself, including detailed descriptions about the interface and signals, as well as the Finite State Machines (FSMs) for all modules forming the interface.

3.1 The MIG and its User Interface

As previously stated, Xilinx offers an IP overlay for interfacing with memory modules. Because the DDR3 interface standard is fairly complex and rigid when it comes to timings, among other properties, it has been decided to use the available IP.

Figure 3.1 shows the overview of the design generated by the MIG[1]. The module generated by the MIG is the one labeled *7 Series FPGAs Memory Interface Solution*, and the module called *User FPGA Logic* is where the communication interface to the UI is located, in combination with the transcoder module. The signals the UI consists of, and whom are illustrated in Figure 3.1, are listed and described in Table 3.1. For the proposed design, the values for `APP_DATA_WIDTH` and `ADDR_WIDTH` are 128 bits and 29 bits, respectively.

The MIG also offers the possibility of issuing additional refresh and calibration commands, through the *User Refresh* option. This has not been done, as the memory controller handles this in a fashion that complies with the JEDEC standards. At startup of the system, memory initialization and calibration is performed, and the `init_calib_complete` signal is asserted when this is completed.

Another option is the physical layer (PHY) to memory controller clock ratio. This feature states the ratio of the memory clock frequency to the user interface clock frequency. Xilinx state that the 2:1 ratio has lower latency, while the 4:1 ratio is needed for achieving

[1]System clock (`sys_clk_p` and `sys_clk_n`/`sys_clk_i`), reference clock (`clk_ref_p` and `clk_ref_n`/`clk_ref_i`), and system reset (`sys_rst_n`) port connections are not shown in the overview. [22]

the highest data rates. [19, p. 22] Because high data rates are necessary for this design, the 4:1 mode is selected, with a PHY frequency of 400 MHz. The all clocks are made through a clock generator which uses a reference clock, running at 200 MHz.

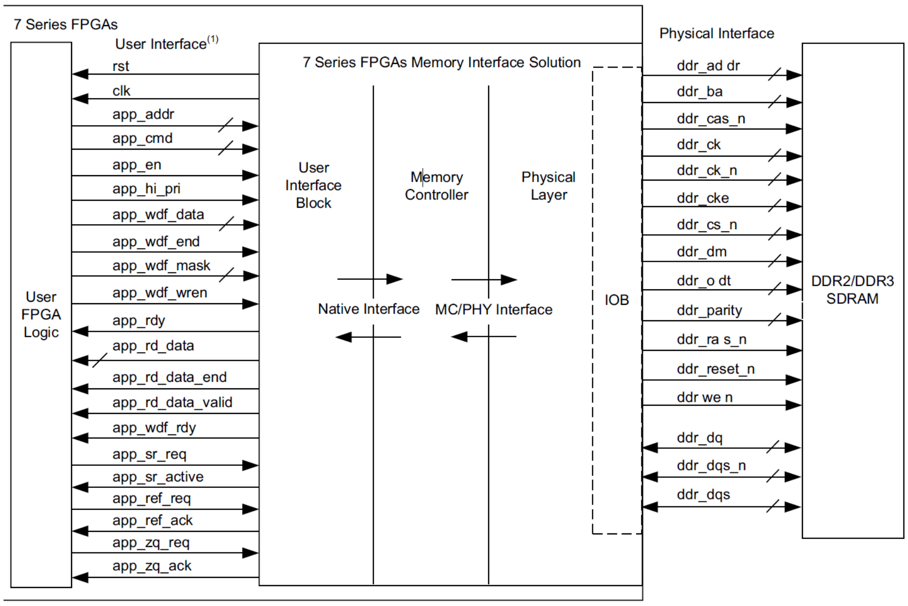


Figure 3.1: Block Overview for the 7 series MIG, with the UI. [22, p. 82] Illustration from [22] is used because the figure in [19] is inconsistent with the code generated, with regards to the direction of the `rst` and `clk` signals.

The User Interface (UI) aggregates the address fields of the external DDR3 memory and presents a flat address space to interface with, as well as the ability of buffering both read and write data. [19, p. 64] The relation between the UI address space and the physical memory row, bank and column is illustrated in Figure 3.2. Furthermore, unlike the native interface, the User Interface (UI) returns the data in order, much like a FIFO.

The interaction to the UI is divided in three paths - the *Command Path*, the *Write Path* and the *Read Path*. These are described in the following sections.

3.1.1 The UI Command Path

The command path is the path for sending write or read commands, together with the associated address and enable signal. The outgoing command values are `000` for *writing* and `001` for *reading*. As illustrated in Figure 3.3, a command is accepted by the memory controller when the `app_rdy` signal is high. If the signal is low when the `app_cmd` signal is transmitted, the signal has to wait until the signal is high. This means that the corresponding `app_addr` and `app_wdf_data` signals must be maintained until the `app_rdy` signal is asserted.

Table 3.1: Signal Names and Descriptions, for the UI. [19, p. 65]

Signal Name	Width	Description
app_en	1 bit	Strobe for submitting a request, containing address and command.
app_addr	ADDR_WIDTH	Target address in the UI flat address space. Sent alongside app_en, accepted when app_rdy is asserted.
app_cmd	3 bits	Command signal, "001" for reading and "000" for writing.
app_rdy	1 bit	Signal indicating that the UI is ready to accept commands.
app_wdf_data	APP_DATA_WIDTH	Data to be transferred.
app_wdf_wren	1 bit	High strobe for app_wdf_data
app_wdf_end	1 bit	Indicating the last cycle of app_wdf_data. The same as app_wdf_wren when in 4:1 mode.
app_rd_data	APP_DATA_WIDTH	Data returned from the requested address, after a read command has been issued.
app_rd_data_valid	1 bit	Data is valid when this is asserted.

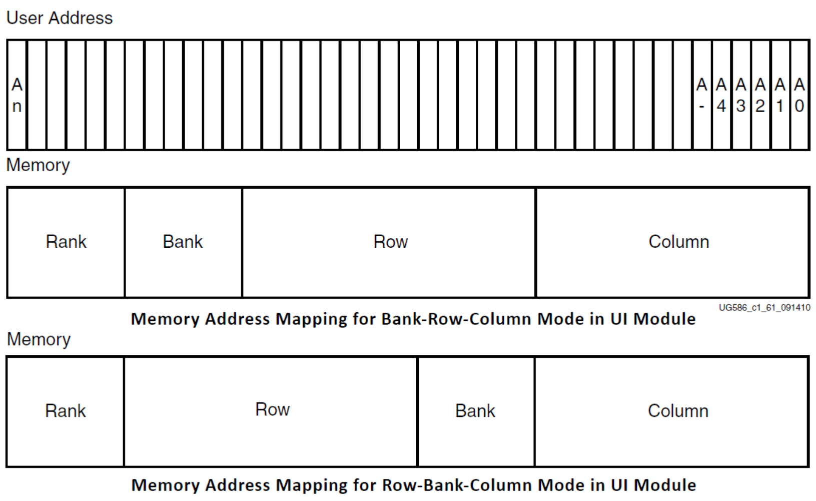

User Address
A n
A -
A 4
A 3
A 2
A 1
A 0
Memory
Rank
Bank
Row
Column
UG586_c1_61_091410
Memory Address Mapping for Bank-Row-Column Mode in UI Module
Memory
Rank
Row
Bank
Column
Memory Address Mapping for Row-Bank-Column Mode in UI Module

Figure 3.2: Memory address mapping for Bank-Row-Column and Row-Bank-Column mode in the UI. Slightly modified from [19, pp. 127-128].

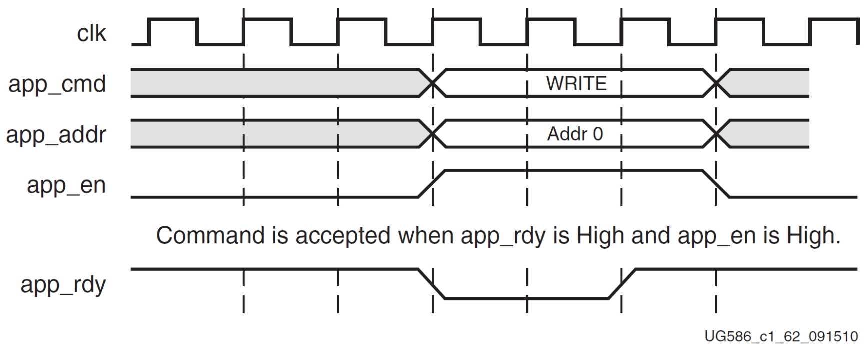

clk
app_cmd
WRITE
app_addr
Addr 0
app_en
Command is accepted when app_rdy is High and app_en is High.
app_rdy
UG586_c1_62_091510

Figure 3.3: Timing Diagram for the UI command path. [19]

3.1.2 The UI Write Path

As previously stated, the UI has a FIFO-like way of handling data. This is utilized by the write path. The written data is stored in the FIFO when the `app_wdf_rdy` signal is high, and `app_wdf_wren` is asserted at the same time. Just like the command and address signals for the command path, the `app_wdf_wren` signal must be held high until `app_wdf_rdy` is asserted.

The `app_wdf_end` signal is used to indicate the last cycle of data on `app_wdf-_data`. For the 4:1 mode, this means that the signals `app_wdf_wren` and `app_wdf_end` are equal.

Figure 3.4 shows three non-back-to-back write scenarios, as described below:

1. Write ata is transferred and accepted at the same time as the corresponding write command is accepted.
2. Write data is transferred and accepted one clock cycle *before* the corresponding write command is accepted.
3. Write data is transferred and accepted at most two clock cycles *after* the corresponding write command is accepted.

The MIG also supports back-to-back writing. An example of a back-to-back data transfer is illustrated in Figure 3.5. While the `app_wdf_rdy` signal is high, data can be written back-to-back. The figure also indicates that it is possible to keep writing data after the command path goes low. The documentation states that there is no maximum time delay between the write data and its associated write command, when issuing back-to-back write commands. [19, p. 130]

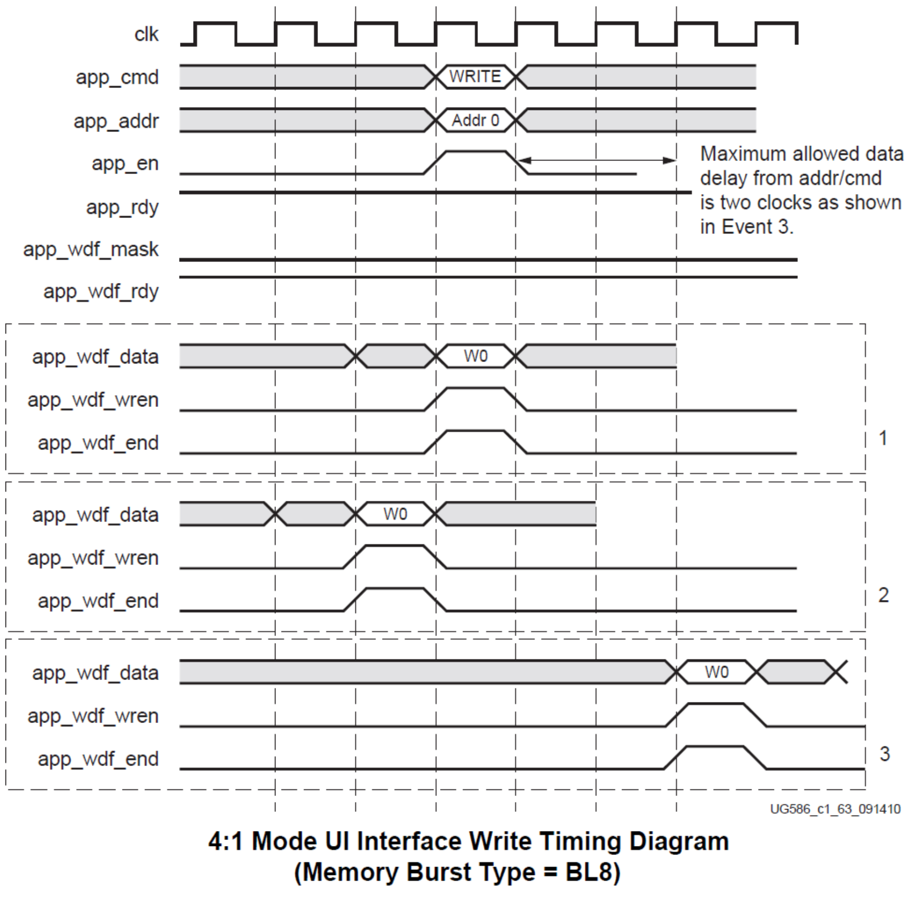

clk
app_cmd
WRITE
app_addr
Addr 0
app_en
Maximum allowed data delay from addr/cmd is two clocks as shown in Event 3.
app_rdy
app_wdf_mask
app_wdf_rdy
app_wdf_data
W0
app_wdf_wren
app_wdf_end
1
2
3
UG586_c1_63_091410
4:1 Mode UI Interface Write Timing Diagram
(Memory Burst Type = BL8)

Figure 3.4: Timing diagram for the UI write path. [19, p. 129]

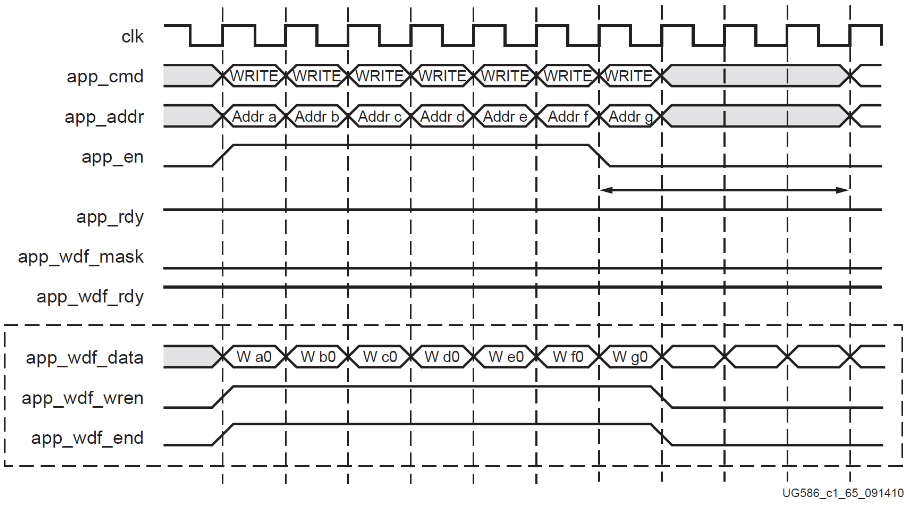
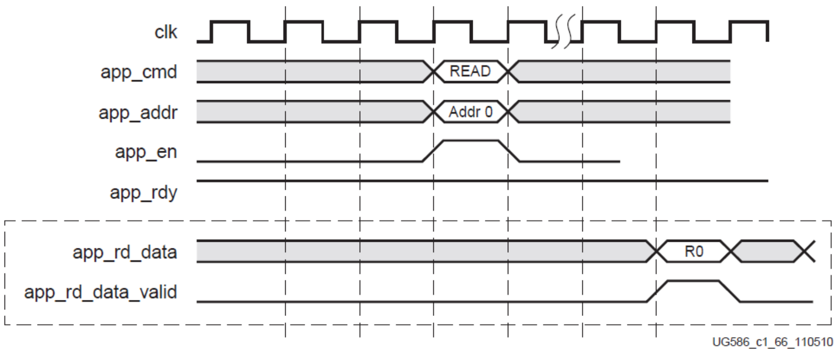


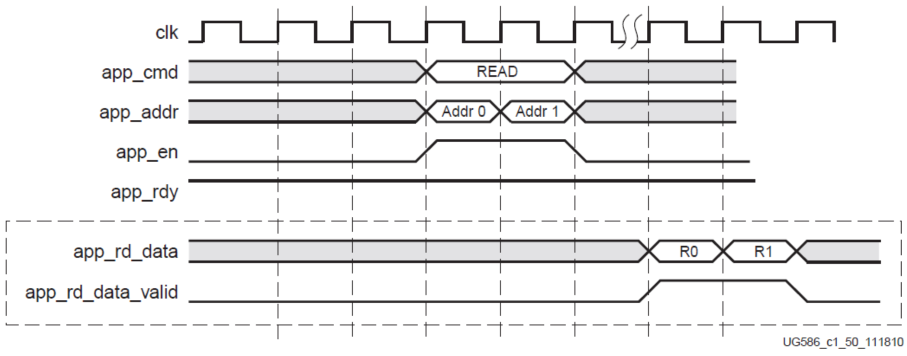
Figure 3.5: Timing diagram for back-to-back writing, in 4:1 mode. [19, p. 130]

3.1.3 The UI Read Path

The communication for the read path is initiated over the command path, through the command, enable and address signals. After some time delay, data is received from the DDR3 memory, through the signals `app_rd_data` and `app_rd_data_valid`. The first is the data itself, while the last one indicates that the data currently on the bus is valid. In addition, there is a signal called `app_rd_data_end`, which indicates the end of a read command burst. Because the MIG user guide states that this is not needed, it is left unused. [19, p. 132]



4:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL8)



4:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL4 or BL8)

Figure 3.6: Timing Diagram for UI Read Path. [19, p. 132]

The timing diagram for the read path is shown in Figure 3.6. The upper part shows the issuing of reading from a single address. The lower part shows the issuing of to back-to-back read commands from two addresses, and how they are received in the correct, requested order. It can be seen in both illustrations that the time it takes from the read command is accepted, until the data is returned, can vary. This is denoted by the break in the timing diagram, seen after the read command is successfully issued.

3.2 Communication Interface Architecture

This section described the proposed interface, as well as the interface between the inner submdules. The designs can serve as a bridge between the H.264 transcoder and the external DDR3 SDRAM memory.

Some of the significant design decisions are described first, before the top level interface is presented. It then goes on to the architecture of the design, and describing all modules from the top to the bottom.

3.2.1 Design Decisions

The design of the proposed interface assumes that a pixel is eight bits long. This has been done to easily match a whole number of pixels on the data buses, as both the FIFO and MIG offer data widths in factors of eight. A data width of 128 bits has been chosen, and the generated FIFO has room for 512 elements. If it is necessary to modify the data width at a later time, this can be done by oversizing the data buses to exceed the size from the transcoder, and pad the rest. The MIG also offers the possibility of masking data, which can also be used if necessary.

3.2.2 The Communication Interface and Top Level Architecture

Interfacing to DDR3 is fairly complex, as it requires very precise timing of many signals. This is why the offered memory interface IP by Xilinx has been used. The MIG IP is used as an overlay, and controls the interface to the memory. The proposed communication interface is connected to the MIG as illustrated in Figure 3.7. The proposed interface is connected to the MIG and the DDR3 SDRAM memory model by replacing the traffic generator module in the example design (`example_top.vhd`), as shown in Figure 2.5, on page 10, with the communication top module. Figure 3.7 also shows the signals forming the interface for the transcoder, which are listed and described in the following.

Because the transcoder can request data in different modes, a dedicated reading module has been designed. It currently supports the. 4×4 and 8×8 modes.

- `mod_dataIn_en` – Active high input strobe for the `mod_dataIn` signal.
- `mod_dataIn` – The input data to be written to the external memory.
- `mod_dataOut` – Data output read from the memory, sent to the transcoder. This signal should eventually be removed, and replaced by the last three in this list.
- `mod_readReq` – Active high input for requesting a read from the memory. This signal should eventually be removed, and replaced by a request signal from the reading module.
- `mod_read4x4_req` – Active high input, from the transcoder, for requesting data in 4×4 mode.
- `mod_read8x8_req` – Active high input, from the transcoder, for requesting data in 8×8 mode.

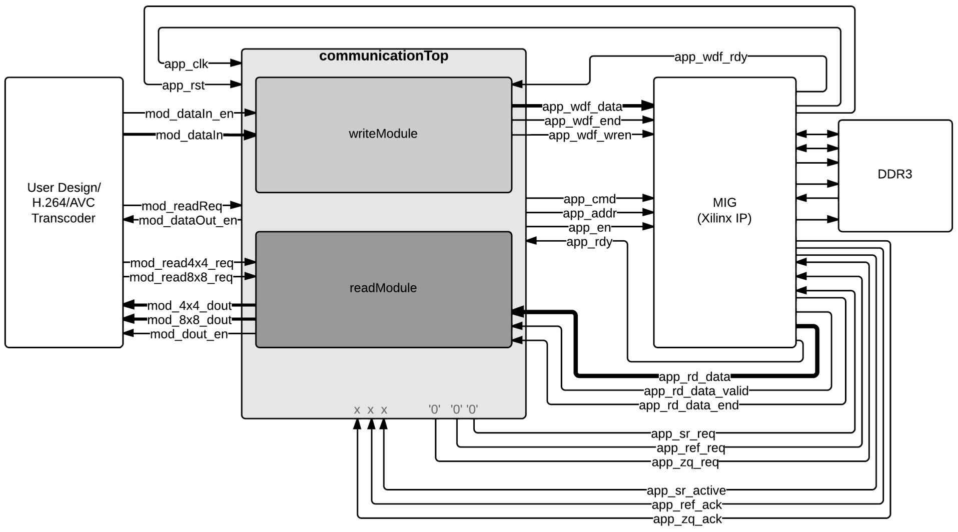


Figure 3.7: Design Overview

- `mod_4x4_dout` – Output data, to the transcoder, when in 4×4 mode. A 128 bit long vector.

- `mod_8x8_dout` – Output data, to the transcoder, when in 8×8 mode. A 64 bit long vector.

- `mod_dout_en` – Active high signal, for the `mod_NxN_dout` signals.

The last five signals listed have been partly implemented, but the reading module is yet to be fully connected to the communication top module. The goal is to eventually remove the `mod_readReq` and `mod_dataOut` signals completely, and replace the read requesting with a signal from the reading module.

Theoretical Use-Case

A theoretical use-case scenario would be that a complete video frame has been loaded to the external DDR3 SDRAM memory. The reading module is notified that a frame is available[2], and loads one macroblock to the local storage. According to the request signals from the transcoder, the module transfers parts of the macroblock, divided in the fashion desired by the transcoder. Processed data can be transferred to the interface using the `mod_dataIn_en` and `mod_dataIn` signals, at any point. When a complete macroblock has been received by the transcoder, a new can be constructed and is then ready to be transferred.

[2]This has not been implemented, at this time.

3.2.3 Communication Top Module

The top module, to which the transcoder is to be connected, is called the communication top. This is the module that handles the *command path* part of the UI. It contains a dedicated writing module, which handles the data which is to be written to the SDRAM. The module also contains a reading module, but this has not been completed. This is because further modification of the design is needed, to handle the first loading of a macroblock, from the memory.For this reason, the communication top module also handles the reading from memory, based on commands issued by the simulated transcoder.

The current design issues writes to consecutive addresses, starting from address number eight (8), and continues in increments of eight. The same is the case when reading from the memory. This can be modified to use a register, with a predefined address order for reading or writing, if it is necessary. All data widths are set to 128 bits, meaning that both data buses in and out, as well as the data bus to the MIG.

The mediating between the read and write address is handled by a separate process within the communication top. This simply depends on the `writing` and `reading` signals, with priority on the reading. This is because the state machine also prioritizes in the same manner. The writing address is received from the writing module, while the reading address is incremented within a state machine. All internal signals are clock synchronous, in the submodules as well, by using current `(c_)` and next `(n_)` signals. The current signals obtain the next value at a positive clock edge, or are reset when a reset signal is received. It should also be noted that all the presented FSMs, for all the modules, return to the `IDLE` state at reset.

Communication Top State Machine

As the communication top module contains a submodule for writing to the memory, as well as issues read request to the memory, a state machine is used. The State Transition Diagram (STD) for the communication top module is illustrated in Figure 3.8. Please note that the STDs presented throughout this section are not extensive, in the sense that only the general assignments in each state are shown, while several other are done depending other signals in addition to the current state.

The FSM starts in the `IDLE` state, where it waits for either a read request (`mod-_readReq`) from the transcoder or a write request (`write_req`) from the writing submodule. Reading is given priority over writing, because a transition to the reading (`S_READ-_WAIT`) state only is performed when a read request is received. It counts the number of received read requests from the transcoder, as well as the number of read requests issued to the MIG, but these are do not regarded in the `IDLE` state.

If a reading request is received, a transition is made to the `S_READ_WAIT` state. If the MIG is ready to receive commands, meaning that `app_rdy` is high, it goes on to controlling the number of issued and received commands. It compares the number of read commands issued (`readCount`) with the number of received requests (`readCommand-_count`). At the same time, to avoid read requests past the addresses which have had data written to them, it compares the number of issued read requests to the number of data blocks written (`acceptedWrite_count`). If the amount of issued requests is less than both of the other two counters and the `app_rdy` signal is asserted, it issues a read

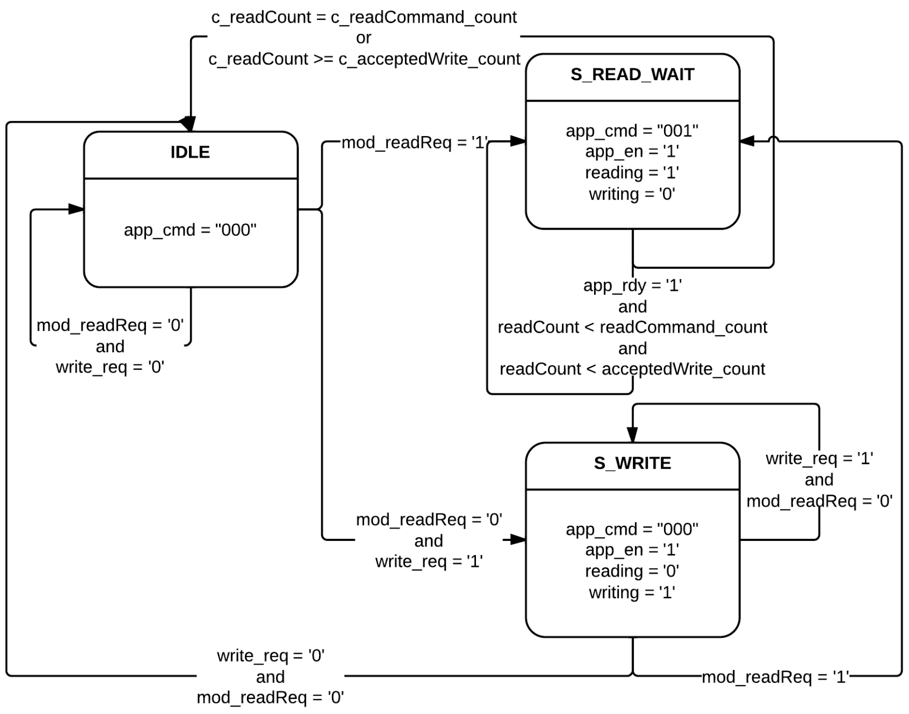


Figure 3.8: State Transition Diagram for the communication top module.

```
if app_rdy = '1' and c_readCount < c_readCommand_count and c_readCount <
    c_acceptedWrite_count then
    app_cmd <= "001";
    app_en <= '1';
    n_state <= S_READ_WAIT;
    n_reading_addr <= c_reading_addr + 8;
    n_readCount <= c_readCount + 1;
elsif app_rdy = '1' and (c_readCount = c_readCommand_count or c_readCount
    >= c_acceptedWrite_count) then
    app_cmd <= "000";
    app_en <= '0';
    n_state <= IDLE;
    n_reading_addr <= c_reading_addr;
    n_readCount <= c_readCount;
```

Listing 3.1: How the counters are used when read requests are issued.

request to the MIG, and increments the reading address (`reading_addr`) by eight. If the `app_rdy` signal is low, and the comparison yields for more requests, it stays in the `S_READ_WAIT` state until enough requests are issued to the MIG. When an adequate amount of read requests are successfully issued to the MIG, the state machine returns to the `IDLE` state. A part of the implementation of this control can be seen in Listing 3.1.

If a writing request (`write_req='1'`) is received, while the FSM is in `IDLE` and the `mod_readReq` is low, the FSM goes to the `S_WRITE` state. As stated previously, reading has priority over writing, as the written data is stored within the writing module, and this is the case in this state as well. If no read request is received, it issues a write request to the MIG over the command path. The address is received from the writing module, and set as the output through a separate process. It stays in the writing state until no more requests are received, when it goes back to the `IDLE` state. If a read request is received, it is issued, and the FSM goes to the `S_READ_WAIT` state.

3.2.4 Writing Module

From the simulations of the example design, it was clear that the memory was not necessarily ready to receive data at each clock cycle, as were the specifications for this design. To deal with this, an internal write buffer is used, realized through the Xilinx FIFO generator IP. Some of the possibilities offered by this IP are described in a separate paragraph, at the end of this section. The overview of the signals between the write and communication top module, as well as with the inner FIFO module, are illustrated in Figure 3.9.

As shown in the figure, data received from the transcoder is buffered directly to the FIFO. The `mod_dataIn_en` signal serves as the write enable (`wr_en`) and the data is connected to the incoming data port (`din`). The FIFO has full and empty signals, which notify the writing module of its status. While the FIFO has data stored, shown by a low `empty` assignment, the writing module requests write access to the MIG. This is done by asserting the `top_wr_req` signal, as well as transmitting the current writing address through the `write_app_addr` signal. At the same time, the first element of the FIFO is requested, and made available on the next clock cycle. The behavior of the module is

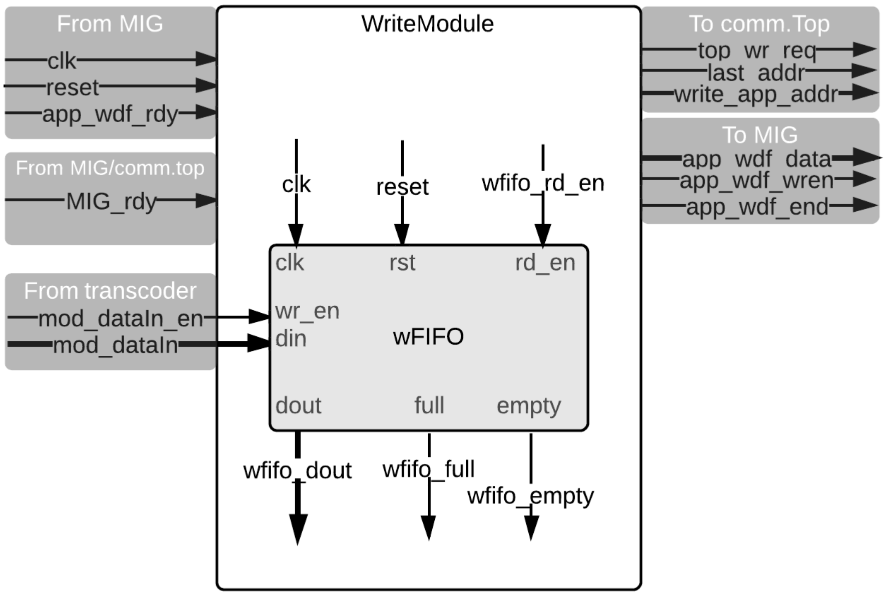


Figure 3.9: Overview of the writing module.

described further in the following, by explaining the state machine controlling it.

Writing Module State Machine

Because the writing requests have to be acknowledged by the MIG, the writing module has a state machine with behavior designed to handle this. The STD for the writing module's state machine is illustrated in Figure 3.10. The IDLE state is the initial starting point. It stays in IDLE until some data is written to the FIFO.

When the FIFO is no longer empty (wfifo_empty='0'), it goes to the S_WAIT state. At this point, it requests permission to write from the communication top module, by setting the top_wr_req signal high. It stays there until the write request has been accepted. In this context, *acceptance* means that the issued write command has been registered by the MIG, as illustrated in Figure 3.4 on page 19. This is detected by the writing module through the signal called MIG_rdy, which is handled by a process within the communication top module. The code for this is shown in Listing 3.2, and the writing signal is equivalent to the issuing of a write command (app_cmd = ``000'' and app_en='1'). If the MIG_rdy signal is high, it means that a write command is successfully issued and accepted by the MIG. When it is accepted, there are three different possible transitions, as listed in the following.

1. If the app_wdf_rdy signal is high while there still are elements in the FIFO, it goes to S_WRITE. This means that the current data write is accepted, and thus it goes on to writing more data.

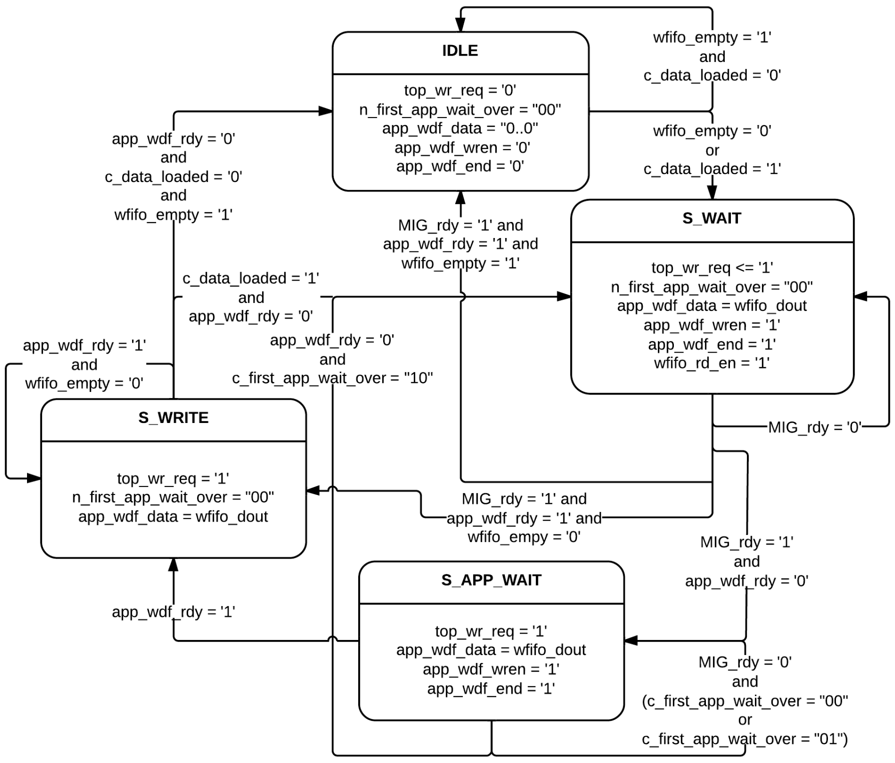

IDLE
top_wr_req = '0'
n_first_app_wait_over = "00"
app_wdf_data = "0..0"
app_wdf_wren = '0'
app_wdf_end = '0'
wfifo_empty = '1' and c_data_loaded = '0'
wfifo_empty = '0' or c_data_loaded = '1'
S_WAIT
top_wr_req <= '1'
n_first_app_wait_over = "00"
app_wdf_data = wfifo_dout
app_wdf_wren = '1'
app_wdf_end = '1'
wfifo_rd_en = '1'
MIG_rdy = '0'
app_wdf_rdy = '0' and c_data_loaded = '0' and wfifo_empty = '1'
MIG_rdy = '1' and app_wdf_rdy = '1' and wfifo_empty = '1'
c_data_loaded = '1' and app_wdf_rdy = '0'
app_wdf_rdy = '0' and c_first_app_wait_over = "10"
app_wdf_rdy = '1' and wfifo_empty = '0'
S_WRITE
top_wr_req = '1'
n_first_app_wait_over = "00"
app_wdf_data = wfifo_dout
MIG_rdy = '1' and app_wdf_rdy = '1' and wfifo_empy = '0'
MIG_rdy = '1' and app_wdf_rdy = '0'
S_APP_WAIT
top_wr_req = '1'
app_wdf_data = wfifo_dout
app_wdf_wren = '1'
app_wdf_end = '1'
app_wdf_rdy = '1'
MIG_rdy = '0' and (c_first_app_wait_over = "00" or c_first_app_wait_over = "01")

Figure 3.10: State Transition Diagram for the writing module.

```
write_process : process( writing , app_rdy )
    begin
        if writing = '1' and app_rdy = '1' then
            MIG_rdy <= '1';
        else
            MIG_rdy <= '0';
        end if ;
    end process ; -- write_process
```

Listing 3.2: The process for the `MIG_rdy` signal, in the communication top module.

2. If the `app_wdf_rdy` signal is high and the FIFO is empty, it means that there is no more data left to send. It will then return the `IDLE` state until more data is received.

3. If `app_wdf_rdy` is low when the write command is accepted, it goes to `S_APP-_WAIT`.

Otherwise, while the write request is not accepted, it keeps waiting in the `S_WAIT` state. To utilize the MIG's ability for delayed writes, the `S_APP_WAIT` state serves as a second waiting state. If the `MIG_rdy` signal has gone low during the transition, it starts counting. It stays here for at most two clock cycles, waiting for the `app_wdf_rdy` signal to be asserted. This is as the last case specified in Figure 3.4 on page 19. If the signal is asserted, it goes on to the `S_WRITE`. Otherwise, it returns to the `S_WAIT` state, to wait for a new acceptance of the write command.

In the `S_WRITE` state, the system stays until either the FIFO is empty, or `app_wdf_rdy` is deserted. If the FIFO is empty, it goes back to the `IDLE` state. If data has been loaded from the FIFO but has not been successfully written, it goes back to the `S_WAIT` while keeping the current data.

In all situations where the initial write command is accepted, the signals `app_wdf-_wren` and `app_wdf_end` are kept high, and the data on `app_wdf_data` is kept constant until it is successfully written.

The FIFO

Due to the possibility of the MIG not being ready to receive data, a FIFO is instantiated within the writing module. The FIFO is another Xilinx IP, made using the FIFO generator. The FIFO has several design options and specifications, which define how it should be used. One option offered is the First-Word-Fall-Through (FWFT), meaning that the `read_en` signal is an acknowledgment signal, rather than a read request. This means that data on the output port is replaced by the next element, when the read enable is asserted. [24, p. 13] This has been evaluated as possibly useful, by adding the ability to pop the next element when the data is transmitted, without having to wait for one clock cycle. For the proposed interface, this has been evaluated as not necessary, so it uses the regular `read_en` interpretation. This means that data is available on the FIFO output one clock cycle after a read request. It also has the possibility of using handshake operations for reading and writing, but these are also not used. This is because it would add some complexity to the writing module.

The FIFO used within the writing module is 128 bits wide, and can hold up to 512 elements. The built-in FIFO has been selected, as the Xilinx 7 series FPGAs contain dedicated logic in the block RAM This means that no additional Configurable Logic Block (CLB) logic is used for implementing it. [25, p. 45] An extensive list of the selected properties for the generated FIFO can be found in Appendix B.

3.2.5 Reading Module

The reading module only partially implemented with the rest of the design. This is because some reconstruction of the communication top module is needed, to request the initial

```vhdl
    type half1x8elem is array (0 to 3) of STD_LOGIC_VECTOR(7 downto 0);
    type FourXfourElem is array (0 to 3) of half1x8elem;
    type macroblock_type is array (0 to 15) of FourXfourElem;
    signal c_macroBlock, n_macroBlock : macroblock_type;
```

Listing 3.3: VHDL implementation of the macroblock type.

macroblock from the memory. However, it has been designed to interact with the rest of the system, and is described in the following.

The design assumes that each complete macroblock is read in either 4×4 mode, or 8×8, before a new macroblock is loaded. This is due to the way the amount of read data and requesting of new data is handled, as described in a separate paragraph.

The macroblock type is defined in hardware as shown in Listing 3.3 and the composition of a macroblock is also illustrated in Figure 3.11. The figure also shows the assumed order of data, in both transcoder read modes. The large numbers indicate the ordering of the 4×4 blocks, while the smaller indicate 1×8 vectors in 8×8 mode.

As may be seen by the VHDL implementation, each macroblock consists of 16 4×4 elements. Each one of these elements contains a quarter of a 8×8 block, or eight halves of 1×8 blocks. At last, every 1×8 block contains eight pixels.

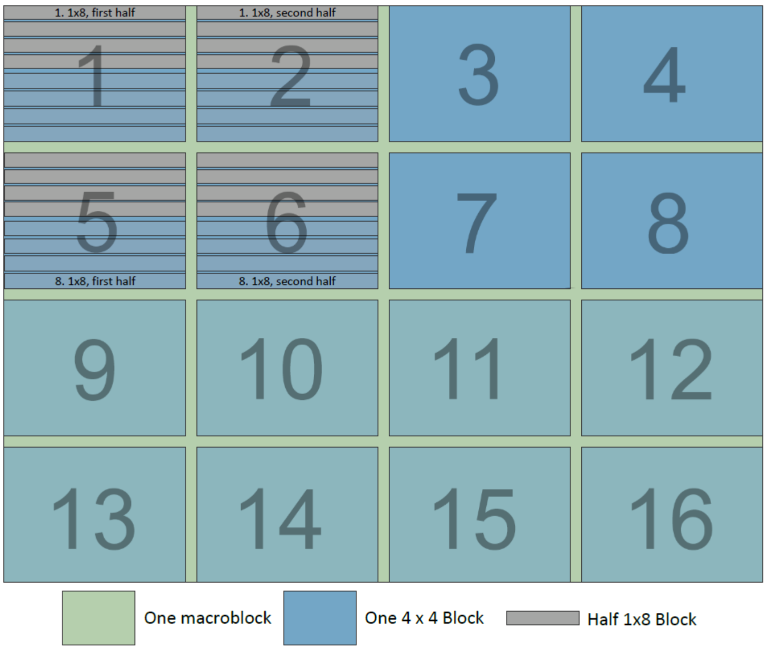


Figure 3.11: Macroblock composition.

By using such a composition to store a macroblock, a design has been made for the reading module which is able to send either 4×4 or 1×8 (in 8×8 read mode) when requested by the transcoder. The outputs to the transcoder are thus either a 128 bit or a 64 bit vector, respectively.

The building of a macroblock is done in a process within the reading module, sensitive to the `app_rd_data` and `app_rd_data_valid` signals. When the valid data is returned, it is stored in the appropriate place within the macroblock. Read requests are issued from within the state machine, as described in the following subsection.

Because the system, with the current MIG and FIFO configuration, reads 128 bits of data each clock cycle, 16 transfers from the memory are needed to construct one complete macroblock. This is handled by a separate process, as described in its own paragraph.

Reading Module State Machine

The state machine controlling the reading module is fairly simple. It reacts on read requests from the transcoder, and transits to the appropriate state. It sends data to the transcoder while the read request signal is high, for each mode. If the request is stopped, it returns to the `IDLE` state, waiting for a new request.

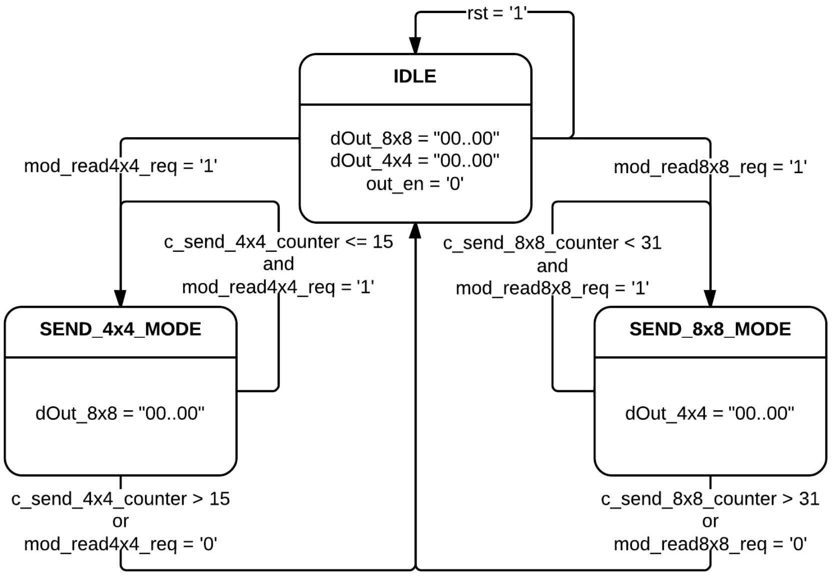
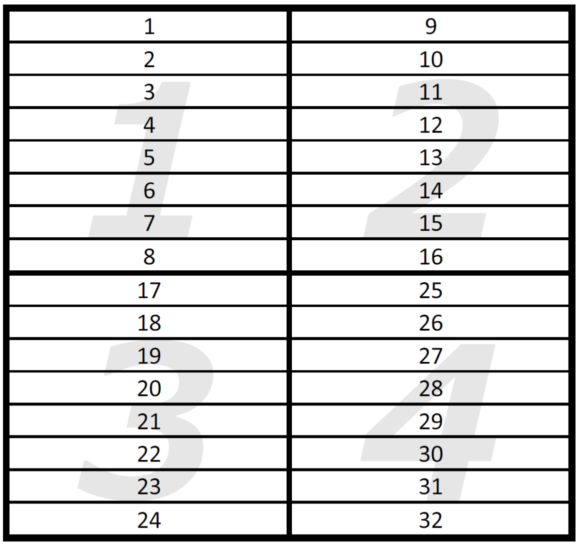


Figure 3.12: State Transition Diagram for the reading module.

When a 8×8 mode read-request is issued, the STD transfers to the appropriate state. As requested, the data is sent as illustrated in Figure 3.13, with eight pixels (64 bits) each clock cycle, when the `mod_read8x8_req` signal is asserted. The large numbers in gray denote the ordering of the 8×8 blocks. The same is the case, however with different ordering and data sizes, for the 4×4 mode. It should be noted that there is a delay of one clock cycle from a request is issued until the data is transferred. This means that one extra clock cycle is needed, when each read-request is issued, which has to be taken into account by the transcoder.



1	9
2	10
3	11
4	12
5	13
6	14
7	15
8	16
17	25
18	26
19	27
20	28
21	29
22	30
23	31
24	32

Figure 3.13: Illustration of the order in which data is sent, when in 8×8 mode.

Process for Constructing a Macroblock

Every 128 bits read from the memory are divided into two 1×8 blocks. The current implementation constructs the macroblock by going from the top to the bottom, 128 bits at a time. This means that the data which is stored first are 1×8 data blocks one and nine. Then two and ten, and so on. For a complete list of the receiving order, see Table 3.2.

Because of the specific way the macroblock is constructed within the reading module, it is required that each transfer of a macroblock is followed through in either 4×4 or 8×8 mode. As 64 bits are read each clock cycle when in 8×8 mode, and 128 bits when in 4×4 mode, the amount of data that can be fetched from the memory differs between the two. How this is handled, is described in the next paragraph.

Requesting New Data from the Memory

When appropriate amounts of data are read, the next macroblock needs to be constructed. This is handled by a separate process, sensitive on the reading mode the state machine is in. A latch has been used to store what type of reading mode was requested last. If 4×4 reading is requested, it starts issuing request for new data after the fourth 4×4 block is transferred to the transcoder. It continues requesting until the complete macroblock is transferred, and then continues for four more clock cycles.

If the mode is 8×8, it issues requests to the memory after data block number 15 is sent to the transcoder. It then keeps requesting until the complete macroblock is read, before it issues eight more requests. At this point, the next macroblock is fully constructed, and

Table 3.2: Order of received (128 bit) blocks.

Order of received data			
1.	First	and	Ninth
2.	Second	and	Tenth
3.	Third	and	Eleventh
4.	Fourth	and	Twelfth
5.	Fifth	and	Thirteenth
6.	Sixth	and	Fourteenth
7.	Seventh	and	Fifteenth
8.	Eighth	and	Sixteenth
9.	Seventeenth	and	Twenty-fifth
10.	Eighteenth	and	Twenty-sixth
11.	Nineteenth	and	Twenty-seventh
12.	Twentieth	and	Twenty-eighth
13.	Twenty-first	and	Twenty-ninth
14.	Twenty-second	and	Thirtieth
15.	Twenty-third	and	Thirty-first
16.	Twenty-fourth	and	Thirty-second

ready to be read. In both cases, 16 requests are issued to the memory, making sure that a complete macroblock is loaded from the memory.

Chapter 4

Results

This section presents simulation results for the designed modules, from top to bottom, as well as the combined communication interface design. As the design is yet to be verified on the development board, only simulation results are presented. The design has been successfully implemented using the Xilinx ISE PROJECT NAVIGATOR, and the results from the implementation reports are presented as well.

4.1 Simulation Results and Verification

All tests of the designed interface are only performed using simulations, as it is yet to be placed on the FPGA. This section presents several simulation results for all the designed submodules, as described in Section 2.5. The presented results for the communication top and writing modules are from complete system tests, while the presented results for the reading module are from a separate test.

Do note that a curved line (similar to the '≀' character) is seen in some figures, as for instance in Figure 4.2. This is used when no significant changes occur in the simulated signals over time, to make the presentation of the results clearer.

4.1.1 Communication Top Module

The top level testbench is simple, in the sense that the data values are equal to the addresses. This is done to make it easy to confirm whether or not the read data is valid, and if it is retrieved in the correct order. Do note that the simulation values for the address and data are in base 16, also known as hexadecimal.

Figure 4.1 shows the behavior of the communication top module when it receives the first data to write to memory. This is accepted and acknowledged through the `writing` signal going high. It now sets the address through `app_addr` and asserts `app_en`. When this is accepted by the MIG, the `MIG_rdy` signal is set high to notify the writing module that writing is accepted. This means that the subsequent data can be written, continuing until the `app_wdf_rdy` signal is set to zero.

The figure also shows that the writing address is kept constant after the `app_rdy` goes low, until it is high. At the same time, several data elements are transferred to the writing FIFO, as the `app_wdf_rdy` signal is high.

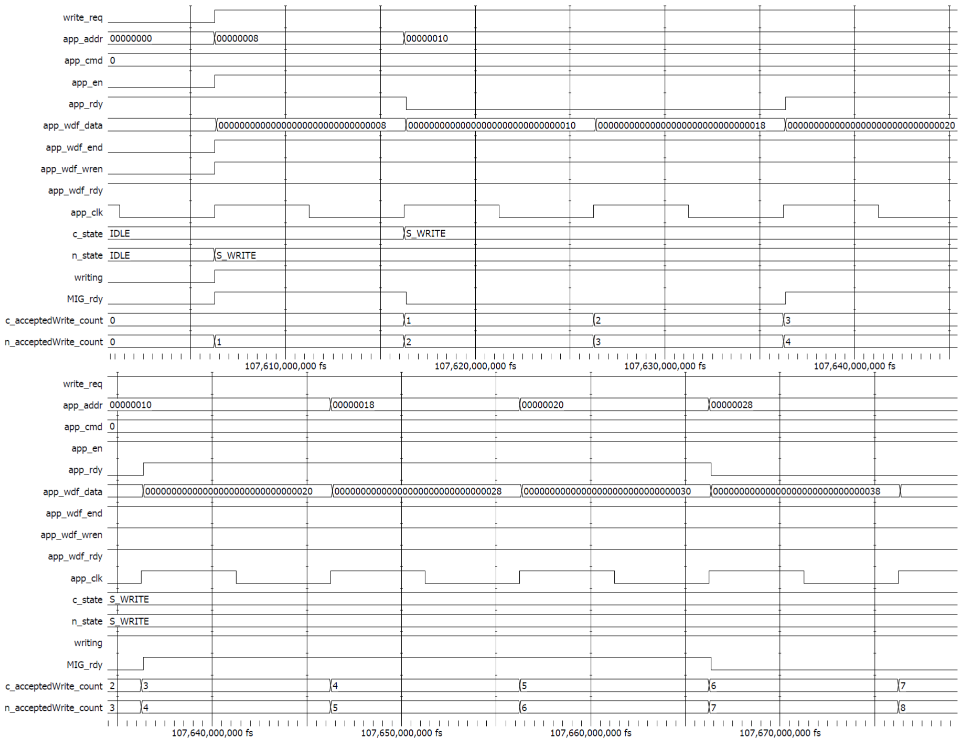


Figure 4.1: Simulation results for the communication top module receiving the first data to write, and corresponding write command, and transferring it to the MIG.

Further confirmation of the write success can be seen in Figure 4.2. It shows the transfer of the second value written, `128'h0010`. It can also be seen that the rank and bank selection is conducted, through the `ddr3_addr`, `ddr3_ras_n` and `ddr3_cas_n` signals, as well as writing through the `ddr3_we_n` signal. In addition, as opposed to the erroneous results presented in Section 4.1.4, one can also see that the address the data is written to is different.

As the current implementation of the communication top module also handles reading from the memory, this behavior has also been tested. Figure 4.3 shows how the communication top receives a high reading request through the `mod_readReq` signal, and goes on the requesting reading from the memory. One can see that every request is held until its accepted, before the address is incremented. At a later time, the data is successfully received from the memory. This is shown in Figure 4.4. It is possible to see that the counting of the received data, with the `read_rec_count` has a change at the end of each received block. This is because the `app_rd_data` and `app_rd_data_valid` signals are not completely synchronized with the clock, and are set low a short period after the

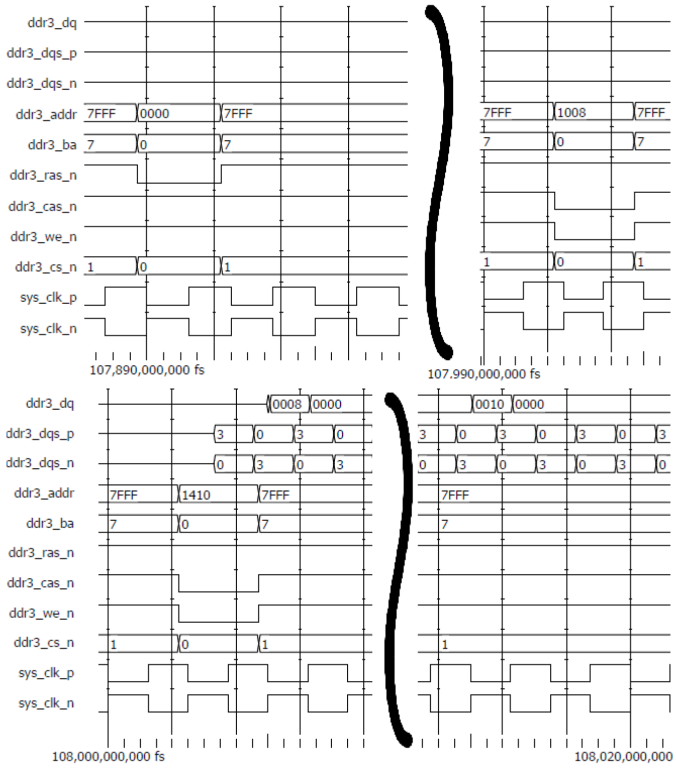


Figure 4.2: The DDR3 signals from the MIG, confirming that data from the MIG's writing FIFO is successfully written to the DDR3 memory.

positive clock edge. The counting is still correct, as the increment is discarded as soon as the app_rd_data_valid signal is set to low.

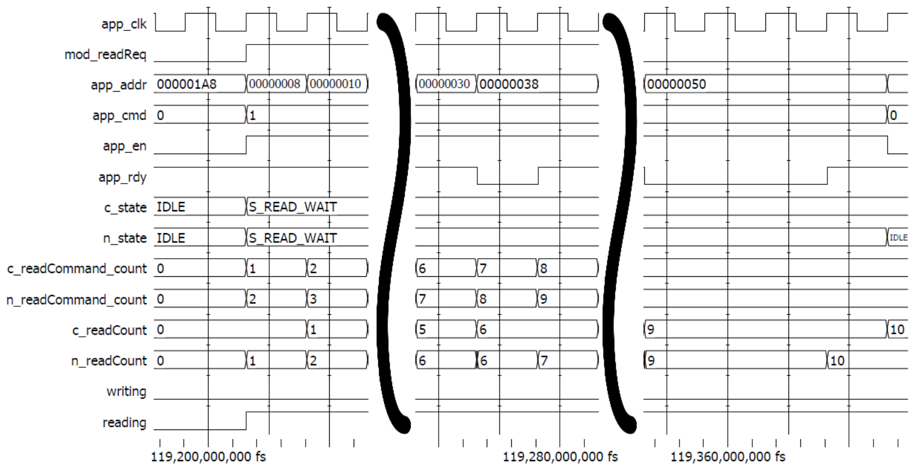


Figure 4.3: Simulation results showing the how the communication top module successfully issues read request, based on the received mod_readReq signal.

Reading and Writing Latency

A noteworthy observation has been made with regards to the latency between the time a read or write request is issued, and the time the data is received from or written to the memory. Table 4.1 lists the different latency values for reading, from one of the tests run. The clocks running the UI and proposed design clocks were set to 100 MHz. Because the MIG is not necessarily ready to receive commands, the values have an additional difference between different blocks of data. The mean values for the measured reading latencies are 307 ns from a request is accepted, and 317.5 ns from a request is issued.

It should be noted that this is not an extensive test, as only twenty measurements are used. The values presented in the table are from a test with a long initial writing request sequence, followed by several reading requests. To examine if the latency changed when a sequential combination of reading and writing requests was issued, this was also tested. These results indicated that the latency was slightly lower, but this was probably coincidental. Even though the presented figures might change for a more extensive work load on the memory, they are useful as an indication of the reading latency.

For the case of writing, the latency between time a command was accepted until the data was transferred to the memory was longer. In most cases it was close to the reading latency, at about 300 ns, while it could also be as high as 600 ns. These results are, however, also depending on that data has been transferred to the MIG's writing FIFO.

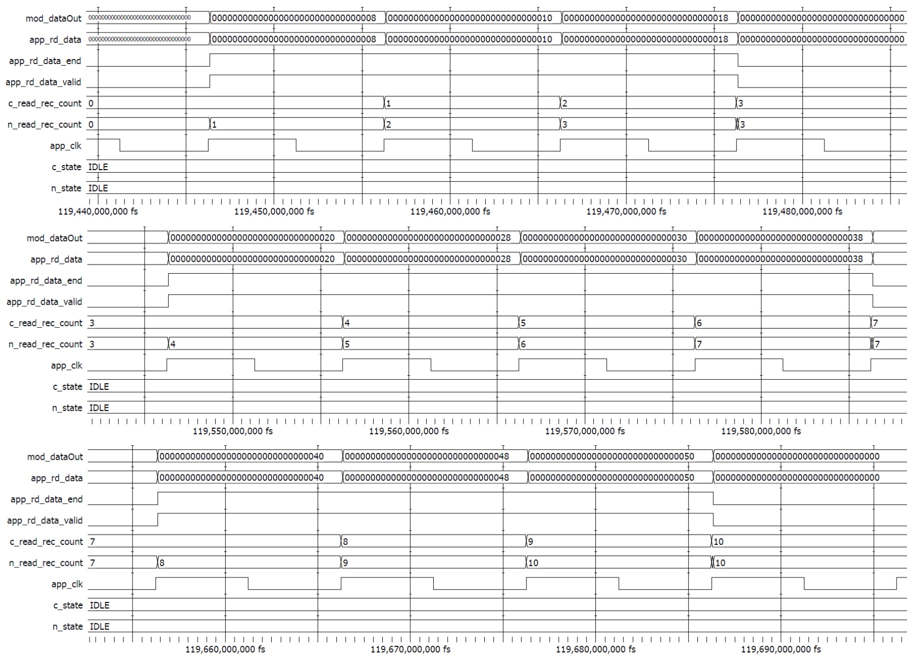


Figure 4.4: Simulation results showing the data being received from the memory, and sent further on to the transcoder, through the `mod_dataOut` signal.

Table 4.1: Latency from issuing read requests to the time of data read-back.

Reading Address	**Time after request was accepted [ns]**	**Time after request was first issued [ns]**
`128′h0008`	240	240
`128′h0010`	240	240
`128′h0018`	240	240
`128′h0020`	310	310
`128′h0028`	310	310
`128′h0030`	310	310
`128′h0038`	300	310
`128′h0040`	370	370
`128′h0048`	330	370
`128′h0050`	300	330
`128′h0058`	240	240
`128′h0060`	310	310
`128′h0068`	310	310
`128′h0070`	310	310
`128′h0078`	310	310
`128′h0080`	380	380
`128′h0088`	330	380
`128′h0090`	330	380
`128′h0098`	300	330
`128′h00A0`	370	370

4.1.2 Writing Module

Due to the stochastic behavior within the SDRAM model, the initial test for the writing module was not exhausting. By this, it is meant that the behavior of the `app_wdf_rdy` and `app_rdy` and the timing between them was not perfectly replicated in the testbench. It still did serve as an indication of whether or not the designed module behaved as expected, and reacted to the inputs as it should. The results presented in this subsection are from the complete system test, as in the previous section.

As can be seen from the figure, the `mod_dataIn_en` signal goes high, when the data on the `mod_dataIn` is valid. When this is successfully stored in the FIFO, the `wfifo_empty` goes low and a write request is issued from the writing module to the top, through the `top_wr_req` signal.

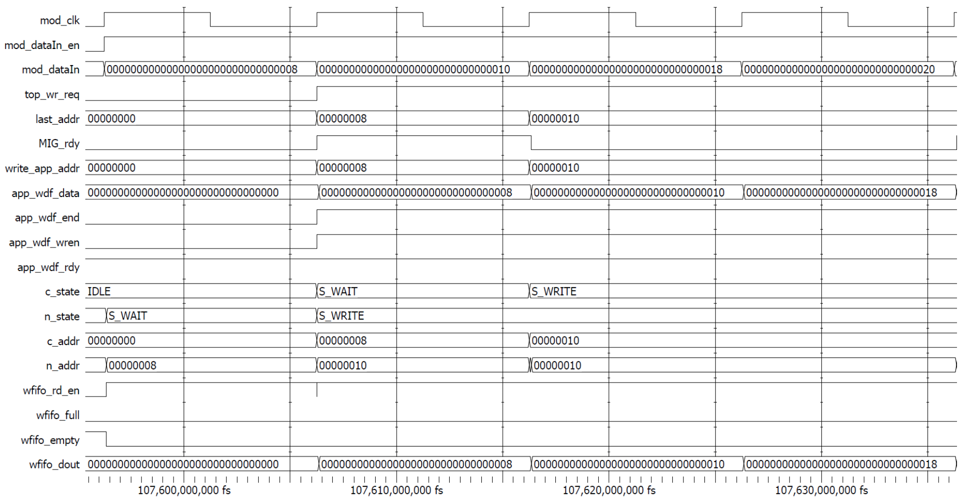


Figure 4.5: Simulation results for the writing module receiving the first data to write, and transferring them through the communication top module, further on to the MIG.

As stated in Section 3.2.4, the writing module handles the case when waiting at most two clock cycles after the `app_rdy` signal goes low. Results showing this are presented in Figure 4.6, and it is clear from the figure that the implementation works. Two cursors are used to emphasize that the time between `app_rdy` going low and `app_wdf_rdy` going high is two clock cycles, as they are not synchronized with the clock.

4.1.3 Reading Module

The reading module has mostly been tested individually, and the results presented here are from these tests. It has also been tested with the complete system, but because some functionality still needs to be added, it has not been completely validated with the complete system. It has been tested to correctly receive data from the memory, but the reading requests are still done through an input signal.

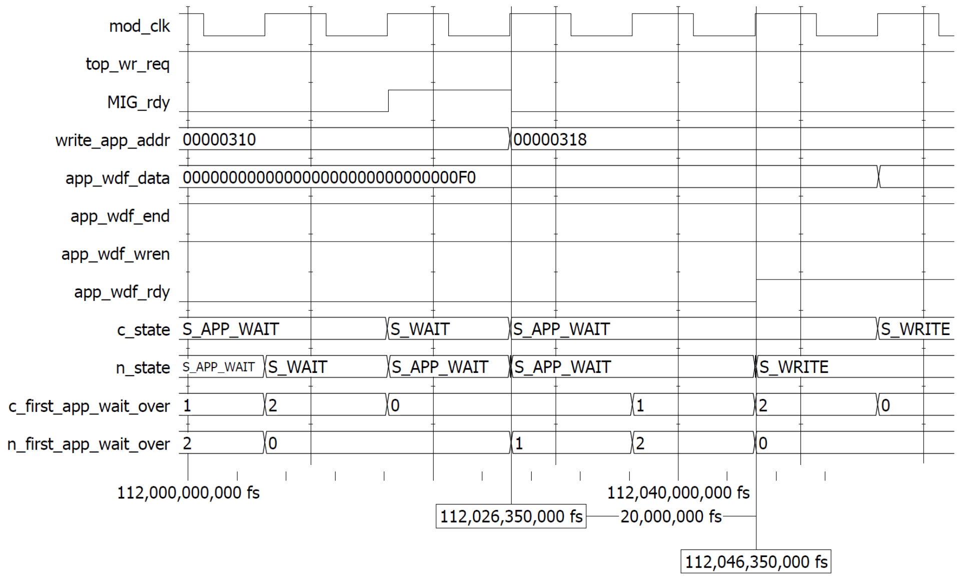


Figure 4.6: Simulation results confirming how the writing module handles the two clock cycle delay of `app_wdf_rdy` after an accepted write request.

The testbench simulates the behavior of the read path of the MIG, with `readData` and `read_valid` corresponding to the MIG's `app_rd_data` and `app_rd_data_valid`. The macroblock used in the test is consistent with the data shown in Table 4.2. Each square in the table represent one 4×4 block.

Data is received in the order specified in Section 3.2.5, from top to bottom, 16 pixels at a time. The receiving order is also is illustrated in Table 4.3, for clarity.

The successful construction of the macroblock is shown in Figure 4.7. Each color corresponds with one 128 bit block of data received, and shows where it is stored. It is clear that the counters are working properly, and that all data is stored correctly.

The macroblock structure can also be seen, by looking at each vector within the `n_macroBlock` signal. After four clock cycles, the two first 4×4 blocks are filled with data, corresponding with the values shown in Table 4.2.

Figure 4.8 shows how the module successfully sends data eight pixels at a time. The figure only shows the first and last six, but the others are also successfully transferred. A read request in 4×4 mode is presented in Figure 4.9, showing the transfer of the first two 4×4 vectors.

Table 4.2: Value assignments (hexadecimal) in the macroblock used for testing the reading module.

01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20
21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30
31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	40
41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50
51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60
61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70
71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	80
80	7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71
70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61
60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51
50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41
40	3F	3E	3D	3C	3B	3A	39	38	37	36	35	34	33	32	31
30	2F	2E	2D	2C	2B	2A	29	28	27	26	25	24	23	22	21
20	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11
10	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01

Table 4.3: Order of the data written used in the testbench for the reading module.

128 Bit Hexadecimal Value	
1.	`128'h0102030405060708090A0B0C0D0E0F10`
2.	`128'h1112131415161718191A1B1C1D1E1F20`
3.	`128'h2122232425262728292A2B2C2D2E2F30`
4.	`128'h3132333435363738393A3B3C3D3E3F40`
5.	`128'h4142434445464748494A4B4C4D4E4F50`
6.	`128'h5152535455565758595A5B5C5D5E5F60`
7.	`128'h6162636465666768696A6B6C6D6E6F70`
8.	`128'h7172737475767778797A7B7C7D7E7F80`
9.	`128'h807F7E7D7C7B7A797877767574737271`
10.	`128'h706F6E6D6C6B6A696867666564636261`
11.	`128'h605F5E5D5C5B5A595857565554535251`
12.	`128'h504F4E4D4C4B4A494847464544434241`
13.	`128'h403F3E3D3C3B3A393837363534333231`
14.	`128'h302F2E2D2C2B2A292827262524232221`
15.	`128'h201F1E1D1C1B1A191817161514131211`
16.	`128'h100F0E0D0C0B0A090807060504030201`

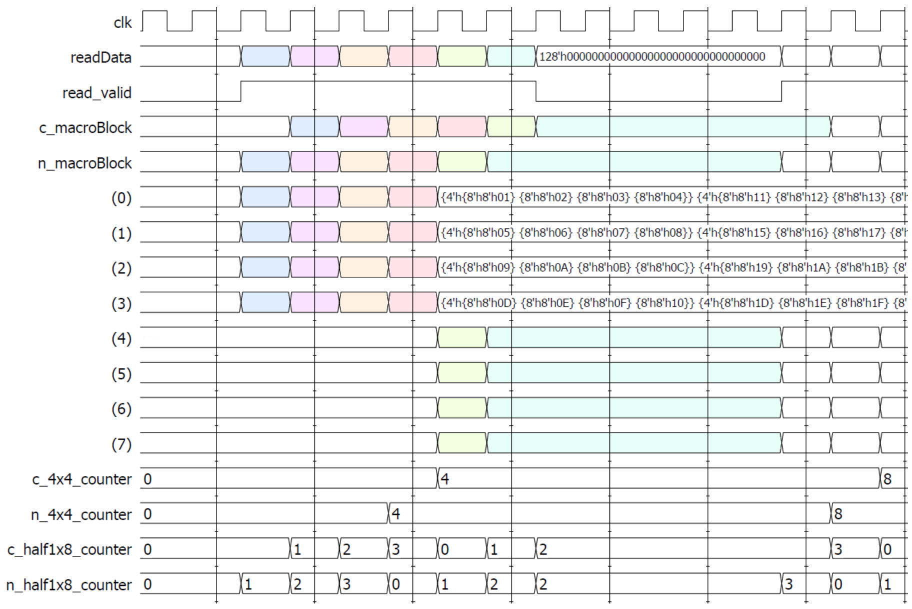


Figure 4.7: Reading module receiving six blocks of 128 bit data.

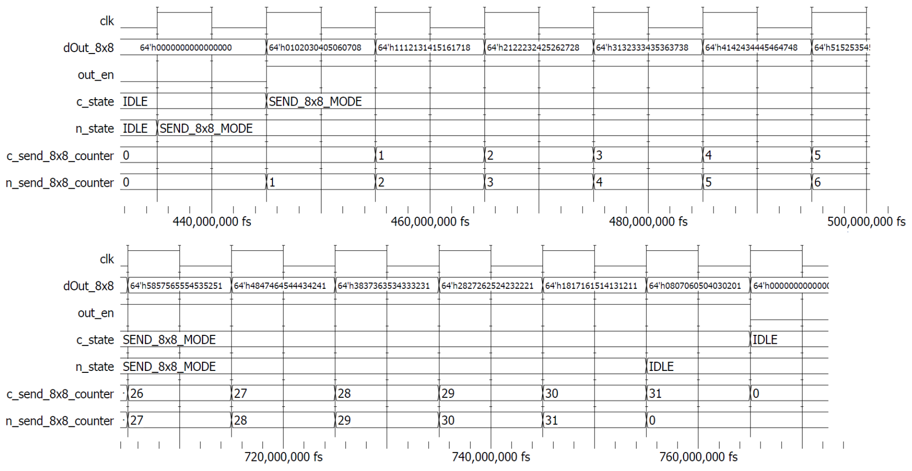


Figure 4.8: Reading module transferring 1×8 pixel data, in 8×8 mode.

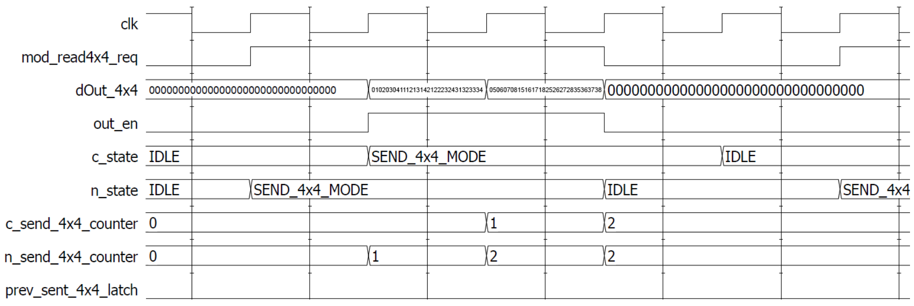


Figure 4.9: Reading module transferring data 4×4 pixel data, in 4×4 mode.

4.1.4 Overwriting Data and Erroneous Reads

An issue arose early in the design process, with erroneous data being read from the memory. Late in the design process, it was discovered that erroneous data read from the memory was due to write requests not being issued correctly. In addition, the way they were issued lead to overwriting of data, on the rare occasions they were accepted. The result from this can be seen in Figure 4.12. Here, one can see how the selected column address is the same for both `128'h28`, `128'h30` and `128'h38`, in the same rank. This means that the first two data values are written, but replaced by the last one. The figure also shows that the value `128'h40` is successfully written.

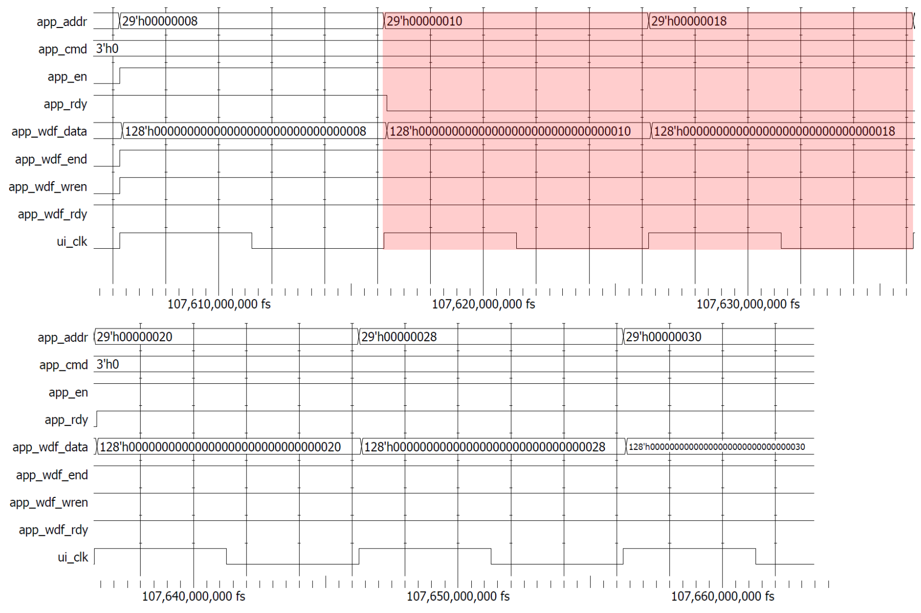


Figure 4.10: Erroneous issuing of write commands - part one.

Figure 4.10 shows the issuing of write commands, leading to the described erroneous behavior. As shown in the figure, first address `29'h8` is accepted, before `29'h20`, `29'h28` and `29'h30` are registered. In the meantime, data `128'h10` and `128'h18` are accepted by the MIG's FIFO, while the addresses are not accepted. This is the area in red in the figure. After this, `app_rdy` is low until the scenario shown in Figure 4.11, where address `29'h90` is accepted three times. Two of the three are emphasized in red in the figure. This is the reason for the two overwrites seen in the DDR3 signals in Figure 4.12.

After the presumed writes, read requests are issued for the addresses from `29'h8` to `29'h98`. 19 values are returned with the `app_rd_data_valid` set high. However, only six of these return written data, while the rest is just initial values in the different parts of the SDRAM. Figure 4.13 shows the read data, and Table 4.4 lists all the returned values, and the corresponding address they are read from.

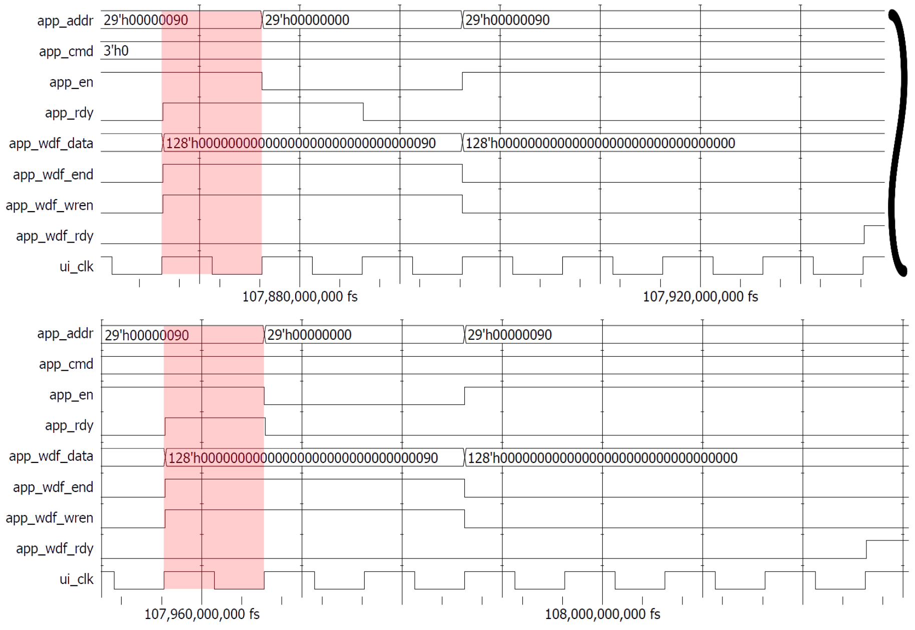


Figure 4.11: Erroneous issuing of write commands - part two.

Table 4.4: Data read from memory when the writes were incorrectly issued, with their corresponding addresses.

Address	Data
`29'h0008`	`128'h0008`
`29'h0020`	`128'h0010`
`29'h0028`	`128'h0018`
`29'h0030`	`128'h0020`
`29'h0090`	`128'h0038`
`29'h0098`	`128'h0040`
`Others`	Initial value `(erroneous)`

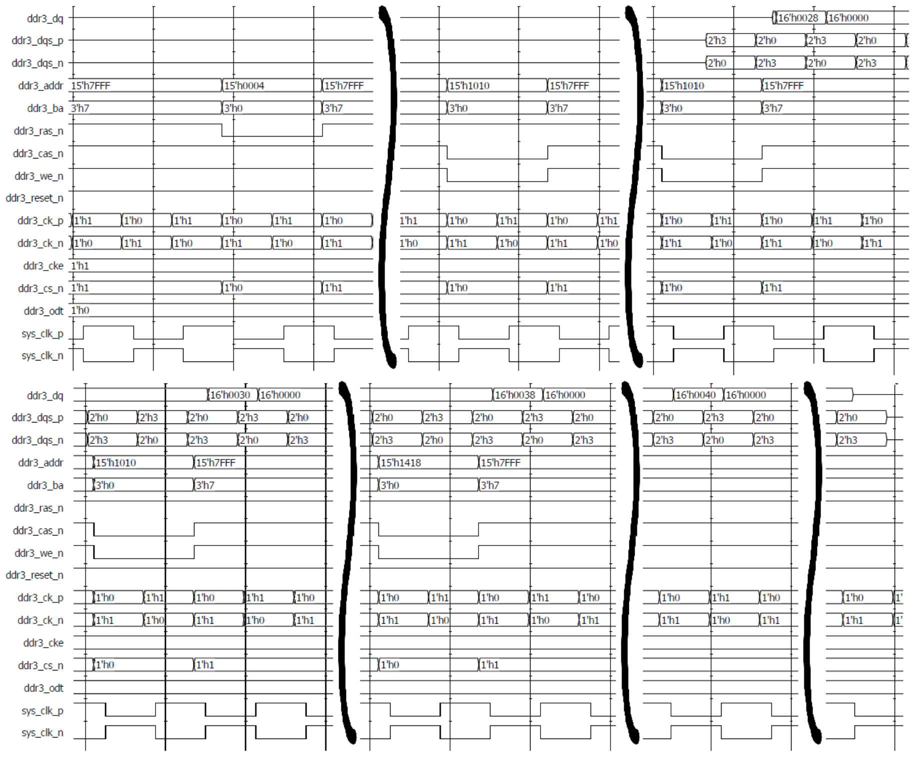


Figure 4.12: Overwrites seen on the DDR3 signals.

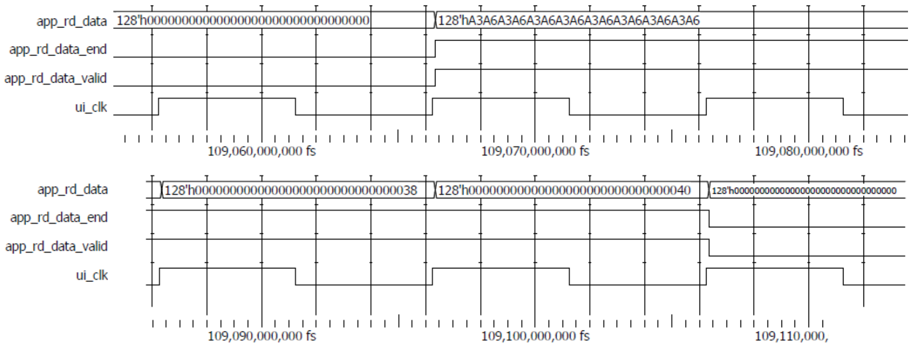


Figure 4.13: Erroneous data received from the memory.

4.2 Implementation Results

Further verification of the design was conducted by successfully running the implementation process on the different modules. The design has been successfully synthesized and implemented using ISE PROJECT NAVIGATOR, without any errors. Implementing reports issues with the design, as well as the utilization of the targeted FPGA. Excerpts from the utilization reports can be seen in Tables 4.5 and 4.6, and the complete reports can be found in Appendix C.

The communication top module has not been implemented separately, as the great amount of Input Output Blocks (IOBs), made the implementation process fail. Extensive modification was needed to reduce this, as opposed to the example top design, where the `mod_dataOut` and `mod_4x4_dout` were simply disconnected.

From the reported values, it is clear that the design does not require a large part of the FPGA. The most utilized blocks are the IOBs, which will be reduced as the design will be a part of a larger system. Table 4.5 shows an increase in the design with the interface, compared to original example, where the traffic generator is used. The increase is not of great significance, as it is small in comparison.

The greatest part of the available slice registers and Look-up-tables (LUTs) are utilized by the reading module. This is because of the way the macroblock is constructed and stored, which requires a lot of area.

Table 4.5: Slice logic utilization reported after implementation, for example top with and without the proposed interface.

Slice Logic Utilization	Available	Original example top		Modified example top	
Slice Registers	407,600	5,165	1 %	7,164	1 %
Slice LUTs	203,800	6,132	3 %	8,123	3 %
Occupied Slices	50,950	2,537	4 %	3,283	6 %
Bonded IOBs	500	54	10 %	251	50 %
RAMB36E1/FIFO36E1s	445	0	0 %	2	1 %

Table 4.6: Slice logic utilization reported after implementation, for the writing and reading modules.

Slice Logic Utilization	Available	Writing module		Reading module	
Slice Registers	407,600	43	1 %	2,077	1 %
Slice LUTs	203,800	108	1 %	3,240	1 %
Occupied Slices	50,950	62	1 %	1,073	2 %
Bonded IOBs	500	322	64 %	327	65 %
RAMB36E1/FIFO36E1s	445	2	1 %	0	0 %

As may be seen in the complete reports, in Appendix C, some warnings are reported. For the writing module, these are caused by two unused bits in the address signal, caused by incrementing by eight, in addition to unconstrained location mapping of its IOBs.

Many of them are the same for the reading module, being caused by unconstrained locations for IOB and incrementing counters by four and eight. The last ones are because of a latch, which has been intentionally used to keep track of the last requested read mode. It does report that the latch is both set and reset asynchronously, as it gets its value from the combinatorial state machine. The synthesis report states that this might work, but recommends that an alternative solution is used.

For the proposed interface, which is part of the modified example top, many warnings are issued. The majority of these are for parts generated through the MIG, which are present in the implementation of the original version as well. The only additional warnings are the same as specified above.

The synthesis part of the process also reported maximum frequencies for the different modules. These are listed in Table 4.7. Initially, the implementation of the reading module reported a failing time constraint, when no constraints were defined. When a timing constraint for the clock was added, at 100 MHz, it was reported that all constraints were met. The modified example top design also meets all the timing constraints, which were automatically made when the MIG was generated.

Table 4.7: Maximum frequencies reported after synthesis.

Module	**Maximum frequency**
Modified example top	308.414 MHz
Writing module	353.851 MHz
Reading module	591.226 MHz

4.3 Simulation Difficulties

Throughout the development process, there have been several difficulties with running the example design simulation, generated by the MIG. The main issue has been the inability of simulating the design files, when VHDL was the chosen language. When simulation was run in ISIM, an error regarding assignment of the `phy_dout` signal, reporting that it is out of bound, causing the simulation initialization to fail. Because of this, different alternatives have been tried. Before a working edition of MODELSIM was obtained, simulation using ACTIVE-HDL 9.3 STUDENT EDITION was attempted.

It was found viable because of its support for mixed language designs, as well as the including of pre-compiled Xilinx libraries. Xilinx libraries can also be compiled as mentioned in Section 2.6.2, as described in [26].

ISE DESIGN SUITE was set up according to the specifications at [27], and simulation was initiated. However, the simulation of the VHDL version stopped after a while, reporting the error shown in Figure 4.14.

When the Verilog version was run, the error did not stop the simulation. However, it still stopped due to the size restriction of the AHDL Student Edition, as seen in Figure 4.15.

```
# ELAB2: Fatal Error: ELAB2_0031 ...
mig_7series_v1_9_iodelay_ctrl.v (167):
Value ’A’ out of range (false to true).
```

Figure 4.14: Error in Active-HDL for VHDL version of the MIG.

```
# ELAB2: Elaboration final pass complete - time: 45.5 [s].
# KERNEL: SLP loading done - time: 0.2 [s].
# KERNEL: Error: The size of your design has exceeded the
  maximum capacity of the Student Edition.
# KERNEL: Error: For upgrade options please visit our
  University Program page at www.aldec.com.
# KERNEL: stopped at time: 0 ps
# Error: Fatal error occurred during simulation
  initialization.
```

Figure 4.15: Error message in Active-HDL during simulation of Verilog version of MIG.

Chapter 5

Discussion

This section discusses the developed interface, and different aspects of it. The achieved performance and results are evaluated, and the quality of the tests performed is also addressed. Because some difficulties and misbehaving occurred during the design process, the reasons for this are also discussed. Assessment of the most significant choices leading to the presented design are also assessed. In the end, the possible future improvements to the current design are presented.

5.1 System Architecture

The proposed interface consists of the communication top module, which is connected to the MIG. The communication top is used to mediate between read and write access, with priority on reading. The reason for this has been that the only way to transition to the reading state is by a read request signal. This should be modified, for instance by comparing the number of received and issued reading and writing commands. If no more write commands are received, it should go to the `READ` state and issue read requests if more commands are received from the transcoder than commands issued to the MIG. The command path is working as specified in the documentation, and successfully serves as an interface for the transcoder. The following sections discuss the two submodules instantiated within it, as well some other aspects of the proposed design.

5.1.1 Writing Module

As previously stated, the proposed design uses the memory interface IP, the MIG, as an abstraction level above the low level DDR3 SDRAM signaling and control. A long time was spent on understanding the interface specifications as they were described in [19], analyzing the generated simulation testbench, and comparing the two.

The most important challenge was interpreting the specifications presented in Section 3.1. More specifically, how to handle writing as described in the scenario illustrated in Figure 3.4 on page 19. Because both the address and data are labeled as '0', it could be interpreted as that a write command and data write transfer should occur with at most two clock cycles delay. It is also stated in the guide that the maximum delay between the write data and its associated write command is two clock cycles. [19, p. 130] In combination with the ability of writing data back-to-back, after an initial write command is accepted (as depicted in Figure 3.5, on page page 20), this was misconceived as an ability of writing successfully, regardless of the command path's `app_rdy` signal going low. This was the reason for the faulty design behavior, which produced the faulty results presented in Section 4.1.4. Fortunately, this was discovered and the behavior was corrected.

It is not explicitly specified in the documentation what should be done in the event that the command path is no longer accepting commands while the MIG's writing FIFO still is ready. In the proposed design, a choice is made to continue transferring data until the `app_wdf_rdy` signal is set low. In addition, the write path's `app_wdf_end` and `app_wdf_wren` signals are set high as soon as data has been stored in the module's FIFO, while loading data on the `app_wdf_data` bus. These decisions were made because they were evaluated as a better utilization of the available time and bandwidth. The described choices do introduce two possible issues, due to the way the writing module has been designed.

The first issue is the possibility of the second scenario depicted in Figure 3.4, where the `app_rdy` signal goes high, and a write command is accepted, one clock cycle after `app_wdf_rdy` has gone low. The reason for this possible weakness is that a control checking for that particular situation has not been implemented. The scenario does rarely occur in the simulations, but when it does, data is written to two constitutive addresses.

The second issue is caused by the way the write commands are issued to the MIG. The presented design issues write commands only while the write data FIFO within the writing module is not empty. In the simulations that have been run, the MIG's write data FIFO is ready more often than the commands are accepted. After some time, this leads to the data preceding its corresponding write command, and associated address. Because of this, the data transfer is done before an adequate amount of write commands are issued, and the module falsely believes that it has successfully finished the data transfer.

One solution to the first issue could be introducing a new state for waiting for the `app_rdy` signal for one clock cycle, analogous to the solution for the opposite scenario with the two cycle delay of `app_wdf_rdy`. Alternatively, by only transferring data after a command is accepted, the issue would be avoided completely, at the possible expense of a decrease in the utilization of the available bandwidth.

The second problem could possibly be solved by extracting the issuing of write commands from the writing modules Finite State Machine (FSM). It should still be initiated by a control signal from the state machine, to keep the relation between the issuing of the two. Alternatively, the FSM could be modified to issue write commands in a different manner. The last option could be leaving the issuing of write commands completely to the communication top module, as is the case for the reading requests. In any of the three cases, a control should be introduced to verify that the amount of write commands and transferred data blocks are equal at some intervals. Issuing enough write commands is more important, as data transferred at a later time still would align with the addressing order.

Overwriting Issue

At an early point of the development process, an issue with receiving erroneous data from the SDRAM arose. Analysis of the DDR3 signals indicated that data was successfully written to the memory, but was erroneous when read back. This lead to a misguided conclusion about the issuing of the read commands being faulty, and a great amount of time was spent trying to find the reason for this issue.

It was later discovered that the issue was caused by an erroneous interpretation of the relation between the write and command paths. It was seen that data was written to the memory, but it took a long time before it was discovered that the rank, bank and address selection for different data was the same, when it was transferred to the DDR3. Fortunately, this issue was resolved, by modifying the writing module.

5.1.2 Reading Module

As stated in Section 3.2.5, the proposed reading module is not completely developed. A solution for constructing macroblocks and reacting to requests from the transcoder is completed, but some challenges are still left. The main challenge is handling the requesting of data from the external DDR3 memory, while avoiding the possibility of overwriting the current data. One preliminary solution has been developed, but as stated in Section 4.2, it introduces an asynchronous latch. This is because it uses a latch to keep track of what type of mode was requested last by the transcoder - either 4×4 or 8×8 mode, using asynchronous setting and resetting. This would have to be examined further, and possibly modified.

One solution could be using a FIFO as a buffer. This way, one could issue read requests to the memory as soon as one macroblock is constructed, without the need of the more complex read request issuing. By buffering the data to the FIFO first, the possibility of overwriting data in the current macroblock would be avoided, and data could be read in a more regular fashion, controlled by the FSM.

Another possible issue is with the current used ordering. For consistency and simplicity during the propositional design process, it has been assumed that the order the transcoder would read the data is as described in Figure 3.11 on page 30. This way, the further relation between 4×4 and 8×8 blocks has been utilized, during macroblock construction and read-back to the transcoder. If this is not the case, as other orders are presented in [5], this will have to be modified.

5.1.3 Using a FIFO Write Buffer

By using the Xilinx FIFO, several limitations are introduced. As implied by the the name, data transferred through a FIFO is forwarded to the memory in the received order. If this is not the case, as some of the suggested reading orders for 4×4 blocks in [5], the data must be reordered. One way would be reordering data before the transcoder transmits it, but this would introduce some undesirable delay. An alternative solution would be modifying the address order in such a way that data is read back from the SDRAM memory in the desired order.

It should also be noted that the proposed design does not respect the FIFO's `full` signal. This is because the current FIFO size of 512 elements of 128 bits has been more than enough for testing purposes, and it has not been filled up completely. Furthermore, this size can be expanded, which could make it possible to avoid the FIFO ever becoming full. Additional information about the frequency and relation between writing and reading from the transcoder would be needed to select an adequate FIFO size. Initial testing indicates that if the amount of reading and writing commands is similar, the FIFO would never be filled up completely.

5.1.4 Keeping Track of Reads and Writes

Both the communication top and the writing module are using unconstrained counters to keep track of the number of received data packets, the number of reads issued and the number of received read commands. These are of type `natural`, and are currently only reset at reset. This means that they are vulnerable for reaching overflow if the number of read or write commands exceed ($2^{32}-1$). This would certainly cause unexpected behavior.

An attempt has been made to make a counter which adds one when a data block is transferred, and subtracts when a read command has been issued. This was unsuccessful, which is why the current implementation uses separate counters for each action. This should be modified to avoid the possibility of overflow. Furthermore, the control within the communication top module, for not issuing more read requests than successful writes, could be faulty. Because the proposed design counts accepted data transfers, instead of accepted write commands, the possibility of requesting data from an empty address is possible. It is unclear if this would be an issue in a system with the transcoder, as data addressing could be arranged in a different order, but should be modified. Although the error did not present itself in the simulations that have been run, it should be modified to comparing the number of read requests to the number of accepted write commands and associated data transfers. There would still be a chance of error, due to the write latency from the MIG to the memory, as mentioned in Section 4.1.1. This is difficult to account for, apart from introducing an adequate delay between the writing and reading to and from the same address.

5.2 Verification Results

The proposed communication interface has only been verified through simulation testbenches. This is because a framework for testing design on hardware has not been developed. Although simulation might be limited, it can be used to indicate correctness in certain scenarios. In addition, using the DDR3 memory model supplied by Micron Technology, was it possible to simulate realistic behavior for the external memory.

All test of the communication top module verified that the proposed interface is able to both write and read data to and from the simulated memory. The design has been tested with both a long initial sequence of write requests, followed by a number of read requests, as well as a combinations of reading and write request sequences. It is possible that the real scenario, when connected to the transcoder would be different. To be able to test this, it would be necessary with more information about the behavior of the transcoder.

The reading module has been successfully tested as a part of the communication top, but also separately. Simulating it alone verified that it reacted correctly to the different modes from the transcoder, by transferring correct data in the requested format.

There are two possible scenarios that have not been tested. Both have to do with immense amounts of data transfers. Because the proposed design only increments both the reading and writing addresses, as well as many of the counters used, unexpected behavior could occur. From the documentation, it is uncertain how the MIG would handle addresses out of its range. The other case is when the unconstrained counters reach overflow. These two scenarios should be verified, and precautions made as necessary.

To make an absolute assessment of the design's correctness, more extensive verification must be conducted. One option could also be using formal verification techniques and tools, which would be useful to verify the correctness of the designed modules further. Unfortunately, only initial testing has been done, due to time constraints. However, the obtained results from the performed tests do indicate that the design is promising as a memory interface.

5.3 Implementation Results

The design has been successfully implemented, without any errors. Some warning are issued, as described in Section 4.2, but only the one about the asynchronous set and reset latch is of importance. The latch was intentionally used, but the asynchronous behavior should be avoided. One solution is suggested in Section 5.1.2.

The utilization report shows that the proposed design does not utilize a large part of the targeted Kintex-7 XC7K325T FPGA. By occupying only 8,123 of the FPGA's LUTs, and small amount of registers, a great amount of area is left for the transcoder. This means that is should be possible to implement it aside the existing design.

5.4 Further Work

Throughout this thesis, a design has been developed which might serve as a memory interface for the H.264/AVC transcoder. There are, however, still some aspects of it that need addressing.

The most pressing improvement would be modifying the design to accommodate for use with the DDR3 SDRAM controller available on the FPGA. As stated earlier, the memory part number was not found until late, causing the selection of a different memory part. Because the available part is different and has a wider data width than the one selected, this would require some reconstruction of the design.

The next thing would be, as stated previously, to rearrange the order of transferred blocks, if it is different than the current implementation. If the pixel size is different from eight bits, this would also require some modifications.

In addition to the possible vulnerabilities mentioned in the previous sections, the system must also wait for the `init_calib_complete` signal from the MIG to be asserted, before transferring data. This is important as this signal indicates that the initialization and calibration of the memory is complete, and that the interface is ready.

Chapter 6

Conclusion

The main goal for this thesis has been to develop a working interface to an external DDR3 memory. A working communication interface has been developed, which contains separate modules for handling reading and writing operations. The proposed design utilizes the MIG IP supplied by Xilinx, as a memory interface controller. The proposed design also has the ability to transfer data to the transcoder in either 4×4 or 8×8 mode, according the request issued by the transcoder.

Before the proposed interface can replace the current implementation using the MicroBlaze for memory handling, more extensive testing is needed. The design has only been verified through testbench simulations, at this point, by running different combinations of read and write requests. It has also been implemented using Xilinx' ISE PROJECT NAVIGATOR, to verify that the design has no possible errors. The reported area utilization for the complete system was 8,123 slices, of which 3,283 are occupied.

The interface runs at 100 MHz, and is according to specifications. Synthesis reports have indicated that 308 MHz is the maximum frequency for the complete system, which suggests that it would support the running frequency of the transcoder.

From the obtained results, it seems like the proposed design could be a viable alternative memory interface for the existing H.264/AVC transcoder. Although initial results seem promising, more extensive testing must be conducted to assure complete correctness. Some improvements are also needed, as described in the previous chapter. The most pressing would be to use the available memory part, as well as conduct more extensive tests of it. It must also be tested on the targeted FPGA, and be verified to work with the transcoder.

Bibliography

[1] Xilinx Ltd., "UG761 v14.1 - AXI Reference Guide." http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v14_1/ug761_axi_reference_guide.pdf, 2012.

[2] ITU, "Joint Video Team." http://www.itu.int/en/ITU-T/studygroups/com16/video/Pages/jvt.aspx, [Online 2014].

[3] Avnet, "Optimal Memory Interface Design with Xilinx 7 Series - Xfest 2012." http://www.em.avnet.com/en-us/design/trainingandevents/Documents/X-FEST%202012%20PRESENTATIONS/xfest12_pdf_memory_v1_2_may15.pdf, 2012.

[4] M. Orlandić and K. Svarstad, "A Low-Complexity MPEG-2 to H.264/AVC Intra-Frame Transcoder Architecture for HD Video Sequences." (Subitted for publication), 2014.

[5] M. Orlandić and K. Svarstad, "A low complexity h.264/avc 4×4 intra prediction architecture with macroblock/block reordering," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, pp. 1–6, Dec 2013.

[6] I. Richardson, *The H.264 Advanced Video Compression Standard*. Wiley, 2011.

[7] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier Science, 2008.

[8] JEDEC Solid State Technology Association and others, "DDR3 SDRAM Standard," 2010.

[9] Elpida, "User's Manual - New Features of DDR3 SDRAM - Document No. E1503E10 (Ver.1.0)." http://www.micron.com/-/media/Documents/Products/Technical%20Note/DRAM/E1503E10.pdf, 2009.

[10] J. Hennessy, D. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*. Computer Architecture: A Quantitative Approach, Morgan Kaufmann/Elsevier, 2012.

[11] Altera Corporation, "Timing Diagrams for UniPHY IP." http://www.altera.com/literature/hb/external-memory/emi_uniphy_ref_timing_diagram.pdf, 2013.

[12] Xilinx Ltd., "Xilinx Kintex-7 FPGA KC705 Evaluation Kit." http://www.xilinx.com/products/boards-and-kits/EK-K7-KC705-G.htm, [Online, 2014].

[13] Xilinx Ltd., "KINTEX-7 FPGA KC705 Evaluation Kit - Product Brief." http://www.xilinx.com/publications/prod_mktg/KC705-Kit-Product-Brief.pdf.

[14] Xilinx Ltd., "DS180 v1.15 - 7 Series FPGAs Overview." http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, 2014.

[15] Xilinx Ltd., "KC705 Evaluation Board for the Kintex-7 FPGA - User Guide (UG819 v1.4)." http://www.xilinx.com/support/documentation/boards_and_kits/kc705/ug810_KC705_Eval_Bd.pdf, 2013.

[16] Micron Technology Inc., "Data Sheet - DDR3 SDRAM SODIMM - MT8JTF12864HZ – 1GB." http://www.micron.com/-/media/Documents/Products/Data%20Sheet/Modules/SODIMM/jtf8c128_256_512x64hz.pdf, 2014.

[17] Xilinx Ltd., "Xilinx CORE Generator System." http://www.xilinx.com/tools/coregen.htm, 2014.

[18] Xilinx Ltd., "Xilinx Design Tools - ISE Design Suite - Software Matrix." http://www.xilinx.com/publications/matrix/Software_matrix.pdf.

[19] Xilinx Ltd., "7 Series FPGAs Memory Interface Solutions v1.9 and v1.9a - User Guide." http://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v1_9/ug586_7Series_MIS.pdf, March 20, 2013.

[20] Xilinx Ltd., "Memory Interface Generator (MIG)." http://www.xilinx.com/products/intellectual-property/MIG.htm, 2014.

[21] D. Brooks, "Differential Signals, The Differential Difference," *Printed Circuit Design*, vol. 18, pp. 36–37, 2001.

[22] Xilinx Ltd., "7 Series FPGAs Memory Interface Solutions v2.0 - User Guide." http://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v2_0/ug586_7Series_MIS.pdf, December 18, 2013.

[23] Xilinx Ltd., “Opening a Static Simulation.” http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ism_c_opening_static_simulation.htm, 2014.

[24] Xilinx Ltd., “LogiCORE IP FIFO Generator v9.3 - Product Guide.” http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v9_3/pg057-fifo-generator.pdf, 2012.

[25] Xilinx Ltd., “7 Series FPGAs Memory Resources - User Guide v.1.10.” http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf, 2014.

[26] ALDEC, “Compile Xilinx Libraries for Aldec using compxlib.” http://www.aldec.com/en/support/resources/documentation/articles/1310.

[27] ALDEC, “Starting Active-HDL as the Default Simulator in Xilinx ISE.” http://www.aldec.com/en/support/resources/documentation/articles/1259.

Appendix A

Settings Selection for the MIG

The MIG can be modified to fit the designer's purpose. The selected MIG settings for this design are listed in Table A.1. More information about each property is available in [19].

Table A.1: Selected MIG Properties

Property	Selected Option
Number of Controllers	1
Pin Compatible FPGAs	None selected
Controller Type	DDR3 SDRAM
Clock Period	2500 ps/400 MHz
PHY to Controller Clock Ratio	4:1
Memory Part	MT41J256M8XX-107
Data Width	16
ECC	Unavailable, available for Data Width = 72
Data Mask	Disabled
Ordering	Normal
Input Clock Period	2500 ps (400 MHz) [10000 ps (100 MHz is also available)]
Read Burst Type and Length	Sequential (or Interleaved)
Output Driver Impedance Control	RZQ/7
Controller Chip Select Pin	Enable
RTT (nominal) - On Die Termination (ODT)	RZQ/4
Memory Address Mapping Selection	Bank Row Column
System Clock	Differential
Reference Clock	Differential
System Reset Polarity	Active Low

Debug Signals Control	OFF (for ChipScope debugging)
Internal Vref	Disabled
IO Power Reduction	ON
XADC Instantiation	Enabled
Internal Termination Impedance	50 Ohms
Digitally Controlled Impedance (DCI) Cascade	Disabled

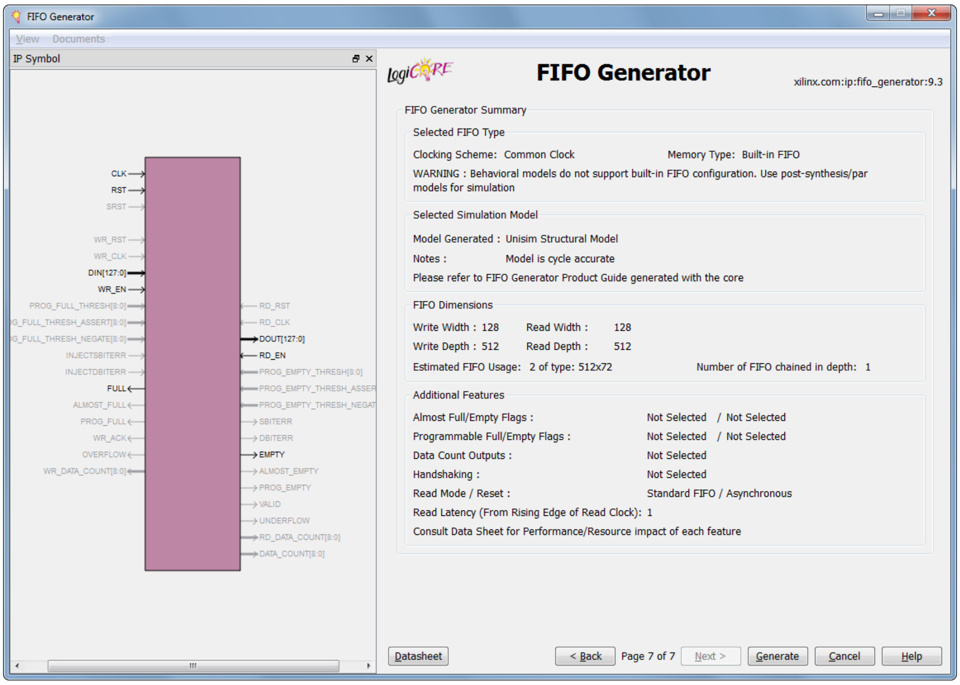
Appendix B

Settings Selection for the FIFO

The FIFO properties can be modified to achieve different behavior. The selected FIFO properties used in this design are listed in Table B.1. Figure B.1 shows the final summary displayed by the generator. More information about the FIFO IP can be found in [24].

Table B.1: Selected FIFO Properties

Property	**Selected Option**
Interface Type	Native
Read/Write Clock Domain	Common Clock
Memory Type	Built-in FIFO
Read Mode	Standard FIFO
Write Width	128
Write Depth	512
Enable ECC	Disabled
Use Embedded Registers in BRAM or FIFO	Disabled
Handshaking Options	Disabled
Programmable Flags	Disabled



FIFO Generator

xilinx.com:ip:fifo_generator:9.3

FIFO Generator Summary

Selected FIFO Type

Clocking Scheme: Common Clock   Memory Type: Built-in FIFO

WARNING : Behavioral models do not support built-in FIFO configuration. Use post-synthesis/par models for simulation

Selected Simulation Model

Model Generated : Unisim Structural Model

Notes : Model is cycle accurate

Please refer to FIFO Generator Product Guide generated with the core

FIFO Dimensions

Write Width :	128	Read Width :	128
Write Depth :	512	Read Depth :	512

Estimated FIFO Usage: 2 of type: 512x72   Number of FIFO chained in depth: 1

Additional Features

Almost Full/Empty Flags :	Not Selected / Not Selected
Programmable Full/Empty Flags :	Not Selected / Not Selected
Data Count Outputs :	Not Selected
Handshaking :	Not Selected
Read Mode / Reset :	Standard FIFO / Asynchronous
Read Latency (From Rising Edge of Read Clock):	1

Consult Data Sheet for Performance/Resource impact of each feature

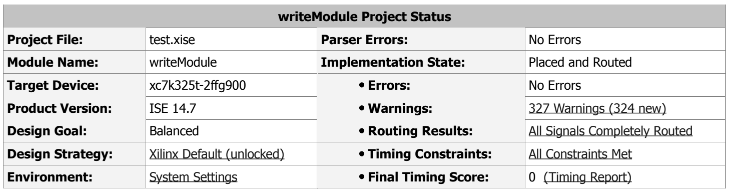
Figure B.1: FIFO Generator Summary

Appendix C

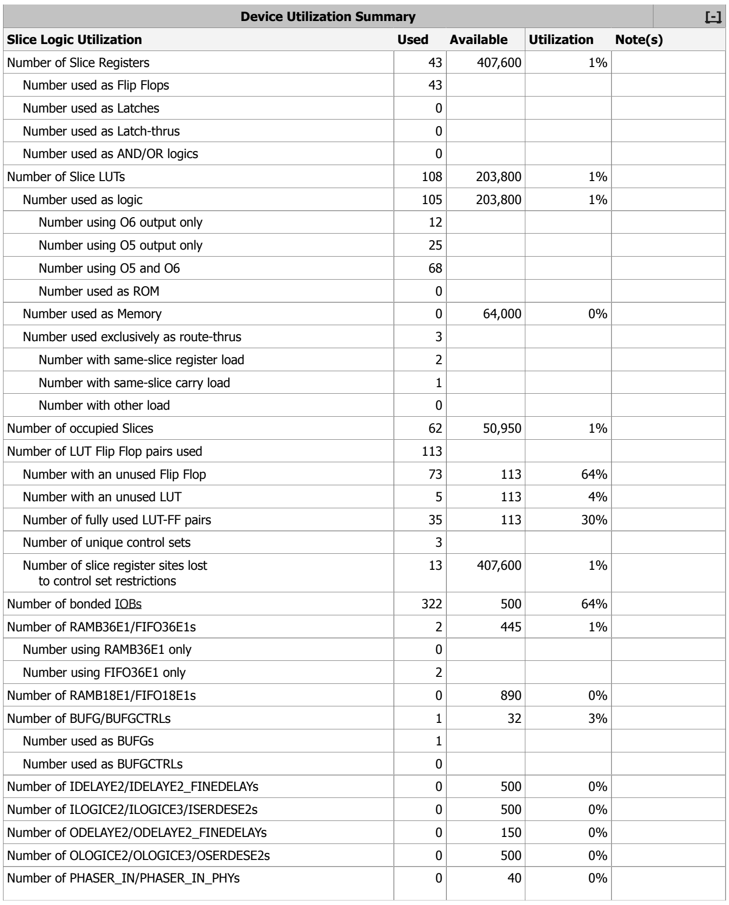
Xilinx Design Summaries

This part of the appendix includes the design summaries after running successful synthesis and implementations of the different modules.

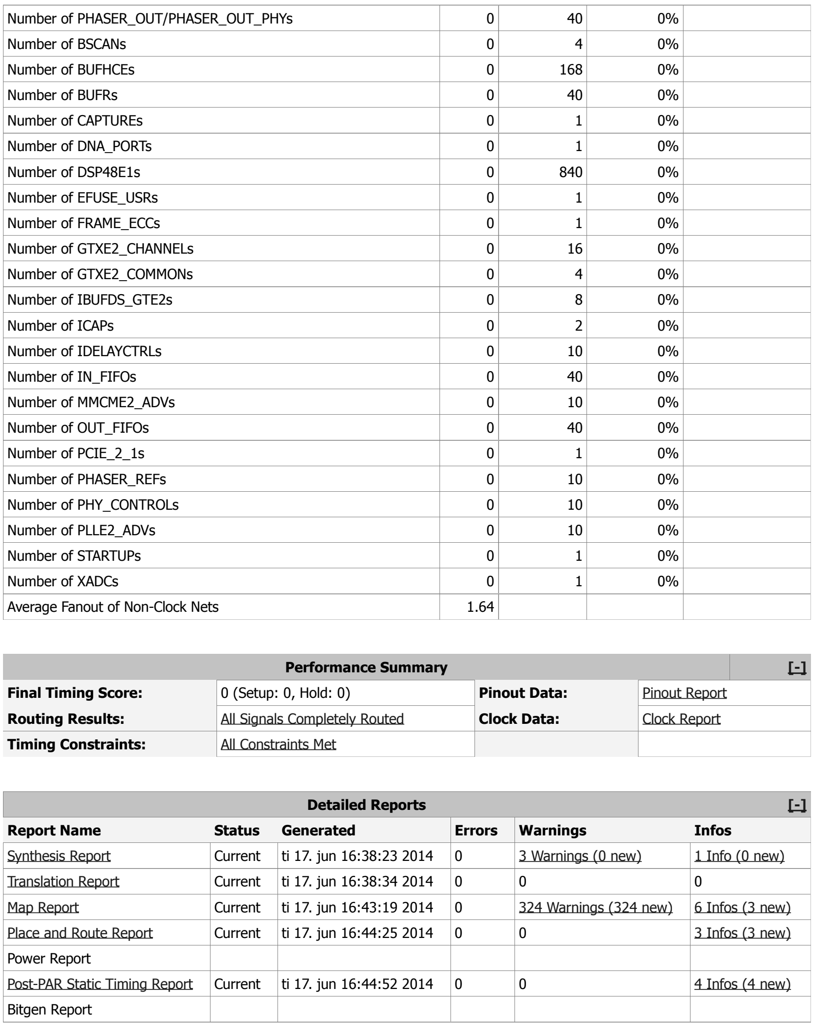
C.1 Summary for the Writing Module



writeModule Project Status			
Project File:	test.xise	**Parser Errors:**	No Errors
Module Name:	writeModule	**Implementation State:**	Placed and Routed
Target Device:	xc7k325t-2ffg900	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	327 Warnings (324 new)
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)



Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	43	407,600	1%	
Number used as Flip Flops	43			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	108	203,800	1%	
Number used as logic	105	203,800	1%	
Number using O6 output only	12			
Number using O5 output only	25			
Number using O5 and O6	68			
Number used as ROM	0			
Number used as Memory	0	64,000	0%	
Number used exclusively as route-thrus	3			
Number with same-slice register load	2			
Number with same-slice carry load	1			
Number with other load	0			
Number of occupied Slices	62	50,950	1%	
Number of LUT Flip Flop pairs used	113			
Number with an unused Flip Flop	73	113	64%	
Number with an unused LUT	5	113	4%	
Number of fully used LUT-FF pairs	35	113	30%	
Number of unique control sets	3			
Number of slice register sites lost to control set restrictions	13	407,600	1%	
Number of bonded IOBs	322	500	64%	
Number of RAMB36E1/FIFO36E1s	2	445	1%	
Number using RAMB36E1 only	0			
Number using FIFO36E1 only	2			
Number of RAMB18E1/FIFO18E1s	0	890	0%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number used as BUFGCTRLs	0			
Number of IDELAYE2/IDELAYE2_FINEDELAYs	0	500	0%	
Number of ILOGICE2/ILOGICE3/ISERDESE2s	0	500	0%	
Number of ODELAYE2/ODELAYE2_FINEDELAYs	0	150	0%	
Number of OLOGICE2/OLOGICE3/OSERDESE2s	0	500	0%	
Number of PHASER_IN/PHASER_IN_PHYs	0	40	0%	



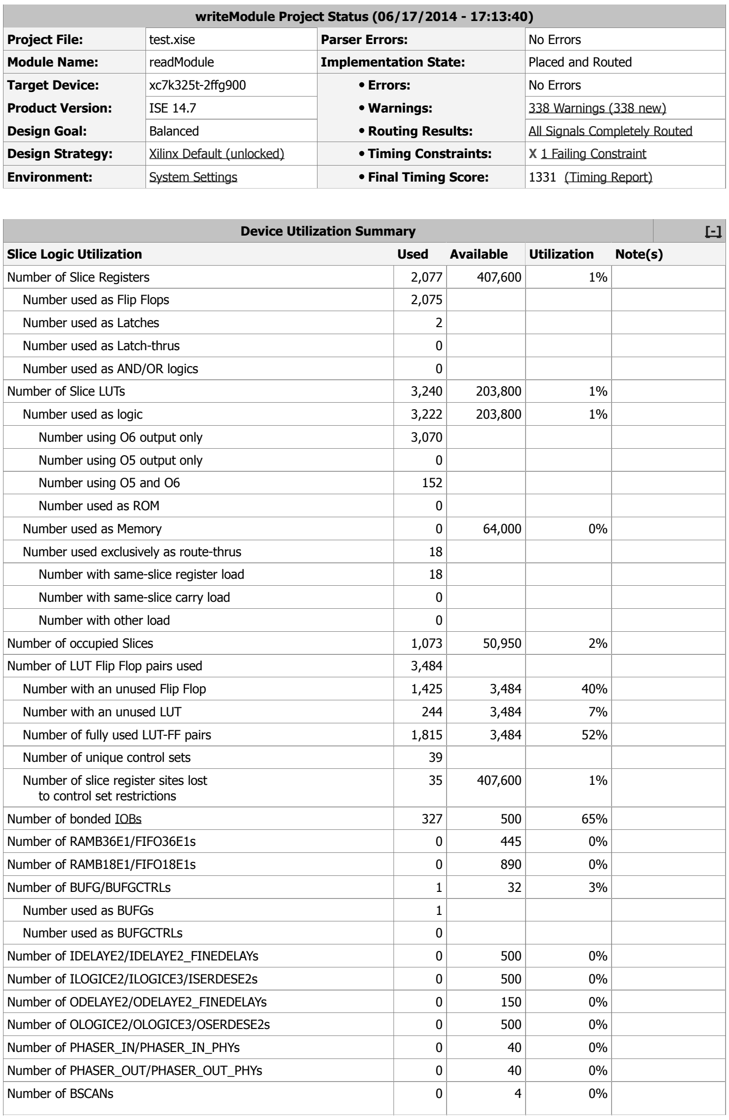
Number of PHASER_OUT/PHASER_OUT_PHYs	0	40	0%	
Number of BSCANs	0	4	0%	
Number of BUFHCEs	0	168	0%	
Number of BUFRs	0	40	0%	
Number of CAPTUREs	0	1	0%	
Number of DNA_PORTs	0	1	0%	
Number of DSP48E1s	0	840	0%	
Number of EFUSE_USRs	0	1	0%	
Number of FRAME_ECCs	0	1	0%	
Number of GTXE2_CHANNELs	0	16	0%	
Number of GTXE2_COMMONs	0	4	0%	
Number of IBUFDS_GTE2s	0	8	0%	
Number of ICAPs	0	2	0%	
Number of IDELAYCTRLs	0	10	0%	
Number of IN_FIFOs	0	40	0%	
Number of MMCME2_ADVs	0	10	0%	
Number of OUT_FIFOs	0	40	0%	
Number of PCIE_2_1s	0	1	0%	
Number of PHASER_REFs	0	10	0%	
Number of PHY_CONTROLs	0	10	0%	
Number of PLLE2_ADVs	0	10	0%	
Number of STARTUPs	0	1	0%	
Number of XADCs	0	1	0%	
Average Fanout of Non-Clock Nets	1.64			

Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports [-]

Report Name	Status	Generated	Errors	Warnings	Infos
Synthesis Report	Current	ti 17. jun 16:38:23 2014	0	3 Warnings (0 new)	1 Info (0 new)
Translation Report	Current	ti 17. jun 16:38:34 2014	0	0	0
Map Report	Current	ti 17. jun 16:43:19 2014	0	324 Warnings (324 new)	6 Infos (3 new)
Place and Route Report	Current	ti 17. jun 16:44:25 2014	0	0	3 Infos (3 new)
Power Report					
Post-PAR Static Timing Report	Current	ti 17. jun 16:44:52 2014	0	0	4 Infos (4 new)
Bitgen Report					

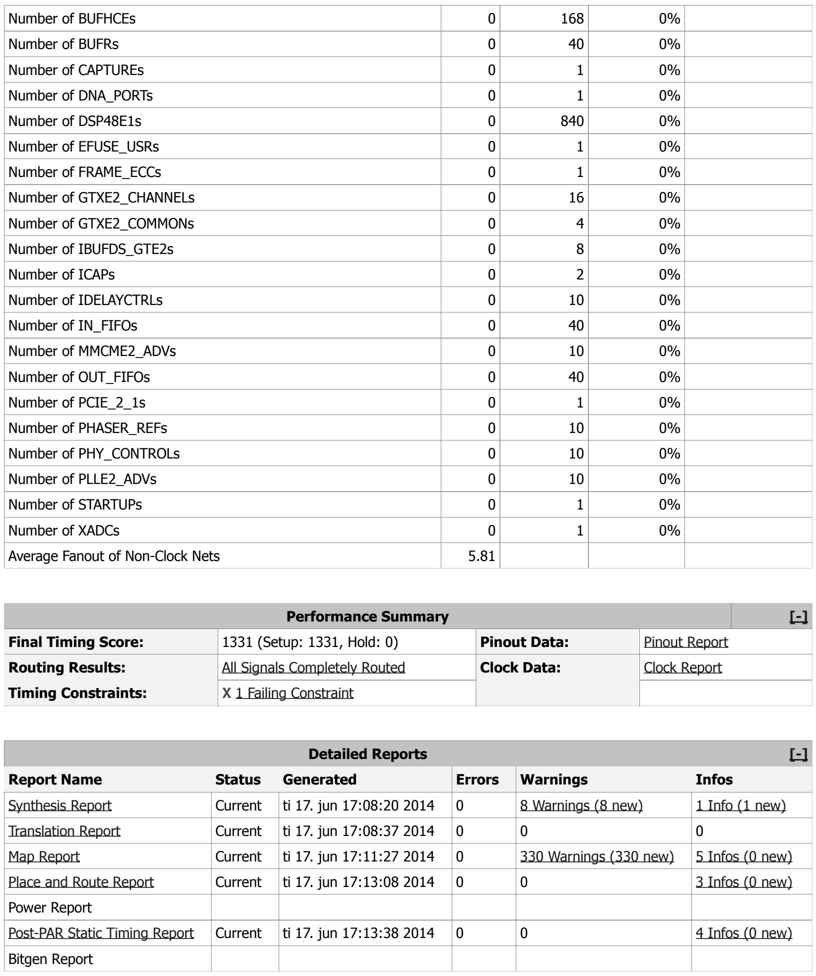
C.2 Summary for the Reading Module



writeModule Project Status (06/17/2014 - 17:13:40)			
Project File:	test.xise	**Parser Errors:**	No Errors
Module Name:	readModule	**Implementation State:**	Placed and Routed
Target Device:	xc7k325t-2ffg900	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	338 Warnings (338 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	X 1 Failing Constraint
Environment:	System Settings	• **Final Timing Score:**	1331 (Timing Report)

Device Utilization Summary [-]

Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	2,077	407,600	1%	
Number used as Flip Flops	2,075			
Number used as Latches	2			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	3,240	203,800	1%	
Number used as logic	3,222	203,800	1%	
Number using O6 output only	3,070			
Number using O5 output only	0			
Number using O5 and O6	152			
Number used as ROM	0			
Number used as Memory	0	64,000	0%	
Number used exclusively as route-thrus	18			
Number with same-slice register load	18			
Number with same-slice carry load	0			
Number with other load	0			
Number of occupied Slices	1,073	50,950	2%	
Number of LUT Flip Flop pairs used	3,484			
Number with an unused Flip Flop	1,425	3,484	40%	
Number with an unused LUT	244	3,484	7%	
Number of fully used LUT-FF pairs	1,815	3,484	52%	
Number of unique control sets	39			
Number of slice register sites lost to control set restrictions	35	407,600	1%	
Number of bonded IOBs	327	500	65%	
Number of RAMB36E1/FIFO36E1s	0	445	0%	
Number of RAMB18E1/FIFO18E1s	0	890	0%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number used as BUFGCTRLs	0			
Number of IDELAYE2/IDELAYE2_FINEDELAYs	0	500	0%	
Number of ILOGICE2/ILOGICE3/ISERDESE2s	0	500	0%	
Number of ODELAYE2/ODELAYE2_FINEDELAYs	0	150	0%	
Number of OLOGICE2/OLOGICE3/OSERDESE2s	0	500	0%	
Number of PHASER_IN/PHASER_IN_PHYs	0	40	0%	
Number of PHASER_OUT/PHASER_OUT_PHYs	0	40	0%	
Number of BSCANs	0	4	0%	

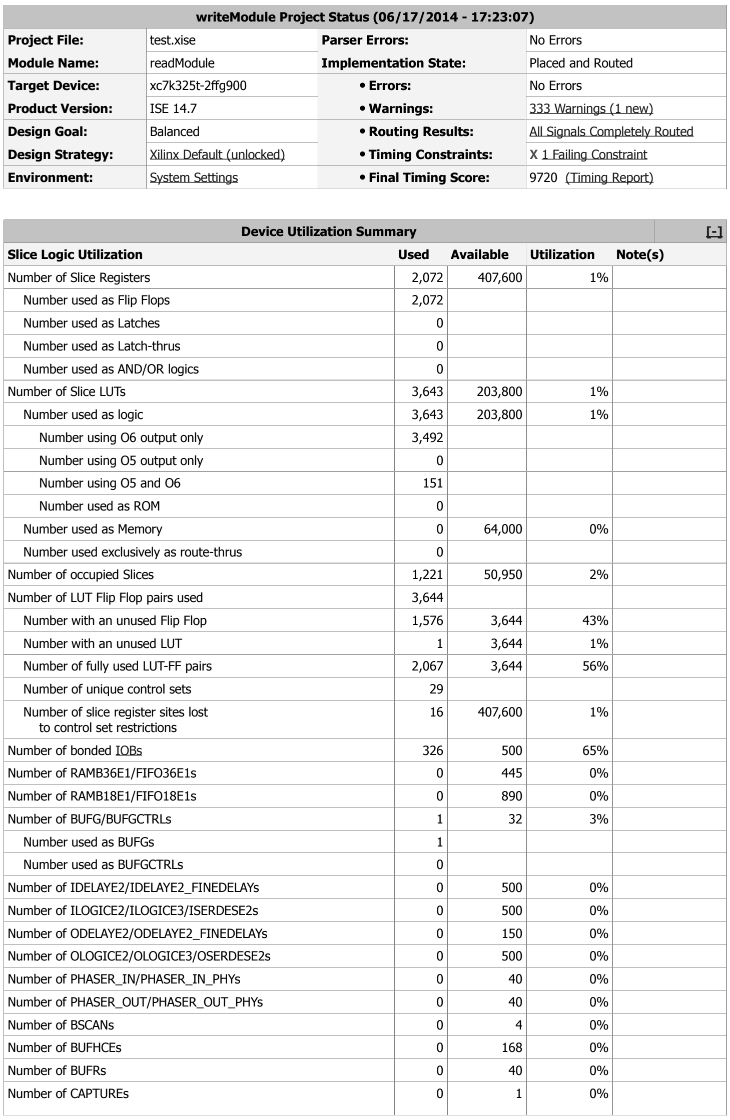


Number of BUFHCEs	0	168	0%	
Number of BUFRs	0	40	0%	
Number of CAPTUREs	0	1	0%	
Number of DNA_PORTs	0	1	0%	
Number of DSP48E1s	0	840	0%	
Number of EFUSE_USRs	0	1	0%	
Number of FRAME_ECCs	0	1	0%	
Number of GTXE2_CHANNELs	0	16	0%	
Number of GTXE2_COMMONs	0	4	0%	
Number of IBUFDS_GTE2s	0	8	0%	
Number of ICAPs	0	2	0%	
Number of IDELAYCTRLs	0	10	0%	
Number of IN_FIFOs	0	40	0%	
Number of MMCME2_ADVs	0	10	0%	
Number of OUT_FIFOs	0	40	0%	
Number of PCIE_2_1s	0	1	0%	
Number of PHASER_REFs	0	10	0%	
Number of PHY_CONTROLs	0	10	0%	
Number of PLLE2_ADVs	0	10	0%	
Number of STARTUPs	0	1	0%	
Number of XADCs	0	1	0%	
Average Fanout of Non-Clock Nets	5.81			

Performance Summary			[-]
Final Timing Score:	1331 (Setup: 1331, Hold: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	X 1 Failing Constraint		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	ti 17. jun 17:08:20 2014	0	8 Warnings (8 new)	1 Info (1 new)
Translation Report	Current	ti 17. jun 17:08:37 2014	0	0	0
Map Report	Current	ti 17. jun 17:11:27 2014	0	330 Warnings (330 new)	5 Infos (0 new)
Place and Route Report	Current	ti 17. jun 17:13:08 2014	0	0	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	ti 17. jun 17:13:38 2014	0	0	4 Infos (0 new)
Bitgen Report					

C.3 Summary for the Reading Module without Read Request



writeModule Project Status (06/17/2014 - 17:23:07)			
Project File:	test.xise	**Parser Errors:**	No Errors
Module Name:	readModule	**Implementation State:**	Placed and Routed
Target Device:	xc7k325t-2ffg900	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	333 Warnings (1 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	X 1 Failing Constraint
Environment:	System Settings	• **Final Timing Score:**	9720 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	2,072	407,600	1%	
Number used as Flip Flops	2,072			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	3,643	203,800	1%	
Number used as logic	3,643	203,800	1%	
Number using O6 output only	3,492			
Number using O5 output only	0			
Number using O5 and O6	151			
Number used as ROM	0			
Number used as Memory	0	64,000	0%	
Number used exclusively as route-thrus	0			
Number of occupied Slices	1,221	50,950	2%	
Number of LUT Flip Flop pairs used	3,644			
Number with an unused Flip Flop	1,576	3,644	43%	
Number with an unused LUT	1	3,644	1%	
Number of fully used LUT-FF pairs	2,067	3,644	56%	
Number of unique control sets	29			
Number of slice register sites lost to control set restrictions	16	407,600	1%	
Number of bonded IOBs	326	500	65%	
Number of RAMB36E1/FIFO36E1s	0	445	0%	
Number of RAMB18E1/FIFO18E1s	0	890	0%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number used as BUFGCTRLs	0			
Number of IDELAYE2/IDELAYE2_FINEDELAYs	0	500	0%	
Number of ILOGICE2/ILOGICE3/ISERDESE2s	0	500	0%	
Number of ODELAYE2/ODELAYE2_FINEDELAYs	0	150	0%	
Number of OLOGICE2/OLOGICE3/OSERDESE2s	0	500	0%	
Number of PHASER_IN/PHASER_IN_PHYs	0	40	0%	
Number of PHASER_OUT/PHASER_OUT_PHYs	0	40	0%	
Number of BSCANs	0	4	0%	
Number of BUFHCEs	0	168	0%	
Number of BUFRs	0	40	0%	
Number of CAPTUREs	0	1	0%	

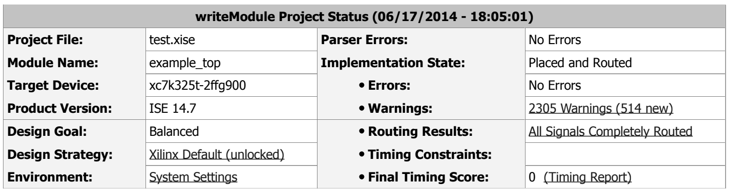
Number of DNA_PORTs	0	1	0%	
Number of DSP48E1s	0	840	0%	
Number of EFUSE_USRs	0	1	0%	
Number of FRAME_ECCs	0	1	0%	
Number of GTXE2_CHANNELs	0	16	0%	
Number of GTXE2_COMMONs	0	4	0%	
Number of IBUFDS_GTE2s	0	8	0%	
Number of ICAPs	0	2	0%	
Number of IDELAYCTRLs	0	10	0%	
Number of IN_FIFOs	0	40	0%	
Number of MMCME2_ADVs	0	10	0%	
Number of OUT_FIFOs	0	40	0%	
Number of PCIE_2_1s	0	1	0%	
Number of PHASER_REFs	0	10	0%	
Number of PHY_CONTROLs	0	10	0%	
Number of PLLE2_ADVs	0	10	0%	
Number of STARTUPs	0	1	0%	
Number of XADCs	0	1	0%	
Average Fanout of Non-Clock Nets	6.98			

Performance Summary			[-]
Final Timing Score:	9720 (Setup: 9720, Hold: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	X 1 Failing Constraint		

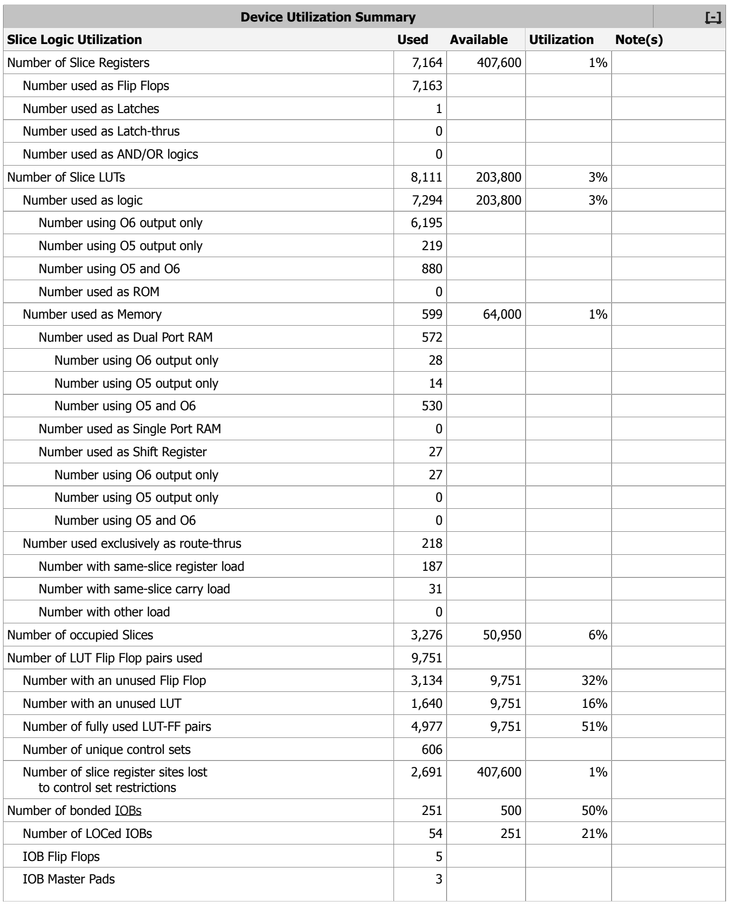
Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	ti 17. jun 17:17:54 2014	0	5 Warnings (1 new)	0
Translation Report	Current	ti 17. jun 17:18:11 2014	0	0	0
Map Report	Current	ti 17. jun 17:20:53 2014	0	328 Warnings (0 new)	6 Infos (1 new)
Place and Route Report	Current	ti 17. jun 17:22:32 2014	0	0	3 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	ti 17. jun 17:23:03 2014	0	0	4 Infos (0 new)
Bitgen Report					

C.4 Summary for the Example Top module

To be able to map the `example_top.vhd` file, which contains both the MIG and the proposed design, several signals had to be removed. This is because the design used too many IOBs. The temporarily removed signals were `mod_dataOut` and `mod_4x4_dout`.



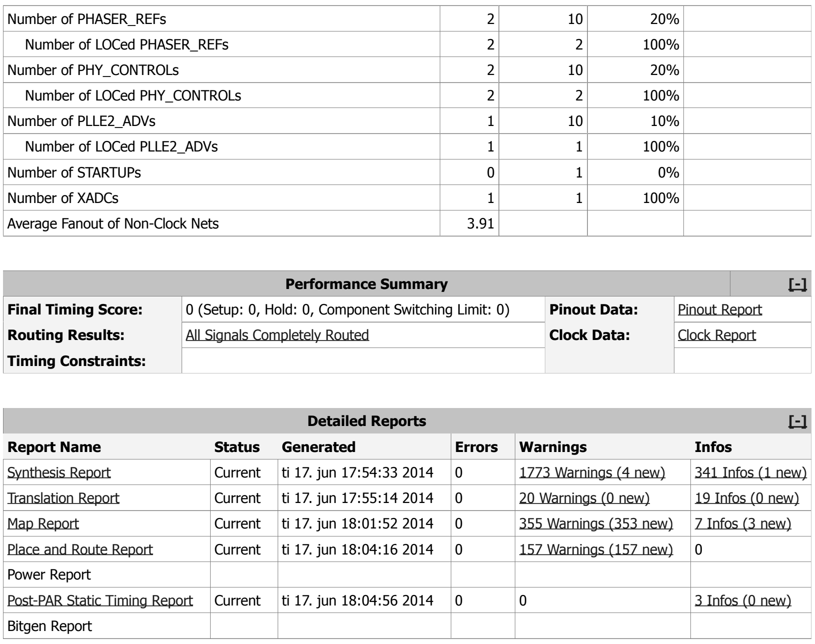
writeModule Project Status (06/17/2014 - 18:05:01)			
Project File:	test.xise	**Parser Errors:**	No Errors
Module Name:	example_top	**Implementation State:**	Placed and Routed
Target Device:	xc7k325t-2ffg900	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	2305 Warnings (514 new)
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)



Device Utilization Summary [-]

Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	7,164	407,600	1%	
Number used as Flip Flops	7,163			
Number used as Latches	1			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	8,111	203,800	3%	
Number used as logic	7,294	203,800	3%	
Number using O6 output only	6,195			
Number using O5 output only	219			
Number using O5 and O6	880			
Number used as ROM	0			
Number used as Memory	599	64,000	1%	
Number used as Dual Port RAM	572			
Number using O6 output only	28			
Number using O5 output only	14			
Number using O5 and O6	530			
Number used as Single Port RAM	0			
Number used as Shift Register	27			
Number using O6 output only	27			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used exclusively as route-thrus	218			
Number with same-slice register load	187			
Number with same-slice carry load	31			
Number with other load	0			
Number of occupied Slices	3,276	50,950	6%	
Number of LUT Flip Flop pairs used	9,751			
Number with an unused Flip Flop	3,134	9,751	32%	
Number with an unused LUT	1,640	9,751	16%	
Number of fully used LUT-FF pairs	4,977	9,751	51%	
Number of unique control sets	606			
Number of slice register sites lost to control set restrictions	2,691	407,600	1%	
Number of bonded IOBs	251	500	50%	
Number of LOCed IOBs	54	251	21%	
IOB Flip Flops	5			
IOB Master Pads	3			

IOB Slave Pads	3			
Number of RAMB36E1/FIFO36E1s	2	445	1%	
Number using RAMB36E1 only	0			
Number using FIFO36E1 only	2			
Number of RAMB18E1/FIFO18E1s	0	890	0%	
Number of BUFG/BUFGCTRLs	2	32	6%	
Number used as BUFGs	2			
Number used as BUFGCTRLs	0			
Number of IDELAYE2/IDELAYE2_FINEDELAYs	16	500	3%	
Number used as IDELAYE2s	16			
Number used as IDELAYE2_FINEDELAYs	0			
Number of ILOGICE2/ILOGICE3/ISERDESE2s	16	500	3%	
Number used as ILOGICE2s	0			
Number used as ILOGICE3s	0			
Number used as ISERDESE2s	16			
Number of ODELAYE2/ODELAYE2_FINEDELAYs	0	150	0%	
Number of OLOGICE2/OLOGICE3/OSERDESE2s	45	500	9%	
Number used as OLOGICE2s	3			
Number used as OLOGICE3s	0			
Number used as OSERDESE2s	42			
Number of PHASER_IN/PHASER_IN_PHYs	2	40	5%	
Number used as PHASER_INs	0			
Number used as PHASER_IN_PHYs	2			
Number of LOCed PHASER_IN_PHYs	2	2	100%	
Number of PHASER_OUT/PHASER_OUT_PHYs	5	40	12%	
Number used as PHASER_OUTs	0			
Number used as PHASER_OUT_PHYs	5			
Number of LOCed PHASER_OUT_PHYs	5	5	100%	
Number of BSCANs	0	4	0%	
Number of BUFHCEs	0	168	0%	
Number of BUFRs	0	40	0%	
Number of CAPTUREs	0	1	0%	
Number of DNA_PORTs	0	1	0%	
Number of DSP48E1s	0	840	0%	
Number of EFUSE_USRs	0	1	0%	
Number of FRAME_ECCs	0	1	0%	
Number of GTXE2_CHANNELs	0	16	0%	
Number of GTXE2_COMMONs	0	4	0%	
Number of IBUFDS_GTE2s	0	8	0%	
Number of ICAPs	0	2	0%	
Number of IDELAYCTRLs	2	10	20%	
Number of IN_FIFOs	2	40	5%	
Number of LOCed IN_FIFOs	2	2	100%	
Number of MMCME2_ADVs	1	10	10%	
Number of LOCed MMCME2_ADVs	1	1	100%	
Number of OUT_FIFOs	5	40	12%	
Number of LOCed OUT_FIFOs	5	5	100%	
Number of PCIE_2_1s	0	1	0%	

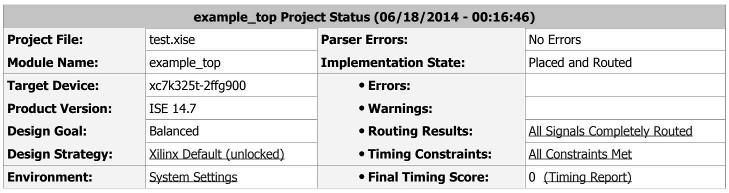


Number of PHASER_REFs	2	10	20%	
Number of LOCed PHASER_REFs	2	2	100%	
Number of PHY_CONTROLs	2	10	20%	
Number of LOCed PHY_CONTROLs	2	2	100%	
Number of PLLE2_ADVs	1	10	10%	
Number of LOCed PLLE2_ADVs	1	1	100%	
Number of STARTUPs	0	1	0%	
Number of XADCs	1	1	100%	
Average Fanout of Non-Clock Nets	3.91			

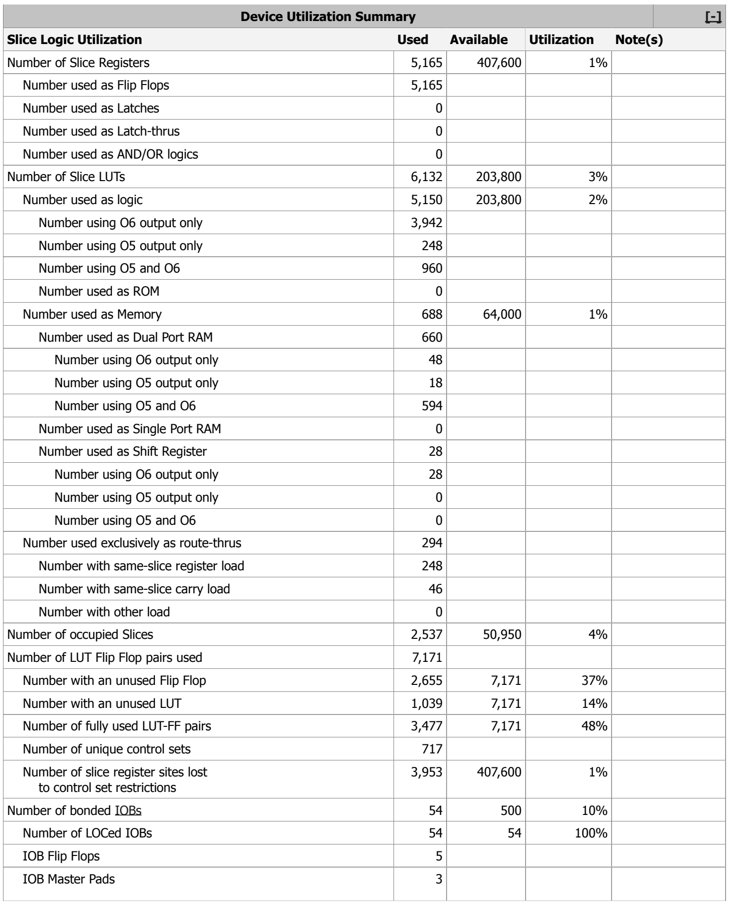
Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:			

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	ti 17. jun 17:54:33 2014	0	1773 Warnings (4 new)	341 Infos (1 new)
Translation Report	Current	ti 17. jun 17:55:14 2014	0	20 Warnings (0 new)	19 Infos (0 new)
Map Report	Current	ti 17. jun 18:01:52 2014	0	355 Warnings (353 new)	7 Infos (3 new)
Place and Route Report	Current	ti 17. jun 18:04:16 2014	0	157 Warnings (157 new)	0
Power Report					
Post-PAR Static Timing Report	Current	ti 17. jun 18:04:56 2014	0	0	3 Infos (0 new)
Bitgen Report					

C.5 Summary for the Original Example Top module

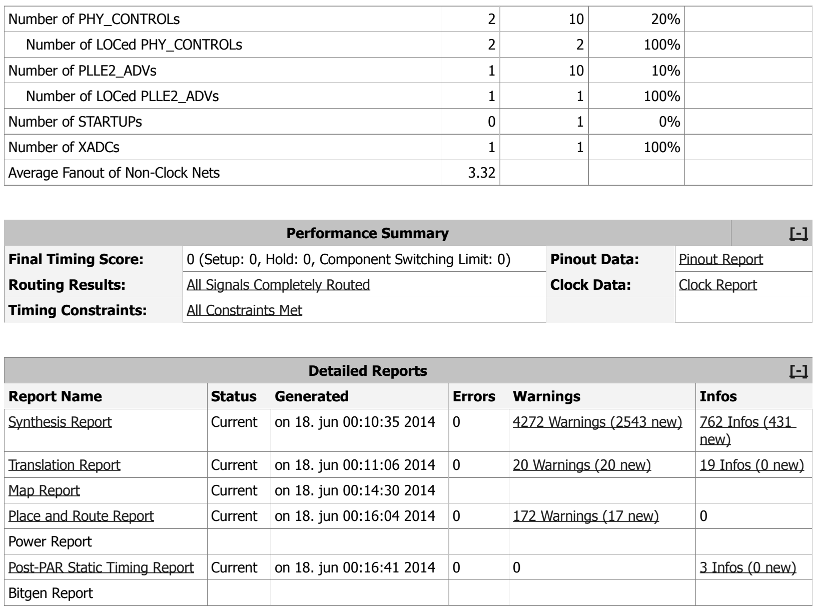


example_top Project Status (06/18/2014 - 00:16:46)			
Project File:	test.xise	**Parser Errors:**	No Errors
Module Name:	example_top	**Implementation State:**	Placed and Routed
Target Device:	xc7k325t-2ffg900	**• Errors:**	
Product Version:	ISE 14.7	**• Warnings:**	
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)



Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	5,165	407,600	1%	
Number used as Flip Flops	5,165			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	6,132	203,800	3%	
Number used as logic	5,150	203,800	2%	
Number using O6 output only	3,942			
Number using O5 output only	248			
Number using O5 and O6	960			
Number used as ROM	0			
Number used as Memory	688	64,000	1%	
Number used as Dual Port RAM	660			
Number using O6 output only	48			
Number using O5 output only	18			
Number using O5 and O6	594			
Number used as Single Port RAM	0			
Number used as Shift Register	28			
Number using O6 output only	28			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used exclusively as route-thrus	294			
Number with same-slice register load	248			
Number with same-slice carry load	46			
Number with other load	0			
Number of occupied Slices	2,537	50,950	4%	
Number of LUT Flip Flop pairs used	7,171			
Number with an unused Flip Flop	2,655	7,171	37%	
Number with an unused LUT	1,039	7,171	14%	
Number of fully used LUT-FF pairs	3,477	7,171	48%	
Number of unique control sets	717			
Number of slice register sites lost to control set restrictions	3,953	407,600	1%	
Number of bonded IOBs	54	500	10%	
Number of LOCed IOBs	54	54	100%	
IOB Flip Flops	5			
IOB Master Pads	3			

IOB Slave Pads	3			
Number of RAMB36E1/FIFO36E1s	0	445	0%	
Number of RAMB18E1/FIFO18E1s	0	890	0%	
Number of BUFG/BUFGCTRLs	2	32	6%	
Number used as BUFGs	2			
Number used as BUFGCTRLs	0			
Number of IDELAYE2/IDELAYE2_FINEDELAYs	16	500	3%	
Number used as IDELAYE2s	16			
Number used as IDELAYE2_FINEDELAYs	0			
Number of ILOGICE2/ILOGICE3/ISERDESE2s	16	500	3%	
Number used as ILOGICE2s	0			
Number used as ILOGICE3s	0			
Number used as ISERDESE2s	16			
Number of ODELAYE2/ODELAYE2_FINEDELAYs	0	150	0%	
Number of OLOGICE2/OLOGICE3/OSERDESE2s	45	500	9%	
Number used as OLOGICE2s	3			
Number used as OLOGICE3s	0			
Number used as OSERDESE2s	42			
Number of PHASER_IN/PHASER_IN_PHYs	2	40	5%	
Number used as PHASER_INs	0			
Number used as PHASER_IN_PHYs	2			
Number of LOCed PHASER_IN_PHYs	2	2	100%	
Number of PHASER_OUT/PHASER_OUT_PHYs	5	40	12%	
Number used as PHASER_OUTs	0			
Number used as PHASER_OUT_PHYs	5			
Number of LOCed PHASER_OUT_PHYs	5	5	100%	
Number of BSCANs	0	4	0%	
Number of BUFHCEs	0	168	0%	
Number of BUFRs	0	40	0%	
Number of CAPTUREs	0	1	0%	
Number of DNA_PORTs	0	1	0%	
Number of DSP48E1s	0	840	0%	
Number of EFUSE_USRs	0	1	0%	
Number of FRAME_ECCs	0	1	0%	
Number of GTXE2_CHANNELs	0	16	0%	
Number of GTXE2_COMMONs	0	4	0%	
Number of IBUFDS_GTE2s	0	8	0%	
Number of ICAPs	0	2	0%	
Number of IDELAYCTRLs	2	10	20%	
Number of IN_FIFOs	2	40	5%	
Number of LOCed IN_FIFOs	2	2	100%	
Number of MMCME2_ADVs	1	10	10%	
Number of LOCed MMCME2_ADVs	1	1	100%	
Number of OUT_FIFOs	5	40	12%	
Number of LOCed OUT_FIFOs	5	5	100%	
Number of PCIE_2_1s	0	1	0%	
Number of PHASER_REFs	2	10	20%	
Number of LOCed PHASER_REFs	2	2	100%	



Number of PHY_CONTROLs	2	10	20%	
Number of LOCed PHY_CONTROLs	2	2	100%	
Number of PLLE2_ADVs	1	10	10%	
Number of LOCed PLLE2_ADVs	1	1	100%	
Number of STARTUPs	0	1	0%	
Number of XADCs	1	1	100%	
Average Fanout of Non-Clock Nets	3.32			

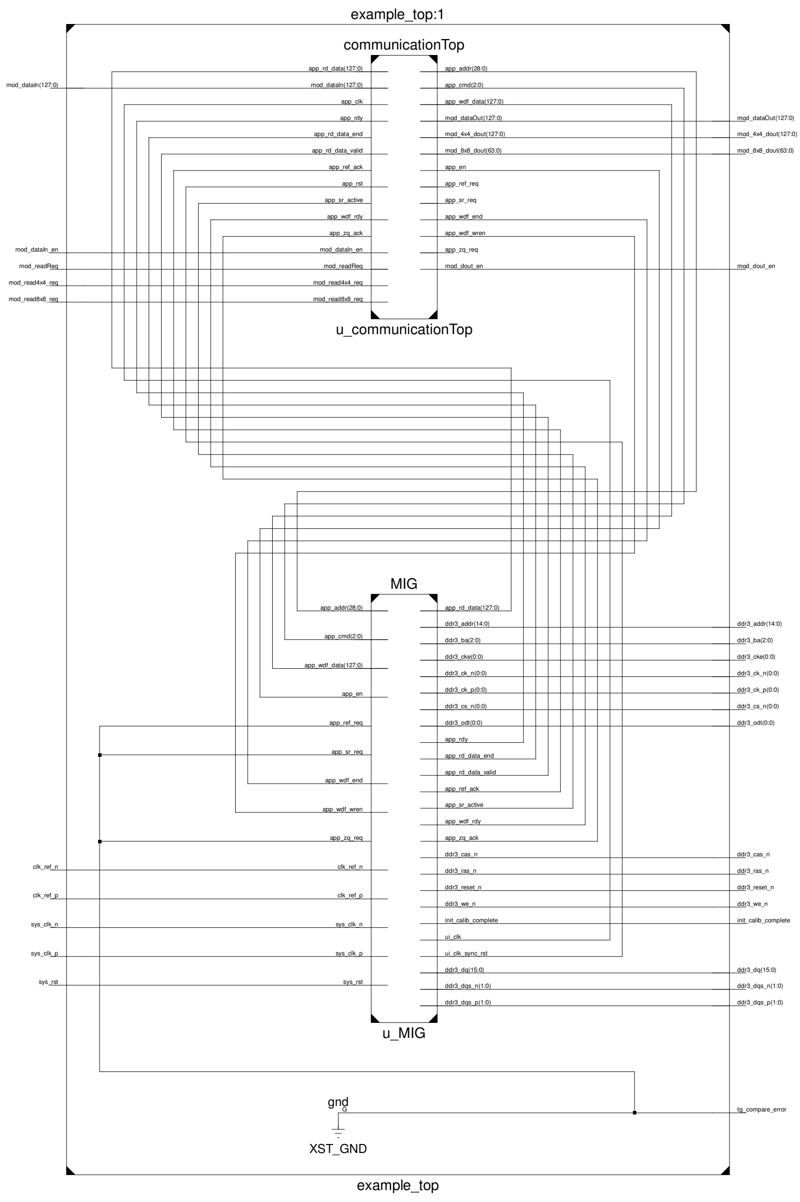
Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	on 18. jun 00:10:35 2014	0	4272 Warnings (2543 new)	762 Infos (431 new)
Translation Report	Current	on 18. jun 00:11:06 2014	0	20 Warnings (20 new)	19 Infos (0 new)
Map Report	Current	on 18. jun 00:14:30 2014			
Place and Route Report	Current	on 18. jun 00:16:04 2014	0	172 Warnings (17 new)	0
Power Report					
Post-PAR Static Timing Report	Current	on 18. jun 00:16:41 2014	0	0	3 Infos (0 new)
Bitgen Report					

Appendix D

Top Level Design Overview

Design overview of the top module (modified `example_top.vhd`) as shown in Xilinx RTL Schematic.


example_top:1
communicationTop
app_rd_data(127:0)
mod_dataIn(127:0)
app_clk
app_rdy
app_rd_data_end
app_rd_data_valid
app_ref_ack
app_rst
app_sr_active
app_wdf_rdy
app_zq_ack
mod_dataIn_en
mod_readReq
mod_read4x4_req
mod_read8x8_req
app_addr(28:0)
app_cmd(2:0)
app_wdf_data(127:0)
mod_dataOut(127:0)
mod_4x4_dout(127:0)
mod_8x8_dout(63:0)
app_en
app_ref_req
app_sr_req
app_wdf_end
app_wdf_wren
app_zq_req
mod_dout_en
u_communicationTop
MIG
app_addr(28:0)
app_cmd(2:0)
app_wdf_data(127:0)
app_en
app_ref_req
app_sr_req
app_wdf_end
app_wdf_wren
app_zq_req
clk_ref_n
clk_ref_p
sys_clk_n
sys_clk_p
sys_rst
app_rd_data(127:0)
ddr3_addr(14:0)
ddr3_ba(2:0)
ddr3_cke(0:0)
ddr3_ck_n(0:0)
ddr3_ck_p(0:0)
ddr3_cs_n(0:0)
ddr3_odt(0:0)
app_rdy
app_rd_data_end
app_rd_data_valid
app_ref_ack
app_sr_active
app_wdf_rdy
app_zq_ack
ddr3_cas_n
ddr3_ras_n
ddr3_reset_n
ddr3_we_n
init_calib_complete
ui_clk
ui_clk_sync_rst
ddr3_dq(15:0)
ddr3_dqs_n(1:0)
ddr3_dqs_p(1:0)
u_MIG
gnd
G
XST_GND
tg_compare_error
example_top