ALPHA DATA

ADM-XRC Gen 3
SDK 1.5.0
User Guide

Revision: V1.6

	Head Office	US Office
Address:	4 West Silvermills Lane, Edinburgh, EH3 5BD, UK	3507 Ringsby Court Suite 105, Denver, CO 80216
Telephone:	+44 131 558 2600	(303) 954 8768
Fax:	+44 131 558 2700	(866) 820 9956 toll free
email:	sales@alpha-data.com	sales@alpha-data.com
website:	http://www.alpha-data.com	http://www.alpha-data.com

Table Of Contents

List of Tables

List of Figures

Page Intentionally left blank

1 Introduction

This document describes the ADM-XRC Gen 3 Software Development Kit (SDK), which provides resources for developers working with the third generation of reconfigurable computing hardware from Alpha Data. The key features of the SDK are:

- Example applications that use the ADMXRC3 API.
- Example HDL FPGA designs that target third generation Alpha Data hardware such as the ADM-XRC-6TL. These designs are built from a number of HDL components that are also provided in this SDK.
- Utilities for working with third generation Alpha Data hardware.

1.1 Document conventions

In order to avoid unnecessary repetition of information pertaining to both Windows and Linux environments, the directory separator character for pathnames in this document is either the forward slash (**/**) or backward slash (****). They are used interchangeably, and it should be understood that to transform a Windows path to a Linux path, backward slashes are replaced by forward slashes. Inversely, to transform a Linux path to Windows path, forward slashes are replaced by backward slashes.

A pathname ending in a slash implies that the pathname refers to a directory as opposed to a file. For example, **apps/src/** is the path of a directory.

Unless stated otherwise or preceded by a slash or a Windows drive letter, pathnames and filenames in this document are relative to where this SDK has been installed on the development or host machine. For example:

- **C:/Program Files/Alpha Data/** is an absolute pathname that translates to the directory **C:\Program Files\Alpha Data** in a Windows environment.
- **apps/src/itest/itest.c** is a pathname relative to the root of the SDK that translates to the file **/opt/admxrcg3sdk-1.5.0/apps/src/itest/itest.c** in a Linux environment, assuming that the root of the SDK is **/opt/admxrcg3sdk-1.5.0/**.

It is assumed that the environment variable ADMXRC3_SDK is set to point to the root of the SDK. This environment variable is referenced in Linux shell commands as **$ADMXRC3_SDK** and as **%ADMXRC3_SDK%** in Windows shell commands. The installer for the Windows SDK normally sets this environment variable automatically so that it is present in the user's environment, but in Linux a user must manually add this variable to his or her environment.

1.2 Supported operating systems

This SDK supports the following operating systems:

- Windows NT-based operating systems beginning with Windows 2000. Both 32-bit and 64-bit editions are supported.
- Linux distributions running a 2.6.x or 3.x.y kernel.

Beginning with release 1.2.0, this SDK includes header files and example code for VxWorks. For VxWorks development, it is assumed that a host / development machine is available that runs one of the above operating systems.

1.3 Supported Alpha Data hardware

The example applications and HDL code in this SDK support the following models in Alpha Data's range of reconfigurable computing hardware:

- ADM-XRC-6TL
- ADM-XRC-6T1
- ADM-XRC-6T-DA1
- ADM-XRC-6TGE
- ADM-XRC-6T-ADV8
- ADPE-XRC-6T and ADPE-XRC-6T-L
- ADM-XRC-7K1 with 7K325T Initial Engineering Silicon (IES) or 7K410T Initial Engineering Silicon (devices marked with SCD code "ES9937")

  Support for Kintex-7 General Engineering Silicon and Production Silicon will be added in a future release of this SDK.
- ADM-XRC-7V1 with 7VX485T Initial Engineering Silicon (IES) (devices marked with SCD code "ES9937")

  Support for Virtex-7 General Engineering Silicon and Production Silicon will be added in a future release of this SDK.

1.4 Installation

1.4.1 Installation in Windows

The default installation location depends upon whether the operating system is a 32-bit or 64-bit edition of Windows:

- **%ProgramFiles%/ADMXRCG3SDK-1.5.0/** in 32-bit editions of Windows.
- **%ProgramFiles(x86)%/ADMXRCG3SDK-1.5.0/** in 64-bit editions of Windows.

During installation, the installer automatically creates an environment variable **ADMXRC3_SDK** that points to where the SDK is installed. Certain example applications use this environment variable to locate FPGA bitstream (.BIT) files. A user need not manually set this variable, but if using several versions of the SDK, it can be set manually according to which version of the SDK is in use.

1.4.2 Installation in Linux

This SDK is supplied as a tarball (.tar.gz extension) that should normally be extracted to the **/opt/** directory, which places the root of the SDK at**/opt/admxrcg3sdk-1.5.0/**.

After installation, an environment variable **ADMXRC3_SDK** must be defined that points to where the SDK is installed. Certain example applications use this environment variable to locate FPGA bitstream (.BIT) files. A convenient way to permanently define this variable for a given user is to add the following to the user's **.bash_profile**:

```
ADMXRC3_SDK=/opt/admxrcg3sdk-1.5.0
export ADMXRC3_SDK
```

1.4.3 Installation in VxWorks

Since VxWorks normally requires a Windows, Linux or UNIX host, this SDK must be installed on a Windows or Linux host as described in Section 1.4.1 or Section 1.4.2.

1.5 Structure of this SDK

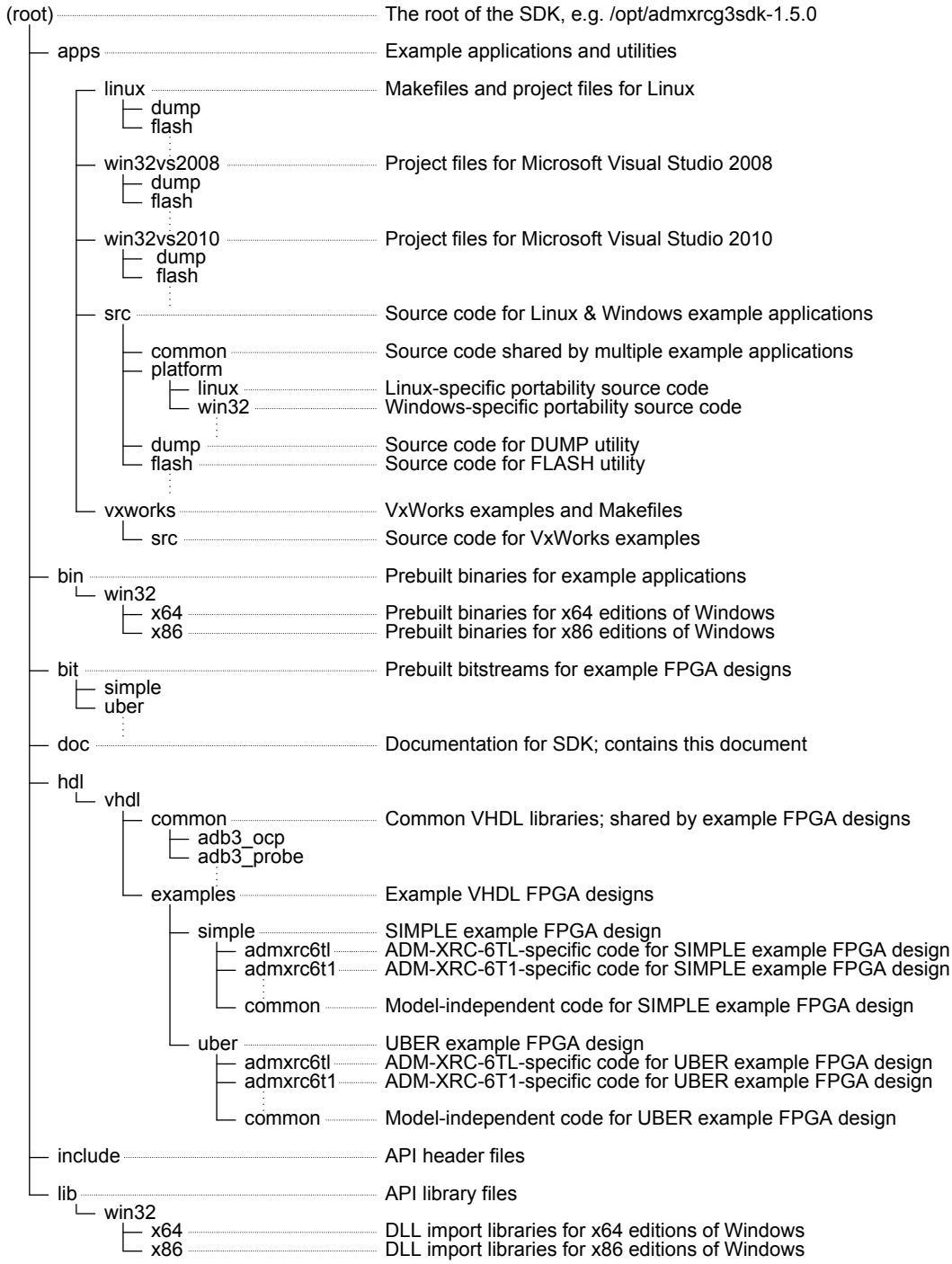

Figure 1 : Structure of the ADM-XRC Gen 3 SDK

2 Getting started

2.1 Getting started in Windows 2000 / XP / Server 2003

> **Note**
> This section also applies to Windows Vista and later when User Account Control (UAC) is disabled.

This section describes how to run a basic confidence test on Alpha Data hardware, in Windows 2000 / XP / Server 2003. This confidence test assumes the following:

1. All features of the SDK were installed, as described in Section 1.4.
2. Any model from Alpha Data's reconfigurable computing range that is supported by this SDK is installed in the machine. For a list of hardware supported, refer to Section 1.3.
3. The ADB3 driver is installed. The ADB3 driver for Windows is available from Alpha Data's public FTP site: Web ftp://ftp.alpha-data.com/pub/admxrcg3/windows ftp://ftp.alpha-data.com/pub/admxrcg3/windows.
4. You are logged on as a user that is a member of the Administrators group.

First, start an SDK command prompt by clicking on the 'SDK Command Prompt' shortcut from the 'ADM-XRC Gen 3 SDK' group on the Windows start menu. This command prompt automatically starts with the working directory set to the **bin/win32/x86/** folder of the SDK and also ensures that the **ADMXRC3_SDK** environment variable is set correctly.

Next, run the **info** utility. The output looks like this:

```
API information
API library version           1.1.2
Driver version                1.1.2

Card information
Model                         ADM-XRC-6TL
Serial number                 106(0x6A)
Number of programmable clocks 1
Number of DMA channels        2
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             10
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF

Target FPGA information
FPGA 0                        xc6vlx365tff1759-2C stepping ES

Memory bank information
Bank 0                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
```

```
Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF5800000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0xFB400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base     0xFB2FF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis  Bus base     0xFB2FE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

Now run the **simple** example application. It prompts the user to enter hexadecimal values (up to 32 bits), and displays these value nibble-reversed. The nibble-reversal is performed by the target FPGA. Pressing CTRL-Z exits this example. The output looks like this:

```
=========================
Enter values for I/O
(CTRL-D / CTRL-Z to exit)
=========================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
deadbeef
OUT = 0xdeadbeef, IN = 0xfeebdaed
cafeface
OUT = 0xcafeface, IN = 0xecafefac
```

If everything works as described above, then the hardware, driver and SDK are all correctly installed and substantially working. Possible next steps are:

- Experiment with modifying and rebuilding the **simple** example application in order to become familiar with the basics of the ADMXRC3 API.
- Experiment with modifying and rebuilding the **simple** example FPGA design in order to become familiar with creating FPGA designs for Alpha Data hardware.

2.2 Getting started in Windows Vista and later

> **Note**
> If User Account Control is disabled, please refer instead to the instructions in Section 2.1.

This section describes how to run a basic confidence test on Alpha Data hardware, in versions of Windows that have User Account Control (UAC) such as Windows Vista and later. This confidence test assumes the following:

1. All features of the SDK were installed, as described in Section 1.4.
2. Any model from Alpha Data's reconfigurable computing range that is supported by this SDK is installed in the machine. For a list of hardware supported, refer to section Section 1.3.
3. The ADB3 driver is installed. The ADB3 driver for Windows is available from Alpha Data's public FTP site: Web ftp://ftp.alpha-data.com/pub/admxrcg3/windows ftp://ftp.alpha-data.com/pub/admxrcg3/windows.
4. You are logged on as a user that is a member of the Administrators group.

Because of User Account Control (UAC), it is not possible to make use of the 'SDK Command Prompt' shortcut that is installed along with the SDK. Instead, start a command prompt by right-clicking on the 'Command Prompt' shortcut in the 'Accessories' program group and selecting **'Run as administrator'**. This will typically incur a UAC

confirmation prompt. Then, enter the following command (do not omit the double quotes):

```
"%ADMXRC3_SDK%\env.bat"
```

This executes the **env.bat** batch file, which sets up the environment and changes to the folder containing the prebuilt example application binaries. In order for this to work correctly, the **ADMXRC3_SDK** system environment variable must be correctly defined. The installer normally sets this variable, but if not, it must be set using the Windows Control Panel as a **system** environment variable to point to where the SDK is installed.

Next, run the **info** utility. The output looks like this:

```
API information
API library version           1.1.2
Driver version                1.1.2

Card information
Model                         ADM-XRC-6TL
Serial number                 106(0x6A)
Number of programmable clocks 1
Number of DMA channels        2
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             10
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF

Target FPGA information
FPGA 0                        xc6vlx365tff1759-2C stepping ES

Memory bank information
Bank 0                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1

Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF5800000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0xFB400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base     0xFB2FF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis  Bus base     0xFB2FE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

Now run the **simple** example application. It prompts the user to enter hexadecimal values (up to 32 bits), and displays these value nibble-reversed. The nibble-reversal is performed by the target FPGA. Pressing CTRL-Z exits this example. The output looks like this:

```
=========================
Enter values for I/O
(CTRL-D / CTRL-Z to exit)
=========================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
deadbeef
OUT = 0xdeadbeef, IN = 0xfeebdaed
cafeface
OUT = 0xcafeface, IN = 0xecafefac
```

If everything works as described above, then the hardware, driver and SDK are all correctly installed and substantially working. Possible next steps are:

- Make a copy of the SDK in your own filespace, and use the copy to experiment with modifying and rebuilding the **simple** example application in order to become familiar with the basics of the ADMXRC3 API.
- Make a copy of the SDK in your own filespace, and use the copy to experiment with modifying and rebuilding the **simple** example FPGA design in order to become familiar with creating FPGA designs for Alpha Data hardware.

2.3 Getting started in Linux

This section describes how to run a basic confidence test on Alpha Data hardware, in Linux. This confidence test assumes the following:

1 This SDK is installed as described in Section 1.4, and the **ADMXRC3_SDK** environment variable is set to point to where the SDK has been installed.

2 Any model from Alpha Data's reconfigurable computing range that is supported by this SDK is installed in the machine. For a list of hardware supported, refer to Section 1.3.

3 The ADB3 driver is installed. The ADB3 driver for Linux is available from Alpha Data's public FTP site: Web ftp://ftp.alpha-data.com/pub/admxrcg3/linux ftp://ftp.alpha-data.com/pub/admxrcg3/linux.

Note
In the following text, it is assumed that it is possible to log in as 'root'. If a Linux distribution is used where users are expected to use 'sudo' rather than logging in as root, then in all of the following instructions, commands should be prefixed with 'sudo' so that the effect is the same as 'su' to 'root'.

Log in as root (if possible), change directory to where the SDK has been installed, and then run the **configure** script:

```
$ cd $ADMXRC3_SDK/apps/linux
$ ./configure
```

This detects certain features of the operating system environment so that the example applications can be built. Now build the example applications:

```
$ make clean all
```

Having built the example applications, run the **info** utility:

```
$ info/info
```

The output looks like this:

```
API information
```

```
API library version           1.1.2
Driver version                1.1.2

Card information
Model                         ADM-XRC-6TL
Serial number                 106(0x6A)
Number of programmable clocks 1
Number of DMA channels        2
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             10
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF

Target FPGA information
FPGA 0                        xc6vlx365tff1759-2C stepping ES

Memory bank information
Bank 0                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1

Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF5800000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0xFB400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base     0xFB2FF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis  Bus base     0xFB2FE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

Now run the **simple** example application:

```
$ simple/simple
```

It prompts the user to enter hexadecimal values (up to 32 bits), and displays these value nibble-reversed. The nibble-reversal is performed by the target FPGA. Pressing CTRL-D exits this example. The output looks like this:

```
=========================
Enter values for I/O
(CTRL-D / CTRL-Z to exit)
=========================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
deadbeef
OUT = 0xdeadbeef, IN = 0xfeebdaed
cafeface
OUT = 0xcafeface, IN = 0xecafefac
```

If everything works as described above, then the hardware, driver and SDK are all correctly installed and substantially working. Possible next steps are:

- Experiment with modifying and rebuilding the **simple** example application in order to become familiar with the basics of the ADMXRC3 API.
- Experiment with modifying and rebuilding the **simple** example FPGA design in order to become familiar with creating FPGA designs for Alpha Data hardware.

2.4 Getting started in VxWorks

> **Note**
> Before attempting to follow the instructions in this section, we recommend first building the **ADB3 Driver for VxWorks** and following the instructions for getting started, verifying that the driver appears to start correctly on the target system. For details, please refer to the release notes for the **ADB3 Driver for VxWorks**.

The example VxWorks applications in this SDK are supplied only in source code form because it is impractical to provide binaries for the near-infinite number of possible VxWorks configurations. As a result, a downloadable module binary for the examples must be built using one of the supported Wind River VxWorks toolchains (DIAB or GNU).

A second consideration is how the target system will access the downloadable module that you build. In the following discussion, the term *staging area* refers to the some location on the development machine's filesystem (s) that the target system can access via FTP, NFS, or whatever other method the target system uses for host file access. There are two main approaches:

- Copy the entire SDK into the staging area, and build the examples there into a downloadable module. The target system can then access the downloadable module from the staging area. This approach is convenient as no manual copying of files is required after building, but may be problematic on some host operating systems if file permissions in the staging area do not permit the execution of build commands in the staging area.
- Copy the SDK to an arbitrary location (e.g. your personal folder on the development machine) and build the examples there into a downloadable module. The downloadble module must then be copied to the staging area, and the target system can then access it. This approach is compatible with restrictive file permissions in the staging area, but the downside is the inconvenience of manually copying of the downloadable module into the staging area each time it is built.

Whichever approach is chosen, the next step is build the example applications as described in Section 4.1 or Section 4.2. This yields a file **admxrc3Apps.out** containing all of the examples that can be downloaded to the target system. The location of this file is as shown in Table 2.

To download the file onto the target system, use the target system's console or a VxWorks host shell on the target system in order to enter the following command:

```
-> ld <host:/path/to/admxrc3Apps.out
```

where *host:/path/to/* is replaced by the host and folder that contains **admxrc3Apps.out**.

Now the **INFO** utility can be run as a basic confidence test that the applications were built correctly. Enter the following command:

```
-> admxrc3Info
```

The output looks like this:

```
API information
```

```
API library version           1.1.2
Driver version                1.1.2

Card information
Model                         ADM-XRC-6TL
Serial number                 106(0x6A)
Number of programmable clocks 1
Number of DMA channels        2
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             10
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF

Target FPGA information
FPGA 0                        xc6vlx365tff1759-2C step ES

Memory bank information
Bank 0                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1

Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF1400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0xF0400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base     0xF0800000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis  Bus base     0xF0801000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

Now run the **simple** example:

```
-> admxrc3Simple
```

It prompts the user to enter hexadecimal values (up to 32 bits), and displays these value nibble-reversed. The nibble-reversal is performed by the target FPGA. Pressing CTRL-D exits this example. The output looks like this:

```
=========================
Enter values for I/O
(CTRL-D / CTRL-Z to exit)
=========================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
deadbeef
OUT = 0xdeadbeef, IN = 0xfeebdaed
cafeface
OUT = 0xcafeface, IN = 0xecafefac
```

If everything works as described above, then the hardware, driver and SDK are all correctly installed and substantially working. Possible next steps are:

- Experiment with modifying and rebuilding the **simple** example application in order to become familiar with the basics of the ADMXRC3 API.
- Experiment with modifying and rebuilding the **simple** example FPGA design in order to become familiar with creating FPGA designs for Alpha Data hardware.

3 Example applications for Windows and Linux

The example applications and utilities are described in the following subsections.

BITSTRIP	Utility for removing the header off a .bit file, leaving only the SelectMap data
DUMP	Utility for reading and writing memory access windows
FLASH	Utility for programming FPGA bitstream (.BIT) files in user-programmable Flash memory
INFO	Utility for displaying information about a reconfigurable computing device
ITEST	Example demonstrating how to consume target FPGA interrupt notifications in an application
LOADER	Utility for configuring a target FPGA with a bitstream file
MEMTESTH	Example demonstrating host-driven memory test
MONITOR	Utility that displays sensor readings
SIMPLE	Example demonstrating how to read and write registers in a target FPGA
SIMPLEDMA	Example that tests the SimpleDMA Example FPGA Design
SYSMON	Utility that combines the functionality of the INFO and MONITOR utilities in a graphical user interface
VPD	Utility that allows the Vital Product Data of a reconfigurable computing device to be read or written

Table 1 : Example applications for Windows and Linux

Source code for the example Windows and Linux applications is provided in the **apps/src** directory, relative to the root of the SDK.

3.1 Building the example applications with Visual Studio 2008

A Microsoft Visual Studio 2008 solution **apps/win32vs2008/apps.sln** is provided, containing all of the Windows examples. To build all of the examples, use the "Batch Build" command in the Microsoft Visual Studio 2008 IDE.

3.2 Building the example applications with Visual Studio 2010

A Microsoft Visual Studio 2010 solution **apps/win32vs2010/apps.sln** is provided, containing all of the Windows examples. To build all of the examples, use the "Batch Build" command in the Microsoft Visual Studio 2010 IDE.

3.3 Building the example applications in Linux

To build all of the example applications, excluding the SYSMON utility, at once, enter the following shell commands in a BASH shell:

```
$ cd $ADMXRC3_SDK/apps/linux
```

```
$ ./configure
$ make clean all
```

When compiling on 64-bit bi-architecture machine such as x86_64, two executables are built for each example application: a 64-bit native version and a 32-bit version. For example, the native version of INFO is named **info**, and the 32-bit version is **info32**. For machines that are not bi-architecture, only the native version is built. The **configure** script determines whether or not to build bi-architecture versions of the example applications.

The SYSMON utility must be built separately, because it depends upon certain packages being present in the system. For further details, refer to Section 3.14.2.

3.4 BITSTRIP utility

Command line

```
bitstrip [option ...] <input .bit filename>
```

where the following options are accepted:

-o <output filename>	Specifies the name of the file into which the SelectMap data is written.

Summary

Reads an FPGA bitstream (.bit) file, displays certain information from the header, and optionally writes the SelectMap data to another file.

Description

To simply display information from a .bit file's header, use

```
bitstrip <input .bit filename>
```

To display information from a .bit file's header and write the SelectMap data to another file, use

```
bitstrip -o <output filename> <input .bit filename>
```

The data written to <*output filename*> is suitable for sending to a target FPGA using ADMXRC3_ConfigureFromBuffer.

3.5 DUMP utility

Command line

```
dump [option ...] rb window offset [n]
dump [option ...] rw window offset [n]
dump [option ...] rd window offset [n]
dump [option ...] rq window offset [n]
dump [option ...] wb window offset [n] [data ...]
dump [option ...] ww window offset [n] [data ...]
dump [option ...] wd window offset [n] [data ...]
dump [option ...] wq window offset [n] [data ...]
```

where

window	is the memory window to read or write.
offset	is the offset into the window at which to begin reading or writing.
n	is the number of bytes to read or write.
data	is an optional data item, valid for write commands.

and the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-be	Causes the data to be read or written to be treated as little-endian (default).
+be	Causes the data to be read or written to be treated as big-endian.
-hex	Causes write values to be interpreted as decimal unless prefixed by '0x' (default).
+hex	Causes write values to be interpreted as hexadecimal always.

Summary

Displays data read from a memory access window, or writes data to a memory access window.

Description

The **DUMP** utility operates in of two modes:

- Reading data from a memory access window and displaying it; for this mode, use the **rb**, **rw**, **rd** or **rq** commands.
- Writing data to a memory access window; for this mode, use the **wb**, **ww**, **wd** or **wq** commands.

In either mode, the option **+be** may be passed, before the command. This causes the **DUMP** utility to adopt big-endian byte ordering convention as opposed to little-endian (the default).

Read mode

The read command implies the radix for displaying data:

- **rb**

  Byte (8-bit) reads; data is displayed as bytes.

- **rw**

Word (16-bit) reads; data is displayed as words.

- **rd**

  Doubleword (32-bit) reads; data is displayed as doublewords.

- **rq**

  Quadword (64-bit) reads; data is displayed as quadwords.

After the read command, a window index and an offset must be supplied, in that order. This specifies the memory access window to be read, and where in that window to begin reading data. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the read command. If present, the length parameter specifies how many bytes to read and display. The length should be an integer multiple of the width; if not, the length is rounded down.

For example, the command

```
dump rw 0 0x80000 0x60
```

produces output of the form

```
Window 0 offset 0x80000 mapped @ 0x00150000
Dump of memory at 0x00150000 + 96(0x60) bytes:
              00   02   04   06   08   0a   0c   0e
0x00150000: 000e 000f 000c b456 c567 d678 5a5a eeee ......V.g.x.ZZ..
0x00150010: eeee eeee ee22 eeee eeee eeee eeee eeee ....".........."
0x00150020: eeee eeee eeee eeee eeee eeee eeee eeee ................
0x00150030: afa7 f596 445d 8232 163f 8414 1d1e 171b ....]D2.?.......
0x00150040: c294 fa5c cd61 d464 d39d 1eed 69f8 f13d ..\.a.d.....i=.
0x00150050: 5858 f489 20ff b77b ef92 a43a 6a27 e620 XX... {...:.'j .
```

Write mode

The write command implies the radix (that is, word size) to be used when performing writes:

- **wb**

  Data is written as bytes (8-bit).

- **ww**

  Data is written as words (16-bit).

- **wd**

  Data is written as doublewords (32-bit).

- **wq**

  Data is written as quadwords (64-bit).

After the write command, a window index and an offset must be supplied, in that order. This specifies the memory access window to be read, and where in that window to begin writing data. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the write command. If present, the length parameter specifies how many bytes to write. The length should be an integer multiple of the width; if not, the length is rounded down.

The program obtains the values to be written in two ways: from any additional parameters on the command line after the length parameter, and then from the standard input stream (stdin). This works as follows:

1. Any remaining command line arguments, if present after the length parameter, are interpreted as data values to be written. These values are assumed to be of the radix implied by the command, and are written to the memory window, incrementing the offset with each word written. If there are enough values

passed on the command line to satisfy the byte count, the program terminates.

2 If there are insufficient data values passed on the command line, the program waits for values to be entered on the standard input stream. Values entered this way are also assumed to be of the radix implied by the command, and are written to the memory window, incrementing the offset with each word written. When the entire byte count that was specified in the length parameter has been satisfied or end-of-file is encountered, the program terminates.

An example session looks like this:

```
C>dump rd 0 0x80000 0x40
Window 0 offset 0x80000 mapped @ 0x002D0000
Dump of memory at 0x002D0000 + 80(0x40) bytes:
                  00       04       08       0c
0x002d0000: 00000000 00000000 00000000 00000000 ................
0x002d0010: 00000000 00000000 00000000 00000000 ................
0x002d0020: 00000000 00000000 00000000 00000000 ................
0x002d0030: 00000000 00000000 00000000 00000000 ................

C>dump wd 0 0x80004 0x8 0xdeadbeef
Window 0 offset 0x80004 mapped @ 0x00110004
0x80004: 0xDEADBEEF
0x80008: 0xcafeface

C>dump rd 0 0x80000 0x40
Window 0 offset 0x80000 mapped @ 0x00110000
Dump of memory at 0x00110000 + 64(0x40) bytes:
                  00       04       08       0c
0x00110000: 00000000 deadbeef cafeface 00000000 ................
0x00110010: 00000000 00000000 00000000 00000000 ................
0x00110020: 00000000 00000000 00000000 00000000 ................
0x00110030: 00000000 00000000 00000000 00000000 ................
```

Remarks

When entering data for write commands, values are expressed in decimal by default. To express data as hexadecimal, prefix it with '0x' or use the **+hex** option.

The **DUMP** utility uses store instructions for writes that are equal to the width specified on the command line, if possible. This is not possible if the CPU architecture in use does not have store instructions of the required width or if the offset specified on the command line would result in unaligned stores. In the case of an unaligned offset, writes are performed as a sequence of byte stores, because unaligned stores are illegal on some CPU architectures.

3.6 FLASH utility

WARNING
Incorrect use of the **+failsafe** option may impact long-term reliability of a reconfigurable computing card. Please refer to Section 3.6.1 below for an explanation of the **+failsafe** option and how it may be used.

Command line

```
flash [option ...] info     target-index
flash [option ...] chkblank target-index
flash [option ...] erase    target-index
flash [option ...] program  target-index filename
flash [option ...] verify   target-index filename
```

where

target-index	is the index of a target FPGA.
filename	is the name of a .BIT file (program or verify commands only).

and the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-failsafe	Causes the default image to be erased / programmed / verified (default).
+failsafe	Causes the failsafe image to be erased / programmed / verified; see Failsafe bitstream mechanism below.
-force	Causes a mismatch between the target FPGA device and the .BIT file device to result in an error (default).
+force	Causes a mismatch between the target FPGA device and the .BIT file device to be ignored.

Summary

Blank-checks, erases, programs or verifies a target FPGA bitstream image in the user-programmable Flash memory of a device.

Description

The **FLASH** utility has five commands:

- **chkblank** <*target-index*>

  Verifies that an image is blank, i.e. all bytes are 0xFF.

- **erase** <*target-index*>

  Erases an image so that it becomes blank, i.e. all bytes are 0xFF.

- **info** <*target-index*>

  Displays information about the Flash memory that holds an image.

- **program** <*target-index*> <*filename*>

  Programs the specified bitstream (.BIT) file into an image so that the target FPGA is configured from the

image at power-on or reset.

- **verify** <*target-index*> <*filename*>

  Verifies that an image contains the specified bitstream (.BIT) file.

chkblank command

The **chkblank** command verifies that a target FPGA image is blank, i.e. all bytes are 0xFF, but does not modify the Flash memory bank. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

For example, to blank-check the default image for target FPGA 0:

```
flash chkblank 0 /path/to/my_design.bit
```

erase command

The **erase** command erases a target FPGA image so that it becomes blank, i.e. all bytes are 0xFF. It automatically performs a blank-check after erasing. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

For example, to erase the default image for target FPGA 0:

```
flash erase 0
```

info command

The **info** command displays information about the Flash memory and then exits, without doing anything else. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

program command

The **program** command programs a target FPGA image with the data in the specified bitstream (.BIT) file. Following the command, an index of a target FPGA in the device and the name of a bitstream (.BIT) filename must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

If the device in the .BIT file does not match the target FPGA, this command fails with an error and does not program the target FPGA image, unless the **+force** option is passed. Verification is automatically performed after programming.

For example, to program the default image for target FPGA 0 with a bitstream file called **my_design.bit**:

```
flash program 0 /path/to/my_design.bit
```

verify command

The **verify** command verifies that a target FPGA image contains the data in the specified bitstream (.BIT) file, but does not modify the Flash memory bank. Following the command, an index of a target FPGA in the device and the name of a bitstream (.BIT) filename must be specified. The index of the target FPGA is normally zero but may be nonzero in in models with multiple target FPGAs.

If the device in the .BIT file does not match the target FPGA, this command fails with an error unless the **+force** option is passed. If discrepancies between the target FPGA image and the data in the .BIT file are found, they are displayed (up to a certain number of erroneous bytes), followed by a failure message.

For example, to verify that the default image for target FPGA 0 contains the data in a bitstream file called **my_design.bit**:

```
flash verify 0 /path/to/my_design.bit
```

3.6.1 Failsafe bitstream mechanism

Due to errata in certain Xilinx FPGA families, the following Gen 3 models have a "failsafe bitstream" mechanism:

- ADM-XRC-6TL
- ADM-XRC-6T1
- ADM-XRC-6TGE
- ADM-XRC-6T-ADV8

In the above models, each target FPGA has two images: a default image, and a failsafe image. Alpha Data factory-programs a known-good "null bitstream" into the failsafe image. When power is applied to a card, the firmware on the card first looks for a valid bitstream in the default image. If no bitstream is found, the firmware uses the null bitstream in the failsafe image to configure the target FPGA. In this way, the firmware ensures that the target FPGA is always configured with something when it is powered-on.

Because the purpose of the failsafe image is to protect the target FPGA from sub-micron effects that would otherwise degrade the performance of the target FPGA over time, Alpha Data recommends that the failsafe image should never be erased. If overwritten, a customer must ensure that the bitstream is valid, known-good and satisfies the requirements for protecting the target FPGA from sub-micron effects.

web http://www.xilinx.com/support/answers/35055.htm Xilinx answer record 35055 elaborates on protecting Virtex-6 GTX transceivers from performance degradation over time.

3.7 INFO utility

Command line

```
info [option ...]
```

where the following options are accepted:

-flash	Causes Flash bank information not to be shown (default).
+flash	Causes Flash bank information to be shown.
-index <*index*>	Specifies the index of the card to open (default 0).
-io	Causes I/O module information not to be shown (default).
+io	Causes I/O module information to be shown.
-sensor	Causes sensor information not to be shown (default).
+sensor	Causes sensor information to be shown.
-sn <#>	Specifies the serial number of the card to open.

Summary

Displays information about a reconfigurable computing device.

Description

The **INFO** utility demonstrates the use of most of the informational functions in the ADMXRC3 API. It uses ADMXRC3_OpenEx to open a device in passive mode, meaning that an unprivileged user can successfully run it. The output consists of several sections, the first of which is obtained using ADMXRC3_GetVersionInfo:

```
API information
API library version           1.1.1
Driver version                1.1.1
```

The second section shows information obtained using ADMXRC3_GetCardInfoEx, and shows the information in the ADMXRC3_CARD_INFOEX structure:

```
Card information
Model                         ADM-XRC-6TL
Serial number                 101(0x65)
Number of programmable clocks 1
Number of DMA channels        1
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             10
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF
```

The third section uses the **NumTargetFpga** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetFpgaInfo to enumerate the target FPGAs in the device:

```
Target FPGA information
FPGA 0                        xc6vlx240tff1759
```

The fourth section uses the **NumMemoryBank** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetBankInfo to enumerate the memory banks (non-Flash) in the device:

```
Memory bank information
```

```
Bank 0                          SDRAM, DDR3, 65536 kiWord x 32+0 bits
                                303.0 MHz - 533.3 MHz
                                Connectivity mask 0x1
Bank 1                          SDRAM, DDR3, 65536 kiWord x 32+0 bits
                                303.0 MHz - 533.3 MHz
                                Connectivity mask 0x1
Bank 2                          SDRAM, DDR3, 65536 kiWord x 32+0 bits
                                303.0 MHz - 533.3 MHz
                                Connectivity mask 0x1
Bank 3                          SDRAM, DDR3, 65536 kiWord x 32+0 bits
                                303.0 MHz - 533.3 MHz
                                Connectivity mask 0x1
```

The fourth section uses the **NumWindow** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetWindowInfo to enumerate the memory access windows in the device:

```
Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF5400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0xFAC00000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base     0xFAAFF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis  Bus base     0xFAAFE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

The next section appears if the **+flash** option is passed on the command line. It uses the **NumFlashBank** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetFlashInfo to enumerate the Flash memory banks in the device:

```
Flash bank information
Bank 0                          Intel 28F256P30, 65536(0x10000) kiB
                                Useable area 0x1200000-0x3FFFFFF
```

The next section appears if the **+io** option is passed on the command line. It uses the **NumModuleSite** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetModuleInfo to enumerate the I/O module sites in the device and show what is fitted, if anything:

```
I/O module information
Module 0                        Not present
```

The final section appears if the **+sensor** option is passed on the command line. It uses the **NumSensor** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetSensorInfo to enumerate the sensors in the device:

```
Sensor information
Sensor 0                        1V supply rail
                                V, double, exponent 0, error 0.0
Sensor 1                        1.5V supply rail
                                V, double, exponent 0, error 0.0
Sensor 2                        1.8V supply rail
                                V, double, exponent 0, error 0.0
Sensor 3                        2.5V supply rail
                                V, double, exponent 0, error 0.1
Sensor 4                        3.3V supply rail
                                V, double, exponent 0, error 0.1
Sensor 5                        5V supply rail
                                V, double, exponent 0, error 0.1
```

```
Sensor 6                    XMC variable power rail
                            V, double, exponent 0, error 0.2
Sensor 7                    XRM I/O voltage
                            V, double, exponent 0, error 0.1
Sensor 8                    LM87 internal temperature
                            deg. C, double, exponent 0, error 3.0
Sensor 9                    Target FPGA temperature
                            deg. C, double, exponent 0, error 4.0
```

3.8 ITEST example

Command line

```
itest [option ...]
```

where the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-uber	Uses the ITest FPGA design bitstream (default).
+uber	Uses the UBER FPGA design bitstream.

Summary

Demonstrates consumption of FPGA interrupt notifications.

Description

This example demonstrates how to consume FPGA interrupt notifications in an application. By default, it uses the ITest example FPGA design as a means of generating FPGA interrupt notifications, and starts a thread whose purpose is to wait for and acknowledge interrupts from the target FPGA.

When ITEST is started, the main thread first configures target FPGA 0 with the bitstream (.bit file) for the ITest example FPGA design. The main thread then launches an interrupt thread that waits for notifications, in a loop. The main thread then proceeds to wait for input, also in a loop. At this point, the user may press RETURN to generate an interrupt, or enter 'q' to terminate the program. On termination, the program displays the number of FPGA interrupt notifications that the interrupt thread consumed during execution.

A sample session looks like this:

```
Enter 'q' to quit, or anything else to generate an interrupt:
Interrupt thread started

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:
q
Generated 5 interrupts
Interrupt thread saw 5 interrupt(s)
```

The blank lines in the above session are simply empty lines where the user has pressed return. As can be seen, each of the 5 interrupts generated results in the interrupt thread consuming a notification.

Remarks

As noted in the ADMXRC3 API Specification (see functions ADMXRC3_RegisterWin32Event, ADMXRC3_RegisterVxwSem and ADMXRC3_StartNotificationWait), the ADMXRC3 API does not queue each type of notification. Therefore, this example works as expected as long as the frequency of target FPGA interrupt notifications is not too fast for the interrupt thread. Since the rate of generation of notifications in this example is limited the user's keyboard input rate, the interrupt thread should be able to keep up (as long as the machine is not heavily loaded with other processes). Nevertheless, it is important to note that in this simple example, there is

no mechanism for throttling the rate of notifications so that notifications cannot be lost. In a real application, the preferred design approaches are:

1. Architect the FPGA design and host application so that they tolerate *out-of-date* notifications being missed. For example, if the target FPGA generates an interrupt when data arrives via an I/O interface, it does not matter if the host application does not succeed in consuming every target FPGA interrupt notification, because the notifications before the latest one are considered out-of-date. When the host application handles a notification, it reads a register in the target FPGA to determine the amount of new data rather than using the number of notifications consumed. What matters is that regardless of how many times the target FPGA generates an interrupt, the host application is guaranteed to eventually wake up and check for new data.
2. Use a fully handshaked system, where the host application must positively acknowledge a target FPGA interrupt before the target FPGA generates a new interrupt.

In fact, the above two approaches are best used together, because minimizing the number of FPGA interrupts minimizes unnecessary context switches in the operating system.

3.9 LOADER utility

Command line

```
loader [option ...] <target FPGA index> <bitstream filename>
```

where the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-binary	Treat <*bitstream filename*> as a .bit file, and attempt to parse it accordingly.
+binary	Treat <*bitstream filename*> as a binary file containing only SelectMap data.
-checklink	Do not verify that communications with the target FPGA have been established before exiting.
+checklink	Verify that communications with the target FPGA have been established before exiting (default).

Summary

Configures a target FPGA with a .bit file, and then exits.

Description

The **LOADER** utility configures the target FPGA, identified by <*target FPGA index*>, within a reconfigurable computing device with the bitstream file identified by <*bitstream filename*>, and then exits.

By default, **LOADER** expects <*bitstream filename*> to name a .bit file, i.e. a file generated by the **bitgen** tool in the Xilinx ISE toolset, and attempts to parse the file accordingly. However, the **+binary** option causes **LOADER** to treat <*bitstream filename*> as a file containing raw SelectMap data. Such a file can be obtained in a number of ways, including a user-created program or by using the BITSTRIP utility.

Because the **LOADER** utility uses ADMXRC3_ConfigureFromFile or ADMXRC3_ConfigureFromBuffer, it normally checks that communications with the target FPGA have been established before exiting. Passing the **-checklink** option causes this check to be omitted, and is needed for an FPGA design that has no host interface (i.e. no OCP communication) with the host, such as a stand-alone Ethernet design.

3.10 MEMTESTH example

Command line

```
memtesth [option ...]
```

where the following options are accepted:

-banks <*bitmask*>	Specifies which banks to test, as a bitmask (default all banks).
-dma	Use CPU-initiated data transfer instead of DMA data transfer during the test; this is relatively slow and may increase runtime to minutes instead of seconds.
+dma	Use DMA transfers for transferring data between host memory and the target FPGA (default).
-index <*index*>	Specifies the index of the card to open (default 0).
-maxerror <#>	Specifies the maximum number of data verification errors to display; note that further errors are still counted and a total is displayed at the end of the test (default 20).
-repeat <#>	Specifies the number of times to repeat the data test; 0 means "for ever" (default 1).
-sn <#>	Specifies the serial number of the card to open.

Summary

Performs a host-driven test of the memory banks on a reconfigurable computing card.

Description

The MEMTESTH example demonstrates the transfer of data between host memory and on-board memory devices (for example, DDR3 SDRAM on the ADM-XRC-6T1), via the target FPGA. A number of test phases are performed, each with a different data generation method, such as alternating an 55 / AA pattern, "own address" etc. In each phase, each bank is tested by first filling the bank with data and then reading it back in order to verify that data transfers are error-free.

This example makes use of the Uber example FPGA design. Assuming no errors are detected, running it produces output of the form:

```
Bank 00: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank 01: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank 02: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank 03: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank test mask is 0x000f
Performing host-driven memory test...
Phase 1 - 0x55 pattern
Phase 2 - 0xAA pattern
Phase 3 - own address pattern
Phase 4 - pseudorandom data
Measuring throughput...
Throughput from host to memory is 439.7 MiB/s
Throughput from memory to host is 1009.6 MiB/s
PASSED
```

3.11 MONITOR utility

Command line

```
monitor [option ...]
```

where the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-period <*delay*>	Specifies the update period, in seconds.
-repeat <*n*>	Specifies the number of updates to perform (default 0); a value of zero means "repeat for ever".
-sn <#>	Specifies the serial number of the card to open.

Summary

Displays readings from all sensors.

Description

The **MONITOR** utility repeatedly displays sensor readings in the command shell at the interval specified by the **-period** option. The number of updates to perform before terminating can be specified on the command line using the **-repeat** option, but by default, the program runs until interrupted with CTRL-C.

It makes use of the ADMXRC3_GetSensorInfo and ADMXRC3_ReadSensor functions from the ADMXRC3 API, and because it opens a device in passive mode using ADMXRC3_OpenEx, it can run alongside other reconfigurable computing applications without disturbing them.

The output looks like this:

```
Model:                                  257 (0x101) => ADM-XRC-6TL
Serial number:                          101 (0x65)
Number of sensors:                      10
  Sensor  0               1V supply rail: 0.987000 V
  Sensor  1             1.5V supply rail: 1.509186 V
  Sensor  2             1.8V supply rail: 1.803192 V
  Sensor  3             2.5V supply rail: 2.508896 V
  Sensor  4             3.3V supply rail: 3.268082 V
  Sensor  5               5V supply rail: 5.017990 V
  Sensor  6      XMC variable power rail: 12.000000 V
  Sensor  7              XRM I/O voltage: 2.495712 V
  Sensor  8    LM87 internal temperature: 49.000000 deg C
  Sensor  9      Target FPGA temperature: 57.000000 deg C
```

3.12 SIMPLE example

Command line

```
simple [option ...]
```

where the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-uber	Uses the SIMPLE FPGA design bitstram (default).
+uber	Uses the UBER FPGA design bitstream.

Summary

Demonstrates access to target FPGA registers.

Description

The SIMPLE example application demonstrates accessing FPGA registers in its simplest form. It first configures target FPGA 0 with the Simple example FPGA design, or the Uber example FPGA design if the **+uber** option is specified. It then waits for input from the user. The user enters hexadecimal values (up to 32 bits in length), and for each value:

1. The program writes the value to a register in the target FPGA.
2. The target FPGA nibble-reverses the value and makes the reversed value available to be read via a register. Here, nibble-reversing means that the FPGA swaps bits 31:28 with 3:0, 27:24 with 7:4 etc.
3. The program reads back and displays the nibble-reversed value.

The program terminates on CTRL-D (Linux) or CTRL-Z (Windows). A sample session looks like this:

```
=========================
Enter values for I/O
(CTRL-D / CTRL-Z to exit)
=========================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
deadbeef
OUT = 0xdeadbeef, IN = 0xfeebdaed
cafeface
OUT = 0xcafeface, IN = 0xecafefac
```

3.13 SIMPLEDMA example

Command line

```
simpledma [option ...]
```

where the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <*#*>	Specifies the serial number of the card to open.
-address <*64-bit hex value*>	Specifies the OCP address at which to begin reading data (default 0x0).
-maxerror <*n*>	Specifies the maximum number of data verification errors to display (default 20).
-n <*number of bytes*>	Specifies the number of bytes to read and verify (default 0x100); must be no greater than the constant **BUFFER_SIZE** (0x100000).

Summary

Basic test program for the SimpleDMA Example FPGA Design.

Description

The SIMPLEDMA example application performs a basic test of the SimpleDMA Example FPGA Design, by reading 256 bytes (by default) from a given OCP address on DMA channel 0 and displaying them (as 32-bit words). The default OCP address at which reading begins is 0, but this can be specified on the command line using the **-address** option. As well as reading and displaying some data read from the FPGA, the program also measures and displays the rate at which data can be transferred via DMA channel 0.

In the SimpleDMA Example FPGA Design, reads on DMA channel 0 return a pattern of incrementing 32-bit integers, based on the OCP address of each 32-bit word of data divided by 4. The output of the program, when run with the default options, looks like this:

```
Reading using DMA channel 0 at OCP address 0x0...
Dump of data read from OCP address 0x0 + 0x100 B:
                           00       04       08       0c
0x00000000_00000000: 00000000 00000001 00000002 00000003 ................
0x00000000_00000010: 00000004 00000005 00000006 00000007 ................
0x00000000_00000020: 00000008 00000009 0000000a 0000000b ................
0x00000000_00000030: 0000000c 0000000d 0000000e 0000000f ................
0x00000000_00000040: 00000010 00000011 00000012 00000013 ................
0x00000000_00000050: 00000014 00000015 00000016 00000017 ................
0x00000000_00000060: 00000018 00000019 0000001a 0000001b ................
0x00000000_00000070: 0000001c 0000001d 0000001e 0000001f ................
0x00000000_00000080: 00000020 00000021 00000022 00000023  ...!..."...#...
0x00000000_00000090: 00000024 00000025 00000026 00000027 $...%...&...'...
0x00000000_000000a0: 00000028 00000029 0000002a 0000002b (...)...*...+...
0x00000000_000000b0: 0000002c 0000002d 0000002e 0000002f ,...-......./...
0x00000000_000000c0: 00000030 00000031 00000032 00000033 0...1...2...3...
0x00000000_000000d0: 00000034 00000035 00000036 00000037 4...5...6...7...
0x00000000_000000e0: 00000038 00000039 0000003a 0000003b 8...9...:...;...
0x00000000_000000f0: 0000003c 0000003d 0000003e 0000003f <...=...>...?...
Measuring throughput...
Throughput from host to FPGA is 703.8 MiB/s
Throughput from FPGA to host is 751.0 MiB/s
PASSED
```

3.14 SYSMON utility

Command line

```
sysmon
```

Summary

Utility presenting device information and hardware sensors in a graphical user interface.

Description

The **SYSMON** utility combines the information shown by the INFO and MONITOR utilities with a graphical user interface. Its main function is graphical display of hardware sensor data, and it can be minimized to the notification area of the desktop (the "System Tray" in Windows) in order to run unobtrusively.

It makes use of the ADMXRC3_GetSensorInfo and ADMXRC3_ReadSensor functions from the ADMXRC3 API, and because it opens a device in passive mode using ADMXRC3_OpenEx, it can run alongside other reconfigurable computing applications without disturbing them.

The user interface of the Linux version of SYSMON is as follows, upon starting the utility:

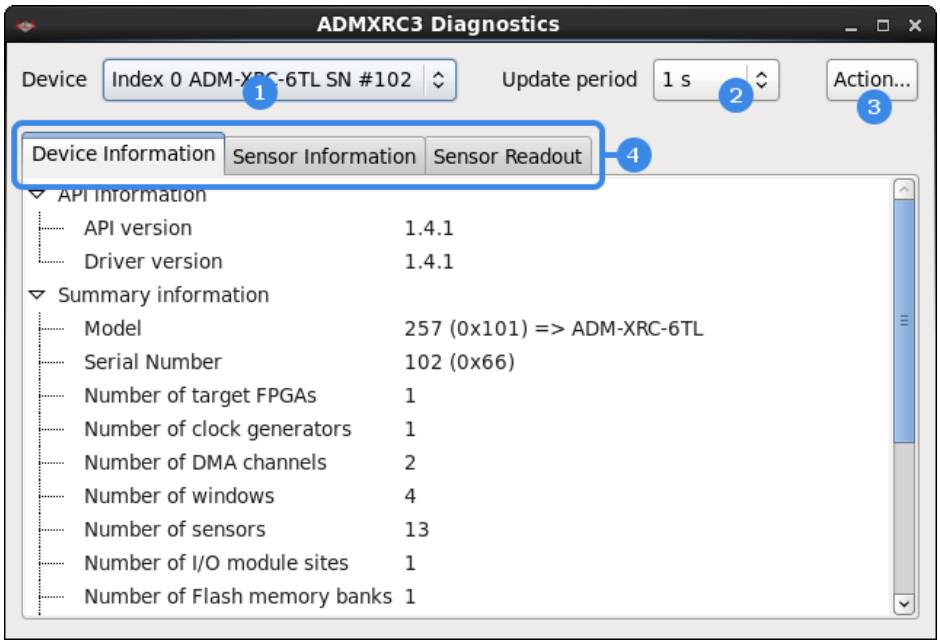

Figure 2 : SYSMON user interface

Referring to Figure 2, the user interface elements are as follows:

1 A combo box that specifies which reconfigurable computing device to use.

2 A combo box that selects the time interval between sensor readings.

3 A button that reveals the **Action** menu when clicked. The **Action** menu allows sensor data logging as described in Section 3.14.1 below. The Windows version of SYSMON does not have this button, but instead hosts equivalent functionality via the system menu.

4 A tab control whose tabs are as follows:

- The device information tab shows information about the currently selected device.

- The sensor information tab shows information about the available sensors in the currently selected device.
- The sensor readout tab displays sensor data graphically in up to four 'scopes.

When minimized (item 5), **sysmon** appears in the notification area of the desktop:

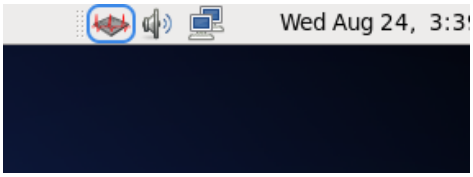

Figure 3 : SYSMON notification area icon

The icon shown in the notification area has a context menu activated by a right-click, and this can be used to restore the application to the desktop, as well as offering the same logging functions as the **Action** menu. Refer to Section 3.14.1 for a description of data logging.

To actually close the application as opposed to minimizing it, click the close button of the window.

SYSMON device information tab

The set of information shown in the device information tab is approximately the same as that shown by the command-line INFO utility, but with a collapsible tree structure.

SYSMON sensor information tab

The sensor information tab is a tabular view of the available sensors, including the current reading for each sensor:

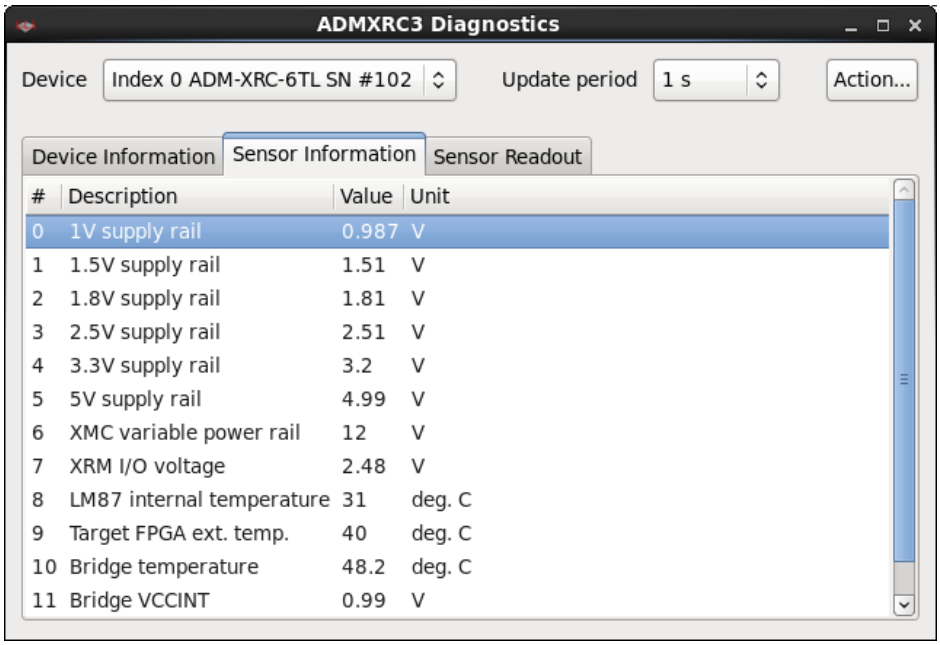

Figure 4 : SYSMON sensor information tab

SYSMON sensor readout tab

The sensor readout tab displays sensor readings in graphical form:

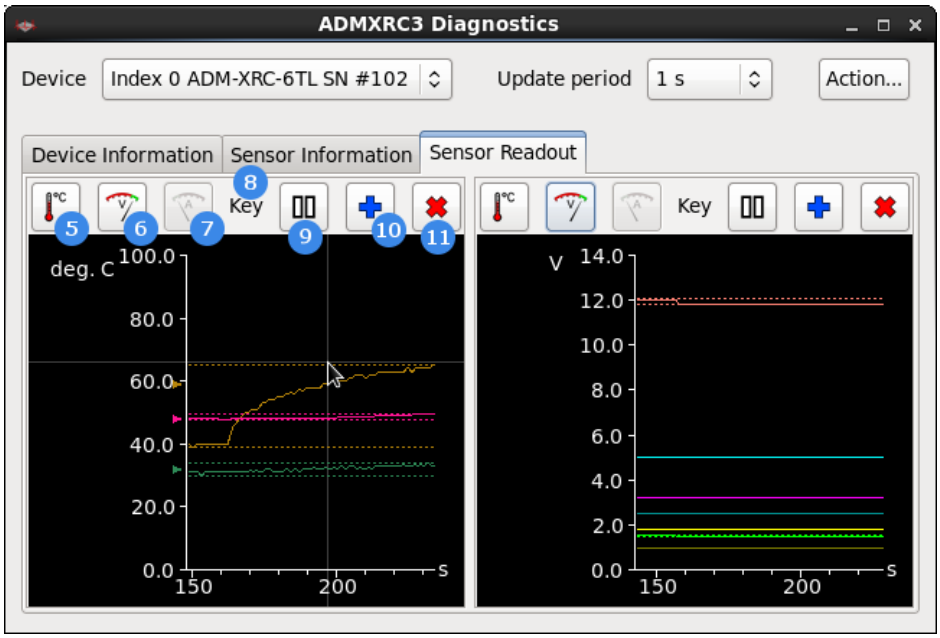

Figure 5 : SYSMON sensor readout tab

Initially, the 'scope is empty and displays no sensors. The above figure shows two scopes, one showing temperatures and the other showing voltages. The user interface elements of the 'scope toolbar are as follows:

5 The temperature button sets the 'scope to display all temperature sensors in the device, and starts updates.

6 The voltage button sets the 'scope to display all voltage sensors in the device, and starts updates.

7 The current button sets the 'scope to display all current sensors in the device, and starts updates.

8 Mouse over the key to see which sensor corresponds to which colored trace.

9 The pause / resume button can be used to pause and resume update of the 'scope.

10 A button that adds another 'scope when clicked, to a maximum of 4, so that various types of sensor can be viewed at the same time.

11 A button that destroys a 'scope when clicked. If there is only one 'scope, the button is disabled.

3.14.1 SYSMON sensor data logging

In Linux, SYSMON can log sensor data over an arbitrary time period via the **Action** menu:

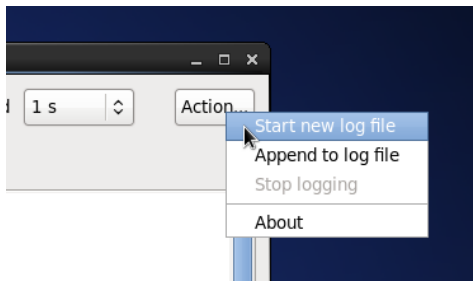

Figure 6 : SYSMON Action menu in Linux

In Windows, the **Action** button is not present, and the **Action** menu items are located in the system menu:

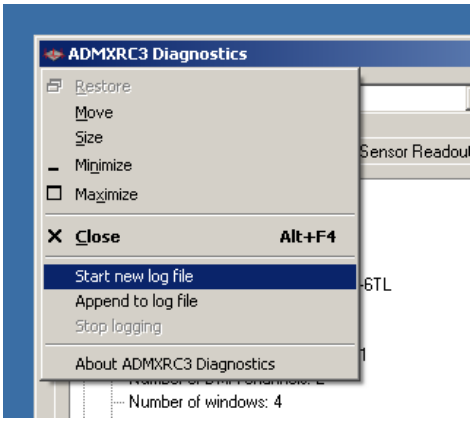

Figure 7 : SYSMON Action menu in Windows

Data logging works as follows:

- The **Start new log file** option prompts for a filename into which sensor data is to be logged. If a file of that name already exists, it will be overwritten.
- The **Append to log file** option prompts for a filename into which sensor data is to be logged, but unlike **Start new log file**, if a file of that name already exists, new data will be appended to it.
- The **Stop logging** option is only enabled after logging has successfully been started using **Start new log file** or **Append to log file**, and causes SYSMON to cease logging data.

The files created are in comma-separated value (CSV) format (some rows and columns deleted for brevity):

```
START,11:59:07 23 Aug 2011
COMMENT,MODEL,SERIAL#
DEVICE,ADM-XRC-6TL,102
COMMENT,SENSOR#,Description,Unit
SENSOR,0,1V supply rail,V
SENSOR,1,1.5V supply rail,V
SENSOR,2,1.8V supply rail,V
SENSOR,3,2.5V supply rail,V
...
SENSOR,12,Bridge VCCAUX,V
COMMENT,TIMESTAMP,1V supply rail,1.5V supply rail,1.8V supply rail,2.5V supply
rail,...
COMMENT,ms,V,V,V,V,...
DATA,583,0.987000,1.509186,1.812988,2.508896,...
DATA,1584,0.987000,1.509186,1.812988,2.508896,...
DATA,2645,0.987000,1.509186,1.812988,2.508896,...
DATA,3646,0.987000,1.509186,1.812988,2.508896,...
...
DATA,13661,0.987000,1.509186,1.812988,2.508896,...
DATA,14663,0.987000,1.509186,1.812988,2.508896,...
STOP,11:59:22 23 Aug 2011
```

The string in column 1 of each row indicates what information a row contains:

- **START** signifies the start of a logging session, in case the file contains multiple sessions that were obtained using the **Append to log file** option.
- **STOP** signifies the end of a logging session, in case the file contains multiple sessions that were obtained using the **Append to log file** option.

- **COMMENT** signifies a comment, for the benefit of human readers, and can be filtered out by a program that reads the file.
- **DEVICE** identifies the model and serial number, in the second and third cells respectively, of the physical card from which the data was collected.
- **SENSOR** signifies information about a sensor. The second cell is the sensor index, the third cell is the sensor's description and the fourth cell is the unit for that sensor.
- **DATA** signifies a set of sensor readings at a given instant. The second cell is a timestamp, in milliseconds, relative to the time and date in the **START** row. The third and subsequent cells are individual sensor values, where the third cell corresponds to the **SENSOR** row whose sensor index is 0, the fourth cell corresponds to the **SENSOR** row whose sensor index is 1 etc.

3.14.2 Building SYSMON in Linux

The Linux version of the **SYSMON** utility uses **GTKMM-2.4**. This package is present in recent Linux distributions, but may not be present in some Linux distributions. For this reason, **SYSMON** is built separately from the other example applications. A non-exhaustive list of the packages that are required to build **SYSMON** is as follows:

gtkmm24-devel	cairomm-devel
libsigc++20-devel	glibmm24-devel
pangomm-devel	pkgconfig

To run **SYSMON**, the corresponding runtime packages are required:

gtkmm24	cairomm
libsigc++20	glibmm24
pangomm	

To build the "Release" configuration of **SYSMON**, enter the following commands in a BASH shell:

```
$ cd $ADMXRC3_SDK/apps/linux
$ ./configure
$ cd sysmon
$ make CONFIG=Release clean all
```

The executable's path is then **apps/linux/sysmon/bin/Release/sysmon**.

3.15 VPD utility

Command line

```
vpd [option ...] fb address n [data]
vpd [option ...] fw address n [data]
vpd [option ...] fd address n [data]
vpd [option ...] fq address n [data]
vpd [option ...] fs address n [string]
vpd [option ...] rb address [n]
vpd [option ...] rw address [n]
vpd [option ...] rd address [n]
vpd [option ...] rq address [n]
vpd [option ...] wb address [n] [data ...]
vpd [option ...] ww address [n] [data ...]
vpd [option ...] wd address [n] [data ...]
vpd [option ...] wq address [n] [data ...]
vpd [option ...] ws address [n] [string ...]
```

where

address	is the address in VPD memory at which to begin reading or writing.
n	is the number of bytes to read or write.
data	is a numeric data item, valid for fill and write commands.
string	is a string data item, valid for fill and write commands.

and the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-hex	Causes numeric data values to be interpreted as decimal unless prefixed by '0x' (default).
+hex	Causes numeric data values to be interpreted as hexadecimal always.

Summary

Displays data read from VPD memory, or writes data to VPD memory.

Description

The **VPD** utility operates in one of three modes:

- Filling a region of VPD memory with a value or string; for this mode, use the **fb**, **fw**, **fd**, **fq** or **fs** commands.
- Reading data from VPD memory and displaying it; for this mode, use the **rb**, **rw**, **rd** or **rq** commands.
- Writing numeric or string data to a region of VPD memory; for this mode, use the **wb**, **ww**, **wd**, **wq** or **ws** commands.

Fill mode

When filling a region of VPD memory with data, the fill command specifies whether the data is numeric or string data. In the case of numeric data, the command also implies the radix (i.e. word size) of the data. The available fill commands are:

- **fb**

Fill value is a byte (8-bit).

- **fw**

  Fill value is a word (16-bit).

- **fd**

  Fill value is a doubleword (32-bit).

- **fq**

  Fill value is a quadword (64-bit).

- **fs**

  Fill value is an ASCII string (8-bit characters).

The next 3 arguments after the fill command must be:

- *address* - the byte address within VPD memory at which to begin filling.
- *n* - byte count; the number of bytes of VPD memory to fill.
- *data* or *string* - the numeric or string value to place in the specified region of VPD memory.

If the command is **fs** and the string value is shorter than the byte count *n*, the string is repeated until the byte count is satisfied. If the string is longer than the byte count *n*, only the first *n* characters are used. If a string contains spaces, it must be quoted on the command line so that it is not interpreted by the shell as two or more separate arguments.

For the numeric fill commands **fb**, **fw**, **fd** and **fq**, the numeric value is repeated until the byte count is satisfied.

Read mode

The read command implies the radix (i.e. word size) used for displaying the data:

- **rb**

  Byte (8-bit) reads; data is displayed as bytes.

- **rw**

  Word (16-bit) reads; data is displayed as words.

- **rd**

  Doubleword (32-bit) reads; data is displayed as doublewords.

- **rq**

  Quadword (64-bit) reads; data is displayed as quadwords.

After the read command, an address must be supplied, which specifies where in VPD memory to begin reading. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the read command. If present, the length parameter specifies how many bytes to read and display. The length should be an integer multiple of the width; if not, the length is rounded down.

Write mode

The write command specifies whether the data is numeric or string data. In the case of numeric data, the command also implies the radix (i.e. word size) of the data. The available write commands are:

- **wb**

  Data is written as bytes (8-bit).

- **ww**

  Data is written as words (16-bit).

- **wd**

  Data is written as doublewords (32-bit).

- **wq**

  Data is written as quadwords (64-bit).

- **ws**

  Data is supplied as one or more ASCII strings (8-bit characters).

After the write command, an address must be supplied, which specifies where in VPD memory to begin writing data. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the write command. If present, the length parameter specifies how many bytes to write. The length should be an integer multiple of the width; if not, the length is rounded down.

The program obtains the values to be written in two ways: from any additional parameters on the command line after the length parameter, and then from the standard input stream (stdin). This works as follows:

1. Any remaining command line arguments, if present after the length parameter, are interpreted as data values to be written. Numeric values are assumed to be of the radix implied by the command parameter. As each value it written to VPD memory, the address is incremented. If there are enough values passed on the command line to satisfy the byte count, the program terminates.
2. If there are insufficient data values passed on the command line, the program waits for values to be entered on the standard input stream. Numeric values entered this way are also assumed to be of the radix implied by the command. As each value it written to VPD memory, the address is incremented. When the entire byte count that was specified in the length parameter has been satisfied or end-of-file is encountered, the program terminates.

Example session

The following session was captured under Linux using an ADM-XRC-6TL. The base address 0x100000 is used because that is the VPD-space address of the user-definable area of VPD memory in the ADM-XRC-6TL.

```
$ ./vpd rb 0x100000 0x60
Dump of VPD at 0x100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x00100000: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100010: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100020: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100030: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100050: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
$ ./vpd fs 0x100008 20 'hello world!'
$ ./vpd wd 0x100020 12
0x00100020: 0xdeadbeef
0x00100024: 0xcafeface
0x00100028: 0x12345678
$ ./vpd fw 0x100031 10 0xa55a
$ ./vpd rb 0x100000 0x60
Dump of VPD at 0x100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x00100000: ff ff ff ff ff ff ff ff 68 65 6c 6c 6f 20 77 6f ........hello wo
0x00100010: 72 6c 64 21 68 65 6c 6c 6f 20 77 6f ff ff ff ff rld!hello wo....
0x00100020: ef be ad de ce fa fe ca 78 56 34 12 ff ff ff ff ........xV4.....
0x00100030: ff 5a a5 5a a5 5a a5 5a a5 5a a5 ff ff ff ff ff .Z.Z.Z.Z.Z......
0x00100040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100050: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
```

NOTE: the above sesssion assumes that VPD write protection has been disabled as described in the release notes for the ADB3 Driver for Linux or Windows (as appropriate).

Remarks

When entering data for fill or write commands, values are expressed in decimal by default. To express data as hexadecimal, prefix it with '0x' or use the **+hex** option.

In the current version of the **VPD** utility, data is always read from and written to VPD memory in little-endian byte order.

4 Example applications for VxWorks

The example applications and utilities are described in the following subsections.

FLASH	Utility for programming FPGA bitstream (.BIT) files in user-programmable Flash memory
INFO	Utility for displaying information about a reconfigurable computing device
ITEST	Example demonstrating how to consume target FPGA interrupt notifications in an application
LOADER	Utility for configuring a target FPGA with a bitstream file
MEMTESTH	Example demonstrating host-driven memory test
MONITOR	Utility that displays sensor readings
SIMPLE	Example demonstrating how to read and write registers in a target FPGA
SIMPLEDMA	Example that tests the SimpleDMA Example FPGA Design
VPD	Utility that allows the Vital Product Data of a reconfigurable computing device to be read or written

Source code for the example VxWorks and Linux applications is provided in the **apps/vxworks/src** directory, relative to the root of the SDK.

4.1 Building the example VxWorks applications in Windows

If using a Windows machine for VxWorks hosting and development, follow these steps:

1. Make a copy of the SDK according to the discussion in Section 2.4.
2. Start a **VxWorks Development Shell** via the shortcut on the Windows Start Menu. It is important to use this shortcut in order to obtain the correct environment for performing command-line builds using the Wind River VxWorks toolchains.
3. Change directory to

   ```
   $(ADMXRC3_SDK)/apps/vxworks
   ```

   where *$(ADMXRC3_SDK)* is the root of the copy of the SDK that you have made.
4. Execute the following command, replacing <config> with the name of the configuration that is appropriate for your target system:

   ```
   make CONFIG=<config> clean all
   ```

   For example, the Pentium 4 configuration for VxWorks 6.7 is **p4-6.7**, and the PowerPC 604 configuration

for VxWorks 6.7 is **ppc604-6.7**. The configuration that you use depends on the target system. Alpha Data supplies several predefined configurations, but it is possible that none of these are exactly what is required for your target system. Refer to Section 4.3 for a discussion of configurations and how to create a new configuration.

The full path, by default, of the binary downloadable module is:

```
$(ADMXRC3_SDK)/apps/vxworks/<config>/debug/admxrc3Apps.out
```

However, the **DEBUG** and **VSB** options can modify this path as shown in Table 2.

4.2 Building the example VxWorks applications in Linux

TBA

4.3 MAKE options for the example VxWorks applications

The top-level Makefile for the VxWorks examples accepts a number of options which are passed on the MAKE command line. These are:

- **CONFIG=*<configuration>***

  Specifies a predefined configuration defined by the file **rules.*<configuration>***, located in the same folder as the Makefile. This option affects the directory where the binary is placed; see Table 2 below for details.

  The rules file may contain any of the following options; for an example, see **rules.p4-6.7**.

- **CPU=*<cpu>***

  Specifies the CPU being targetted; for example PPC604 or PENTIUM4 (default). Must be appropriate for the TARGET option.

- **DEBUG=*<false|true>***

  Specifies a release (false) or debug (true, default) build. This option affects the directory where the binary is placed; see Table 2 below for details.

- **EXTRA_CCOPTS=*<extra compiler options>***

  Specifies extra C compiler options.

- **EXTRA_LDOPTS=*<extra linker options>***

  Specifies extra linker options.

- **TARGET=*<target spec>***

  Defines the target specification, which must be appropriate for the **CPU** option. Examples of valid target specifications for the DIAB toolchain are **-tPPC604FH:vxworks55** (PowerPC 604 VxWorks 5.5) and **-tPENTIUM4LH:vxworks67** (default, Pentium 4 VxWorks 6.7). Examples of valid target specifications for the GNU toolchain are **-mcpu=604** (PowerPC 604) and **-mtune=pentium4 -march=pentium4** (Pentium 4).

- **TOOLCHAIN=*<diab|gnu>***

  Specifies the toolchain to be used to build the driver; legal values are **diab** (default) or **gnu**. If the **gnu** toolchain is selected, the following additional options must be specified (which can be in the rules file specified by the **CONFIG** option, for convenience):

- **CC=*<compiler>***

  Specifies the C compiler; must be appropriate for the **CPU** and **TARGET** options. For example, **ccppc** selects the PowerPC GNU compiler.

- **LD=*<linker>***

  Specifies the linker; must be appropriate for the **CPU** and **TARGET** options. For example, **ldppc** selects the PowerPC GNU linker.

- **NM=*<object dumper>***

  Specifies object dumper; must be appropriate for the **CPU** and **TARGET** options. For example, **nmppc** selects the PowerPC GNU object dump utility.

- **VSB=*<variant>***

  Specifies VxWorks source build (VSB) variant libraries, if required. If omitted, the normal libraries are used. The most common value for this option is **smp**. This option affects the directory where the binary is placed; see Table 2 below for details.

When the **CONFIG** option is specified, the SDK's build system reads a rules file that contains values for the other options. For example, the configuration **ppc604-6.7** has a rules file **rules.ppc604-6.7**. This configuration targets a PowerPC 604 CPU running VxWorks 6.7. and by way of illustration, the rules file contains:

```
CPU=PPC604
ifeq ($(TOOLCHAIN),diab)
EXTRA_CCOPTS=-Xcode-absolute-far -Xdata-absolute-far
TARGET=-tPPC604FH:vxworks67
else
ifeq ($(TOOLCHAIN),gnu)
EXTRA_CCOPTS=-mlongcall
CC=ccppc
LD=ldppc
NM=nmppc
TARGET=-mcpu=604
else
$(error "TOOLCHAIN $(TOOLCHAIN) not recognized.")
endif
endif
```

If no **CONFIG** option is specified, the default configuration is **default**. The **rules.default** file contains:

```
CPU=PENTIUM4
ifeq ($(TOOLCHAIN),diab)
TARGET=-tPENTIUM4LH:vxworks67
else
ifeq ($(TOOLCHAIN),gnu)
CC=ccpentium
LD=ldpentium
NM=nmpentium
TARGET=-mtune=pentium4 -march=pentium4
else
$(error "TOOLCHAIN $(TOOLCHAIN) not recognized.")
endif
endif
```

It is possible that none of the predefined configurations supplied by Alpha Data is appropriate for your hardware platform. If that is the case, a new configuration can be created by using one of the existing rules files as a template and modifying it appropriately.

Several options affect the location where the resulting binary is placed, assuming that a build is successful. The naming conventions are as follows:

DEBUG option	VSB option	Path to binary
false	not defined	$(ADMXRC3_SDK)/apps/vxworks/<config>/release/admxrc3Apps.out
true	not defined	$(ADMXRC3_SDK)/apps/vxworks/<config>/debug/admxrc3Apps.out
false	defined	$(ADMXRC3_SDK)/apps/vxworks/<config>/release_<VSB value>/admxrc3Apps.out
true	defined	$(ADMXRC3_SDK)/apps/vxworks/<config>/debug_<VSB value>/admxrc3Apps.out

Table 2 : Naming conventions for VxWorks examples binary

For example, if **DEBUG=true** and **VSB=smp**, the path to the binary is

```
$(ADMXRC3_SDK)/apps/vxworks/<config>/debug_smp/admxrc3Apps.out
```

4.4 FLASH utility (VxWorks)

WARNING
Incorrect use of the FLAG_FAILSAFE value (0x100) for the **flags** parameter may impact long-term reliability of a reconfigurable computing card. Please refer to Section 4.4.1 below for an explanation of the failsafe bitstream mechanism and how it may be used.

Invocation in VxWorks shell

```
admxrc3Flash <index>, <flags>, "info",     <target-index>
admxrc3Flash <index>, <flags>, "chkblank", <target-index>
admxrc3Flash <index>, <flags>, "erase",    <target-index>
admxrc3Flash <index>, <flags>, "program",  <target-index>, <"filename">
admxrc3Flash <index>, <flags>, "verify",   <target-index>, <"filename">
```

where

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index **FLAG_FORCE** (0x10) => a program or verify command proceeds even if the FPGA type in the .BIT file device does not match the FPGA type in the device **FLAG_FAILSAFE** (0x100) => performs the operation on the the failsafe image instead of the default image
target-index	is the index of a target FPGA (default 0).
"filename"	is a string containing the name of a .BIT file (**program** or **verify** commands only).

The **FLASH** utility requires one of the following commands to be passed as a string argument in the third parameter:

- **chkblank**

  Verifies that an image is blank, i.e. all bytes are 0xFF.

- **erase**

  Erases an image so that it becomes blank, i.e. all bytes are 0xFF.

- **info**

  Displays information about the Flash memory.

- **program**

  Programs the specified bitstream (.BIT) file into an image so that the target FPGA is configured from the image at power-on or reset.

- **verify**

  Verifies that an image contains the specified bitstream (.BIT) file.

chkblank command

The **chkblank** command verifies that a target FPGA image is blank, i.e. all bytes are 0xFF, but does not modify the Flash memory bank. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

For example, to blank-check the default image for target FPGA 0 in the reconfigurable computing device whose index is 0:

```
-> admxrc3Flash 0,0,"chkblank",0
```

erase command

The **erase** command erases a target FPGA image so that it becomes blank, i.e. all bytes are 0xFF. It automatically performs a blank-check after erasing. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

For example, to erase the default image for target FPGA 0 in the reconfigurable computing device whose index is 0:

```
-> admxrc3Flash 0,0,"erase",0
```

info command

The **info** command displays information about the Flash memory and then exits, without doing anything else. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

program command

The **program** command programs a target FPGA image with the data in the specified bitstream (.BIT) file. Following the command, an index of a target FPGA in the device and the name of a bitstream (.BIT) filename must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

If the device in the .BIT file does not match the target FPGA, this command fails with an error and does not program the target FPGA image, unless the **+force** option is passed. Verification is automatically performed after programming.

For example, to program the default image for target FPGA 0, in the reconfigurable computing device whose index is 0, with a bitstream file called **my_design.bit**:

```
-> admxrc3Flash 0,0,"program",0,"host:/path/to/my_design.bit"
```

verify command

The **verify** command verifies that a target FPGA image contains the data in the specified bitstream (.BIT) file, but does not modify the Flash memory bank. Following the command, an index of a target FPGA in the device and the name of a bitstream (.BIT) filename must be specified. The index of the target FPGA is normally zero but may be nonzero in models with multiple target FPGAs.

If the device in the .BIT file does not match the target FPGA, this command fails with an error unless the **flags** parameter contains **FLAG_FORCE** (0x10). If discrepancies between the target FPGA image and the data in the .BIT file are found, they are displayed (up to a certain number of erroneous bytes), followed by a failure message.

For example, to verify that the default image for target FPGA 0, in the reconfigurable computing device whose index is 0, contains the data in a bitstream file called **my_design.bit**:

```
-> admxrc3Flash 0,0,"verify",0,"host:/path/to/my_design.bit"
```

4.4.1 Failsafe bitstream mechanism (VxWorks)

Due to errata in certain Xilinx FPGA families, the following Gen 3 models have a "failsafe bitstream" mechanism:

- ADM-XRC-6TL
- ADM-XRC-6T1
- ADM-XRC-6TGE
- ADM-XRC-6T-ADV8

In the above models, each target FPGA has two images: a default image, and a failsafe image. Alpha Data factory-programs a known-good "null bitstream" into the failsafe image. When power is applied to a card, the firmware on the card first looks for a valid bitstream in the default image. If no bitstream is found, the firmware uses the null bitstream in the failsafe image to configure the target FPGA. In this way, the firmware ensures that the target FPGA is always configured with something when it is powered-on.

Because the purpose of the failsafe image is to protect the target FPGA from sub-micron effects that would otherwise degrade the performance of the target FPGA over time, Alpha Data recommends that the failsafe image should never be erased. If overwritten, a customer must ensure that the bitstream is valid, known-good and satisfies the requirements for protecting the target FPGA from sub-micron effects.

web http://www.xilinx.com/support/answers/35055.htm Xilinx answer record 35055 elaborates on protecting Virtex-6 GTX transceivers from performance degradation over time.

4.5 INFO utility (VxWorks)

Invocation in VxWorks shell

```
admxrc3Info <index>, <flags>
```

where

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index **FLAG_SHOWFLASHINFO** (0x10) => show Flash bank information. **FLAG_SHOWMODULEINFO** (0x20) => show I/O module information. **FLAG_SHOWSENSORINFO** (0x40) => show sensor information.

Summary

Displays information about a reconfigurable computing device.

Description

The **INFO** utility demonstrates the use of most of the informational functions in the ADMXRC3 API. It uses ADMXRC3_OpenEx to open a device in passive mode, meaning that an unprivileged user can successfully run it. The output consists of several sections, the first of which is obtained using ADMXRC3_GetVersionInfo:

```
API information
API library version           1.1.2
Driver version                1.1.2
```

The second section shows information obtained using ADMXRC3_GetCardInfoEx, and shows the information in the ADMXRC3_CARD_INFOEX structure:

```
Card information
Model                         ADM-XRC-6TL
Serial number                 106(0x6A)
Number of programmable clocks 1
Number of DMA channels        2
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             10
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF
```

The third section uses the **NumTargetFpga** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetFpgaInfo to enumerate the target FPGAs in the device:

```
Target FPGA information
FPGA 0                        xc6vlx365tff1759-2C stepping ES
```

The fourth section uses the **NumMemoryBank** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetBankInfo to enumerate the memory banks (non-Flash) in the device:

```
Memory bank information
Bank 0                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
```

```
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
```

The fourth section uses the **NumWindow** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetWindowInfo to enumerate the memory access windows in the device:

```
Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF5800000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0xFB400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base     0xFB2FF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis  Bus base     0xFB2FE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

The next section appears if the **FLAG_SHOWFLASHINFO** (0x10) flag is used. It uses the **NumFlashBank** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetFlashInfo to enumerate the Flash memory banks in the device:

```
Flash bank information
Bank 0                        Intel 28F256P30, 65536(0x10000) kiB
                              Useable area 0x1200000-0x3FFFFFF
```

The next section appears if the **FLAG_SHOWMODULEINFO** (0x20) flag is used. It uses the **NumModuleSite** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetModuleInfo to enumerate the I/O module sites in the device and show what is fitted, if anything:

```
I/O module information
Module 0                      Not present
```

The final optional section appears if the **FLAG_SHOWSENSORINFO** (0x40) flag is used. It uses the **NumSensor** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetSensorInfo to enumerate the sensors in the device:

```
Sensor information
Sensor 0                      1V supply rail
                              V, double, exponent 0, error 0.0
Sensor 1                      1.5V supply rail
                              V, double, exponent 0, error 0.0
Sensor 2                      1.8V supply rail
                              V, double, exponent 0, error 0.0
Sensor 3                      2.5V supply rail
                              V, double, exponent 0, error 0.1
Sensor 4                      3.3V supply rail
                              V, double, exponent 0, error 0.1
Sensor 5                      5V supply rail
                              V, double, exponent 0, error 0.1
Sensor 6                      XMC variable power rail
```

```
                                V, double, exponent 0, error 0.2
    Sensor 7                    XRM I/O voltage
                                V, double, exponent 0, error 0.1
    Sensor 8                    LM87 internal temperature
                                deg. C, double, exponent 0, error 3.0
    Sensor 9                    Target FPGA temperature
                                deg. C, double, exponent 0, error 4.0
```

4.6 ITEST example (VxWorks)

Invocation in VxWorks shell

```
admxrc3ITest <index>, <flags>
```

where

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index **FLAG_USEUBER** (0x10) => use UBER bitstream instead of ITest bitstream.

Summary

Demonstrates consumption of FPGA interrupt notifications.

Description

This example demonstrates how to consume FPGA interrupt notifications in an application. It uses the interrupt register test block of the Uber example FPGA design, described in Section 5.9.5.4.2.5 as a means of generating FPGA interrupt notifications, and starts a thread whose purpose is to wait for and acknowledge interrupts from the target FPGA.

When ITEST is started, the main thread first configures target FPGA 0 with the bitstream (.bit file) for the Uber example FPGA design. The main thread then launches an interrupt thread that waits for notifications, in a loop. The main thread then proceeds to wait for input, also in a loop. At this point, the user may press RETURN to generate an interrupt, or enter 'q' to terminate the program. On termination, the program displays the number of FPGA interrupt notifications that the interrupt thread consumed during execution.

A sample session looks like this:

```
Enter 'q' to quit, or anything else to generate an interrupt:
Interrupt thread started

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:
q
Generated 5 interrupts
Interrupt thread saw 5 interrupt(s)
```

The blank lines in the above session are simply empty lines where the user has pressed return. As can be seen, each of the 5 interrupts generated results in the interrupt thread consuming a notification.

Remarks

As noted in the ADMXRC3 API Specification (see functions ADMXRC3_RegisterWin32Event,

ADMXRC3_RegisterVxwSem and ADMXRC3_StartNotificationWait), the ADMXRC3 API does not queue each type of notification. Therefore, this example works as expected as long as the frequency of target FPGA interrupt notifications is not too fast for the interrupt thread. Since the rate of generation of notifications in this example is limited the user's keyboard input rate, the interrupt thread should be able to keep up (as long as the machine is not heavily loaded with other processes). Nevertheless, it is important to note that in this simple example, there is no mechanism for throttling the rate of notifications so that notifications cannot be lost. In a real application, the preferred design approaches are:

1. Architect the FPGA design and host application so that they tolerate *out-of-date* notifications being missed. For example, if the target FPGA generates an interrupt when data arrives via an I/O interface, it does not matter if the host application does not succeed in consuming every target FPGA interrupt notification, because the notifications before the latest one are considered out-of-date. When the host application handles a notification, it reads a register in the target FPGA to determine the amount of new data rather than using the number of notifications consumed. What matters is that regardless of how many times the target FPGA generates an interrupt, the host application is guaranteed to eventually wake up and check for new data.
2. Use a fully handshaked system, where the host application must positively acknowledge a target FPGA interrupt before the target FPGA generates a new interrupt.

In fact, the above two approaches are best used together, because minimizing the number of FPGA interrupts minimizes unnecessary context switches in the operating system.

4.7 LOADER utility (VxWorks)

Command line

```
loader <index>, <flags>, <target index>, <"bitstream filename">
```

where the following options are accepted:

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index **FLAG_BINARY** (0x10) => treat <*"bitstream filename"*> as a binary file containing only SelectMap data. **FLAG_NOLINKCHECK** (0x20) => do not verify that communications with the target FPGA have been established before exiting.

Summary

Configures a target FPGA with a .bit file, and then exits.

Description

The **LOADER** utility configures the target FPGA, identified by <*target FPGA index*>, within a reconfigurable computing device with the bitstream file identified by <*bitstream filename*>, and then exits.

By default, **LOADER** expects <*bitstream filename*> to name a .bit file, i.e. a file generated by the **bitgen** tool in the Xilinx ISE toolset, and attempts to parse the file accordingly. However, the **FLAG_BINARY** flag causes **LOADER** to treat <*bitstream filename*> as a file containing raw SelectMap data. Such a file can be obtained in a number of ways, including a user-created program or by using the BITSTRIP utility.

Because the **LOADER** utility uses ADMXRC3_ConfigureFromFile or ADMXRC3_ConfigureFromBuffer, it normally checks that communications with the target FPGA have been established before exiting. The **FLAG_NOLINKCHECK** flag causes this check to be omitted, and is needed for an FPGA design that has no host interface (i.e. no OCP communication) with the host, such as a stand-alone Ethernet design.

4.8 MEMTESTH example (VxWorks)

Invocation in VxWorks shell

```
admxrc3MemTestH <index>, <bankmask>, <bNoDma>, <numRep>, <maxError>
```

where

index	specifies the index of the card to open (default 0).
bankmask	is a bitmask specifying which banks to test (0 => all).
bNoDma	should be nonzero to use CPU-initiated data transfer instead of DMA data transfer during the test; this is relatively slow and may increase runtime to minutes instead of seconds.
numRep	is the number of repetitions of the test to perform, minus 1 (0 => 1 repetition, -1 => for ever).
maxError	is the maximum number of data verification errors to display; note that further errors are still counted and a total is displayed at the end of the test (0 => default of 20).

Summary

Performs a host-driven test of the memory banks on a reconfigurable computing card.

Description

The MEMTESTH example demonstrates the transfer of data between host memory and on-board memory devices (for example, DDR3 SDRAM on the ADM-XRC-6T1), via the target FPGA. A number of test phases are performed, each with a different data generation method, such as alternating an 55 / AA pattern, "own address" etc. In each phase, each bank is tested by first filling the bank with data and then reading it back in order to verify that data transfers are error-free.

This example makes use of the Uber example FPGA design. Assuming no errors are detected, running it produces output of the form:

```
Bank 00: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank 01: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank 02: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank 03: DDR-3 SDRAM, 262144 (0x40000) kiB
Bank test mask is 0x000f
Performing host-driven memory test...
Phase 1 - 0x55 pattern
Phase 2 - 0xAA pattern
Phase 3 - own address pattern
Phase 4 - pseudorandom data
Measuring throughput...
Throughput from host to memory is 439.7 MiB/s
Throughput from memory to host is 1009.6 MiB/s
PASSED
```

4.9 MONITOR utility (VxWorks)

Invocation in VxWorks shell

```
admxrc3Monitor <index>, <flags>, <period>, <numberOfUpdates>
```

where

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index
period	is the update period, in seconds.
numberOfUpdates	specifies the number of updates to perform (default 0); a value of zero means "repeat for ever".

Summary

Displays readings from all sensors.

Description

The **MONITOR** utility repeatedly displays sensor readings in the VxWorks shell at the interval specified by the **period** parameter. The number of updates to perform before terminating is specified by the **number of updates** parameter. If not specified, the default is 0, which means that the example runs for ever.

This utility makes use of the ADMXRC3_GetSensorInfo and ADMXRC3_ReadSensor functions from the ADMXRC3 API, and because it opens a device in passive mode using ADMXRC3_OpenEx, it can run alongside other reconfigurable computing applications without disturbing them.

The output looks like this:

```
Model:                                  257 (0x101) => ADM-XRC-6TL
Serial number:                          101 (0x65)
Number of sensors:                      10
  Sensor  0              1V supply rail: 0.987000 V
  Sensor  1           1.5V supply rail: 1.509186 V
  Sensor  2           1.8V supply rail: 1.803192 V
  Sensor  3           2.5V supply rail: 2.508896 V
  Sensor  4           3.3V supply rail: 3.268082 V
  Sensor  5             5V supply rail: 5.017990 V
  Sensor  6      XMC variable power rail: 12.000000 V
  Sensor  7            XRM I/O voltage: 2.495712 V
  Sensor  8    LM87 internal temperature: 49.000000 deg C
  Sensor  9      Target FPGA temperature: 57.000000 deg C
```

4.10 SIMPLE example (VxWorks)

Invocation in VxWorks shell

```
admxrc3Simple <index>, <flags>
```

where

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index **FLAG_USEUBER** (0x10) => use UBER bitstream instead of SIMPLE bitstream.

Summary

Demonstrates access to target FPGA registers.

Description

The SIMPLE example application demonstrates accessing FPGA registers in its simplest form. It first configures target FPGA 0 with the Simple example FPGA design, or the Uber example FPGA design if the **flags** parameter includes **FLAG_USEUBER** (0x10). It then waits for input from the user. The user enters hexadecimal values (up to 32 bits in length), and for each value:

1. The program writes the value to a register in the target FPGA.
2. The target FPGA nibble-reverses the value and makes the reversed value available to be read via a register. Here, nibble-reversing means that the FPGA swaps bits 31:28 with 3:0, 27:24 with 7:4 etc.
3. The program reads back and displays the nibble-reversed value.

The program terminates on CTRL-D (Linux) or CTRL-Z (Windows). A sample session looks like this:

```
=========================
Enter values for I/O
(CTRL-D / CTRL-Z to exit)
=========================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
deadbeef
OUT = 0xdeadbeef, IN = 0xfeebdaed
cafeface
OUT = 0xcafeface, IN = 0xecafefac
```

4.11 SIMPLEDMA example (VxWorks)

Command line

```
admxrc3SimpleDma <index>, <flags>, <addressLo>, <addressHi>
```

where

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index.
addressLo	is the low 32 bits of the OCP address at which to begin reading data (default 0).
addressHi	is the high 32 bits of the OCP address at which to begin reading data (default 0).

Summary

Basic test program for the SimpleDMA Example FPGA Design.

Description

The SIMPLEDMA example application performs a basic test of the SimpleDMA Example FPGA Design, by reading 256 bytes from a given OCP address on DMA channel 0 and displaying them (as 32-bit words). The default OCP address at which reading begins is 0, but this can be specified on the command line using the **-address** option. As well as reading and displaying some data read from the FPGA, the program also measures and displays the rate at which data can be transferred via DMA channel 0.

In the SimpleDMA Example FPGA Design, reads on DMA channel 0 return a pattern of incrementing 32-bit integers, based on the OCP address of each 32-bit word of data divided by 4. The output of the program, when run with the default parameters, looks like this:

```
Reading using DMA channel 0 at OCP address 0x0...
Dump of data read from OCP address 0x0 + 0x100 B:
                          00       04       08       0c
0x00000000_00000000: 00000000 00000001 00000002 00000003 ................
0x00000000_00000010: 00000004 00000005 00000006 00000007 ................
0x00000000_00000020: 00000008 00000009 0000000a 0000000b ................
0x00000000_00000030: 0000000c 0000000d 0000000e 0000000f ................
0x00000000_00000040: 00000010 00000011 00000012 00000013 ................
0x00000000_00000050: 00000014 00000015 00000016 00000017 ................
0x00000000_00000060: 00000018 00000019 0000001a 0000001b ................
0x00000000_00000070: 0000001c 0000001d 0000001e 0000001f ................
0x00000000_00000080: 00000020 00000021 00000022 00000023  ...!..."...#...
0x00000000_00000090: 00000024 00000025 00000026 00000027 $...%...&...'...
0x00000000_000000a0: 00000028 00000029 0000002a 0000002b (...)...*...+...
0x00000000_000000b0: 0000002c 0000002d 0000002e 0000002f ,...-......./...
0x00000000_000000c0: 00000030 00000031 00000032 00000033 0...1...2...3...
0x00000000_000000d0: 00000034 00000035 00000036 00000037 4...5...6...7...
0x00000000_000000e0: 00000038 00000039 0000003a 0000003b 8...9...:...;...
0x00000000_000000f0: 0000003c 0000003d 0000003e 0000003f <...=...>...?...
Measuring throughput...
Throughput from host to FPGA is 703.8 MiB/s
Throughput from FPGA to host is 751.0 MiB/s
PASSED
```

4.12 VPD utility (VxWorks)

Invocation in VxWorks shell

```
admxrc3Vpd <index>, <flags>, "fb", <address>, <n>, "num-arg"
admxrc3Vpd <index>, <flags>, "fw", <address>, <n>, "num-arg"
admxrc3Vpd <index>, <flags>, "fd", <address>, <n>, "num-arg"
admxrc3Vpd <index>, <flags>, "fq", <address>, <n>, "num-arg"
admxrc3Vpd <index>, <flags>, "fs", <address>, <n>, "str-arg"
admxrc3Vpd <index>, <flags>, "rb", <address>, <n>
admxrc3Vpd <index>, <flags>, "rw", <address>, <n>
admxrc3Vpd <index>, <flags>, "rd", <address>, <n>
admxrc3Vpd <index>, <flags>, "rq", <address>, <n>
admxrc3Vpd <index>, <flags>, "wb", <address>, <n>[, "num-arg"]
admxrc3Vpd <index>, <flags>, "ww", <address>, <n>[, "num-arg"]
admxrc3Vpd <index>, <flags>, "wd", <address>, <n>[, "num-arg"]
admxrc3Vpd <index>, <flags>, "wq", <address>, <n>[, "num-arg"]
admxrc3Vpd <index>, <flags>, "ws", <address>, <n>[, "str-arg"]
```

where

index	is normally the index of the reconfigurable computing device to use (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains **FLAG_BYSERIAL**.
flags	is the bitwise OR of zero or more of the following flags (default 0): **FLAG_BYSERIAL** (0x1) => *index* is interpreted as a serial number rather than a device index. **FLAG_HEX** (0x10) => causes the utility to interpret all numeric data values as hexadecimal.
address	is the address in VPD memory at which to begin reading or writing.
n	is the number of bytes to read or write.
"num-arg"	is a string containing a numeric data argument; required for the **fb**, **fw**, **fd** & **fq** commands and optional for the **wb**, **ww**, **wd** & **wq** commands.
"str-arg"	is a string argument; required for the **fs** command and optional for the **ws** command.

Summary

Displays data read from VPD memory, or writes data to VPD memory.

Description

The **VPD** utility operates in one of three modes:

- Filling a region of VPD memory with a value or string; for this mode, use the **fb**, **fw**, **fd**, **fq** or **fs** commands.
- Reading data from VPD memory and displaying it; for this mode, use the **rb**, **rw**, **rd** or **rq** commands.
- Writing numeric or string data to a region of VPD memory; for this mode, use the **wb**, **ww**, **wd**, **wq** or **ws** commands.

Fill mode

When filling a region of VPD memory with data, the fill command specifies whether the data is numeric or string data. In the case of numeric data, the command also implies the radix (i.e. word size) of the data. The available fill commands are:

- **fb**

Fill value is a byte (8-bit).

- **fw**

  Fill value is a word (16-bit).

- **fd**

  Fill value is a doubleword (32-bit).

- **fq**

  Fill value is a quadword (64-bit).

- **fs**

  Fill value is an ASCII string (8-bit characters).

The next 3 arguments after the fill command must be:

- *address* - the byte address within VPD memory at which to begin filling.
- *n* - byte count; the number of bytes of VPD memory to fill.
- *data* or *string* - the numeric or string value to place in the specified region of VPD memory.

If the command is **fs** and the string value is shorter than the byte count *n*, the string is repeated until the byte count is satisfied. If the string is longer than the byte count *n*, only the first *n* characters are used. If a string contains spaces, it must be quoted on the command line so that it is not interpreted by the shell as two or more separate arguments.

For the numeric fill commands **fb**, **fw**, **fd** and **fq**, the numeric value is repeated until the byte count is satisfied.

Read mode

The read command implies the radix (i.e. word size) used for displaying the data:

- **rb**

  Byte (8-bit) reads; data is displayed as bytes.

- **rw**

  Word (16-bit) reads; data is displayed as words.

- **rd**

  Doubleword (32-bit) reads; data is displayed as doublewords.

- **rq**

  Quadword (64-bit) reads; data is displayed as quadwords.

After the read command, an address must be supplied, which specifies where in VPD memory to begin reading. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the read command. If present, the length parameter specifies how many bytes to read and display. The length should be an integer multiple of the width; if not, the length is rounded down.

Write mode

The write command specifies whether the data is numeric or string data. In the case of numeric data, the command also implies the radix (i.e. word size) of the data. The available write commands are:

- **wb**

  Data is written as bytes (8-bit).

- **ww**

  Data is written as words (16-bit).

- **wd**

  Data is written as doublewords (32-bit).

- **wq**

  Data is written as quadwords (64-bit).

- **ws**

  Data is supplied as one or more ASCII strings (8-bit characters).

After the write command, an address must be supplied, which specifies where in VPD memory to begin writing data. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the write command. If present, the length parameter specifies how many bytes to write. The length should be an integer multiple of the width; if not, the length is rounded down.

The program obtains the values to be written in two ways: from any additional parameters on the command line after the length parameter, and then from the standard input stream (stdin). This works as follows:

1. Any remaining command line arguments, if present after the length parameter, are interpreted as data values to be written. Numeric values are assumed to be of the radix implied by the command parameter. As each value it written to VPD memory, the address is incremented. If there are enough values passed on the command line to satisfy the byte count, the program terminates.
2. If there are insufficient data values passed on the command line, the program waits for values to be entered on the standard input stream. Numeric values entered this way are also assumed to be of the radix implied by the command. As each value it written to VPD memory, the address is incremented. When the entire byte count that was specified in the length parameter has been satisfied or end-of-file is encountered, the program terminates.

Example session

The following session was captured using an ADM-XRC-6TL. The base address 0x100000 is used because that is the VPD-space address of the user-definable area of VPD memory in the ADM-XRC-6TL.

```
-> admxrc3Vpd 0,0,"rb",0x100000,0x60
Dump of VPD at 0x100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x00100000: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100010: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100020: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100030: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100050: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
value = 0 = 0x0
-> admxrc3Vpd 0,0,"fs",0x100008,20,"hello world!"
value = 0 = 0x0
-> admxrc3Vpd 0,0,"wd",0x100020,12
0x00100020: 0xdeadbeef
0x00100024: 0xcafeface
0x00100028: 0x12345678
value = 0 = 0x0
-> admxrc3Vpd 0,0,"fw",0x100031,10,"0xa55a"
value = 0 = 0x0
-> admxrc3Vpd 0,0,"rb",0x100000,0x60
Dump of VPD at 0x100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x00100000: ff ff ff ff ff ff ff ff 68 65 6c 6c 6f 20 77 6f ........hello wo
0x00100010: 72 6c 64 21 68 65 6c 6c 6f 20 77 6f ff ff ff ff rld!hello wo....
```

```
0x00100020: ef be ad de ce fa fe ca 78 56 34 12 ff ff ff ff ........xV4.....
0x00100030: ff 5a a5 5a a5 5a a5 5a a5 5a ff ff ff ff ff .Z.Z.Z.Z.Z......
0x00100040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100050: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
value = 0 = 0x0
```

NOTE: the above sesssion assumes that VPD write protection has been disabled as described in the release notes for the ADB3 Driver for VxWorks.

Remarks

When entering data for fill or write commands, values are expressed in decimal by default. To express data as hexadecimal, prefix it with '0x' or use the **FLAG_HEX** (0x10) flag.

In the current version of the **VPD** utility, data is always read from and written to VPD memory in little-endian byte order.

5 Example HDL FPGA Designs

5.1 Introduction

A number of example FPGA designs are included with the SDK. The purpose of these is to demonstrate functionality available on Alpha Data's range of reconfigurable computing hardware, and also to serve as customisable starting points for user-developed designs. A testbench and simulation/build scripts are also included with each example design.

The example applications use these example designs to demonstrate how software running on the host CPU can interact with an FPGA design.

The table below lists the example FPGA designs and their related applications:

ITest	Demonstrates generation of host interrupts by the target FPGA. The **ITEST example application** (Windows and Linux / VxWorks) uses this FPGA design's bitstreams by default.
Simple	Minimal design that demonstrates implementation of host-accessible registers. The **SIMPLE example application** (Windows and Linux / VxWorks) uses this FPGA design's bitstreams by default.
SimpleDMA	Minimal design that demonstrates how to sink or source data on an OCP DMA channel. The **SIMPLEDMA example application** (Windows and Linux / VxWorks) uses this FPGA design's bitstreams.
Uber	Demonstrates implementation of host-accessible registers. The **SIMPLE example application** (Windows and Linux / VxWorks) uses this FPGA design's bitstreams when the **+uber** option is passed on the command line.
	Demonstrates generation of host interrupts by the target FPGA. The **ITEST example application** (Windows and Linux / VxWorks) uses this FPGA design's bitstreams when the **+uber** option is passed on the command line.
	Demonstrates interfaces to on-board memory such as DDR3 SDRAM. The **MEMTESTH example application** (Windows and Linux / VxWorks) uses this design.

Table 3 : Example HDL FPGA Designs

These example designs are located in the **hdl/vhdl/examples/** directory.

5.2 Supported Xilinx ISE Versions

Due to differences in net naming, netlist formats, and various issues in the individual tools of the Xilinx ISE Develpment System, this release of the SDK is best used together with particular versions of Xilinx ISE. This section summarizes the recommended Xilinx ISE versions and any patches that should be applied for best results. The Xilinx ISE version to use depends upon the FPGA family being targetted, as detailed in the following subsections.

It is possible to use versions of Xilinx ISE other than what are recommended below, but Alpha Data cannot guarantee that there will not be additional issues to resolve when using those other versions of Xilinx ISE with this release of the SDK.

5.2.1 Virtex-6 FPGA Development

For developing Virtex-6 FPGA designs, Alpha Data recommends the following Xilinx ISE versions:

- Xilinx ISE 13.2, 13.3 or 13.4

  Note: In PlanAhead 13.x, **BITGEN** options cannot be made "sticky" for a project, and must be re-applied every time a project is loaded in the PlanAhead GUI. This issue is discussed further in Section 5.3.1.

- Xilinx ISE 14.1 or 14.2

5.2.2 Kintex-7 Initial Engineering Silicon FPGA Development

In this release of the SDK, for Kintex-7 FPGAs, only Initial Engineering Silicon (IES) is supported. These devices are marked with the SCD code "ES9937". Xilinx ISE 13.4 is recommended, with the following patches:

- Kintex-7 325T IES **BITGEN** patch for ISE 13.4 (Xilinx Answer 43060). This patch is applied to the Xilinx ISE 13.4 tools.
- Kintex-7 325T IES patch for MIG7 v1.4 (Xilinx Answer 43372). This patch consists of mapping some generics when instantiating an instance of the MIG7 v1.4 DDR3 SDRAM interface component. Alpha Data has already applied this to the MIG7 v1.4 DDR3 SDRAM interface wrapper supplied in this SDK, so no user action is required.
- MIG 7 Series v1.4 DDR2/DDR3 Callibration Update patch (Xilinx Answer 45653). This patch consists of a set of Verilog files that must be copied into the MIG7 v1.4 HDL source folder for the ADM-XRC-7K1 **AFTER** generating the core, overwriting whatever files were generated. This is described further in the file **hdl/vhdl/common/ddr3_sdram_if/admxrc7k1/mig7_v1_4/tactical_patch_ar45653.txt**.

Xilinx Answer 43347 is the master record of known issues for Kintex-7 IES, and Xilinx Errata Notification 171 (EN171) describes errata specific to 7K325T IES devices.

Note: In PlanAhead 13.4, **BITGEN** options cannot be made "sticky" for a project, and must be re-applied every time a project is loaded in the PlanAhead GUI. This issue is discussed further in Section 5.3.1.

5.2.3 Virtex-7 Initial Engineering Silicon FPGA Development

In this release of the SDK, for Virtex-7 FPGAs, only Initial Engineering Silicon (IES) is supported. These devices are marked with the SCD code "ES9937". Xilinx ISE 13.4 is recommended, with the following patches:

- MIG 7 Series v1.4 DDR2/DDR3 Callibration Update patch (Xilinx Answer 45653). This patch consists of a set of Verilog files that must be copied into the MIG7 v1.4 HDL source folder for the ADM-XRC-7V1 **AFTER** generating the core, overwriting whatever files were generated. This is described further in the file **hdl/vhdl/common/ddr3_sdram_if/admxrc7v1/mig7_v1_4/tactical_patch_ar45653.txt**.

Xilinx Answer 43423 is the master record of known issues for Virtex-7 IES, and Xilinx Errata Notification 172 (EN172) describes errata specific to 7VX485T IES devices.

Note: In PlanAhead 13.4, **BITGEN** options cannot be made "sticky" for a project, and must be re-applied every time a project is loaded in the PlanAhead GUI. This issue is discussed further in Section 5.3.1.

5.3 Generating PlanAhead Projects

Note
Xilinx ISE version 13.4 or 14.2 is recommended if **PlanAhead** is to be used with this SDK. Refer to Section 5.2 - "Supported Xilinx ISE Versions" for information about recommended Xilinx ISE versions for this release of the SDK.

The SDK provides a script that generates projects for the Xilinx ISE **PlanAhead** tool. In order to use **PlanAhead** to generate bitstreams for the example FPGA designs or to simulate the example FPGA designs, **PlanAhead** projects for the ***<design>-<model>-<device>*** combination(s) of interest must first be generated. The TCL script **genppr.tcl**, located in **hdl/vhdl/examples/**, generates PlanAhead projects and is invoked as follows:

```
xtclsh genppr.tcl [-f] [pattern]
```

The **genppr.tcl** script's default behavior is to avoid overwriting existing **PlanAhead** projects, but the **-f** option causes the script to overwrite existing **PlanAhead** projects with new ones.

The **pattern** argument is optional, but if specified, acts as a filter that can be used to generate projects for a subset of the available ***<design>-<model>-<device>*** combination(s).

- When a pattern is specified, it is interpreted as a POSIX-style regular expression that is matched against every supported ***<design>-<model>-<device>*** combination for all of the example FPGA designs in the SDK.

  Note: when invoking **genppr.tcl** in a UNIX-like shell such as **bash**, it is advisable to enclose the pattern in single quotes so that the shell cannot perform globing.

- When **pattern** is omitted, **genppr.tcl** generates **PlanAhead** projects for every supported ***<design>-<model>-<device>*** combination for all of the example FPGA designs in the SDK. This is equivalent to specifying a pattern of **.+** (match one or more characters).

Some examples follow:

- `xtclsh genppr.tcl simple-admxrc6t1-.+`

  Generates **PlanAhead** projects for all possible devices for the Simple example FPGA design, targetting the ADM-XRC-6T1.

- `xtclsh genppr.tcl .+-admxrc6t1-6vsx315t`

  Generates **PlanAhead** projects for all example FPGA designs for the 6VSX315T device, targetting the ADM-XRC-6T1.

- `xtclsh genppr.tcl -f '.+-admxrc6t1-.+'`

  Generates **PlanAhead** projects for all example FPGA designs targetting the ADM-XRC-6T1. Since the **-f** option is specified, the generated **PlanAhead** projects will overwrite any existing projects. Enclosing the pattern by single quotes, as in this example, is recommended when running the command in UNIX-style shell such as **bash**. This prevents the shell from performing unwanted *globing* for patterns such as **.***.

5.3.1 BITGEN Options in PlanAhead 13.x

In **PlanAhead 13.x**, it is not possible to save **BITGEN** options for a project. Each time a project is opened in the **PlanAhead 13.x** GUI, when a bitstream must be generated, the following must be added to the "More Options" field of the "Generate Bitstream" dialog box that appears when "Program and Debug" -> "Generate Bitstream..." is selected: **-g drivedone:yes -g unusedpin:pullnone -g compress**

These options have the following effects:

- **-g drivedone:yes**

  The FPGA drives the **DONE** pin high when configuration is complete, as opposed to tristating it.

- **-g unusedpin:pullnone**

  Unused FPGA pins are tristated, instead of being pulled high or low.

- **-g compress**

Bitstream compression is enabled, which reduces FPGA configuration time.

It is not necessary to manually set **BITGEN** options when the **genppr.tcl** script is used to generate projects for **PlanAhead 14.x**. The **genppr.tcl** script detailed in Section 5.3 - "Generating PlanAhead Projects" detects that it is running under **PlanAhead 14.x** and automatically sets the required **BITGEN** options for a project, so no user action is required.

5.3.2 Extra BITGEN options in PlanAhead 13.x for the ADM-XRC-6T-ADV8

As well as the **BITGEN** options in Section 5.3.1 - "BITGEN Options in PlanAhead 13.x", the following additional options are recommended when setting **BITGEN** options in **PlanAhead 13.x** for a design that targets the ADM-XRC-6T-ADV8 model: **-g persist:no -g StartupClk:cclk -g BPI_1st_read_cycle:4 -g BPI_page_size:8 -g ConfigRate:26**

These options have the following effects:

- **-g persist:no**

  The SelectMap interface does not persist after configuration of the FPGA.

- **-g StartupClk:cclk**

  The clock used to sequence startup of the FPGA (as the final stage of configuration) is the configuration clock (**CCLK**) pin.

- **-g BPI_1st_read_cycle:4 -g BPI_page_size:8**

  Enables page mode reading of the Flash device that stores the bitstream for the FPGA.

- **-g ConfigRate:26**

  Sets the configuration clock frequency to the maximum permitted while remaining within specifications.

It is not necessary to manually set extra **BITGEN** options when the **genppr.tcl** script is used to generate projects for **PlanAhead 14.x**. The **genppr.tcl** script detailed in Section 5.3 - "Generating PlanAhead Projects" detects that it is running under **PlanAhead 14.x** and automatically sets the required **BITGEN** options for a project, so no user action is required.

5.4 Bitstream Build Using Xilinx ISE

Note: Virtex-6 Models
Xilinx ISE version 13.4 or 14.2 is recommended for rebuilding the Virtex-6 bitstreams for the example FPGA designs in this SDK. Refer to Section 5.2.1 - "Virtex-6 FPGA Development" for information about recommended Xilinx ISE versions for this release of the SDK.

Note: Kintex-7 Series Models
Xilinx ISE version 13.4 is recommended for rebuilding the Kintex-7 bitstreams for the example FPGA designs in this SDK. Refer to Section 5.2.2 - "Kintex-7 Initial Engineering Silicon FPGA Development" for information about recommended Xilinx ISE versions for this release of the SDK.

Note: Virtex-7 Series Models
Xilinx ISE version 13.4 is recommended for rebuilding the Virtex-7 bitstreams for the example FPGA designs in this SDK. Refer to Section 5.2.3 - "Virtex-7 Initial Engineering Silicon FPGA Development" for information about recommended Xilinx ISE versions for this release of the SDK.

Bitstreams for all supported combinations of example FPGA design, model, and device are supplied pre-built in the **bit/** directory of the SDK. This directory is the HDL equivalent of the **bin/** directory for the example C/C++ applications. The source files required to re-build all bitstreams are supplied in the **hdl/** directory.

Bitstreams may be re-built using a Make build tool in the command line environment, as described in the following subsection, or by using the Xilinx **PlanAhead** tool with automatcally generated projects (see Section 5.4.2 - "Building Design Bitstreams Using PlanAhead".

5.4.1 Building Example Bitstreams Using Make

Bitstream build in the Windows command line environment uses the Microsoft Visual Studio **NMAKE** tool. Bitstream build in the Linux command line environment uses **GNU Make** tool. Makefiles compatible with **NMAKE** and **GNU Make** are provided for building bitstreams for each example FPGA design. Refer to the following sections for further details about a particular example FPGA design:

- Build ITest example FPGA design using NMAKE or GNU Make
- Build Simple example FPGA design using NMAKE or GNU Make
- Build SimpleDMA example FPGA design using NMAKE or GNU Make
- Build Uber example FPGA design using NMAKE or GNU Make

5.4.2 Building Design Bitstreams Using PlanAhead

Note: Virtex-6 Models
Xilinx ISE version 13.4 or 14.2 is recommended for rebuilding the Virtex-6 bitstreams for the example FPGA designs in this SDK. Refer to Section 5.2.1 - "Virtex-6 FPGA Development" for information about recommended Xilinx ISE versions for this release of the SDK.

Note: Kintex-7 Series Models
Xilinx ISE version 13.4 is recommended for rebuilding the Kintex-7 bitstreams for the example FPGA designs in this SDK. Refer to Section 5.2.2 - "Kintex-7 Initial Engineering Silicon FPGA Development" for information about recommended Xilinx ISE versions for this release of the SDK.

Note: Virtex-7 Series Models
Xilinx ISE version 13.4 is recommended for rebuilding the Virtex-7 bitstreams for the example FPGA designs in this SDK. Refer to Section 5.2.3 - "Virtex-7 Initial Engineering Silicon FPGA Development" for information about recommended Xilinx ISE versions for this release of the SDK.

Note: PlanAhead 13.x
In **PlanAhead** 13.x, bitstream generation options are not "sticky", and must be applied each time a project is opened. For this reason, it is more convenient to use **PlanAhead 14.x**. See Section 5.3.1 - "BITGEN Options in PlanAhead 13.x" for details about this issue.

Xilinx **PlanAhead** may be used to generate a bitstream for an example FPGA design. First, generate **PlanAhead** project(s) as described in Section 5.3 - "Generating PlanAhead Projects". After opening a project in **PlanAhead**, a bitstream can then be generated in the normal way using the **PlanAhead** GUI.

For details of generating a bitstream for particular example FPGA design using **PlanAhead**, refer to the following sections:

- Build ITest example FPGA design using PlanAhead
- Build Simple example FPGA design using PlanAhead
- Build SimpleDMA example FPGA design using PlanAhead
- Build Uber example FPGA design using PlanAhead

5.5 Design Simulation

Note
VHDL source code is compiled for simulation using IEEE-STD-1076-1993 (**ModelSim**) and IEEE-STD-1076-2000 (**ISIM**).

Two types of simulation are currently available, termed "Full MPTL" and "OCP-only". Selection between the two is achieved by the use of different variants of the package adb3_target_inc_pkg and MPTL interface.

5.5.1 Full MPTL Simulation

This simulates the actual MPTL interface core between the Bridge and Target FPGAs as follows:

- OCP transactions are converted to MPTL data by the example design testbench MPTL interface.
- The example design testbench MPTL interface is connected to the example FPGA design MPTL interface.
- The example FPGA design MPTL interface converts MPTL data back to OCP transactions.

HDL source files are used to simulate the example testbench and example FPGA designs. HDL netlists are used to simulate the MPTL interface.

Advantages

- Simulates the actual MPTL interface core.

Disadvantages

- Requires full initialisation period before MPTL interface is available for OCP transactions.
- Runs more slowly than OCP-only simulation.

In most cases this level of simulation detail is not required and the OCP-only simulation should be used.

5.5.2 OCP-Only Simulation

This replaces the MPTL interface core between the Bridge and Target FPGAs with a direct OCP connection as follows:

- OCP transactions are transferred to a simulation version of the example design testbench MPTL interface.
- The example design testbench simulation MPTL interface is connected to the example FPGA design simulation MPTL interface.
- The example FPGA design simulation MPTL interface transfers the OCP transactions.

HDL source files are used to simulate the example testbench and example FPGA designs. OCP-only simulation HDL source files are used to simulate the MPTL interface.

Advantages

- Requires minimal initialisation period before MPTL interface is available for OCP transactions.
- Runs more quickly than full MPTL simulation.

Disadvantages

- Does not simulate the actual MPTL interface core.

In most cases this type of simulation should be used.

5.5.3 Simulation Using ModelSim

Scripts compatible with **ModelSim** are provided for simulation of each example FPGA design. Refer to the following sections for further details:

- ITest, Simulation Using ModelSim
- Simple, Simulation Using ModelSim
- SimpleDMA, Simulation Using ModelSim
- Uber, Simulation Using ModelSim

5.5.4 Simulation Using PlanAhead

> **Note**
> Xilinx ISE version 13.4 or 14.2 is recommended for simulation using PlanAhead. Refer to Section 5.2 - "Supported Xilinx ISE Versions" for information about recommended Xilinx ISE versions for this release of the SDK.

The Xilinx **PlanAhead** tool may be used to initiate **ISIM** simulation of each example FPGA design. Before performing an **ISIM** simulation, first generate **PlanAhead** projects as described in Section 5.3 - "Generating PlanAhead Projects". After opening a project in **PlanAhead**, a simulation may then be performed in the normal way using the **PlanAhead** GUI.

For details of simulating a particular design using **PlanAhead** / **ISIM**, refer to the following sections:

- ITest, Simulation Using PlanAhead
- Simple, Simulation Using PlanAhead
- SimpleDMA, Simulation Using PlanAhead
- Uber, Simulation Using PlanAhead

5.6 ITest Example FPGA Design

5.6.1 Model Support

The **ITest** FPGA design is compatible with all Virtex-6 and 7 Series models.

5.6.2 File Location

The **ITest** FPGA design is located in **hdl/vhdl/examples/itest/**. Files common to all models are located in the **hdl/vhdl/examples/itest/common/** directory. Files specific to a model are located in the **hdl/vhdl/examples/itest/*<model>*/** directory.

5.6.3 Generating PlanAhead Projects

PlanAhead project files (file extension **.ppr**) for the **ITest** FPGA design are generated using the **genppr.tcl** TCL script that can be found in **hdl/vhdl/examples/**. Generating **PlanAhead** projects is described in general in Section 5.3 - "Generating PlanAhead Projects". To generate **PlanAhead** projects for the **ITest** design, the **pattern** passed to the **genppr.tcl** script should be prefixed by **itest-**. Some examples:

1. To generate a **PlanAhead** project in Windows for an ADM-XRC-6T1 fitted with a 6VLX240T device, for the **ITest** example design, start a shell and run the settings32.bat or settings64.bat script from the ISE tools (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl itest-admxrc6t1-6vlx240t
   ```

   Under Linux, start a shell, source the settings32.sh or settings64.sh script (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'itest-admxrc6t1-6vlx240t'
   ```

2. To generate **PlanAhead** projects in Windows for an ADM-XRC-6T1 for all supported devices, use these commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl itest-admxrc6t1-.+
   ```

   Under Linux, use these commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'itest-admxrc6t1-.+'
   ```

Note: the single quotes around the patterns in the Linux examples prevent the shell from attempting to perform globing on the pattern.

PlanAhead project files generated using **genppr.tcl** are created in the example design's **planahead** directory. For example, **hdl/vhdl/examples/itest/planahead/** contains all of the **PlanAhead** projects for the **ITest** example design. Building a design or running a simulation using **ISIM** is performed in the normal way via the **PlanAhead** GUI.

5.6.4 Design Synthesis and Bitstream Build

5.6.4.1 VHDL Source Files

For a complete list of the source files used during synthesis, refer to the appropriate XST project file located in the model design directory; for example, **hdl/vhdl/examples/itest/admxrc6t1/itest-admxrc6t1.prj** for an ADM-XRC-6T1.

5.6.4.2 XST Files

XST Project files (**.prj**) are located in the model design directory; for example, **hdl/vhdl/examples/itest/admxrc6t1/itest-admxrc6t1.prj** for an ADM-XRC-6T1.

XST Script files (**.scr**) are located in the model design directory; for example, **hdl/vhdl/examples/itest/admxrc6t1/itest-admxrc6t1-6vlx240t.scr** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

XST constraint files (**.xcf**) are located in the model design directory; for example, **hdl/vhdl/examples/itest/admxrc6t1/itest-admxrc6t1.xcf** for an ADM-XRC-6T1.

5.6.4.3 Implementation Constraint Files

Implementation constraint files (**.ucf**) are located in the model design directory; for example, **hdl/vhdl/examples/itest/admxrc6t1/itest-admxrc6t1.ucf** for the ADM-XRC-6T1.

5.6.4.4 Build Using Make

Makefiles are provided for building **ITest** design bitstreams (**.bit** files) from the command line. Depending on the target passed to **NMAKE** (Windows), or **GNU Make** (Linux), bitstreams can be built for a specific model-device combination, or for all supported model-device combinations.

When a **.bit** file is built, it is not automatically used by the example applications unless it is copied into the **bit/itest/** directory. This can be done manually, or by using the Makefile.

The Makefile can also be used to delete **.bit** files and intermediate files. This guarantees that the next time the design is built, it is from VHDL sources as opposed to beginning at some intermediate step.

The Makefile for the **ITest** design has the following targets:

Target	Class	Effect
all	build	Builds all **.bit** files for all supported model and device combinations.
bit_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices.
bit_*<model>*_*<device>*		Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>***.
install	install	Builds and installs all **.bit** files for all supported model and device combinations in the directory **bit/itest/**.
inst_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices and copies them to the directory **bit/itest/**.

Table 4 : ITest Design Makefile Targets **(continued on next page)**

Target	Class	Effect
inst_*<model>*_*<device>*	install	Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>*** and copies it to the directory **bit/itest/**.
clean	clean	Deletes all **.bit** files and intermediate build files for all supported model and device combinations (but does not delete any files from **bit/itest/**).
clean_*<model>*		Deletes **.bit** files and intermediate build files for the model specified by ***<model>*** for all supported devices (but does not delete any files from **bit/itest/**).
clean_*<model>*_*<device>*		Deletes the **.bit** file and intermediate build files for the model specified by ***<model>*** with a device specified by ***<device>*** (but does not delete any files from **bit/itest/**).

Table 4 : ITest Design Makefile Targets

Files that result from the build process, including **.bit** files, are placed in:

hdl/vhdl/examples/itest/build/*<model>*-*<device>*/

Filenames of any bitstreams built are thus of the form:

hdl/vhdl/examples/itest/build/*<model>*-*<device>*/itest-*<model>*-*<device>*.bit.

When a target of class "clean" is executed, the **build/*<model>*-*<device>*** directory is deleted, but files in **bit/itest/** are unaffected.

Note
Before a bitstream can be used by one of the example applications, it must be copied to **bit/itest/** by executing a target of class "install", or by manually copying the **.bit** file.

Some example make commands follow:

1 To perform a build of all **ITest** design bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\itest
nmake all
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/itest
make all
```

2 To perform a build and install the resulting bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\itest
nmake install
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/itest
make install
```

3 To perform a build for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\itest
nmake bit_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/itest
make bit_admxrc6t1_6vlx240t
```

4 To perform a build and install for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\itest
nmake inst_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/itest
make inst_admxrc6t1_6vlx240t
```

5 To delete all **.bit** files and intermediate build files in Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\itest
nmake clean
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/itest
make clean
```

6 To delete the **.bit** file and intermediate build files for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\itest
nmake clean_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/itest
make clean_admxrc6t1_6vlx240t
```

5.6.4.5 Build Using PlanAhead

In order to build the **ITest** design using **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.6.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to implement the design and generate a bitstream in the normal way.

Note
In **PlanAhead** 13.x, **bitgen** options are not sticky, and must be applied each time a project is opened. See Section 5.3.1 - "BITGEN Options in PlanAhead 13.x" for details about this issue. This step is **not** required for

5.6.5 Design Description

The **ITest** example FPGA design demonstrates generation of host interrupts by the target FPGA in Gen 3 Alpha Data reconfigurable computing hardware such as the ADM-XRC-6T1.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 5 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/itest/common/
ADM-XRC-6TL	MPTL	itest.vhd
ADM-XRC-6T1	MPTL	itest.vhd
ADM-XRC-6TGE	MPTL	itest.vhd
ADM-XRC-6T-ADV8	PCIe	itest_pcie.vhd
ADM-XRC-6T-DA1	MPTL	itest.vhd
ADPE-XRC-6T	MPTL	itest.vhd
ADPE-XRC-6T-L	MPTL	itest.vhd
ADM-XRC-7K1	MPTL	itest.vhd
ADM-XRC-7V1	MPTL	itest.vhd

Table 5 : Available Variants of the ITest Example Design

The design consists of:

- Clock And Reset Generation.
- Target MPTL interface, using an instance of mptl_if_target_wrap or, target PCIe interface, using an instance of pcie_if_target_wrap.
- Clock Domain Transfer, using an instance of adb3_ocp_cross_clk_dom.
- OCP Full to OCP Lite Conversion, using an instance of adb3_ocp_full2lite_b.
- An Address Space Splitter, using an instance of adb3_ocp_l_split.
- The Test registers, implemented as VHDL processes.
- An Interrupt Controller, using an instance of adb3_ocp_l_ictrl.

Figure 8 below shows the main elements of the **ITest** design using MPTL interface IP.

Figure 9 below shows the main elements of the **ITest** design using PCIe interface IP.

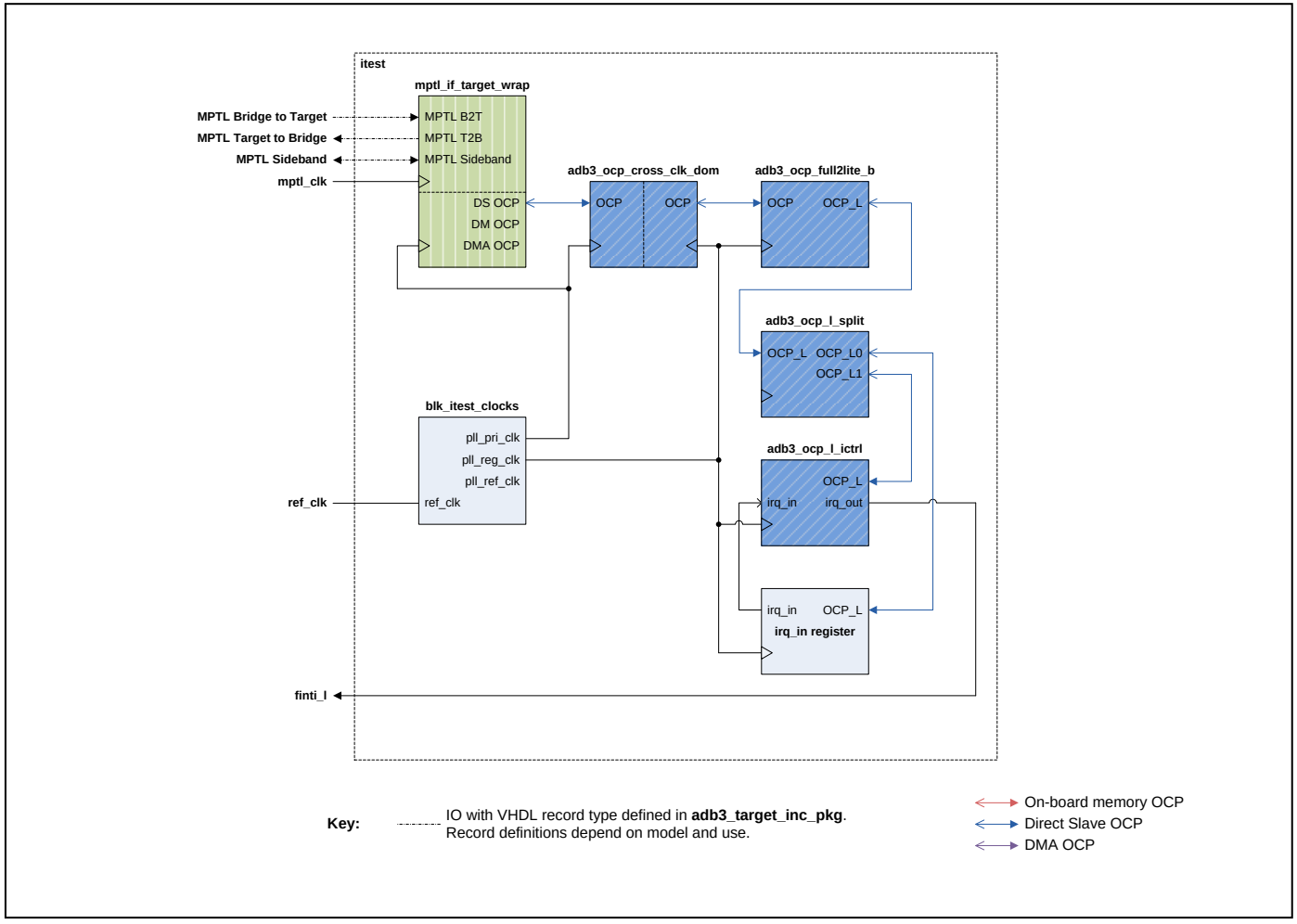

Figure 8 : ITest Design Block Diagram (MPTL)

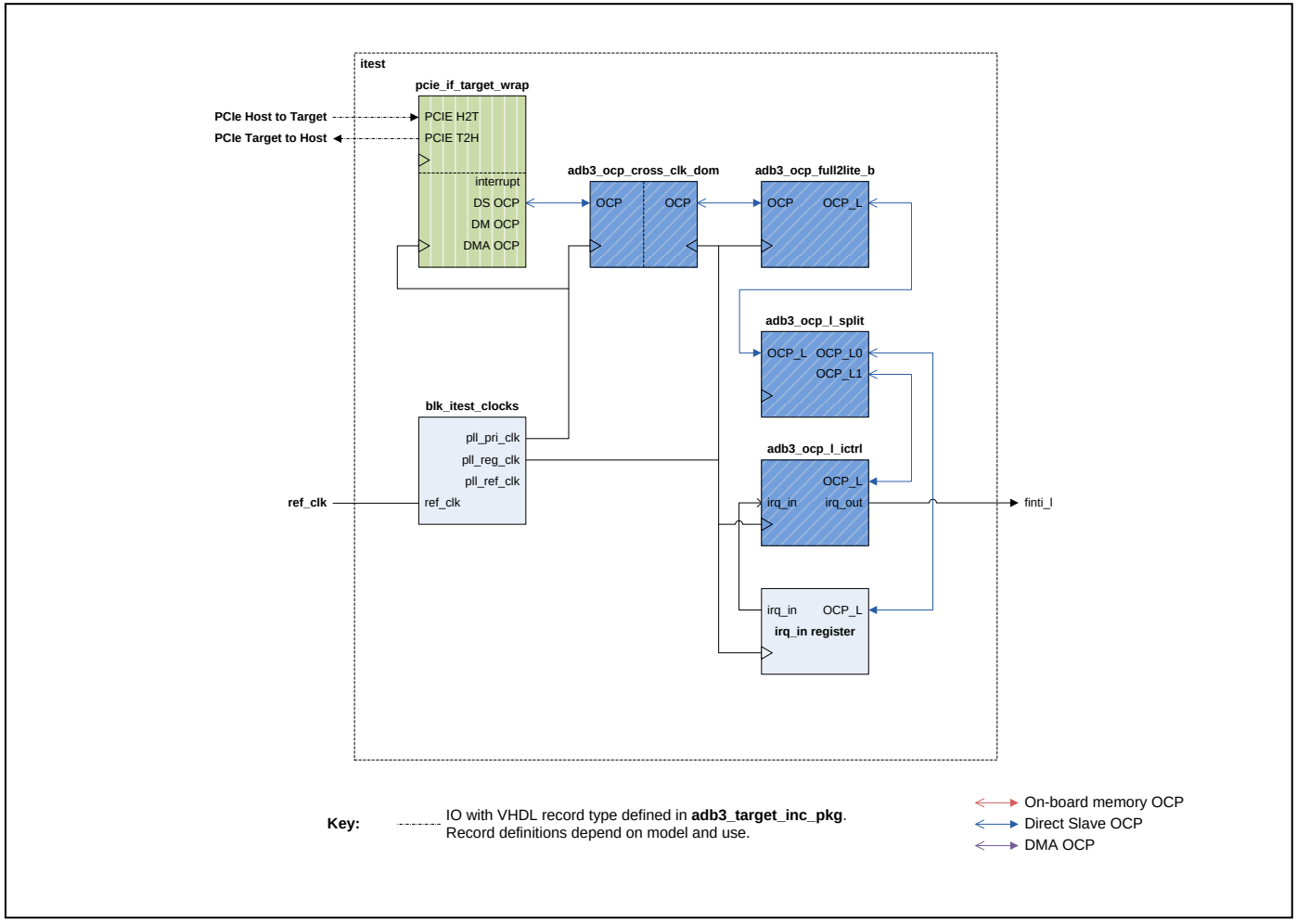

Figure 9 : ITest Design Block Diagram (PCIe)

5.6.5.1 Clock And Reset Generation

This block is implemented by **hdl/vhdl/examples/itest/common/blk_itest_clocks.vhd** and includes the following functional areas:

- Internal clock generation (MMCM)
- Internal reset generation (MMCM)

5.6.5.1.1 Internal Clock Generation (MMCM)

This consists of an instantiation of a Xilinx MMCM block. The MMCM is reset on power on using the **pll_reset** signal, and clocked by a buffered version of the **ref_clk** global clock input. It generates several output clocks depending on the model in use. Refer to Figure 10.

pll_ref_clk

- This clock is generated from a buffered version of the **ref_clk** global clock input.
- It is used as a reference clock by the design.
- On all current models it is fixed at 200 MHz.

pll_pri_clk

- This clock is generated from a buffered version of the MMCM **out_clk0** output.
- It is used as the primary OCP clock by the design.
- On all current models it is generated at 200 MHz.
- It is used to clock the high-frequency OCP logic in the **ITest** design.
- Its frequency need not be related to any of the other clocks.

pll_reg_clk

- This clock is generated from a buffered version of the MMCM **out_clk1** output.
- It is used as a low frequency clock by the design.
- On all current models it is generated at 80 MHz.
- It is used to clock the low-frequency OCP logic in the **ITest** design.
- Its frequency need not be related to any of the other clocks.

5.6.5.1.2 Internal Reset Generation (MMCM)

An active high asynchronous reset **pll_rst** is generated from the MMCM locked signal. Refer to Figure 10.

In models which use the Target MPTL interface, the **pll_rst** signal is used as its **ocp_ready** input. The **ITest** example design is reset by **rst** which is generated from the Target MPTL Interface **mptl_ready** output.

In models which use the Target PCIe interface, the **pll_rst** signal is combined with the **pcie_rst_l** FPGA input to generate the **ITest** example design reset **rst**.

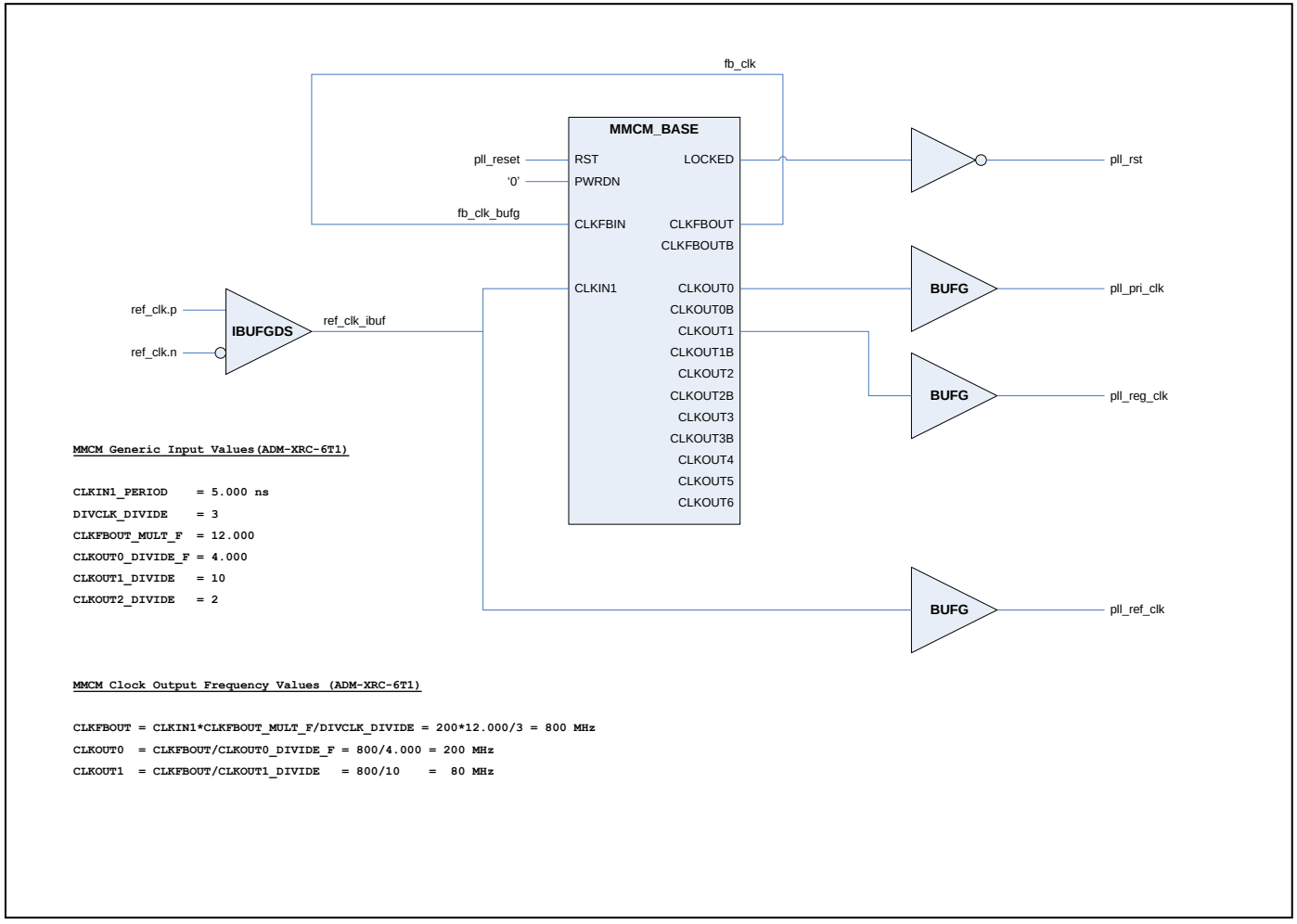

Figure 10 : ITest Design Internal Clock Generation (MMCM)

5.6.5.2 Target MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the target MPTL interface core, instantiating an MPTL to OCP interface appropriate to the model in use. The purpose of the block is to connect the MPTL to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component mptl_if_target_wrap for details.

The **ITest** design output signal **mptl_sb_t2b.mptl_target_configured_l** indicates that the FPGA OCP based blocks are ready to communicate with the bridge via the MPTL interface. This output is generated using the mptl_if_target_wrap input **ocp_ready**. In the case of the **ITest** design, this **ocp_ready** input is driven by a signal derived from the **LOCKED** flag of the design's main MMCM (i.e. the one generating **pll_pri_clk** etc.). This holds off MPTL initialisation until after the MMCM is locked.

The reason for holding off MPTL initialisation is to prevent a race condition that might otherwise occur between (a) software attempting to read or write Target FPGA registers after configuration and (b) the main MMCM in the design achieving lock. Holding off MPTL initialisation between the Bridge and Target until the design's main MMCM has achieved lock causes any call made by software to the **ADMXRC3_ConfigureFromFile** API function to wait until MPTL initialization has been completed, thus guaranteeing that the Target FPGA is in the proper state for software on the host to communicate with it.

The **ITest** design input signal **mptl_sb_b2t.mptl_bridge_gtp_online_l** indicates that the bridge FPGA end of the MPTL interface is active. This input is used to generate the mptl_if_target_wrap output **mptl_ready**. In the case of the **ITest** design, this **mptl_ready** output is used as the asynchronous global reset.

> **Note**
> The Direct Slave and DMA address spaces supported by the Bridge FPGA are smaller than the full ADB3 OCP address space. For the model in use, they are indicated by the **DS_ADDR_WIDTH** and **DMA_ADDR_WIDTH** constants respectively, which are defined in the adb3_target_inc_pkg package.

5.6.5.3 Target PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the target PCIe interface core, instantiating a PCIe to OCP interface appropriate to the model in use. The purpose of the block is to connect the PCIe to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component pcie_if_target_wrap for details.

> **Note**
> The Direct Slave and DMA address spaces supported by the PCIe interface IP are smaller than the full ADB3 OCP address space. For the model in use, they are indicated by the **DS_ADDR_WIDTH** and **DMA_ADDR_WIDTH** constants respectively, which are defined in the adb3_target_inc_pkg package.

5.6.5.4 Clock Domain Transfer

In a typical FPGA design, registers accessible to the host CPU are not directly used as conduits for high volumes of data. By placing such registers in a relatively slow clock domain, clocked by **pll_reg_clk** at 80 MHz, the **ITest** design as a whole more easily meets timing when implemented using the Xilinx ISE tools. To bridge the register clock domain and the the OCP clock domain (clocked by **pll_pri_clk** at 200 MHz), the **ITest** design contains an instance of adb3_ocp_cross_clk_dom.

5.6.5.5 OCP Full to OCP Lite Conversion

As well as placing registers in a slow clock domain as described in Section 5.6.5.4 - "Clock Domain Transfer" above, for a large design, it can be beneficial to reduce the width of OCP data busses that are found within a design, from 128 bits to 32 bits and to eliminate the control signals that enable bursting. The OCP Lite profile fulfils this purpose, providing a lightweight protocol with 32-bit wide data and a reduced number of control signals for handshaking, when compared to the OCP Full profile.

Although the **ITest** design is a small design, an OCP Full to OCP Lite conversion block is instantiated to illustrate this principle.

5.6.5.6 Address Space Splitter

The **ITest** design partitions the OCP Lite address space into two regions using an instance of adb3_ocp_l_split. Region 0, which claims OCP byte addresses 0x0 - 0xFF, is used for the Test Registers. Region 1, which claims OCP byte addresses 0x100 - 0x1FF, connects directly to the Interrupt Controller.

5.6.5.7 Test Registers

A set of VHDL processes implements a single register, connected to port 0 of the address space splitter described in Section 5.6.5.6 - "Address Space Splitter". Although there is a single register in this example, in principle as many registers can be created as are required.

The purpose of this register is to simulate events that originate somewhere else in the design; for example, in an I/O interface. However, since there is no I/O interface in the **ITest** design, the host can simulate interrupt generation by writing ones to individual bits in this register. The addresses in Table 6 - "ITest Design, Test Registers" below are OCP byte addresses after taking into account the splitter's address decoding for region 0.

Name	Type	Address
TEST	WO	0x0

Table 6 : ITest Design, Test Registers

Bits	Mnemonic	Type	Function
0	TEST0	WO	When written with 1, generates interrupt 0
1	TEST1	WO	When written with 1, generates interrupt 1
n	TESTn	WO	When written with 1, generates interrupt n (2 <= n <= 30)
31	TEST31	WO	When written with 1, generates interrupt 31

Table 7 : ITest Design, TEST Register (0x0)

Writing a one to any particular bit of the **TEST** register generates a pulse that the interrupt controller block, described in Section 5.6.5.8 - "Interrupt Controller" receives and latches.

5.6.5.8 Interrupt Controller

The logic for capturing interrupts, i.e. events, is encapsulated in an instance of adb3_ocp_l_ictrl. This inputs the set of interrupt requests, which are pulses that indicate that a particular event has occurred, and latches them until they are cleared by host software. The particular scheme employed should be recognizable to anybody with a background in computer hardware, with registers provided for interrupt status, enabling and clearing. In a typical application, a software thread running on the host CPU interacts with the registers of the interrupt controller in order to respond to and dismiss hardware events as they occur.

Since the adb3_ocp_l_ictrl component has an OCP Lite interface to its registers, no glue logic is required for it. Its OCP Lite interface is connected directly to port 1 of the address space splitter described in Section 5.6.5.6 - "Address Space Splitter". As the splitter claims OCP Lite 0x100 - 0x1FF as region 1, the registers of the interrupt controller are located in this region.

The registers in the interrupt controller component are described in detail in the description of the component adb3_ocp_l_ictrl, but are summarized here for completeness. The addresses in Table 8 - "ITest Design, Interrupt Controller Registers" below are OCP byte addresses after taking into account the splitter's address decoding for region 1.

Name	Type	Address
ENABLE	RW	0x180
CLEAR	RW	0x184
COUNT	RW	0x188
STAT	RW	0x18C

Table 8 : ITest Design, Interrupt Controller Registers

Bits	Mnemonic	Type	Function
0	TEST0	R/W	1 => interrupt 0 is enabled (unmasked) 0 => interrupt 0 is disabled (masked).
1	TEST1	R/W	1 => interrupt 1 is enabled (unmasked) 0 => interrupt 1 is disabled (masked).
n	TESTn	R/W	1 => interrupt n is enabled (unmasked) 0 => interrupt n is disabled (masked) (2 <= n <= 30).
31	TEST31	R/W	1 => interrupt 31 is enabled (unmasked) 0 => interrupt 31 is disabled (masked).

Table 9 : ITest Design, ENABLE Register (0x180)

Bits	Mnemonic	Type	Function
0	TEST0	WO	When written with 1, clears (dismisses) interrupt 0
1	TEST1	WO	When written with 1, clears (dismisses) interrupt 1
n	TESTn	WO	When written with 1, clears (dismisses) interrupt n (2 <= n <= 30)
31	TEST31	WO	When written with 1, clears (dismisses) interrupt 31

Table 10 : ITest Design, CLEAR Register (0x184)

Writing to the **CLEAR** register also has the side-effect of forcing the interrupt line driven out of the interrupt controller to be deasserted for a few cycles (if there are still interrupts after these few cycles, the interrupt line is reasserted). This is known as "rearming" the interrupt line. This is necessary because the interrupt line driven out of the controller is treated as an edge-triggered signal. When multiple interrupt independently-generated events occur in quick succession, rearming the interrupt line avoids a race condition that could otherwise stop interrupts from being delivered the the host.

Bits	Mnemonic	Type	Function
31	NCYCLE	R/W	Increments on every register domain clock cycle in which at least one active interrupt is enabled. May be initialized to a particular value by writing to it.

Table 11 : ITest Design, COUNT Register (0x188)

Bits	Mnemonic	Type	Function
0	IRQ0	RO	1 => interrupt 0 is active 0 => interrupt 0 is not active.
1	IRQ1	RO	1 => interrupt 1 is active 0 => interrupt 1 is not active.
n	IRQn	RO	1 => interrupt n is active 0 => interrupt n is not active (2 <= n <= 30)
31	IRQ31	RO	1 => interrupt 31 is active 0 => interrupt 31 is not active

Table 12 : ITest Design, STAT Register (0x18C)

The **STAT** register indicates the set of active interrupts regardless of the current state of the **ENABLE** register. Thus, in common with a typical interrupt scheme in a real device, the **STAT** register can be used to implement a polled mechanism for responding to events, as opposed to an interrupt-driven scheme.

In effect, the interrupt controller asserts the interrupt output signal to the host when the condition **(ENABLE AND STAT)** evaluates to a nonzero value, i.e. at least one active interrupt is enabled.

5.6.6 Testbench Description

The **ITest** example FPGA design testbench tests operation of the **ITest** example FPGA design.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 13 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/itest/common/
ADM-XRC-6TL	MPTL	test_itest.vhd
ADM-XRC-6T1	MPTL	test_itest.vhd
ADM-XRC-6TGE	MPTL	test_itest.vhd
ADM-XRC-6T-ADV8	PCIe	test_itest_pcie.vhd
ADM-XRC-6T-DA1	MPTL	test_itest.vhd
ADPE-XRC-6T	MPTL	test_itest.vhd
ADPE-XRC-6T-L	MPTL	test_itest.vhd
ADM-XRC-7K1	MPTL	test_itest.vhd
ADM-XRC-7V1	MPTL	test_itest.vhd

Table 13 : Available Variants of the ITest Example Design Testbench

It consists of the following functions:

- Clock generation for the testbench and the Unit Under Test (UUT).
- The Unit Under Test (UUT), which is the one and only instance of the top-level itest block.
- Bridge MPTL interface, using an instance of mptl_if_bridge_wrap or, host PCIe interface, using an instance of pcie_if_host_wrap.
- OCP channel probes, using instances of adb3_ocp_transaction_probe.
- Stimulus generation and verification.
- On-board memory simulation models.

Figure 11 shows the testbench and main elements of the itest FPGA design using MPTL interface IP.

Figure 12 shows the testbench and main elements of the itest FPGA design using PCIe interface IP.

The testbench includes the following packages:

- ADB3 OCP profile definition package (adb3_ocp)
- ADB3 OCP testbench package (adb3_ocp_tb_pkg)
- ADB3 target include package (adb3_target_inc_pkg)
- ADB3 target testbench include package (adb3_target_tb_inc_pkg)

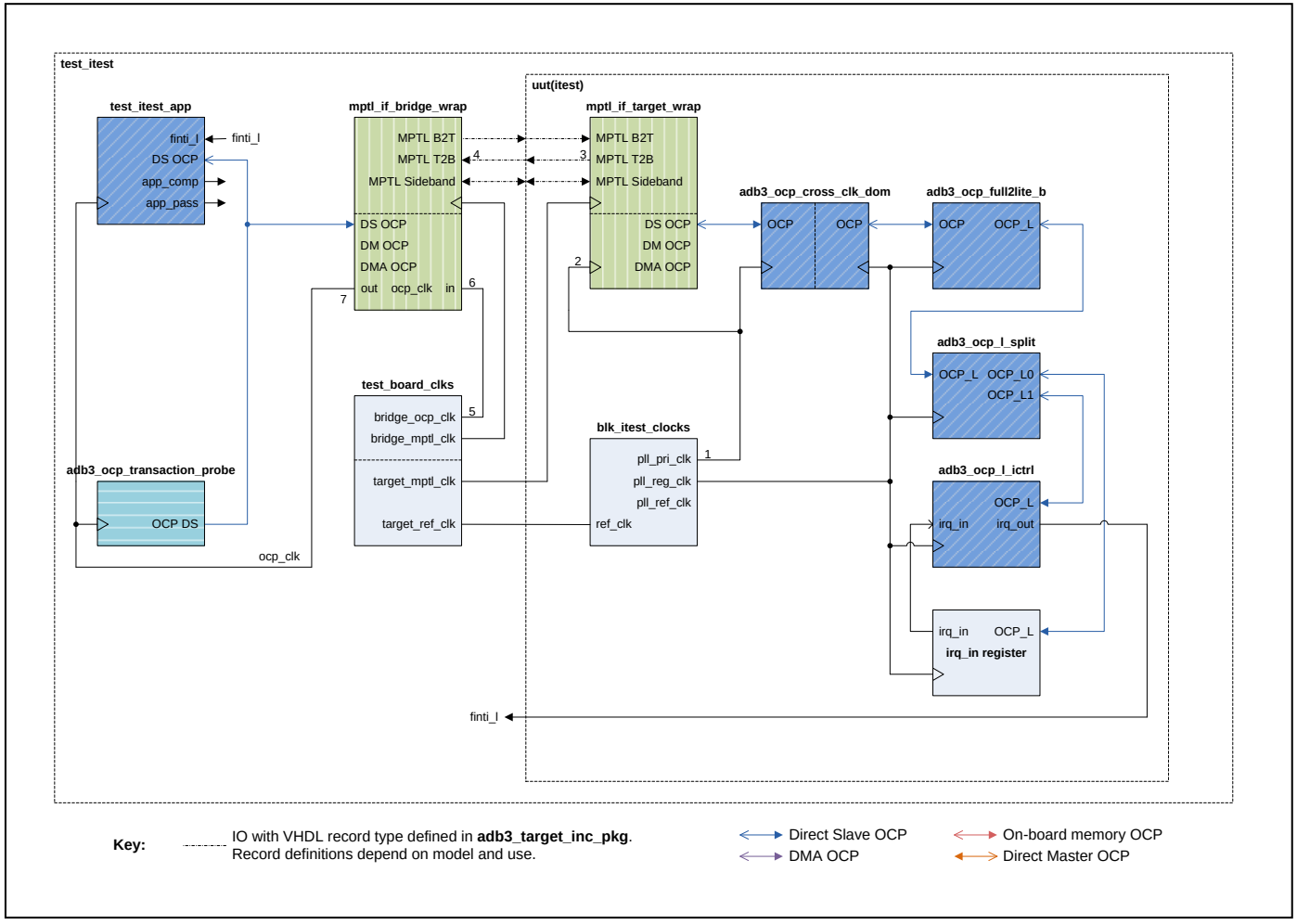

Figure 11 : ITest Design Testbench and Top Level Block Diagram (MPTL)

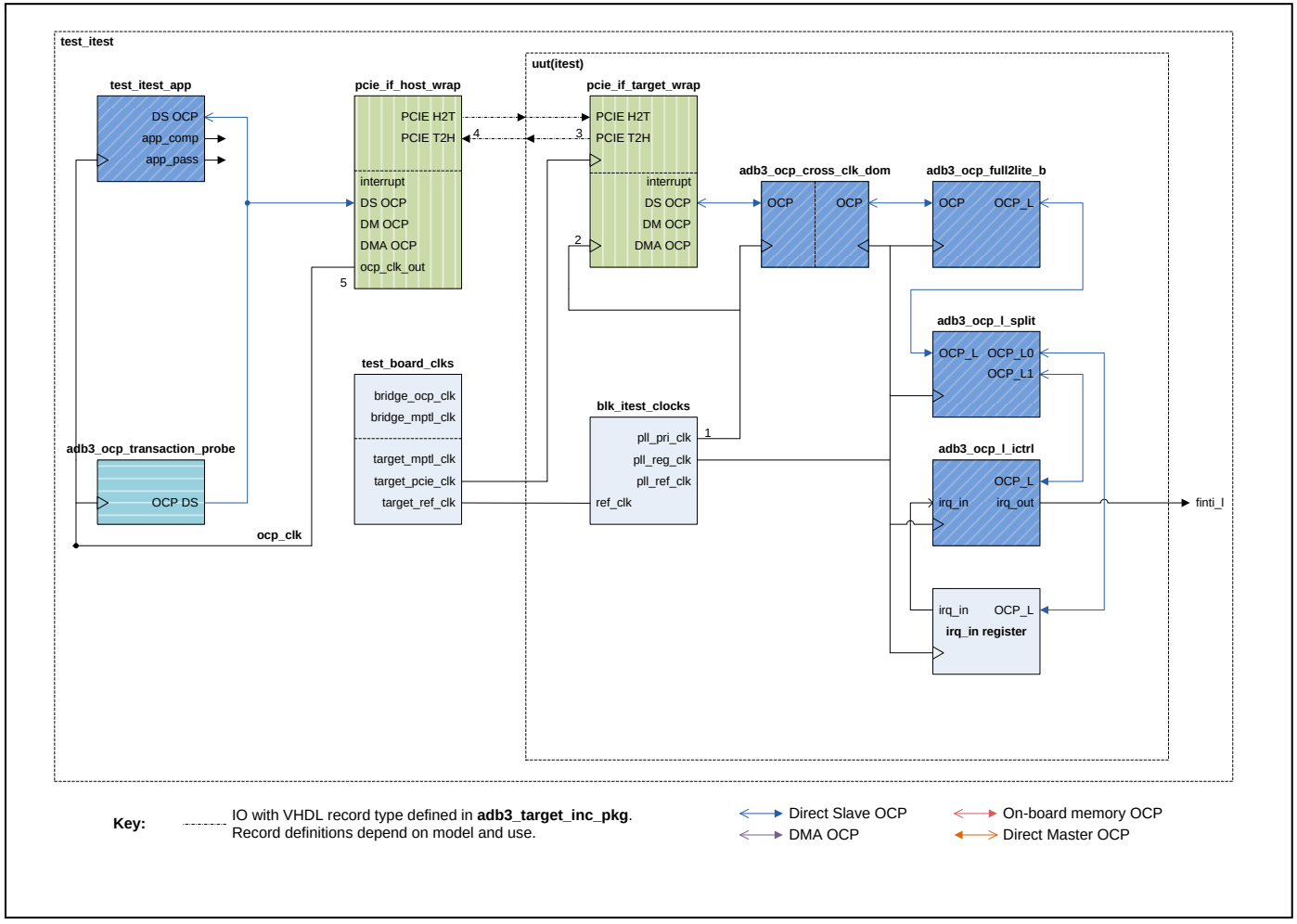

Figure 12 : ITest Design Testbench and Top Level Block Diagram (PCIe)

5.6.6.1 Clock Generation

The testbench uses the **test_itest_clocks** block to generate the clocks used by the testbench and the UUT. This block is implemented by the file **hdl/examples/itest/*<model>*/test_itest_*<model>*.vhd**, for each supported model.

Target Clocks

- It generates the **target_ref_clk** and **target_mptl_clk** clocks according to which model is selected. These clocks drive the unit under test (itest).

Bridge MPTL/PCIe Interface Clock

- It generates the **bridge_mptl_clk** or **bridge_pcie_clk** clock according to which model is selected. This clock drives the **mptl_clk** or **pcie_clk** port, as appropriate for the selected model, on the bridge MPTL interface block.

Bridge OCP Clock (MPTL)

- It generates the **bridge_ocp_clk** clock according to which model is selected. This clock drives the **ocp_clk_in** clock input on the bridge MPTL interface block.
- This clock is only used during Full MPTL simulation. Refer to bridge MPTL interface for details.

5.6.6.2 Bridge MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the bridge MPTL interface core, instantiating an OCP to MPTL interface appropriate to the model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the MPTL. Refer to the component mptl_if_bridge_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the mptl_if_bridge_wrap **mptl_b2t** signals to the mptl_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the mptl_if_target_wrap **mptl_t2b** signals to the mptl_if_bridge_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 11 as the route consisting of points 1, 2, 3, 4 and 7.

Full MPTL simulation

- The testbench Direct Slave and DMA OCP m2s signals are input to the mptl_if_bridge_wrap.
- The UUT Direct Slave and DMA OCP m2s signals are output from the mptl_if_target_wrap.
- Apart from the packetisation, multiplexing and demultiplexing that occurs in the MPTL interfaces (both Bridge and Target), the arrangement is transparent. In other words, behaviour is as if the stimulus were applied directly to the Target FPGA's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 11 as the route consisting of points 5, 6 and 7.

The mptl_if_bridge_wrap output **mptl_online** indicates that the MPTL interface is active and stable. It is used by the testbench to generate the **mptl_online_long** signal which it monitors. Simulation will be terminated with an error message if it becomes inactive. This may occur if, for example, a protocol error arises on the MPTL signals.

The mptl_if_bridge_wrap output **dma_abort** indicates the status of the UUT's **dma_abort** signal.

5.6.6.3 Host PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the host PCIe interface core, instantiating an OCP to PCIe interface appropriate to the model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the PCIe. Refer to the component pcie_if_host_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the pcie_if_host_wrap **pcie_h2t** signals to the pcie_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the pcie_if_target_wrap **pcie_t2b** signals to the pcie_if_host_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 12 as the route consisting of points 1, 2, 3, 4 and 5.

The pcie_if_host_wrap output **dma_abort** indicates the status of the UUT's **dma_abort** signal.

5.6.6.4 OCP Channel Probes

This function monitors the Direct Slave OCP channel for addressing/transaction problems. It generates warnings/errors if it detects an illegal OCP operation. A probe error will result in a 'FAILED' **ITest** simulation result. It uses the component adb3_ocp_transaction_probe.

5.6.6.5 Stimulus Generation and Verification

Stimulus generation and verification of results is performed by the **test_itest_app** component that is part of the **ITest** testbench, and consists of two processes that initiate OCP bursts on the Direct Slave OCP channel in order to stimulate the UUT.

The process **test_generate_p** simulates interrupts (hardware events) being generated within the FPGA. It generates each of the 32 different interrupts implemented by the design exactly once, with some cycles of delay between each. In the testbench for a real FPGA design, this process would be unnecessary (except perhaps for debug purposes) as the FPGA itself would generate interrupts when hardware events occur, such as receiving a packet of data or completing some processing task. This process is described in more detail in Section 5.6.6.5.1 - "Interrupt Generation Process".

The process **host_interrupt_thread_p** represents a software thread running on the host CPU, responding to and clearing interrupts that are generated by the **test_generate_p** process. It verifies that each of the 32 different interrupts is generated exactly once. This process is described in more detail in Section 5.6.6.5.2 - "Interrupt Handler Process".

Because both the **test_generate_p** and **host_interrupt_thread_p** processes need to perform Direct Slave OCP transfers via the the Bridge MPTL Interface instance (for models that incorporate MPTL IP into the design), or via the Host PCIe Interface instance (for models that incorporate PCIe IP into the design), a mutual exclusion mechanism protects the Direct Slave OCP channel so that each process can initiate OCP bursts on it.

5.6.6.5.1 Interrupt Generation Process

The process **test_generate_p** simulates events occuring within somewhere within the FPGA, for example in an I/O interface. In a real hardware design, this process would be unnecessary, except perhaps for debug purposes. Although it is entirely artificial compared to how real hardware would generate interrupts, it exercises the interrupt

controller functionality by iterating over the 32 available interrupt sources, asserting each one exactly once. The steps performed can be expressed in pseudocode as follows:

1. Wait until the testbench top level asserts the **start_test** signal.
2. For i in 0 to 31 do (main loop)
    a. Wait a few cycles (note 1).
    b. Write a '1' to bit *i* of the **TEST** register (note 2).
3. Wait for 100 cycles (note 3).
4. Assert the **generate_complete** signal.

Note 1: This delay prevents the interrupt handler process **host_interrupt_thread_p** seeing most or all of the interrupts at the same time. In real hardware, the timing of interrupts is usually less predictable than in this simple testbench.

Note 2: This makes interrupt source *i* active.

Note 3: This delay gives the interrupt handler process **host_interrupt_thread_p** time to clear the final interrupt request so that it sees exactly the expected number of interrupts. Each of the 32 interrupt sources should be asserted exactly once, both from the point of view of the **test_generate_p** process and the **host_interrupt_thread_p**.

5.6.6.5.2 Interrupt Handler Process

The process **test_generate_p** simulates an interrupt handler running as a thread on the host CPU, interacting with the interrupt controller in the FPGA. The steps performed can be expressed in pseudocode as follows:

1. Enable all 32 interrupt sources by writing X"FFFFFFFF" to the **ENABLE** register in the interrupt controller.
2. While not finished, do (main loop)
    a. Wait until either a falling edge occurs on the interrupt request line **finti_l**, or the **generate_complete** signal is asserted.
    b. If **generate_complete** is asserted, exit the loop.
    c. Read (sample) the **STATUS** register (note 1) to get the set of active interrupts from the interrupt controller.
    d. Report the set of active interrupts (for this iteration of the main loop) on the simulator console (note 2).
    e. Write (only) the set of active interrupts (that was previously sampled) to the **CLEAR** register in the interrupt controller (note 3).
3. Verify that each of the 32 interrupt sources became active exactly once; report any misbehaved interrupt sources on the simulator console.
4. Assert the **host_interrupt_complete** signal.

Note 1: The STATUS register always indicates the set of active interrupts, whether enabled (unmasked) or not.

Note 2: Depending on the rate at which the **test_generate_p** generates interrupts, this may report that several interrupt sources are active on any given iteration of the main loop.

Note 3: Writing to the CLEAR register clears any interrupt for which a '1' bit is written. It also forces the **finti_l** signal deasserted for a few cycles (known as rearming). This avoids a race condition that would otherwise prevent further interrupts being seen; it ensures that if other interrupts become active between steps 2c and 2e, another falling edge will be generated on the **finti_l** line within a few cycles.

5.6.7 Design Simulation

5.6.7.1 Simulation Using ModelSim

For a complete list of the source files used during simulation, refer to the appropriate **ModelSim** macro file (**.do**). These are located in each of the model-specific design directories. The macro file that should be used depends upon the type of simulation required:

- OCP-Only: **hdl/vhdl/examples/itest/*<model>*/itest-*<model>*.do**
- Full MPTL: **hdl/vhdl/examples/itest/*<model>*/itest-*<model>*-mptl.do**

where ***<model>*** corresponds to the model in use; for example **admxrc6t1** for the ADM-XRC-6T1.

ModelSim simulation is initiated using the **vsim** command with the appropriate macro file. For example, to perform an OCP-Only simulation in Windows for the ADM-XRC-6T1, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\itest\admxrc6t1
vsim -do "itest-admxrc6t1.do"
```

In Linux, the commands are:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/itest/admxrc6t1
vsim -do "itest-admxrc6t1.do"
```

Note
The **ModelSim** macro files always delete any previously compiled data before compiling the **ITest** design.

5.6.7.2 Simulation Using PlanAhead

In order to perform an **ISIM** simulation in **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.6.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to perform a simulation as normal.

The **ISIM** simulation set **sim_ocp** should be selected to perform an OCP-Only simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

The **ISIM** simulation set **sim_mptl** should be selected to perform a Full MPTL simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

5.6.7.3 Simulation Results

5.6.7.3.1 Initialisation Results

ModelSim transcript output during initialisation of the simulation is of the form described in the following subsections.

5.6.7.3.1.1 Testbench Status (MPTL)

The testbench produces a summary of the model and simulation type, and then waits for the MPTL interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_itest
# ** Note: Waiting for MPTL online...
```

```
#    Time: 0 ps  Iteration: 0  Instance: /test_itest
# ** Note: MPTL link online
#    Time: 2080 ns  Iteration: 13  Instance: /test_itest
```

5.6.7.3.1.2 Testbench Status (PCIe)

The testbench produces a summary of the model and simulation type, and then waits for the PCIe interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_itest
# ** Note: Waiting for PCIe online...
#    Time: 0 ps  Iteration: 0  Instance: /test_itest
# ** Note: PCIe link online
#    Time: 1285 ns  Iteration: 6  Instance: /test_itest
```

5.6.7.3.2 Direct Slave OCP Channel Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: (host_interrupt_thread_p): Interrupt thread started
#    Time: 2080 ns  Iteration: 14  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 0
#    Time: 2150 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 1
#    Time: 2220 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 2
#    Time: 2290 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 2360 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 3
#    Time: 2360 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 0 is active.
#    Time: 2800 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 1 is active.
#    Time: 2800 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 2 is active.
#    Time: 2800 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 3 is active.
#    Time: 2800 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0x0000000F)
#    Time: 2800 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 4
#    Time: 2810 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0x0000000F)
#    Time: 2825 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 5
#    Time: 2880 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 6
#    Time: 2950 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 7
#    Time: 3020 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 3075 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 4 is active.
#    Time: 3515 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 5 is active.
#    Time: 3515 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 6 is active.
#    Time: 3515 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 7 is active.
#    Time: 3515 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
```

```
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0x000000F0)
#    Time: 3515 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 8
#    Time: 3525 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0x000000F0)
#    Time: 3540 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 9
#    Time: 3595 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 10
#    Time: 3665 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 11
#    Time: 3735 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 3785 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 8 is active.
#    Time: 4225 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 9 is active.
#    Time: 4225 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 10 is active.
#    Time: 4225 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 11 is active.
#    Time: 4225 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0x00000F00)
#    Time: 4225 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 12
#    Time: 4235 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0x00000F00)
#    Time: 4250 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 13
#    Time: 4305 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 14
#    Time: 4375 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 15
#    Time: 4445 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 4500 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 12 is active.
#    Time: 4940 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 13 is active.
#    Time: 4940 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 14 is active.
#    Time: 4940 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 15 is active.
#    Time: 4940 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0x0000F000)
#    Time: 4940 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 16
#    Time: 4950 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0x0000F000)
#    Time: 4965 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 17
#    Time: 5020 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 18
#    Time: 5090 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 19
#    Time: 5160 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 5210 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 16 is active.
#    Time: 5650 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 17 is active.
#    Time: 5650 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 18 is active.
#    Time: 5650 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
```

```
# ** Note: (host_interrupt_thread_p): Interrupt register bit 19 is active.
#    Time: 5650 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0x000F0000)
#    Time: 5650 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 20
#    Time: 5660 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0x000F0000)
#    Time: 5675 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 21
#    Time: 5730 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 22
#    Time: 5800 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 23
#    Time: 5870 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 5925 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 20 is active.
#    Time: 6365 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 21 is active.
#    Time: 6365 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 22 is active.
#    Time: 6365 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 23 is active.
#    Time: 6365 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0x00F00000)
#    Time: 6365 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 24
#    Time: 6375 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0x00F00000)
#    Time: 6390 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 25
#    Time: 6445 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 26
#    Time: 6515 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 27
#    Time: 6585 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 6635 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 24 is active.
#    Time: 7075 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 25 is active.
#    Time: 7075 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 26 is active.
#    Time: 7075 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 27 is active.
#    Time: 7075 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0x0F000000)
#    Time: 7075 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 28
#    Time: 7085 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0x0F000000)
#    Time: 7100 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 29
#    Time: 7155 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 30
#    Time: 7225 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Asserting interrupt bit 31
#    Time: 7295 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt detected...
#    Time: 7350 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 28 is active.
#    Time: 7790 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 29 is active.
#    Time: 7790 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
```

```
# ** Note: (host_interrupt_thread_p): Interrupt register bit 30 is active.
#    Time: 7790 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Interrupt register bit 31 is active.
#    Time: 7790 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Clearing interrupt bits (0xF0000000)
#    Time: 7790 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (test_generate_p): Completed.
#    Time: 7795 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Done clearing interrupt bits (0xF0000000)
#    Time: 7805 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Checking that each interrupt bit was asserted
exactly once...
#    Time: 7810 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
# ** Note: (host_interrupt_thread_p): Completed.
#    Time: 7810 ns  Iteration: 13  Instance: /test_itest/test_itest_app_i
```

5.6.7.3.3 Completion Results

Assuming that all tests passed, **ModelSim** transcript output on successful completion of simulation is of the form:

```
# ** Failure: Test of design ITEST completed: PASSED.
#    Time: 7810 ns  Iteration: 16  Process: /test_itest/test_results_p File:
../common/test_itest.vhd
# Break in Process test_results_p at ../common/test_itest.vhd line 192
# Simulation Breakpoint: Break in Process test_results_p at
../common/test_itest.vhd line 192
# MACRO ./itest-admxrc6t1.do PAUSED at line 87
```

5.7 Simple Example FPGA Design

5.7.1 Model Support

The **Simple** FPGA design is compatible with all Virtex-6 and 7 Series models.

5.7.2 File Location

The **Simple** FPGA design is located in **hdl/vhdl/examples/simple/**. Files common to all models are located in the **hdl/vhdl/examples/simple/common/** directory. Files specific to a model are located in the **hdl/vhdl/examples/simple/*<model>*/** directory.

5.7.3 Generating PlanAhead Projects

PlanAhead project files (file extension **.ppr**) for the **Simple** FPGA design are generated using the **genppr.tcl** TCL script that can be found in **hdl/vhdl/examples/**. Generating **PlanAhead** projects is described in general in Section 5.3 - "Generating PlanAhead Projects". To generate **PlanAhead** projects for the **Simple** design, the **pattern** passed to the **genppr.tcl** script should be prefixed by **simple-**. Some examples:

1. To generate a **PlanAhead** project in Windows for an ADM-XRC-6T1 fitted with a 6VLX240T device, for the **Simple** example design, start a shell and run the settings32.bat or settings64.bat script from the ISE tools (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl simple-admxrc6t1-6vlx240t
   ```

   Under Linux, start a shell, source the settings32.sh or settings64.sh script (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'simple-admxrc6t1-6vlx240t'
   ```

2. To generate **PlanAhead** projects in Windows for an ADM-XRC-6T1 for all supported devices, use these commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl simple-admxrc6t1-.+
   ```

   Under Linux, use these commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'simple-admxrc6t1-.+'
   ```

Note: the single quotes around the patterns in the Linux examples prevent the shell from attempting to perform globing on the pattern.

PlanAhead project files generated using **genppr.tcl** are created in the example design's **planahead** directory. For example, **hdl/vhdl/examples/simple/planahead/** contains all of the **PlanAhead** projects for the **Simple** example design. Building a design or running a simulation using **ISIM** is performed in the normal way via the **PlanAhead** GUI.

5.7.4 Design Synthesis and Bitstream Build

5.7.4.1 VHDL Source Files

For a complete list of the source files used during synthesis, refer to the appropriate XST project file located in the model design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.prj** for an ADM-XRC-6T1.

5.7.4.2 XST Files

XST Project files (**.prj**) are located in the model design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.prj** for an ADM-XRC-6T1.

XST Script files (**.scr**) are located in the model design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1-6vlx240t.scr** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

XST constraint files (**.xcf**) are located in the model design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.xcf** for an ADM-XRC-6T1.

5.7.4.3 Implementation Constraint Files

Implementation constraint files (**.ucf**) are located in the model design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.ucf** for the ADM-XRC-6T1.

5.7.4.4 Build Using Make

Makefiles are provided for building **Simple** design bitstreams (**.bit** files) from the command line. Depending on the target passed to **NMAKE** (Windows), or **GNU Make** (Linux), bitstreams can be built for a specific model-device combination, or for all supported model-device combinations.

When a **.bit** file is built, it is not automatically used by the example applications unless it is copied into the **bit/simple/** directory. This can be done manually, or by using the Makefile.

The Makefile can also be used to delete **.bit** files and intermediate files. This guarantees that the next time the design is built, it is from VHDL sources as opposed to beginning at some intermediate step.

The Makefile for the **Simple** design has the following targets:

Target	Class	Effect
all	build	Builds all **.bit** files for all supported model and device combinations.
bit_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices.
bit_*<model>*_*<device>*		Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>***.
install	install	Builds and installs all **.bit** files for all supported model and device combinations in the directory **bit/simple/**.
inst_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices and copies them to the directory **bit/simple/**.

Table 14 : Simple Design Makefile Targets (continued on next page)

Target	Class	Effect
inst_*<model>*_*<device>*	install	Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>*** and copies it to the directory **bit/simple/**.
clean	clean	Deletes all **.bit** files and intermediate build files for all supported model and device combinations (but does not delete any files from **bit/simple/**).
clean_*<model>*		Deletes **.bit** files and intermediate build files for the model specified by ***<model>*** for all supported devices (but does not delete any files from **bit/simple/**).
clean_*<model>*_*<device>*		Deletes the **.bit** file and intermediate build files for the model specified by ***<model>*** with a device specified by ***<device>*** (but does not delete any files from **bit/simple/**).

Table 14 : Simple Design Makefile Targets

Files that result from the build process, including **.bit** files, are placed in:

hdl/vhdl/examples/simple/build/*<model>*-*<device>*/

Filenames of any bitstreams built are thus of the form:

hdl/vhdl/examples/simple/build/*<model>*-*<device>*/simple-*<model>*-*<device>*.bit.

When a target of class "clean" is executed, the **build/*<model>*-*<device>*** directory is deleted, but files in **bit/simple/** are unaffected.

Note
Before a bitstream can be used by one of the example applications, it must be copied to **bit/simple/** by executing a target of class "install", or by manually copying the **.bit** file.

Some example make commands follow:

1 To perform a build of all **Simple** design bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake all
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make all
```

2 To perform a build and install the resulting bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake install
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make install
```

3 To perform a build for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake bit_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make bit_admxrc6t1_6vlx240t
```

4 To perform a build and install for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake inst_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make inst_admxrc6t1_6vlx240t
```

5 To delete all **.bit** files and intermediate build files in Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake clean
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make clean
```

6 To delete the **.bit** file and intermediate build files for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake clean_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make clean_admxrc6t1_6vlx240t
```

5.7.4.5 Build Using PlanAhead

In order to build the **Simple** design using **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.7.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to implement the design and generate a bitstream in the normal way.

Note
In **PlanAhead** 13.x, **bitgen** options are not sticky, and must be applied each time a project is opened. See Section 5.3.1 - "BITGEN Options in PlanAhead 13.x" for details about this issue. This step is **not** required for

5.7.5 Design Description

The **Simple** example FPGA design demonstrates register access available in Gen 3 Alpha Data reconfigurable computing hardware such as the ADM-XRC-6T1.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 15 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/simple/common/
ADM-XRC-6TL	MPTL	simple_l.vhd
ADM-XRC-6T1	MPTL	simple_l.vhd
ADM-XRC-6TGE	MPTL	simple_l.vhd
ADM-XRC-6T-ADV8	PCIe	simple_l_pcie.vhd
ADM-XRC-6T-DA1	MPTL	simple_l.vhd
ADPE-XRC-6T	MPTL	simple_l.vhd
ADPE-XRC-6T-L	MPTL	simple_l.vhd
ADM-XRC-7K1	MPTL	simple_l.vhd
ADM-XRC-7V1	MPTL	simple_l.vhd

Table 15 : Available Variants of the Simple Example Design

The design consists of:

- Clock and Reset Generation.
- Target MPTL interface, using an instance of mptl_if_target_wrap or, target PCIe interface, using an instance of pcie_if_target_wrap.
- OCP to simple bus interface, using an instance of adb3_ocp_simple_bus_if.
- Simple test registers implemented using VHDL processes.

Figure 13 below shows the main elements of the **Simple** design using MPTL interface IP.

Figure 14 below shows the main elements of the **Simple** design using PCIe interface IP.

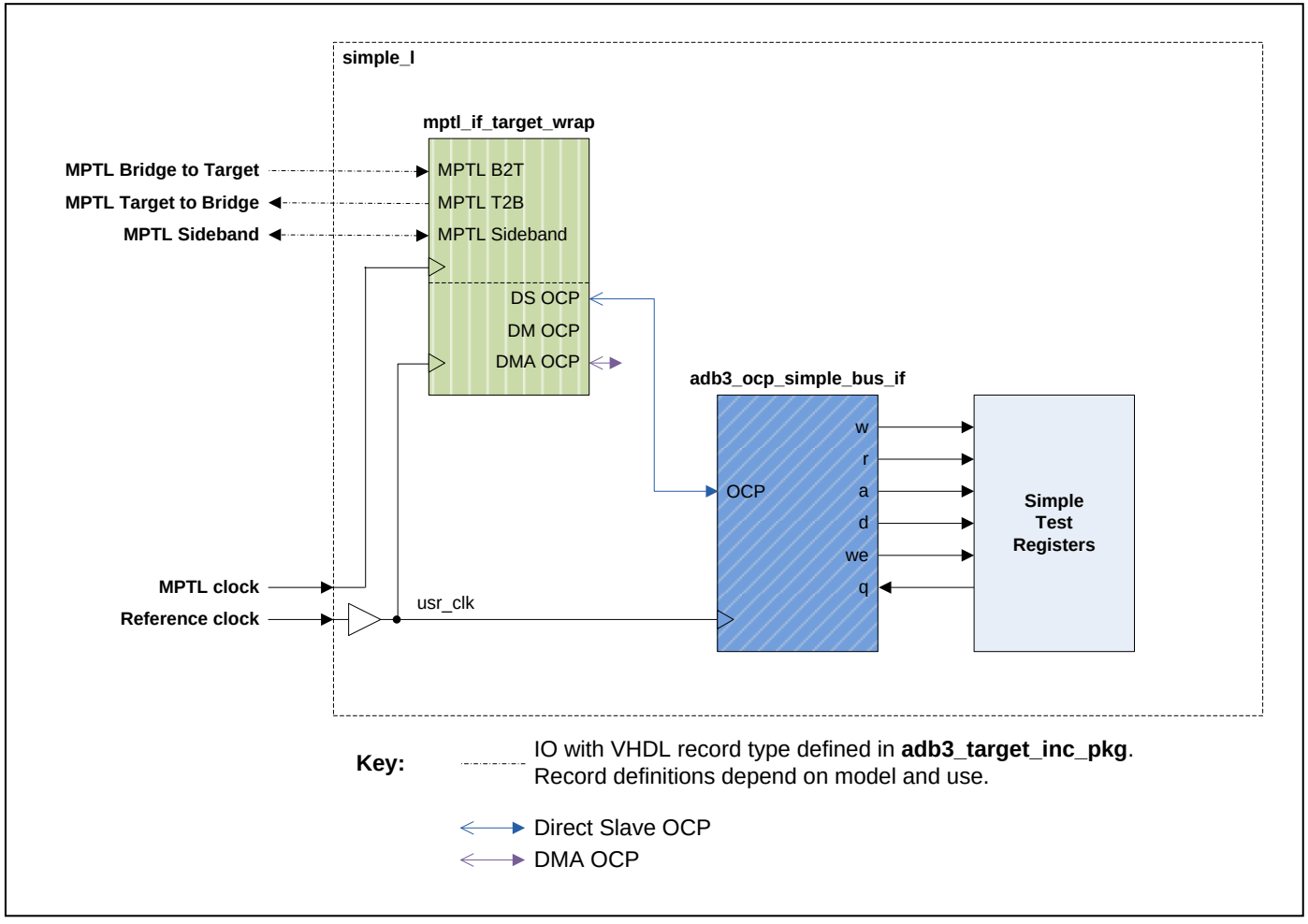

Figure 13 : Simple Design Block Diagram (MPTL)

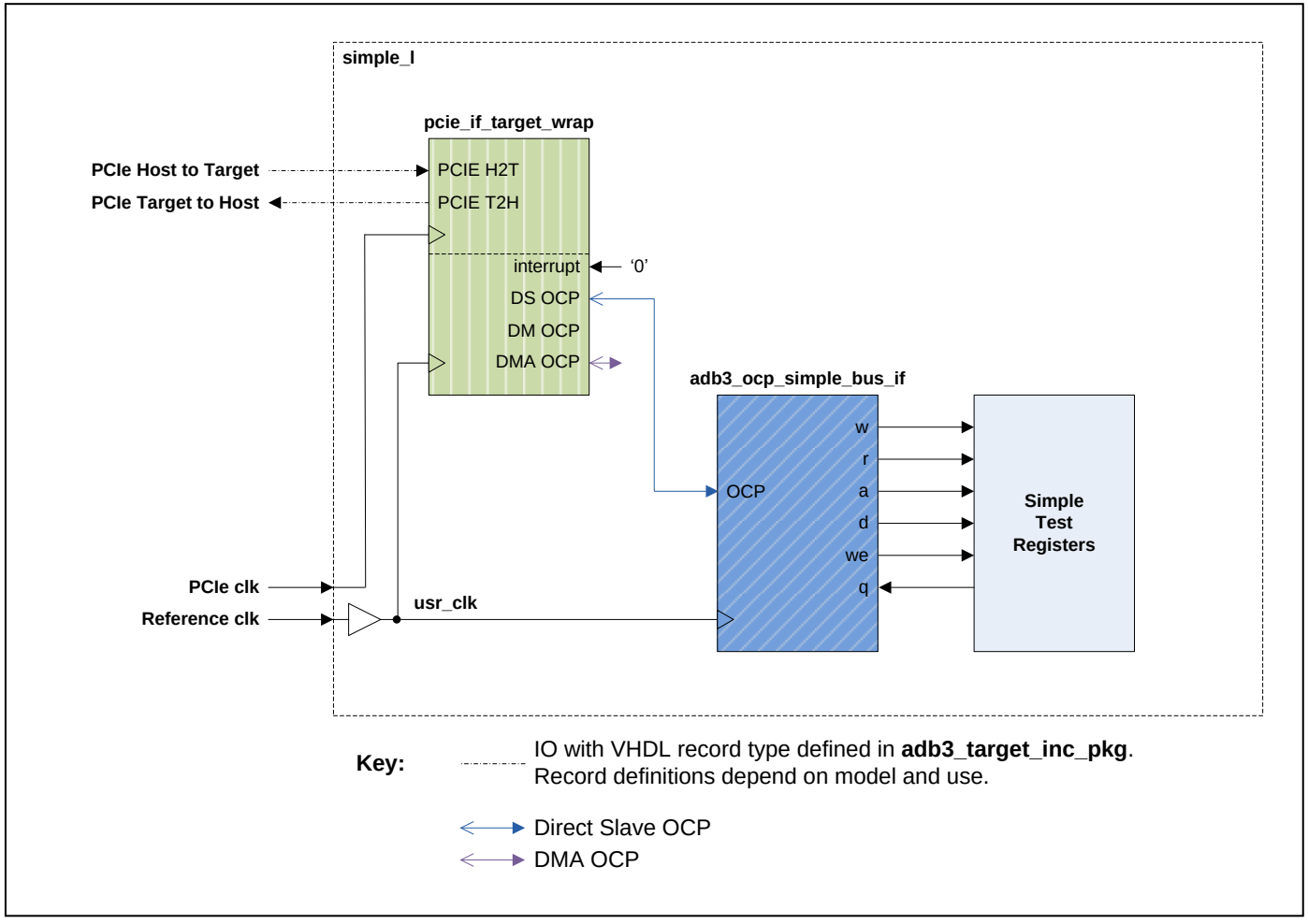

Figure 14 : Simple Design Block Diagram (PCIe)

5.7.5.1 Clock And Reset Generation

OCP Clock

- The **Simple** example design is driven by an OCP clock named **usr_clk**.
- This is a buffered version of the differential reference clock that is input via the top level **ref_clk** port.
- The actual source of the clock in the hardware depends upon the model selected, and is defined in the constraints file located in the model-specific design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.ucf** for an ADM-XRC-6T1.
- On all current models it is fixed at 200 MHz.

MPTL Interface Clock

- The target MPTL interface requires an unbuffered differential MGT clock input.
- It is provided by the **mptl_clk** signal which is buffered within the MPTL interface block.
- The actual source of the clock in the hardware depends upon the model selected, and is defined in the constraints file located in the model-specific design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.ucf** for an ADM-XRC-6T1.

PCIe Interface Clock

- The target PCIe interface requires an unbuffered differential MGT clock input.
- It is provided by the **pcie_clk** signal which is buffered within the PCIe interface block.
- The actual source of the clock in the hardware depends upon the model selected, and is defined in the constraints file located in the model-specific design directory; for example, **hdl/vhdl/examples/simple/admxrc6tadv8/simple-admxrc6tadv8.ucf** for an ADM-XRC-6T-ADV8.

Global Reset

- An active high asynchronous reset **pon_rst** is generated on power up.
- In models which use the Target MPTL interface, the **pon_rst** signal is used as its **ocp_ready** input. The **Simple** example design is reset by **usr_rst** which is generated from the Target MPTL Interface signal **mptl_ready**.
- In models which use the Target PCIe interface, the **pon_rst** signal is combined with the **pcie_rst_l** FPGA input to generate the **Simple** example design reset **usr_rst**.

5.7.5.2 Target MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the target MPTL interface core, instantiating an MPTL to OCP interface appropriate to the model in use. The purpose of the block is to connect the MPTL to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component mptl_if_target_wrap for details.

5.7.5.3 Target PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the target PCIe interface core, instantiating a PCIe to OCP interface appropriate to the model in use. The purpose of the block is to connect the PCIe to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component pcie_if_target_wrap for details.

5.7.5.4 OCP to Simple Bus Interface

An instance of adb3_ocp_simple_bus_if terminates the Direct Slave OCP channel with the Simple test registers, driving a small bus whose signals are as follows:

1. **la_q** - The register address, derived from some low order bits of the Direct Slave OCP address. This is used to select the correct register for writes, and to control a multiplexor that drives **ld_o** for reads.
2. **ds_write** - Indicates that a write cycle is taking place.
3. **lbe_i** - Byte write enables. High when **ds_write** is high and bytes are enabled for writing.
4. **ld_i** - Write data bytes; qualified by **lbe_i** bits.
5. **ds_read** - Indicates that a read cycle is taking place. Valid data must be present on **ld_o** after **read_latency** cycles.
6. **ld_o** - Driven with read data by a multiplexor controlled by **la_q**. The registers of the FPGA design are inputs to the multiplexor.

5.7.5.5 Simple Test Registers

A set of VHDL processes uses the signals **la_q**, **ds_write** etc. described above to implement a single register. Although there is a single register in this example, in principle as many registers can be created as are required.

5.7.5.5.1 Register Description

The **Simple** FPGA design implements registers in the Direct Slave OCP address space as follows:

Name	Type	Address
DATA	RW	0x000000

Table 16 : Simple Design Direct Slave Address Map

Bits	Mnemonic	Type	Function
31:0	DATA	RW	Indicates the nibble-reversed version of the last data written.

Table 17 : Simple Design, DATA Register (0x000000)

Note: there is no address decoding, so this register appears aliased everywhere in the Direct Slave OCP address space.

5.7.6 Testbench Description

The **simple** example FPGA design testbench tests operation of the **simple** example FPGA design.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 18 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/simple/common/
ADM-XRC-6TL	MPTL	test_simple.vhd
ADM-XRC-6T1	MPTL	test_simple.vhd
ADM-XRC-6TGE	MPTL	test_simple.vhd
ADM-XRC-6T-ADV8	PCIe	test_simple_pcie.vhd
ADM-XRC-6T-DA1	MPTL	test_simple.vhd
ADPE-XRC-6T	MPTL	test_simple.vhd
ADPE-XRC-6T-L	MPTL	test_simple.vhd

Table 18 : Available Variants of the Simple Example Design Testbench (continued on next page)

Model	Interface	Filename relative to hdl/vhdl/examples/simple/common/
ADM-XRC-7K1	MPTL	test_simple.vhd
ADM-XRC-7V1	MPTL	test_simple.vhd

Table 18 : Available Variants of the Simple Example Design Testbench

It consists of the following functions:

- Clock generation for the testbench and the Unit Under Test (UUT).
- The Unit Under Test (UUT), which is the one-and-only instance of the simple block.
- Bridge MPTL interface block, using an instance of mptl_if_bridge_wrap or, host PCIe interface block, using an instance of pcie_if_host_wrap.
- Direct Slave OCP channel probe, using an instance of adb3_ocp_transaction_probe.
- Stimulus Generation and Verification.

Figure 15 shows the testbench and embedded simple FPGA design (MPTL).

Figure 16 shows the testbench and embedded simple FPGA design (PCIe).

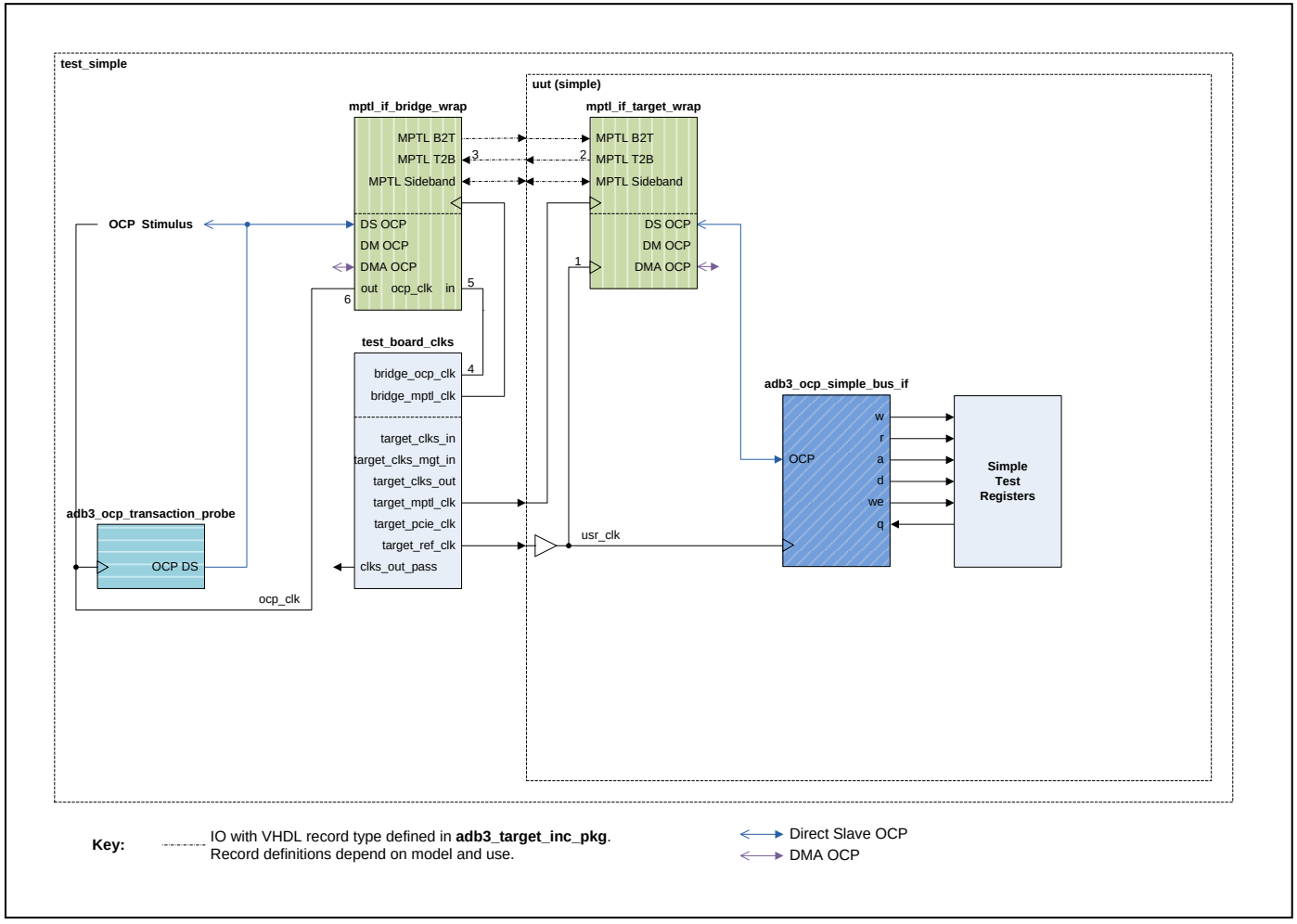

Figure 15 : Simple Design Testbench and Top Level Block Diagram (MPTL)

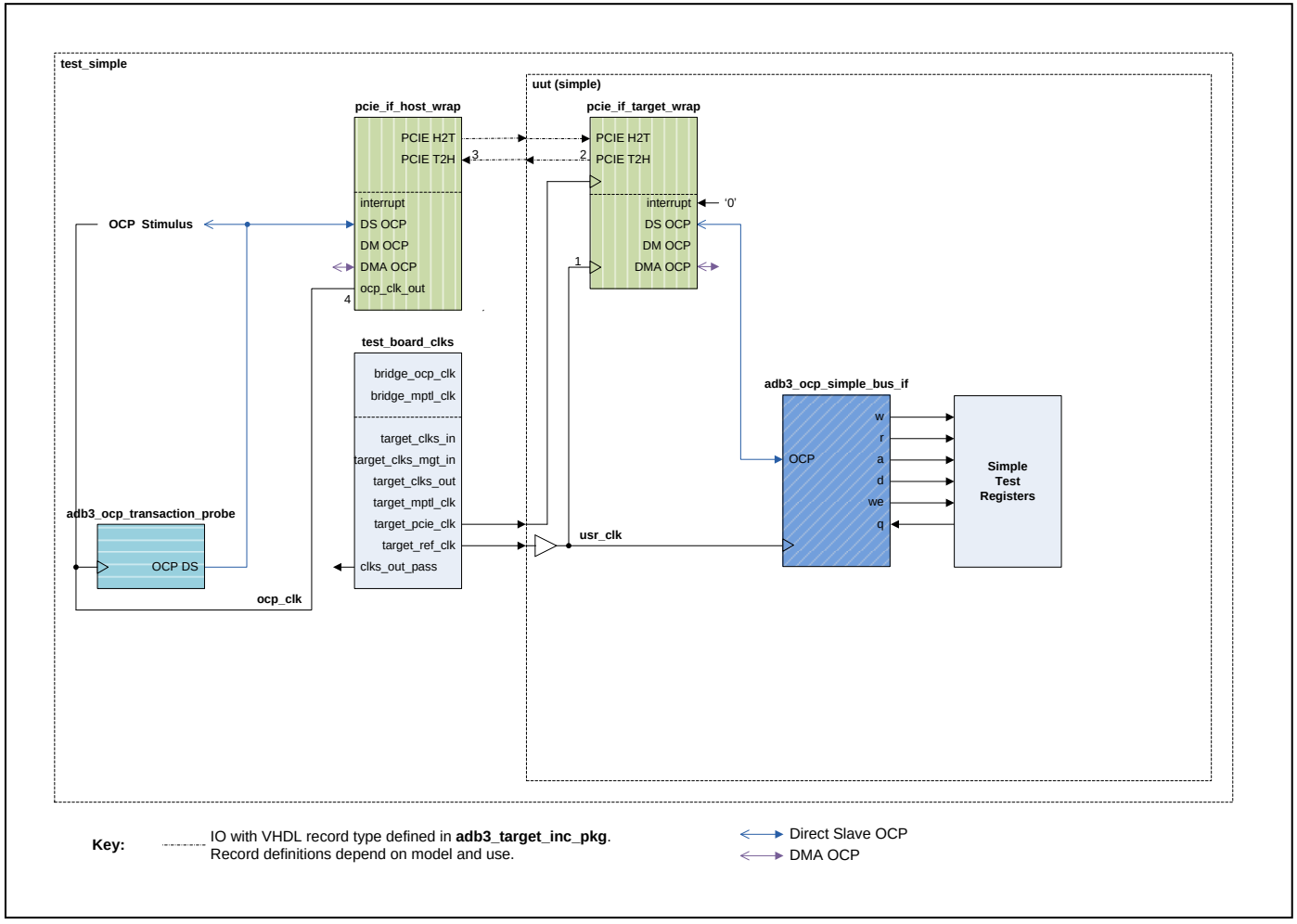

Figure 16 : Simple Design Testbench and Top Level Block Diagram (PCIe)

5.7.6.1 Clock Generation

The testbench uses the test_board_clks block to implement this function.

Target Clocks

- It generates the **ref_clk**, **target_mptl_clk** and **target_pcie_clk** clocks according to which model is selected. These clocks drive the unit under test (simple).

Bridge MPTL Interface Clock

- It generates the **bridge_mptl_clk** clock according to which model is selected. This clock drives the **mptl_clk** differential clock input on the bridge MPTL interface block.

Bridge OCP Clock (MPTL)

- It generates the **bridge_ocp_clk** clock according to which model is selected. This clock drives the **ocp_clk_in** clock input on the bridge MPTL interface block.
- This clock is only used during OCP-Only simulation. Refer to bridge MPTL interface for details.

5.7.6.2 Bridge MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the bridge MPTL interface core, instantiating an OCP to MPTL interface appropriate to the model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the MPTL. Refer to the component mptl_if_bridge_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the mptl_if_bridge_wrap **mptl_b2t** signals to the mptl_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the mptl_if_target_wrap **mptl_t2b** signals to the mptl_if_bridge_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 15 as the route consisting of points 1, 2, 3 and 6.

Full MPTL simulation

- The testbench Direct Slave and DMA OCP m2s signals are input to the mptl_if_bridge_wrap.
- The UUT Direct Slave and DMA OCP m2s signals are output from the mptl_if_target_wrap.
- Apart from the packetisation, multiplexing and demultiplexing that occurs in the MPTL interfaces (both Bridge and Target), the arrangement is transparent. In other words, behaviour is as if the stimulus were applied directly to the Target FPGA's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 15 as the route consisting of points 4, 5 and 6.

The mptl_if_bridge_wrap output **mptl_online** indicates that the MPTL interface is active and stable. It is used by the testbench to generate the **mptl_online_long** signal which it monitors. Simulation will be terminated with an error message if it becomes inactive. This may occur if, for example, a protocol error arises on the MPTL signals.

5.7.6.3 Host PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the host PCIe interface core, instantiating an OCP to PCIe interface appropriate to the

model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the PCIe. Refer to the component pcie_if_host_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the pcie_if_host_wrap **pcie_h2t** signals to the pcie_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the pcie_if_target_wrap **pcie_t2b** signals to the pcie_if_host_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 16 as the route consisting of points 1, 2, 3 and 4.

5.7.6.4 Direct Slave OCP Channel Probe

This function monitors the Direct Slave OCP channel for addressing/transaction problems. It generates warnings/errors if it detects an illegal OCP operation. A probe error will result in a 'FAILED' **Simple** simulation result. It uses the component adb3_ocp_transaction_probe.

5.7.6.5 Stimulus Generation and Verification

This function consists of a set of processes that generate stimulus and verify the results of the simulation.

5.7.6.5.1 Direct Slave OCP Channel

The **simple** testbench provides OCP test stimulus to, and verifies OCP test results from, the UUT's OCP Direct Slave channel.

Tests performed are detailed in the following subsections.

5.7.6.5.1.1 Simple Test

This test exercises the Simple Test Registers as follows:

1. Writes the 32-bit value 0xCAFEFACE to the DATA register.
2. Reads back the DATA register and compares it with the expected value 0xECAFEFAC. If the expected and actual values do not match, the test is considered a failure.

Test complete and pass/fail indications are returned using the **simple_complete** and **simple_passed** testbench signals respectively.

Example results from this test are documented in direct slave OCP channel results.

5.7.7 Design Simulation

5.7.7.1 Simulation Using ModelSim

For a complete list of the source files used during simulation, refer to the appropriate **ModelSim** macro file (**.do**). These are located in each of the model-specific design directories. The macro file that should be used depends upon the type of simulation required:

- OCP-Only: **hdl/vhdl/examples/simple/*<model>*/simple-*<model>*.do**
- Full MPTL: **hdl/vhdl/examples/simple/*<model>*/simple-*<model>*-mptl.do**

where ***<model>*** corresponds to the model in use; for example **admxrc6t1** for the ADM-XRC-6T1.

ModelSim simulation is initiated using the **vsim** command with the appropriate macro file. For example, to perform an OCP-Only simulation in Windows for the ADM-XRC-6T1, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple\admxrc6t1
vsim -do "simple-admxrc6t1.do"
```

In Linux, the commands are:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple/admxrc6t1
vsim -do "simple-admxrc6t1.do"
```

Note
The **ModelSim** macro files always delete any previously compiled data before compiling the **Simple** design.

5.7.7.2 Simulation Using PlanAhead

In order to perform an **ISIM** simulation in **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.7.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to perform a simulation as normal.

The **ISIM** simulation set **sim_ocp** should be selected to perform an OCP-Only simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

The **ISIM** simulation set **sim_mptl** should be selected to perform a Full MPTL simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

5.7.7.3 Simulation Results

5.7.7.3.1 Initialisation Results

ModelSim transcript output during initialisation of the simulation is of the form described in the following subsections.

5.7.7.3.1.1 Testbench Status (MPTL)

The testbench produces a summary of the model and simulation type, and then waits for the MPTL interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Waiting for MPTL online...
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: MPTL link online
#    Time: 22632 ns  Iteration: 2  Instance: /test_simple
```

5.7.7.3.1.2 Testbench Status (PCIe)

The testbench produces a summary of the model and simulation type, and then waits for the PCIe interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Waiting for PCIe online...
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
```

```
# ** Note: PCIe link online
#    Time: 1285 ns  Iteration: 6  Instance: /test_simple
```

5.7.7.3.2 Direct Slave OCP Channel Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote simple DATA 4 bytes 0xCAFEFACE with enable 0b1111 to byte address
0x000000
#    Time: 1625 ns  Iteration: 6  Instance: /test_simple
# ** Note: Read simple DATA 4 bytes 0xECAFEFAC from byte address 0x000000
#    Time: 1687500 ps  Iteration: 7  Instance: /test_simple
# ** Note: Test Simple completed: PASSED.
#    Time: 1687500 ps  Iteration: 7  Instance: /test_simple
```

5.7.7.3.3 Completion Results

Assuming that all tests passed, **ModelSim** transcript output on successful completion of simulation is of the form:

```
# ** Failure: Test of design SIMPLE completed: PASSED.
#    Time: 1687500 ps  Iteration: 9  Process: /test_simple/test_results_p File:
../common/test_simple.vhd
# Break in Process test_results_p at ../common/test_simple.vhd line 230
# Simulation Breakpoint: Break in Process test_results_p at
../common/test_simple.vhd line 230
# MACRO ./simple-admxrc6t1.do PAUSED at line 71
```

5.8 SimpleDMA Example FPGA Design

5.8.1 Model Support

The **SimpleDMA** FPGA design is compatible with all Virtex-6 and 7-series models.

5.8.2 File Location

The **SimpleDMA** FPGA design is located in **hdl/vhdl/examples/simpledma/**. Files common to all models are located in the **hdl/vhdl/examples/simpledma/common/** directory. Files specific to a model are located in the **hdl/vhdl/examples/simpledma/*<model>*/** directory.

5.8.3 Generating PlanAhead Projects

PlanAhead project files (file extension **.ppr**) for the **SimpleDMA** FPGA design are generated using the **genppr.tcl** TCL script that can be found in **hdl/vhdl/examples/**. Generating **PlanAhead** projects is described in general in Section 5.3 - "Generating PlanAhead Projects". To generate **PlanAhead** projects for the **SimpleDMA** design, the **pattern** passed to the **genppr.tcl** script should be prefixed by **simpledma-**. Some examples:

1. To generate a **PlanAhead** project in Windows for an ADM-XRC-6T1 fitted with a 6VLX240T device, for the **SimpleDMA** example design, start a shell and run the settings32.bat or settings64.bat script from the ISE tools (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl simpledma-admxrc6t1-6vlx240t
   ```

   Under Linux, start a shell, source the settings32.sh or settings64.sh script (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'simpledma-admxrc6t1-6vlx240t'
   ```

2. To generate **PlanAhead** projects in Windows for an ADM-XRC-6T1 for all supported devices, use these commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl simpledma-admxrc6t1-.+
   ```

   Under Linux, use these commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'simpledma-admxrc6t1-.+'
   ```

Note: the single quotes around the patterns in the Linux examples prevent the shell from attempting to perform globing on the pattern.

PlanAhead project files generated using **genppr.tcl** are created in the example design's **planahead** directory. For example, **hdl/vhdl/examples/simpledma/planahead/** contains all of the **PlanAhead** projects for the **SimpleDMA** example design. Building a design or running a simulation using **ISIM** is performed in the normal way via the **PlanAhead** GUI.

5.8.4 Design Synthesis and Bitstream Build

5.8.4.1 VHDL Source Files

For a complete list of the source files used during synthesis, refer to the appropriate XST project file located in the model design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1.prj** for an ADM-XRC-6T1.

5.8.4.2 XST Files

XST Project files (**.prj**) are located in the model design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1.prj** for an ADM-XRC-6T1.

XST Script files (**.scr**) are located in the model design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1-6vlx240t.scr** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

XST constraint files (**.xcf**) are located in the model design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1.xcf** for an ADM-XRC-6T1.

5.8.4.3 Implementation Constraint Files

Implementation constraint files (**.ucf**) are located in the model design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1.ucf** for the ADM-XRC-6T1.

5.8.4.4 Build Using Make

Makefiles are provided for building **SimpleDMA** design bitstreams (**.bit** files) from the command line. Depending on the target passed to **NMAKE** (windows), or **GNU Make** (Linux), bitstreams can be built for a specific model-device combination, or for all supported model-device combinations.

When a **.bit** file is built, it is not automatically used by the example applications unless it is copied into the **bit/simpledma/** directory. This can be done manually, or by using the Makefile.

The Makefile can also be used to delete **.bit** files and intermediate files. This guarantees that the next time the design is built, it is from VHDL sources as opposed to beginning at some intermediate step.

The Makefile for the **SimpleDMA** design has the following targets:

Target	Class	Effect
all	build	Builds all **.bit** files for all supported model and device combinations.
bit_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices.
bit_*<model>*_*<device>*		Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>***.
install	install	Builds and installs all **.bit** files for all supported model and device combinations in the directory **bit/simpledma/**.
inst_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices and copies them to the directory **bit/simpledma/**.

Table 19 : SimpleDMA Design Makefile Targets (continued on next page)

Target	Class	Effect
inst_*<model>*_*<device>*	install	Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>*** and copies it to the directory **bit/simpledma/**.
clean	clean	Deletes all **.bit** files and intermediate build files for all supported model and device combinations (but does not delete any files from **bit/simpledma/**).
clean_*<model>*		Deletes **.bit** files and intermediate build files for the model specified by ***<model>*** for all supported devices (but does not delete any files from **bit/simpledma/**).
clean_*<model>*_*<device>*		Deletes the **.bit** file and intermediate build files for the model specified by ***<model>*** with a device specified by ***<device>*** (but does not delete any files from **bit/simpledma/**).

Table 19 : SimpleDMA Design Makefile Targets

Files that result from the build process, including **.bit** files, are placed in:

hdl/vhdl/examples/simpledma/build/*<model>*-*<device>*/

Filenames of any bitstreams built are thus of the form:

hdl/vhdl/examples/simpledma/build/*<model>*-*<device>*/simpledma-*<model>*-*<device>*.bit.

When a target of class "clean" is executed, the **build/*<model>*-*<device>*** directory is deleted, but files in **bit/simpledma/** are unaffected.

Note
Before a bitstream can be used by one of the example applications, it must be copied to **bit/simpledma/** by executing a target of class "install", or by manually copying the **.bit** file.

Some example make commands follow:

1 To perform a build of all **SimpleDMA** design bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simpledma
nmake all
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simpledma
make all
```

2 To perform a build and install the resulting bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simpledma
nmake install
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simpledma
make install
```

3 To perform a build for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simpledma
nmake bit_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simpledma
make bit_admxrc6t1_6vlx240t
```

4 To perform a build and install for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simpledma
nmake inst_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simpledma
make inst_admxrc6t1_6vlx240t
```

5 To delete all **.bit** files and intermediate build files in Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simpledma
nmake clean
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simpledma
make clean
```

6 To delete the **.bit** file and intermediate build files for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simpledma
nmake clean_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simpledma
make clean_admxrc6t1_6vlx240t
```

5.8.4.5 Build Using PlanAhead

In order to build the **SimpleDMA** design using **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.8.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to implement the design and generate a bitstream in the normal way.

Note
In **PlanAhead** 13.x, **bitgen** options are not sticky, and must be applied each time a project is opened. See Section 5.3.1 - "BITGEN Options in PlanAhead 13.x" for details about this issue. This step is **not** required for

5.8.5 Design Description

The **SimpleDMA** example FPGA design demonstrates register access available in Gen 3 Alpha Data reconfigurable computing hardware such as the ADM-XRC-6T1.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 20 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/simpledma/common/
ADM-XRC-6TL	MPTL	simpledma.vhd
ADM-XRC-6T1	MPTL	simpledma.vhd
ADM-XRC-6TGE	MPTL	simpledma.vhd
ADM-XRC-6T-ADV8	PCIe	simpledma_pcie.vhd
ADM-XRC-6T-DA1	MPTL	simpledma.vhd
ADPE-XRC-6T	MPTL	simpledma.vhd
ADPE-XRC-6T-L	MPTL	simpledma.vhd
ADM-XRC-7K1	MPTL	simpledma.vhd
ADM-XRC-7V1	MPTL	simpledma.vhd

Table 20 : Available Variants of the SimpleDMA Example Design

The design consists of:

- Clock and Reset Generation.
- Target MPTL interface, using an instance of mptl_if_target_wrap or, target PCIe interface, using an instance of pcie_if_target_wrap.
- Direct Slave Responder, implemented as a state machine.
- DMA Channel 0 Responder implemented as a state machine.

Figure 17 below shows the main elements of the **SimpleDMA** design using MPTL interface IP.

Figure 18 below shows the main elements of the **SimpleDMA** design using PCIe interface IP.

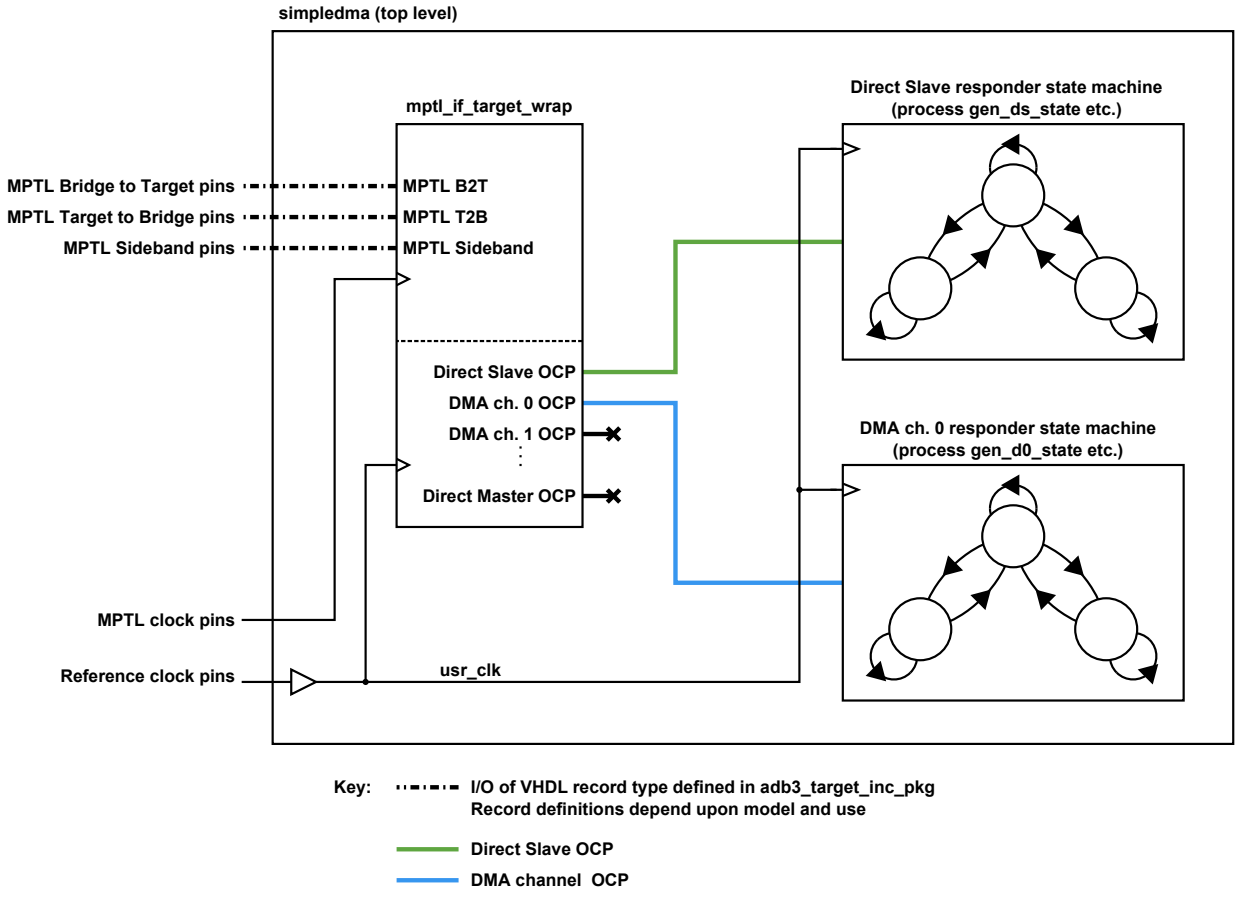

Figure 17 : SimpleDMA Design Block Diagram (MPTL)

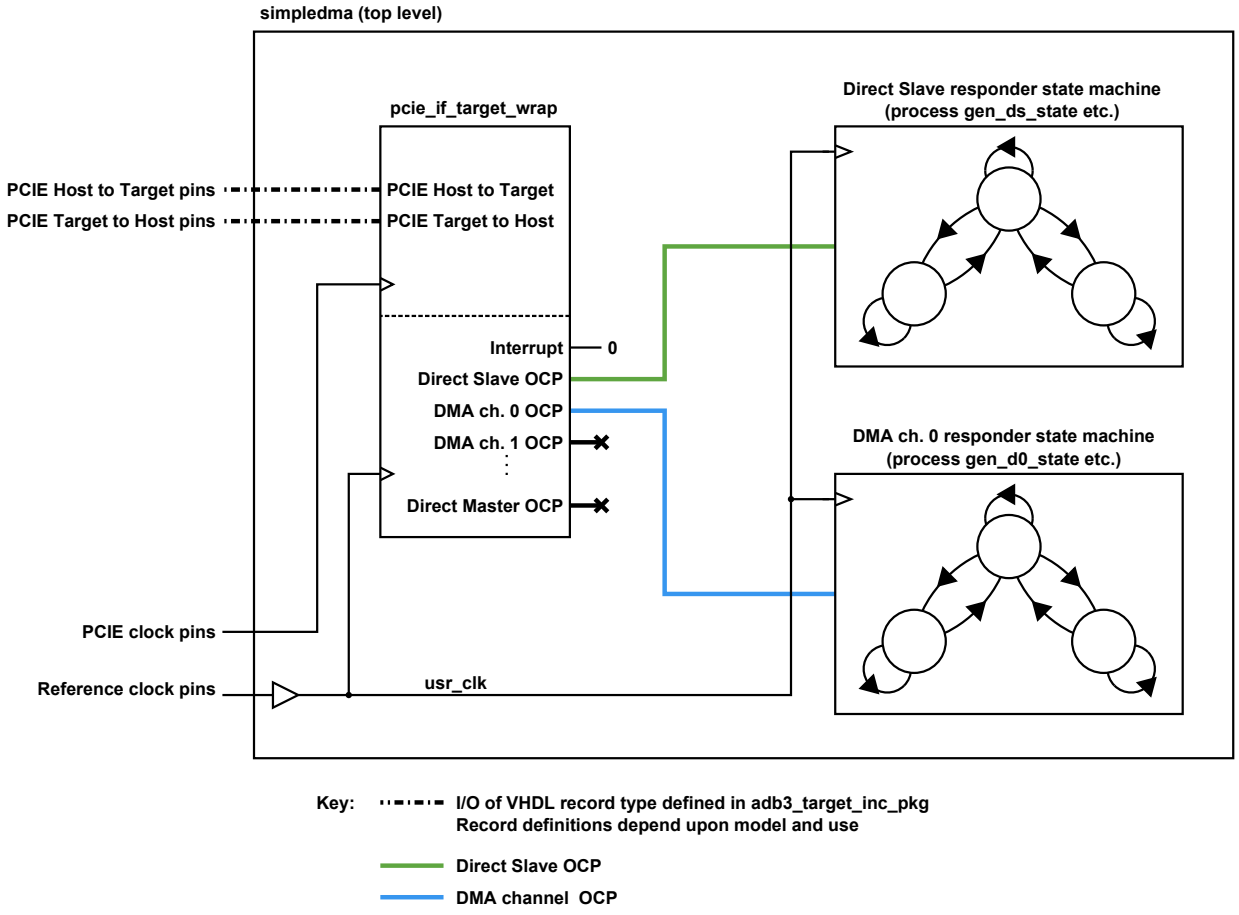

Figure 18 : SimpleDMA Design Block Diagram (PCIe)

5.8.5.1 Clock And Reset Generation

OCP Clock

- The **SimpleDMA** example design is driven by an OCP clock named **usr_clk**.
- This is a buffered version of the differential reference clock that is input via the top level **ref_clk** port.
- The actual source of the clock in the hardware depends upon the model selected, and is defined in the constraints file located in the model-specific design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1.ucf** for an ADM-XRC-6T1.

Target MPTL Interface Clock

- The target MPTL interface requires a clock to be input via its **mptl_clk** port.
- This clock input is differential and is buffered within the MPTL interface block.
- The actual source of the clock in the hardware depends upon the model selected, and is defined in the constraints file located in the model-specific design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1.ucf** for an ADM-XRC-6T1.

Target PCIe Interface Clock

- The target PCIe interface requires a clock to be input via its **pcie_clk** port.
- This clock input is differential and is buffered within the PCIe interface block.
- The actual source of the clock in the hardware depends upon the model selected, and is defined in the constraints file located in the model-specific design directory; for example, **hdl/vhdl/examples/simpledma/admxrc6t1/simpledma-admxrc6t1.ucf** for an ADM-XRC-6T1.

Global Reset

- The **SimpleDMA** example design is reset by **usr_rst** which is generated from the Target MPTL Interface (SimpleDMA) signal **mptl_ready**. Refer to the component mptl_if_target_wrap for details.

5.8.5.2 Target MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the target MPTL interface core, instantiating an MPTL to OCP interface appropriate to the model in use. The purpose of the block is to connect the MPTL to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component mptl_if_target_wrap for details.

5.8.5.3 Target PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the target PCIe interface core, instantiating a PCIe to OCP interface appropriate to the model in use. The purpose of the block is to connect the PCIe to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component pcie_if_target_wrap for details.

5.8.5.4 Direct Slave Responder

Although the **SimpleDMA** design is intended only to demonstrate how to interface to a DMA OCP channel, it is desirable that the design responds to Direct Slave OCP commands and completes them in a timely manner. This avoids timeout errors on PCI Express which, on some platforms, may be treated as fatal and cause the operating system to halt. Thus, as well as for DMA channel 0, the SimpleDMA design includes logic to respond to OCP commands delivered on the Direct Slave OCP channel.

For simplicity, the Direct Slave responder does not buffer OCP commands. Its behaviour can be summarised as follows:

- For an OCP write command (OCP_CMD_WRITE or OCP_CMD_WRITENONPOST), it accepts the command and burst length. It then discards however many words of data for the command were specified by the burst length.
- For an OCP read command (OCP_CMD_READ), it accepts the command and latches the tag and burst length associated with the command. It then returns however many words of data were specified by the burst length, with data consisting of a X"DEADC0DE" pattern, along with the latched tag.

Figure 19 illustrates the state machine at the heart of the Direct Slave responder:

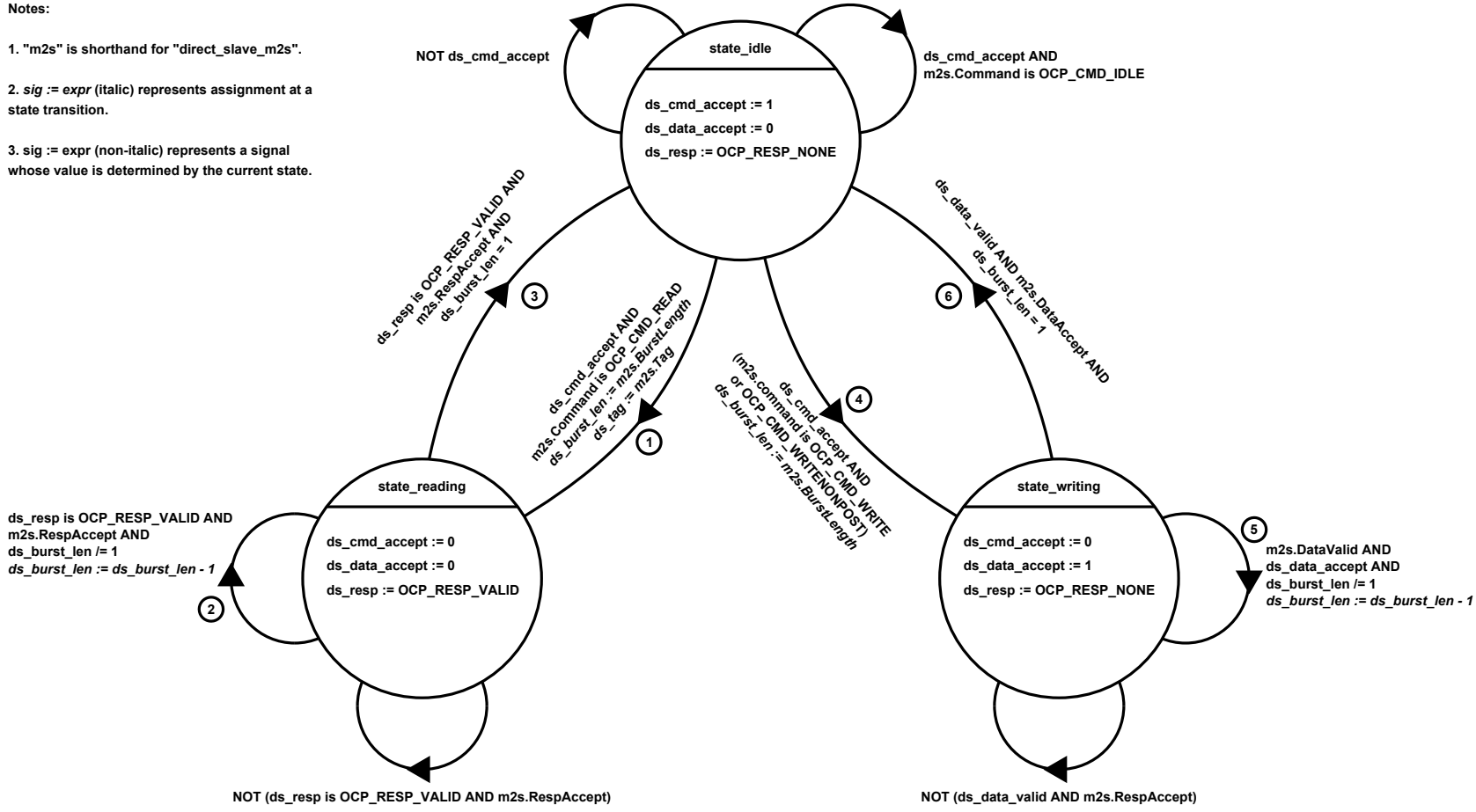

Figure 19 : SimpleDMA Direct Slave Responder State Machine

The state transitions labelled with a numbers inside a circle can be interpreted as follows:

1. This transition represents acceptance of a read command (OCP_CMD_READ) by the OCP slave (the Direct Slave responder); the state machine transitions to the **state_reading** state as it must now supply a number of words of data to the OCP master. The Direct Slave responder latches the burst length and tag, so that it can track the number of remaining words in the burst, and present the correct tag to the OCP master along with the read data.
2. This transition represents acceptance of a word of read data (and tag), presented by the OCP slave (the Direct Slave responder), by the OCP master. The current burst length is not 1, so it is not the final word of the burst and the machine remains in the **state_reading** state. The burst length is decremented.
3. This transition represents acceptance of the final word of read data (and tag) for the burst, presented by the OCP slave (the Direct Slave responder), by the OCP master. The current burst length is 1, so it is the final word of the burst and the next state is **state_idle**.
4. This transition represents acceptance of a write command (OCP_CMD_WRITE or OCP_CMD_WRITENONPOST) by the OCP slave (the Direct Slave responder); the state machine transitions to the **state_writing** state as it must now accept a number of words of data from the OCP master. The Direct Slave responder latches the burst length, so that it can track the number of remaining words in the burst.
5. This transition represents acceptance of a word of write data presented by the OCP master to the OCP slave (the Direct Slave responder). The current burst length is not 1, so it is not the final word of the burst and the machine remains in the **state_writing** state. The burst length is decremented.
6. This transition represents acceptance of the final word of write data presented by the OCP master to the OCP slave (the Direct Slave responder). The current burst length is 1, so it is the final word of the burst and the next state is **state_idle**.

5.8.5.5 DMA Channel 0 Responder

The DMA channel 0 responder section of the SimpleDMA design features a state machine that responds to commands delivered by the DMA channel 0 OCP master. Like the Direct Slave responder, the DMA channel 0 responder does not buffer OCP commands, in order to keep its complexity to a minimum. Its behaviour can be summarised as follows:

- For an OCP write command (OCP_CMD_WRITE or OCP_CMD_WRITENONPOST), it accepts the command and burst length. It then discards however many words of data for the command were specified by the burst length.
- For an OCP read command (OCP_CMD_READ), it accepts the command and latches the tag, burst length and address associated with the command. It then returns however many words of data were specified by the burst length, with data generated using the OCP address of the burst. The effect is that each 32-bit word of data presented to the Direct Slave OCP master is equal to its own 32-bit word index. For example, the 128-bit OCP word returned for a read of OCP address 0x10010 is 0x00004007_00004006_00004005_00004004.

Figure 20 illustrates the state machine at the heart of the DMA channel 0 responder:

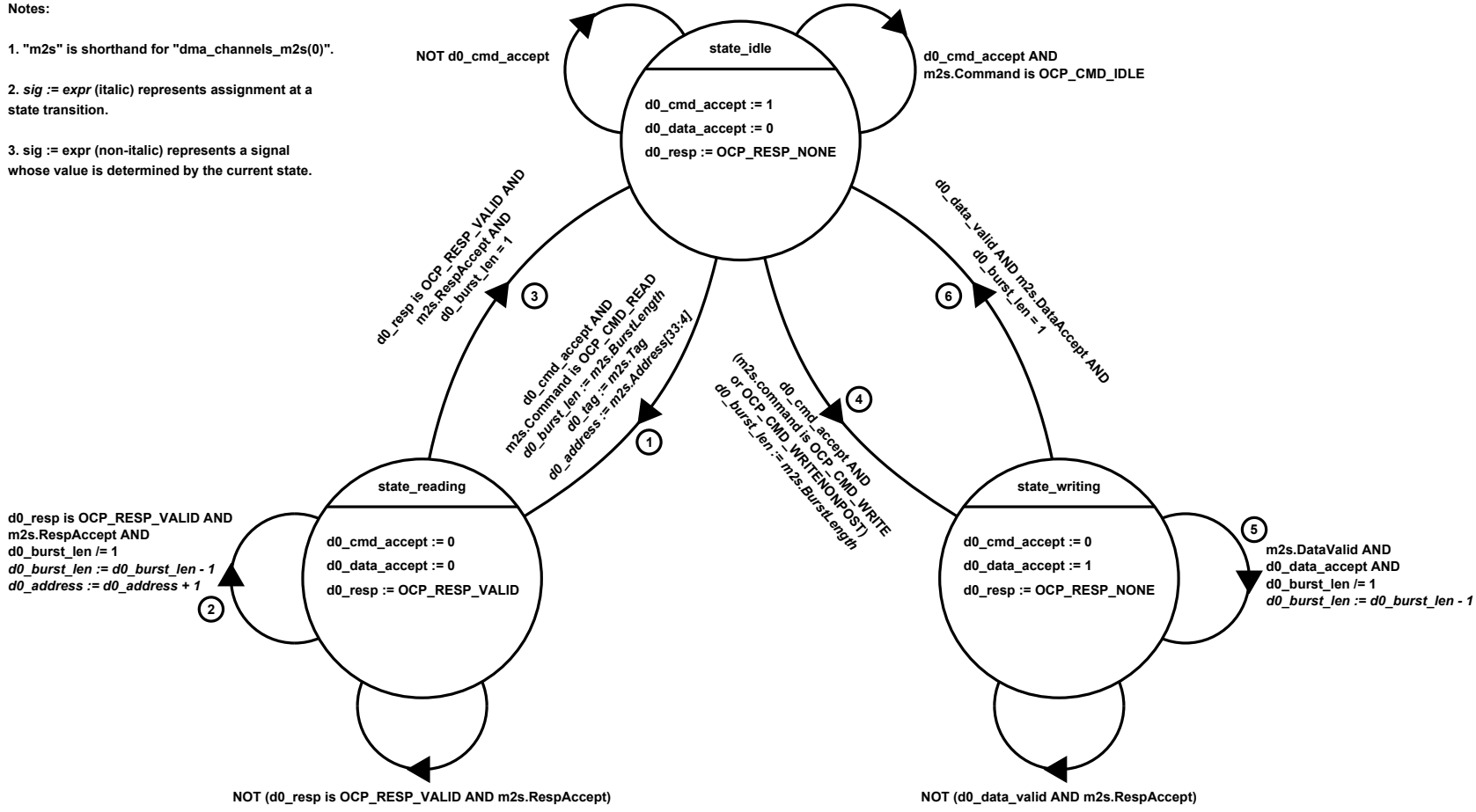

Figure 20 : SimpleDMA DMA Channel 0 Responder State Machine

The state transitions labelled with a numbers inside a circle can be interpreted as follows:

1 This transition represents acceptance of a read command (OCP_CMD_READ) by the OCP slave (the Direct Slave responder); the state machine transitions to the **state_reading** state as it must now supply a number of words of data to the OCP master. The Direct Slave responder latches the burst length, tag and address; this is so that it can track the number of remaining words in the burst, present the correct tag to the OCP master, and use the address to generate the read data.

2 This transition represents acceptance of a word of read data (and tag), presented by the OCP slave (the Direct Slave responder), by the OCP master. The current burst length is not 1, so it is not the final word of the burst and the machine remains in the **state_reading** state. The burst length is decremented and the address is incremented.

3 This transition represents acceptance of the final word of read data (and tag) for the burst, presented by the OCP slave (the Direct Slave responder), by the OCP master. The current burst length is 1, so it is the final word of the burst and the next state is **state_idle**.

4 This transition represents acceptance of a write command (OCP_CMD_WRITE or OCP_CMD_WRITENONPOST) by the OCP slave (the Direct Slave responder); the state machine transitions to the **state_writing** state as it must now accept a number of words of data from the OCP master. The Direct Slave responder latches the burst length, so that it can track the number of remaining words in the burst.

5 This transition represents acceptance of a word of write data presented by the OCP master to the OCP slave (the Direct Slave responder). The current burst length is not 1, so it is not the final word of the burst and the machine remains in the **state_writing** state. The burst length is decremented.

6 This transition represents acceptance of the final word of write data presented by the OCP master to the OCP slave (the Direct Slave responder). The current burst length is 1, so it is the final word of the burst and the next state is **state_idle**.

An isolated OCP read burst, handled by the DMA channel 0 responder, is shown in Figure 21 below:

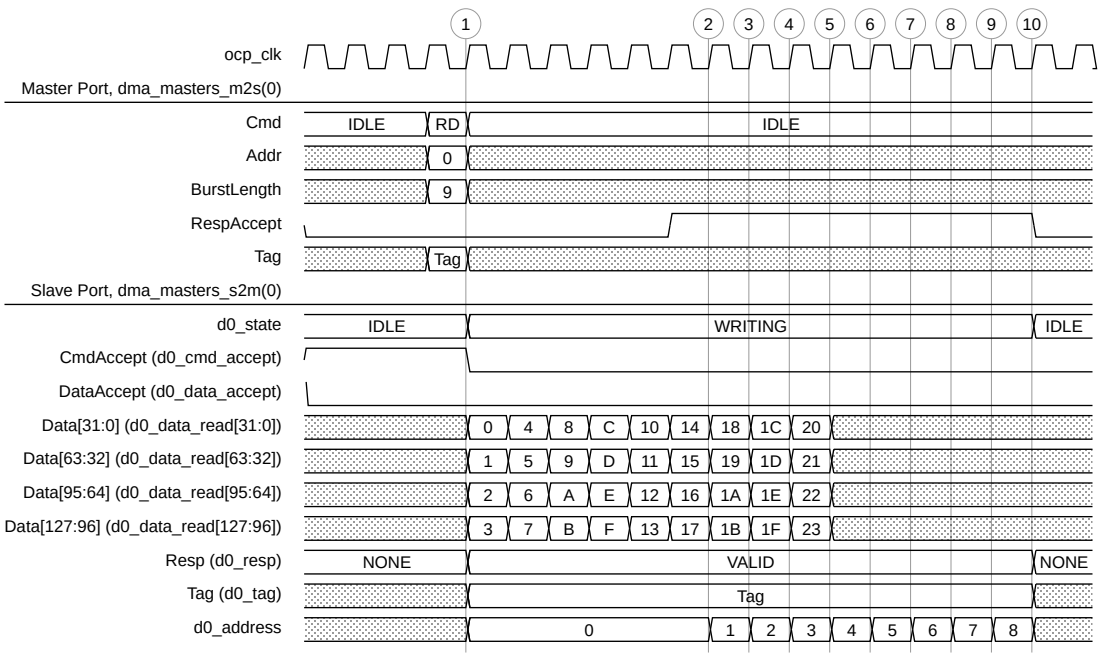

Figure 21 : SimpleDMA Typical Read Burst on DMA Channel 0

In the above timing diagram, the read command from the OCP master is accepted by the DMA channel 0

responder at clock edge 1. The condition for a command to transferred from an OCP master to an OCP slave, expressed in VHDL, is **(m2s.Cmd /= OCP_CMD_IDLE AND s2m.CmdAccept = '1')**.

Data from the DMA channel 0 responder is accepted by the OCP master at clock edges 2 to 10. The condition for a word of read data to be transferred from an OCP slave to an OCP master, expressed in VHDL, is **(s2m.Resp = OCP_RESP_VALID AND m2s.RespAccept = '1')**.

5.8.6 Testbench Description

The **SimpleDMA** example FPGA design testbench tests operation of the **SimpleDMA** example FPGA design.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 21 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/simpledma/common/
ADM-XRC-6TL	MPTL	test_simpledma.vhd
ADM-XRC-6T1	MPTL	test_simpledma.vhd
ADM-XRC-6TGE	MPTL	test_simpledma.vhd
ADM-XRC-6T-ADV8	PCIe	test_simpledma_pcie.vhd
ADM-XRC-6T-DA1	MPTL	test_simpledma.vhd
ADPE-XRC-6T	MPTL	test_simpledma.vhd
ADPE-XRC-6T-L	MPTL	test_simpledma.vhd
ADM-XRC-7K1	MPTL	test_simpledma.vhd
ADM-XRC-7V1	MPTL	test_simpledma.vhd

Table 21 : Available Variants of the SimpleDMA Example Design Testbench

It consists of the following functions:

- Clock generation for the testbench and the Unit Under Test (UUT).
- The Unit Under Test (UUT), which is the one-and-only instance of the simpledma block.
- Bridge MPTL interface block, using an instance of mptl_if_bridge_wrap or, host PCIe interface block, using an instance of pcie_if_host_wrap.
- DMA channel 0 OCP Probe, using an instance of adb3_ocp_transaction_probe.
- Stimulus Generation and Verification.

Figure 22 shows the testbench and embedded SimpleDMA FPGA design (MPTL).

Figure 23 shows the testbench and embedded SimpleDMA FPGA design (PCIe).

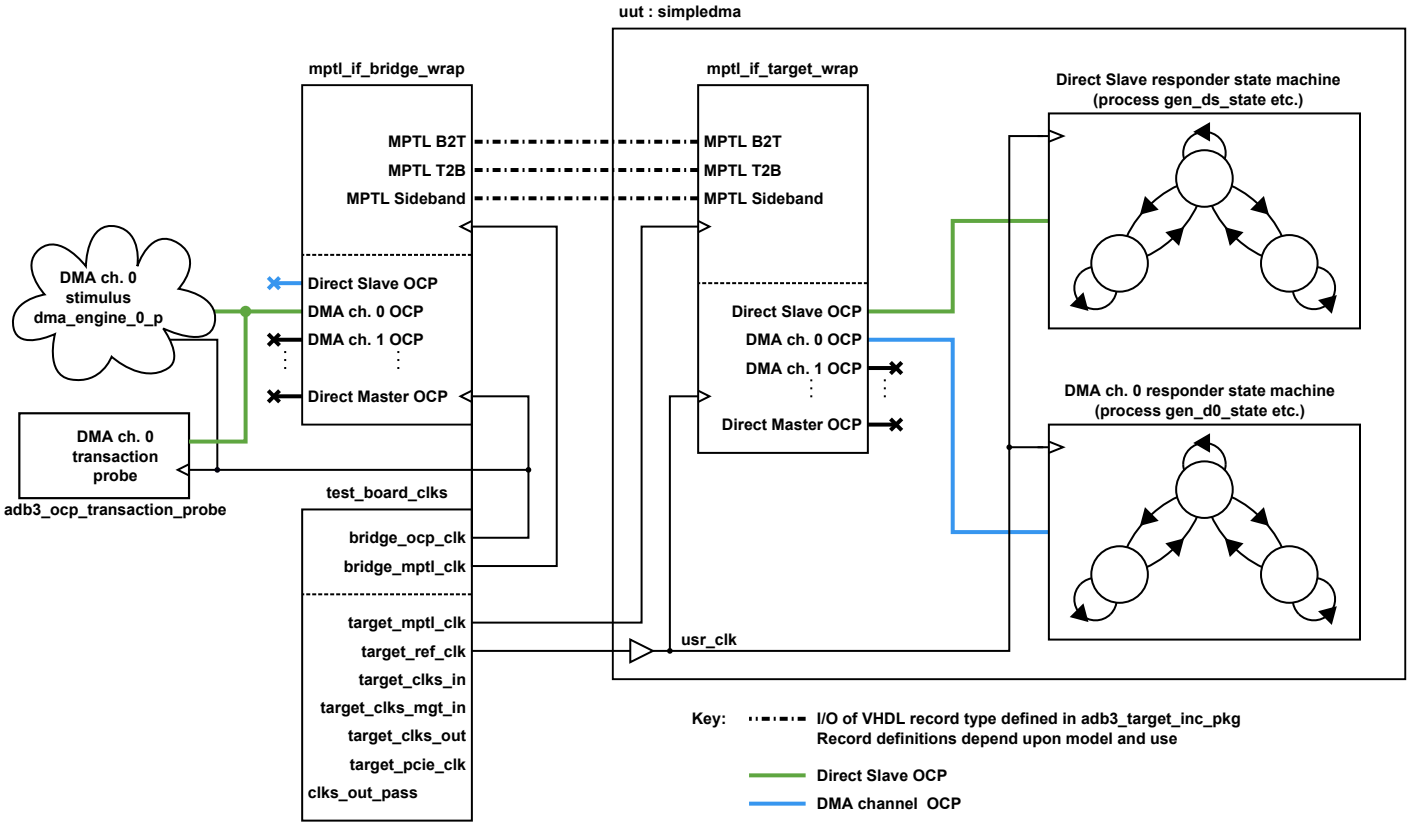

Figure 22 : SimpleDMA Design Testbench and Top Level Block Diagram (MPTL)

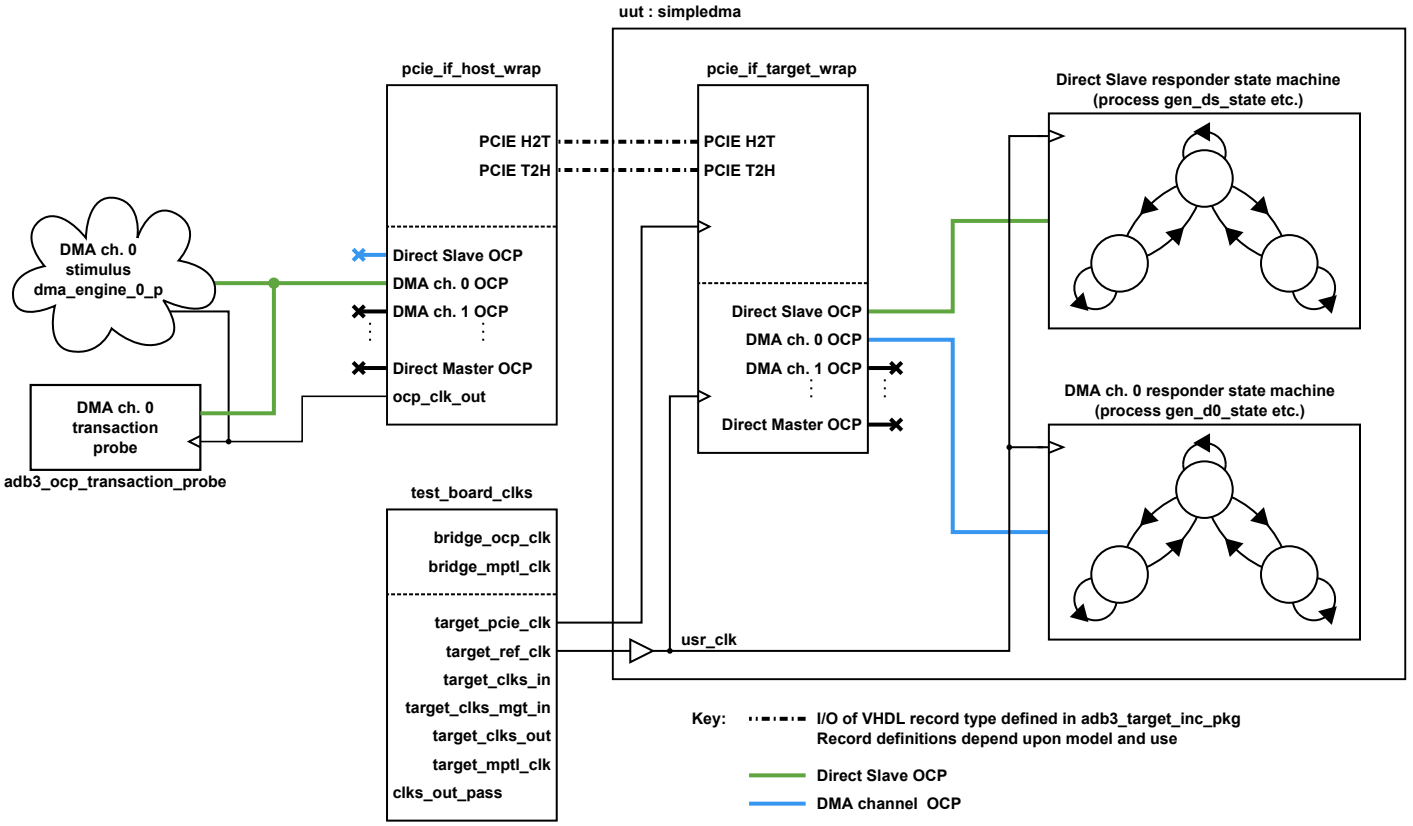

Figure 23 : SimpleDMA Design Testbench and Top Level Block Diagram (PCIe)

5.8.6.1 Clock Generation

The testbench uses the test_board_clks block to implement this function.

Target Clocks

- It generates the **ref_clk**, **target_mptl_clk** and **target_pcie_clk** clocks according to which model is selected. These clocks drive the unit under test (simpledma).

Bridge MPTL Interface Clock

- It generates the **bridge_mptl_clk** clock according to which model is selected. This clock drives the **mptl_clk** differential clock input on the bridge MPTL interface block.

Bridge OCP Clock (MPTL)

- It generates the **bridge_ocp_clk** clock according to which model is selected. This clock drives the **ocp_clk_in** clock input on the bridge MPTL interface block.
- This clock is only used during Full MPTL Simulation. Refer to bridge MPTL interface for details.

5.8.6.2 Bridge MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the bridge MPTL interface core, instantiating an OCP to MPTL interface appropriate to the model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the MPTL. Refer to the component mptl_if_bridge_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the mptl_if_bridge_wrap **mptl_b2t** signals to the mptl_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the mptl_if_target_wrap **mptl_t2b** signals to the mptl_if_bridge_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 22 as the route consisting of points 1, 2, 3 and 6.

Full MPTL simulation

- The testbench Direct Slave and DMA OCP m2s signals are input to the mptl_if_bridge_wrap.
- The UUT Direct Slave and DMA OCP m2s signals are output from the mptl_if_target_wrap.
- Apart from the packetisation, multiplexing and demultiplexing that occurs in the MPTL interfaces (both Bridge and Target), the arrangement is transparent. In other words, behaviour is as if the stimulus were applied directly to the Target FPGA's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 22 as the route consisting of points 4, 5 and 6.

The mptl_if_bridge_wrap output **mptl_online** indicates that the MPTL interface is active and stable. It is used by the testbench to generate the **mptl_online_long** signal which it monitors. Simulation will be terminated with an error message if it becomes inactive. This may occur if, for example, a protocol error arises on the MPTL signals.

5.8.6.3 Host PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the host PCIe interface core, instantiating an OCP to PCIe interface appropriate to the

model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the PCIe. Refer to the component pcie_if_host_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the pcie_if_host_wrap **pcie_h2t** signals to the pcie_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the pcie_if_target_wrap **pcie_t2b** signals to the pcie_if_host_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 23 as the route consisting of points 1, 2, 3 and 4.

5.8.6.4 DMA channel 0 OCP Probe

This function monitors OCP DMA channel 0 for addressing/transaction problems. It generates warnings/errors if it detects an illegal OCP operation. A probe error will result in a 'FAILED' **SimpleDMA** simulation result. It uses the component adb3_ocp_transaction_probe.

5.8.6.5 Stimulus Generation and Verification

This function consists of a set of processes that generate stimulus and verify the results of the simulation.

The process **dma_engine_0_p** drives OCP DMA channel 0, issuing several read and write commands. In all, it writes 256 bytes and reads 256 bytes of data, with reads and writes interleaved. After all commands have been completed, the process verifies that the expected data (a pattern of incrementing 32-bit words) has been received.

If any of the data read appears to be incorrect, the process sets the signal **simpledma_passed** to false, and then, regardless of whether or not any data was incorrect, sets the signal **simpledma_completed** to true. The process **gen_results_p** waits on **simpledma_completed** and displays a message indicating the overall success or failure of the simulation, depending on the value of **simpledma_passed**.

5.8.7 Design Simulation

5.8.7.1 Simulation Using ModelSim

For a complete list of the source files used during simulation, refer to the appropriate **ModelSim** macro file (**.do**). These are located in each of the model-specific design directories. The macro file that should be used depends upon the type of simulation required:

- OCP-Only: **hdl/vhdl/examples/simpledma/*<model>*/simpledma-*<model>*.do**
- Full MPTL: **hdl/vhdl/examples/simpledma/*<model>*/simpledma-*<model>*-mptl.do**

where ***<model>*** corresponds to the model in use; for example **admxrc6t1** for the ADM-XRC-6T1.

ModelSim simulation is initiated using the **vsim** command with the appropriate macro file. For example, to perform an OCP-Only Simulation in Windows for the ADM-XRC-6T1, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simpledma\admxrc6t1
vsim -do "simpledma-admxrc6t1.do"
```

In Linux, the commands are:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simpledma/admxrc6t1
vsim -do "simpledma-admxrc6t1.do"
```

Note
The **ModelSim** macro files always delete any previously compiled data before compiling the **SimpleDMA** design.

5.8.7.2 Simulation Using PlanAhead

In order to perform an **ISIM** simulation in **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.8.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to perform a simulation as normal.

The **ISIM** simulation set **sim_ocp** should be selected to perform an OCP-Only Simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

The **ISIM** simulation set **sim_mptl** should be selected to perform a Full MPTL Simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

5.8.7.3 Simulation Results

5.8.7.3.1 Initialisation Results

ModelSim transcript output during initialisation of the simulation is of the form described in the following subsections.

5.8.7.3.1.1 Testbench Status (MPTL)

The testbench produces a summary of the model and simulation type, and then waits for the MPTL interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_simpledma
# ** Note: Waiting for MPTL online...
#    Time: 0 ps  Iteration: 0  Instance: /test_simpledma
# ** Note: MPTL link online
#    Time: 22632 ns  Iteration: 2  Instance: /test_simpledma
```

5.8.7.3.1.2 Testbench Status (PCIe)

The testbench produces a summary of the model and simulation type, and then waits for the PCIe interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_simpledma
# ** Note: Waiting for PCIe online...
#    Time: 0 ps  Iteration: 0  Instance: /test_simpledma
# ** Note: PCIe link online
#    Time: 1285 ns  Iteration: 6  Instance: /test_simpledma
```

5.8.7.3.2 Direct Slave OCP Channel Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote 256 byte(s) to OCP address 0x0000000000001003
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: Read 256 byte(s) from OCP address 0x0000000000000104
```

```
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: Dump of data:
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note:                            +0        +4        +8        +C
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000104:  00000041  00000042  00000043  00000044
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000114:  00000045  00000046  00000047  00000048
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000124:  00000049  0000004A  0000004B  0000004C
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000134:  0000004D  0000004E  0000004F  00000050
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000144:  00000051  00000052  00000053  00000054
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000154:  00000055  00000056  00000057  00000058
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000164:  00000059  0000005A  0000005B  0000005C
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000174:  0000005D  0000005E  0000005F  00000060
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000184:  00000061  00000062  00000063  00000064
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x0000000000000194:  00000065  00000066  00000067  00000068
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x00000000000001A4:  00000069  0000006A  0000006B  0000006C
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x00000000000001B4:  0000006D  0000006E  0000006F  00000070
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x00000000000001C4:  00000071  00000072  00000073  00000074
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x00000000000001D4:  00000075  00000076  00000077  00000078
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x00000000000001E4:  00000079  0000007A  0000007B  0000007C
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: 0x00000000000001F4:  0000007D  0000007E  0000007F  00000080
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
# ** Note: Test SIMPLEDMA completed: PASSED.
#    Time: 19796 ns  Iteration: 2  Instance: /test_simpledma
```

5.8.7.3.3 Completion Results

Assuming that all tests passed, **ModelSim** transcript output on successful completion of simulation is of the form:

```
# ** Failure: Test of design SIMPLEDMA completed: PASSED.
#    Time: 19796 ns  Iteration: 4  Process: /test_simpledma/test_results_p File:
../common/test_simpledma.vhd
# Break in Process test_results_p at ../common/test_simpledma.vhd line 242
# Simulation Breakpoint: Break in Process test_results_p at
../common/test_simpledma.vhd line 242
# MACRO ./simpledma-admxrc6t1-mptl.do PAUSED at line 78
```

5.9 Uber Example FPGA Design

5.9.1 Model Support

The **Uber** FPGA design is compatible with all Virtex-6 and 7 Series models.

5.9.2 File Location

The **Uber** FPGA design is located in **hdl/vhdl/examples/uber/**. Source files common to all models are located in the **hdl/vhdl/examples/uber/common/** directory. These include the design and testbench top levels.

5.9.3 Generating PlanAhead Projects

PlanAhead project files (file extension **.ppr**) for the **Uber** FPGA design are generated using the **genppr.tcl** TCL script that can be found in **hdl/vhdl/examples/**. Generating **PlanAhead** projects is described in general in Section 5.3 - "Generating PlanAhead Projects". To generate **PlanAhead** projects for the **Uber** design, the **pattern** passed to the **genppr.tcl** script should be prefixed by **uber-**. Some examples:

1. To generate a **PlanAhead** project in Windows for an ADM-XRC-6T1 fitted with a 6VLX240T device, for the **Uber** example design, start a shell and run the settings32.bat or settings64.bat script from the ISE tools (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl uber-admxrc6t1-6vlx240t
   ```

   Under Linux, start a shell, source the settings32.sh or settings64.sh script (this ensures that the environment is correct for running the ISE tools), and then issue the following commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'uber-admxrc6t1-6vlx240t'
   ```

2. To generate **PlanAhead** projects in Windows for an ADM-XRC-6T1 for all supported devices, use these commands:

   ```
   cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
   xtclsh genppr.tcl uber-admxrc6t1-.+
   ```

   Under Linux, use these commands:

   ```
   cd $ADMXRC3_SDK/hdl/vhdl/examples
   xtclsh genppr.tcl 'uber-admxrc6t1-.+'
   ```

Note: the single quotes around the patterns in the Linux examples prevent the shell from attempting to perform globing on the pattern.

PlanAhead project files generated using **genppr.tcl** are created in the example design's **planahead** directory. For example, **hdl/vhdl/examples/uber/planahead/** contains all of the **PlanAhead** projects for the **Uber** example design. Building a design or running a simulation using **ISIM** is performed in the normal way via the **PlanAhead** GUI.

5.9.4 Design Synthesis and Bitstream Build

5.9.4.1 VHDL Source Files for Synthesis

For a complete list of the source files used during synthesis, refer to the appropriate XST project file located in the model design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1-6vlx240t.prj** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

5.9.4.2 XST Files

XST Project files (**.prj**) are located in the model design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1-6vlx240t.prj** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

XST Script files (**.scr**) are located in the model design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1-6vlx240t.scr** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

XST constraint files (**.xcf**) are located in the model design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1.xcf** for an ADM-XRC-6T1.

5.9.4.3 Implementation Constraint Files

Implementation constraint files (**.ucf**) are located in the model design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1-6vlx240t.ucf** for the ADM-XRC-6T1 with a 6VLX240T device.

5.9.4.4 Build Using Make

Makefiles are provided for building the **Uber** design bitstreams (**.bit** files). Depending on the target passed to **NMAKE** or **GNU Make**, for Windows and Linux hosts respectively, bitstreams can be built for a specific model-device combination, or for all supported model-device combinations.

When a **.bit** file is built, it is not automatically used by the example applications unless it is copied into the **bit/uber/** directory. This can be done manually, or by using the Makefile.

The Makefile can also be used to delete **.bit** files and intermediate files, so that the next time the design is built, it is guaranteed to be built from VHDL sources as opposed to beginning at some intermediate step.

Note
Before performing the first bitstream build of **Uber**, HDL files for the Xilinx DDR3 SDRAM Memory Interface Generator (MIG) core must be generated using a TCL script. Refer to DDR3 SDRAM MIG Cores for details.

Note
Changing the constant **CHIPSCOPE_ON** in **hdl/vhdl/examples/uber/common/uber.vhd** from **false** to **true** causes a ChipScope block to be included when building the **Uber** design. Before performing the first bitstream build of **Uber** with **CHIPSCOPE_ON** set to **true**, the ChipScope ILA core **chipscope_ila.ngc** and ICON core **chipscope_icon.ngc** must be generated using the script **gen_chipscope.tcl**. Refer to Xilinx ChipScope Core Generation (ICON/ILA) for details.

The Makefile for the **Uber** design has the following targets:

Target	Class	Effect
all	build	Builds all **.bit** files for all supported model and device combinations.
bit_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices.
bit_*<model>*_*<device>*		Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>***.
install	install	Builds and installs all **.bit** files for all supported model and device combinations in the directory **bit/uber/**.
inst_*<model>*		Builds **.bit** files for the model specified by ***<model>*** for all supported devices and copies them to the directory **bit/uber/**.
inst_*<model>*_*<device>*		Builds the **.bit** file for the model specified by ***<model>*** with a device specified by ***<device>*** and copies it to the directory **bit/uber/**.
clean_*<model>*		Deletes **.bit** files and intermediate build files for the model specified by ***<model>*** for all supported devices (but does not delete any files from **bit/uber/**).
clean	clean	Deletes all **.bit** files and intermediate build files for all supported model and device combinations (but does not delete any files from **bit/uber/**).
clean_*<model>*_*<device>*		Deletes the **.bit** file and intermediate build files for the model specified by ***<model>*** with a device specified by ***<device>*** (but does not delete any files from **bit/uber/**).

Table 22 : Uber Design Makefile Targets (continued on next page)

Target	Class	Effect
	clean	

Table 22 : Uber Design Makefile Targets

Files that result from the build process, including **.bit** files, are placed in:

hdl/vhdl/examples/uber/build/*<model>-<device>*/

Filenames of any bitstreams built are thus of the form:

hdl/vhdl/examples/uber/build/*<model>-<device>*/uber-*<model>-<device>*.bit.

When a target of class "clean" is executed, the **build/*<model>-<device>*** directory is deleted, but files in **bit/uber/** are unaffected.

Note
Before a bitstream can be used by one of the example applications, it must be copied to **bit/uber/** by executing a target of class "install", or by manually copying the **.bit** file.

Some example make commands follow:

1 To perform a build of all **Uber** design bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake all
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make all
```

2 To perform a build and install the resulting bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake install
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make install
```

3 To perform a build for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake bit_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make bit_admxrc6t1_6vlx240t
```

4 To perform a build and install for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake inst_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make inst_admxrc6t1_6vlx240t
```

5 To delete all **.bit** files and intermediate build files in Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake clean
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make clean
```

6 To delete the **.bit** file and intermediate build files for an ADM-XRC-6T1 fitted with a 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake clean_admxrc6t1_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make clean_admxrc6t1_6vlx240t
```

5.9.4.5 Build Using PlanAhead

In order to build the **Uber** design using **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.9.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to implement the design and generate a bitstream in the normal way.

> **Note**
> In **PlanAhead** 13.x, **bitgen** options are not sticky, and must be applied each time a project is opened. See Section 5.3.1 - "BITGEN Options in PlanAhead 13.x" for details about this issue. This step is **not** required for PlanAhead 14.x as it remembers **bitgen** options.

5.9.4.6 Date/Time Package Generation

In the **Make** design flow, if the **makefile** is required to run XST during bitstream build, it will first run the TCL script **hdl/vhdl/examples/uber/gen_today_pkg.tcl** to generate a file containing the **today_pkg** package. This package defines HDL constants containing the SDK version and date/time at which the script was run. The name of the generated file depends upon the model selected and it is located in the model design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/today_pkg_admxrc6t1_6vlx240t.vhd** for the ADM-XRC-6T1 with a 6VLX240T device.

In the **PlanAhead** design flow, the **PlanAhead** project file generation scripts will run the TCL script **hdl/vhdl/examples/uber/gen_today_pkg.tcl** to generate a file containing the **today_pkg** package. This package defines HDL constants containing the SDK version and date/time at which the script was run. The name of the generated file depends upon the model selected and it is located in the **Uber** design **planahead** directory; for example, **hdl/vhdl/examples/uber/planahead/admxrc6t1/today_pkg_admxrc6t1_6vlx240t.vhd** for the

ADM-XRC-6T1 with a 6VLX240T device.

Script output is of the form:

```
--
-- This file was generated automatically by gen_today_pkg.tcl
--
-- SDK:  01.04.00 (Maj/Min/Bld)
-- Date: 08/10/2010 (dd/mm/YYYY)
-- Time: 15:26:46 (HH/MM/SS)
--

library ieee;
use ieee.std_logic_1164.all;

package today_pkg is

  constant SDK_VERSION : std_logic_vector(31 downto 0) := X"00010500";
  constant TODAYS_DATE : std_logic_vector(31 downto 0) := X"08102011";
  constant TODAYS_TIME : std_logic_vector(31 downto 0) := X"15264600";

end package today_pkg;
```

Note
The makefile runs the TCL script using the Xilinx customized TCL distribution TCL shell **xtclsh**. The path to this shell must be defined before initiating synthesis.

5.9.5 Design Description

The **Uber** example FPGA design demonstrates functionality available in Gen 3 Alpha Data reconfigurable computing hardware such as the ADM-XRC-6T1.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 23 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/uber/common/
ADM-XRC-6TL	MPTL	uber.vhd
ADM-XRC-6T1	MPTL	uber.vhd
ADM-XRC-6TGE	MPTL	uber.vhd
ADM-XRC-6T-ADV8	PCIe	uber_pcie.vhd
ADM-XRC-6T-DA1	MPTL	uber.vhd
ADPE-XRC-6T	MPTL	uber.vhd
ADPE-XRC-6T-L	MPTL	uber.vhd
ADM-XRC-7K1	MPTL	uber_s7.vhd
ADM-XRC-7V1	MPTL	uber_s7.vhd

Table 23 : Available Variants of the Uber Example Design

The design includes the following functional areas:

- Clock and reset generation (blk_clocks).
- Target MPTL interface, using an instance of mptl_if_target_wrap or, Target PCIe interface, using an instance of pcie_if_target_wrap.

- OCP Direct Slave block (blk_direct_slave), which includes:
- Direct Slave address space splitter
- Direct Slave clock domain interface, between the **pll_pri_clk** domain and the relatively low frequency **pll_reg_clk** domain.
- Direct Slave register address space splitter
- Simple test register block (blk_ds_simple_test)
- Clock frequency measurement register block (blk_ds_clk_read)
- GPIO test register block (blk_ds_io_test)
- Interrupt test register block (blk_ds_int_test)
- Informational register block (blk_ds_info), including build datestamp and build timestamp
- On-board memory control and status register block (blk_ds_mem_reg)
- Direct Slave access to BRAM
- Direct Slave access to on-board memory
- OCP switching block (blk_dma_switch)
- BRAM block (blk_bram)
- On-board memory interface block (blk_mem_if)
- On-board memory application block (blk_mem_app)
- Optional ChipScope connection block (blk_chipscope)

Figure 24 shows the main elements of the **Uber** design using MPTL interface IP.

Figure 25 shows the main elements of the **Uber** design using PCIe interface IP.

Figure 26 shows the hierarchy of the **Uber** design using MPTL interface IP.

Figure 27 shows the hierarchy of the **Uber** design using PCIe interface IP.

The **Uber** design includes the following packages:

- ADB3 OCP profile definition package (adb3_ocp)
- ADB3 target include package (adb3_target_inc_pkg)
- Design package (uber_pkg)

Figure 28 shows the design package dependencies.

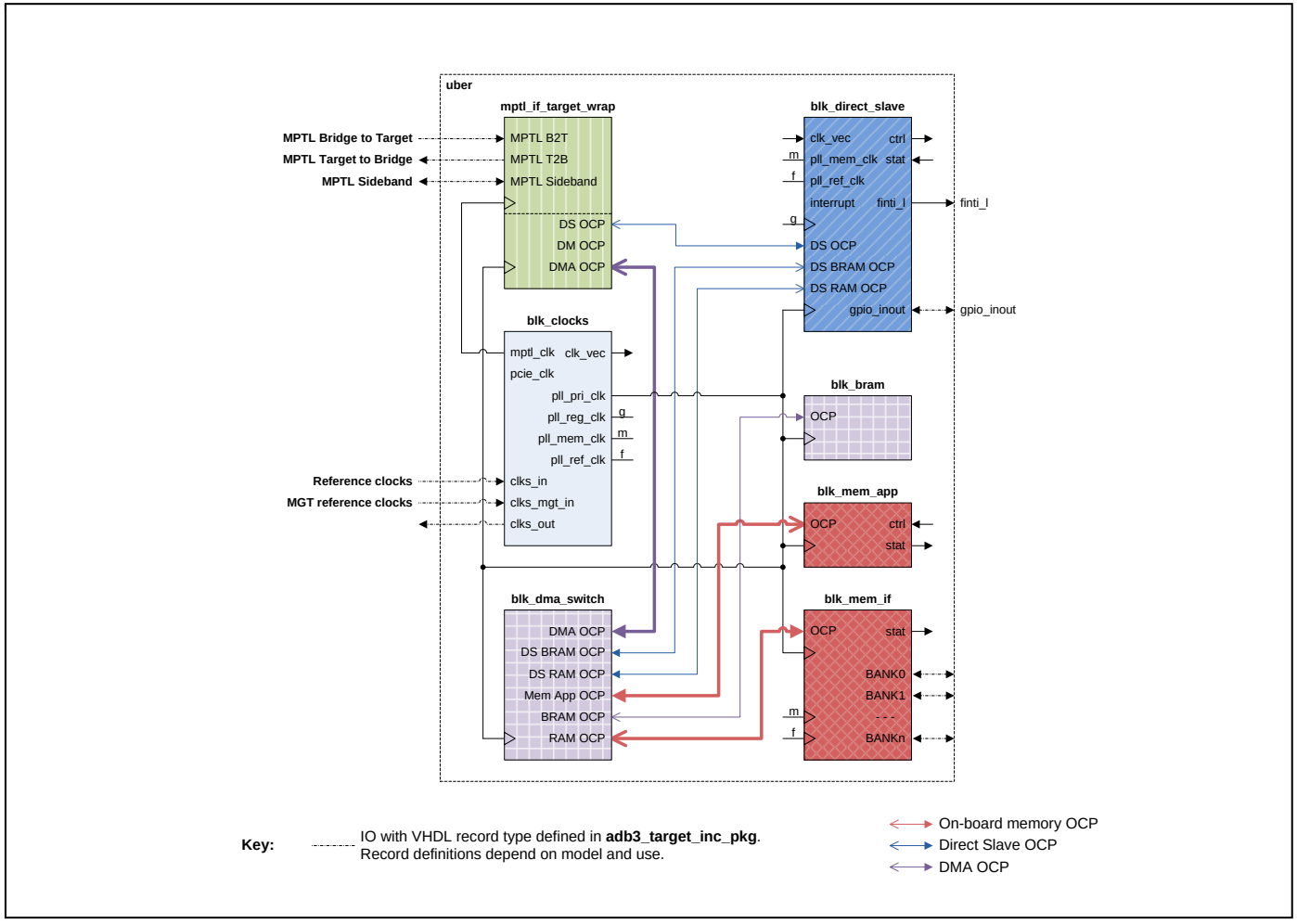

Figure 24 : Uber Design Top Level Block Diagram (MPTL)

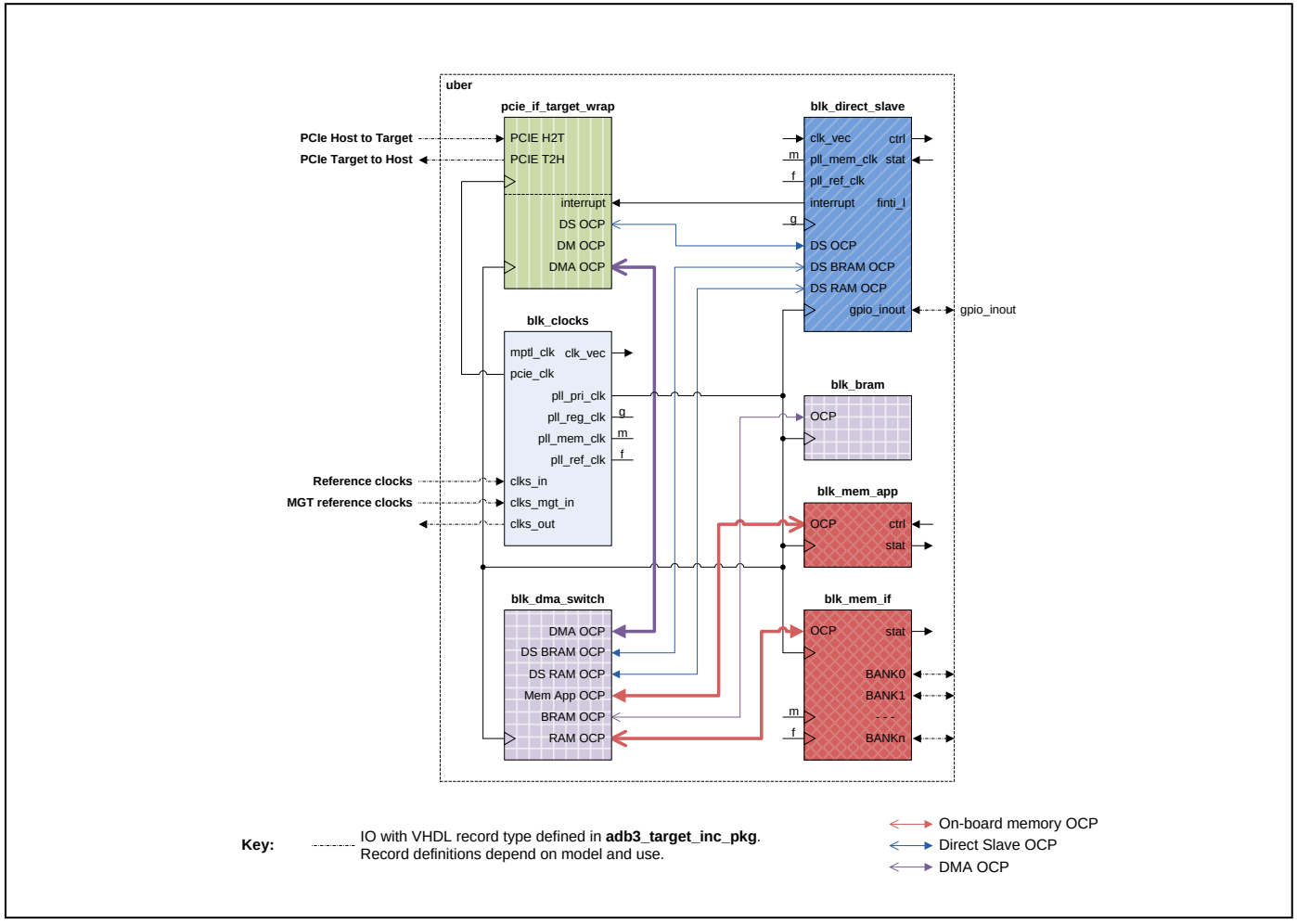

Figure 25 : Uber Design Top Level Block Diagram (PCIe)

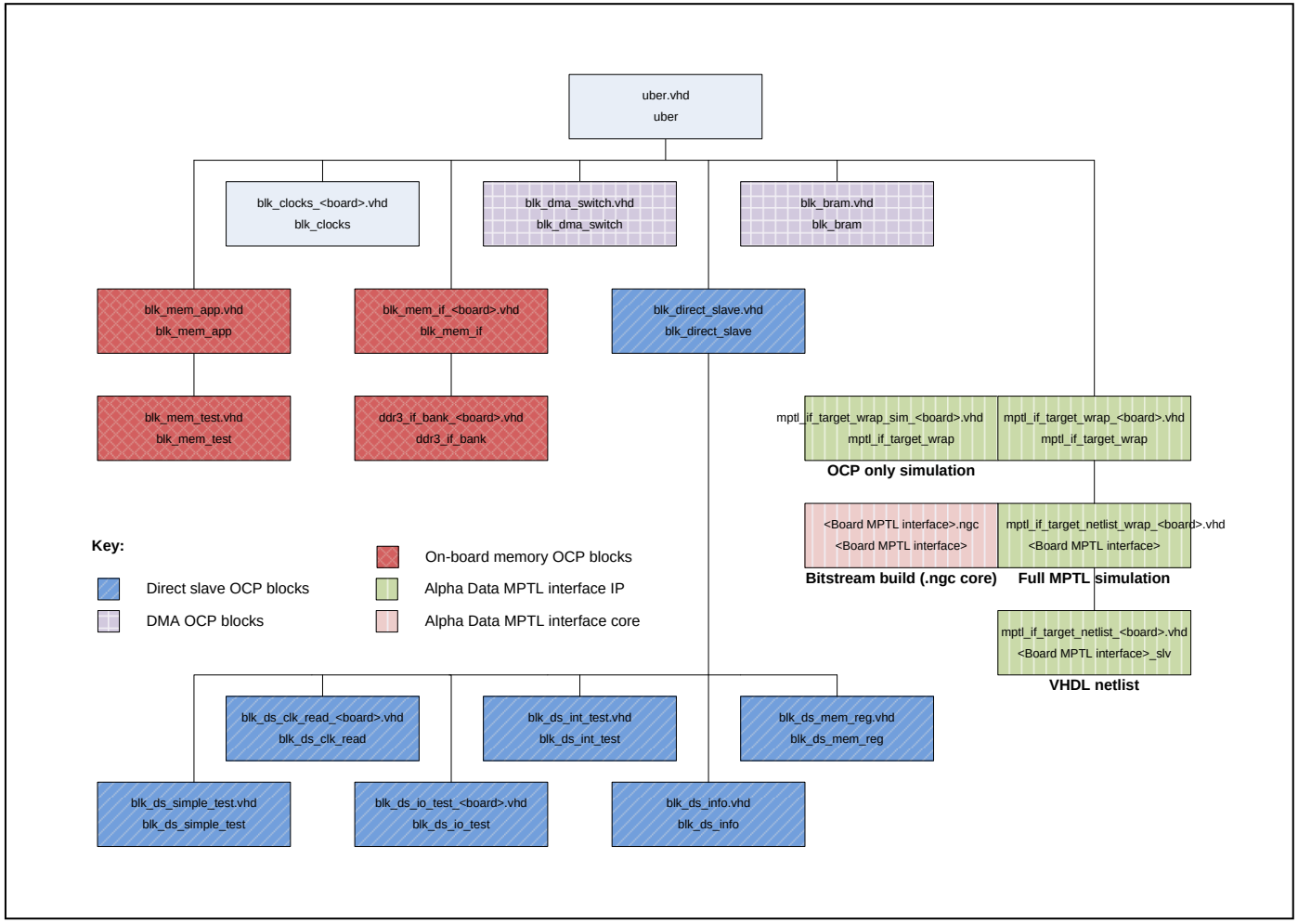

Figure 26 : Uber Design Top Level Hierarchy (MPTL)

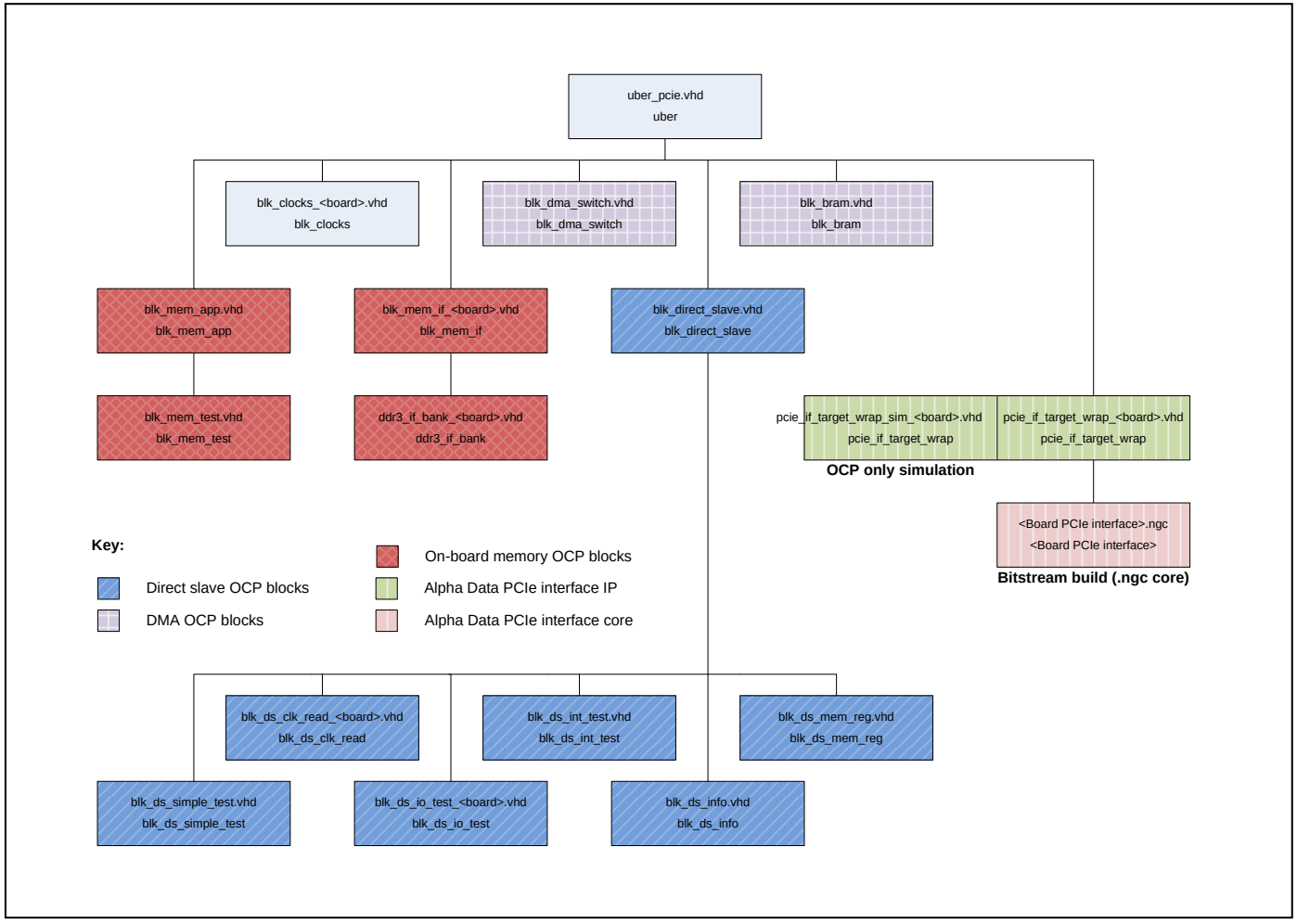

Figure 27 : Uber Design Top Level Hierarchy (PCIe)

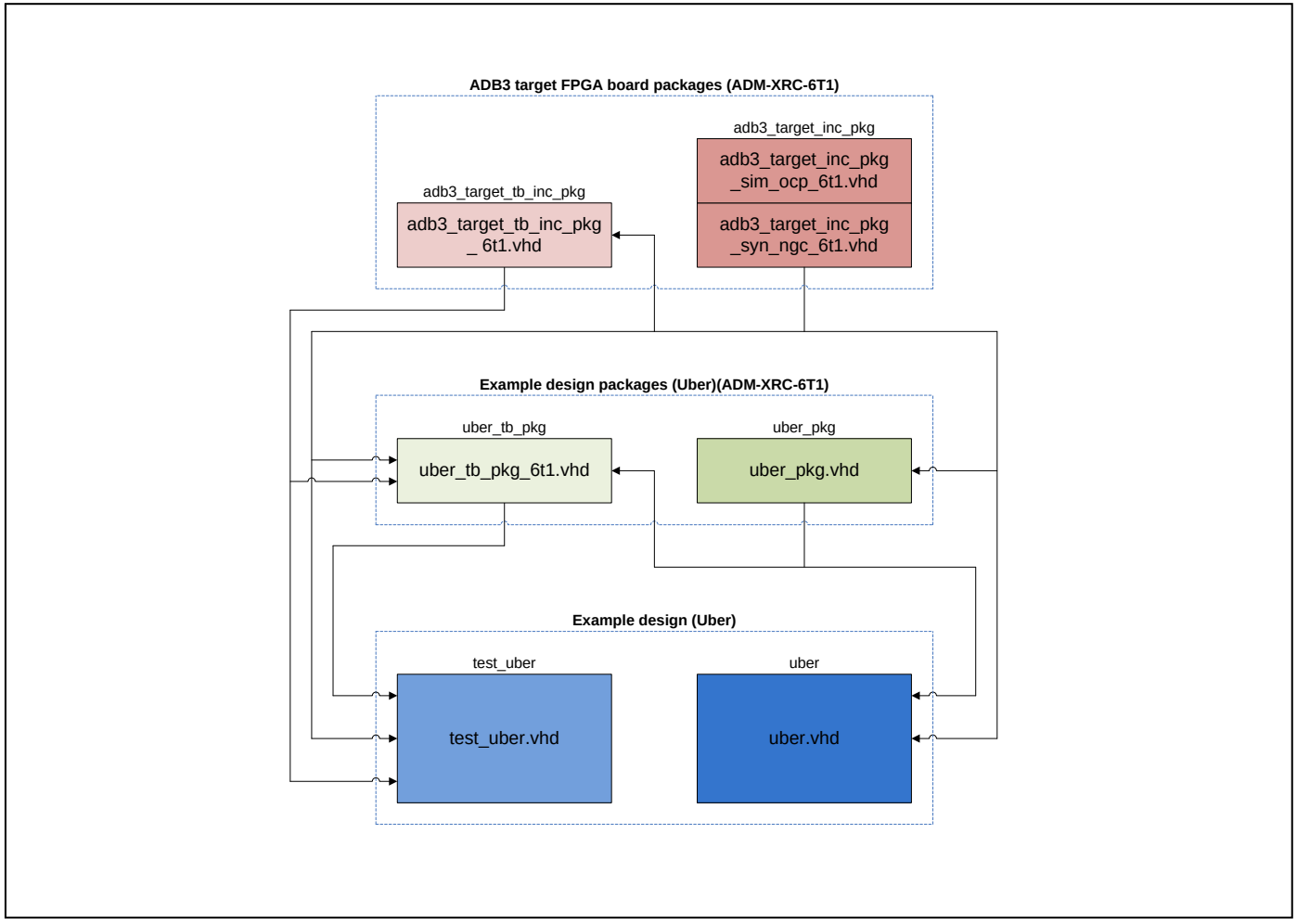

Figure 28 : Uber Design Package Dependencies

5.9.5.1 Clock And Reset Generation

The clock and reset generation block is implemented by the **blk_clks** block which is model dependent.

Table 24 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-6TL	admxrc6tl/blk_clks_6tl.vhd
ADM-XRC-6T1	admxrc6t1/blk_clks_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/blk_clks_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/blk_clks_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/blk_clks_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/blk_clks_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/blk_clks_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/blk_clks_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/blk_clks_admxrc7v1.vhd

Table 24 : Available Variants of blk_clks Block

blk_clks includes the following functional areas:

- Internal clock generation (MMCM)
- Internal reset generation (MMCM)
- MPTL interface clock generation
- PCIe interface clock generation
- Input clock buffering
- Input clock extraction (MGT sourced)
- Output clock generation

5.9.5.1.1 Internal Clock Generation (MMCM)

This consists of an instantiation of a Xilinx MMCM block. The MMCM is reset on power on using the **pll_reset** signal, and clocked by a buffered version of the **clks_in.ref_clk** global clock input. It generates several output clocks depending on the model in use. Refer to Figure 29.

pll_ref_clk

- This clock is generated from a buffered version of the **clks_in.ref_clk** global clock input.
- It is used as a reference clock by the design.
- On all current models it is fixed at 200 MHz.
- It is used by the clock frequency measurement section to measure the frequencies of other clocks in the design.
- It is used as the reference clock for the IODELAYCTRL instances in the DDR3 SDRAM interface.

pll_pri_clk

- This clock is generated from a buffered version of the MMCM **out_clk0** output.
- It is used as the primary OCP clock by the design.
- On the ADM-XRC-6T-ADV8 it is generated at 100MHz. On all other models it is generated at 200 MHz.
- It is used to clock much of the OCP logic in the **Uber** design, including the DMA OCP section.

- Its frequency need not be related to any of the other clocks.

pll_reg_clk

- This clock is generated from a buffered version of the MMCM **out_clk1** output.
- It is used as a low frequency clock by the design.
- On all current models it is generated at 80 MHz.
- It is used to clock the low-frequency OCP logic in the **Uber** design, including the direct slave register section.
- Its frequency need not be related to any of the other clocks.

pll_mem_clk (Virtex-6 models)

- This clock is generated from a buffered version of the MMCM **out_clk2** output.
- It is used as the DDR3 SDRAM memory interface clock by the design.
- On Virtex-6 models it is generated at 400 MHz.
- Its frequency need not be related to any of the other clocks.

pll_mem_clk (7 Series models)

- On 7 Series models, this is an array of clocks, one for each DDR3 SDRAM interface.
- These clocks are generated from buffered versions of the **clks_in.ddr3_clk** global clock inputs.
- They are used as the DDR3 SDRAM memory interface clocks by the design.
- On 7 Series models they are fixed at 200 MHz.
- Their frequency need not be related to any of the other clocks.

5.9.5.1.2 Internal Reset Generation (MMCM)

An active high asynchronous reset **pll_rst** is generated from the MMCM locked signal. Refer to Figure 29.

In models which use the Target MPTL interface, the **pll_rst** signal is used as its **ocp_ready** input. The **Uber** example design is reset by **rst** which is generated from the Target MPTL Interface **mptl_ready** output.

In models which use the Target PCIe interface, the **pll_rst** signal is combined with the **pcie_rst_l** FPGA input to generate the **Uber** example design reset **rst**.

5.9.5.1.3 MPTL Interface Clock Generation

The target MPTL interface requires an unbuffered differential MGT clock input. It is provided by the **mptl_clk** signal which is sourced from a clock in the **clks_mgt_in** input record. Its source is dependent on the model selected. Refer to Figure 30.

5.9.5.1.4 PCIe Interface Clock Generation

The target PCIe interface requires an unbuffered differential MGT clock input. It is provided by the **pcie_clk** signal which is sourced from a clock in the **clks_mgt_in** input record. Its source is dependent on the model selected. Refer to Figure 30.

5.9.5.1.5 Input Clock Buffering

Clocks are input on the **clks_in** input record of type **clks_in_t** and are buffered. Clock support is dependent on the model selected. Type **clks_in_t** is defined in the ADB3 target include package (adb3_target_inc_pkg). The buffered clocks are connected to the **clk_vec** signal. The connection order is defined by the **clk_vec_t** type in the uber_pkg package.

Refer to Figure 30.

5.9.5.1.6 Input Clock Extraction (MGT Sourced)

MGT sourced clocks are input on the **clks_mgt_in** input record of type **clks_mgt_in_t**. MGT sourced clock support is dependent on the model selected. Type **clks_mgt_in_t** is defined in the ADB3 target include package (adb3_target_inc_pkg).

The **MGT_CLKS_VALID** constant defined in the ADB3 target include package (adb3_target_inc_pkg) controls which MGT sourced clocks are extracted, converted to single-ended, and then buffered using a **BUFG**. The buffered clocks are connected to the **clk_vec** signal. The connection order is defined by the **clk_vec_t** type in the uber_pkg package.

Refer to Figure 30.

5.9.5.1.7 Output Clock Generation

Clocks are generated and output on the **clks_out** output record of type **clks_out_t**. Clock support is dependent on the model selected. Type **clks_out_t** is defined in the ADB3 target include package (adb3_target_inc_pkg).

Refer to Figure 30.

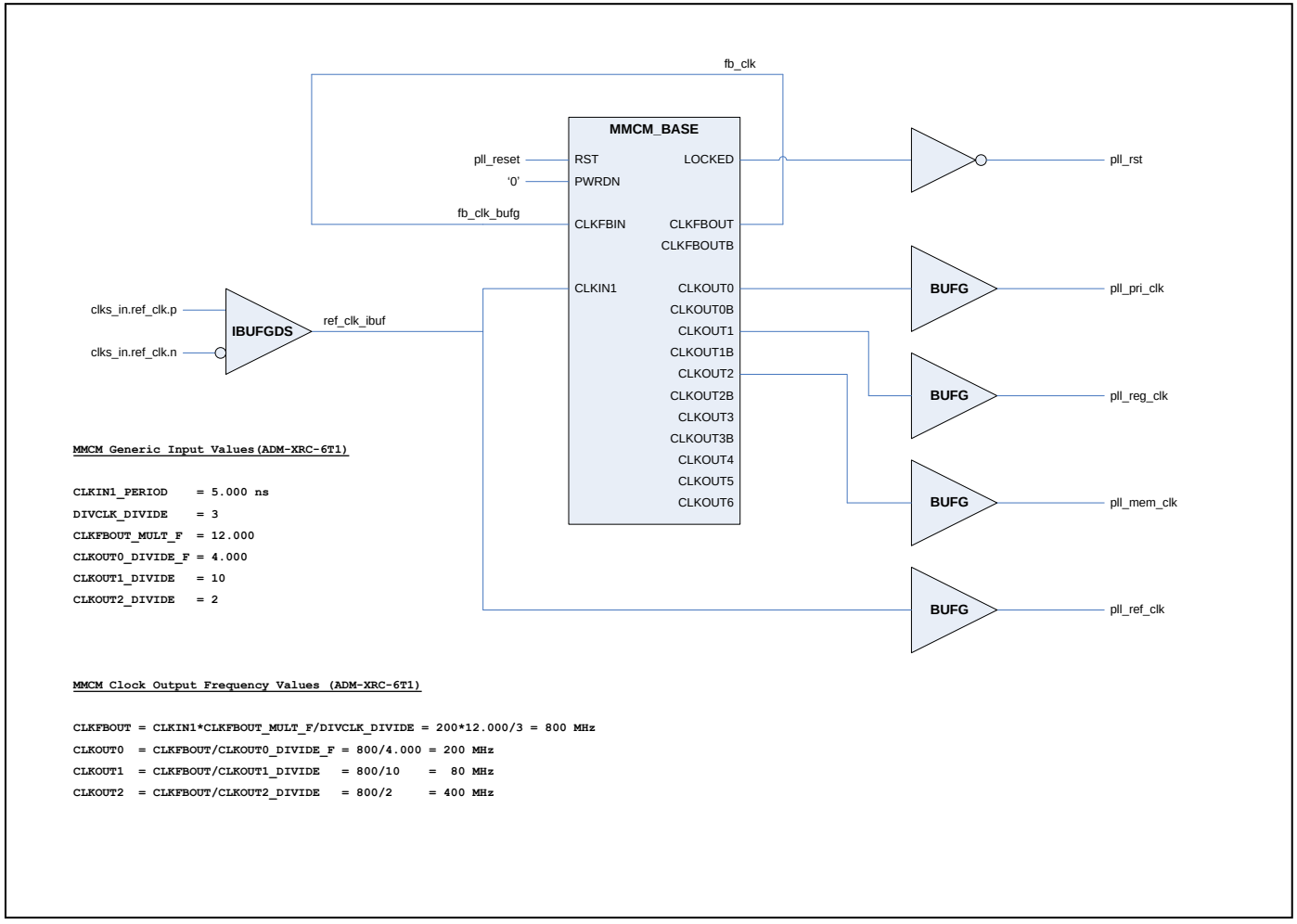

Figure 29 : Uber Design Internal Clock Generation (MMCM)

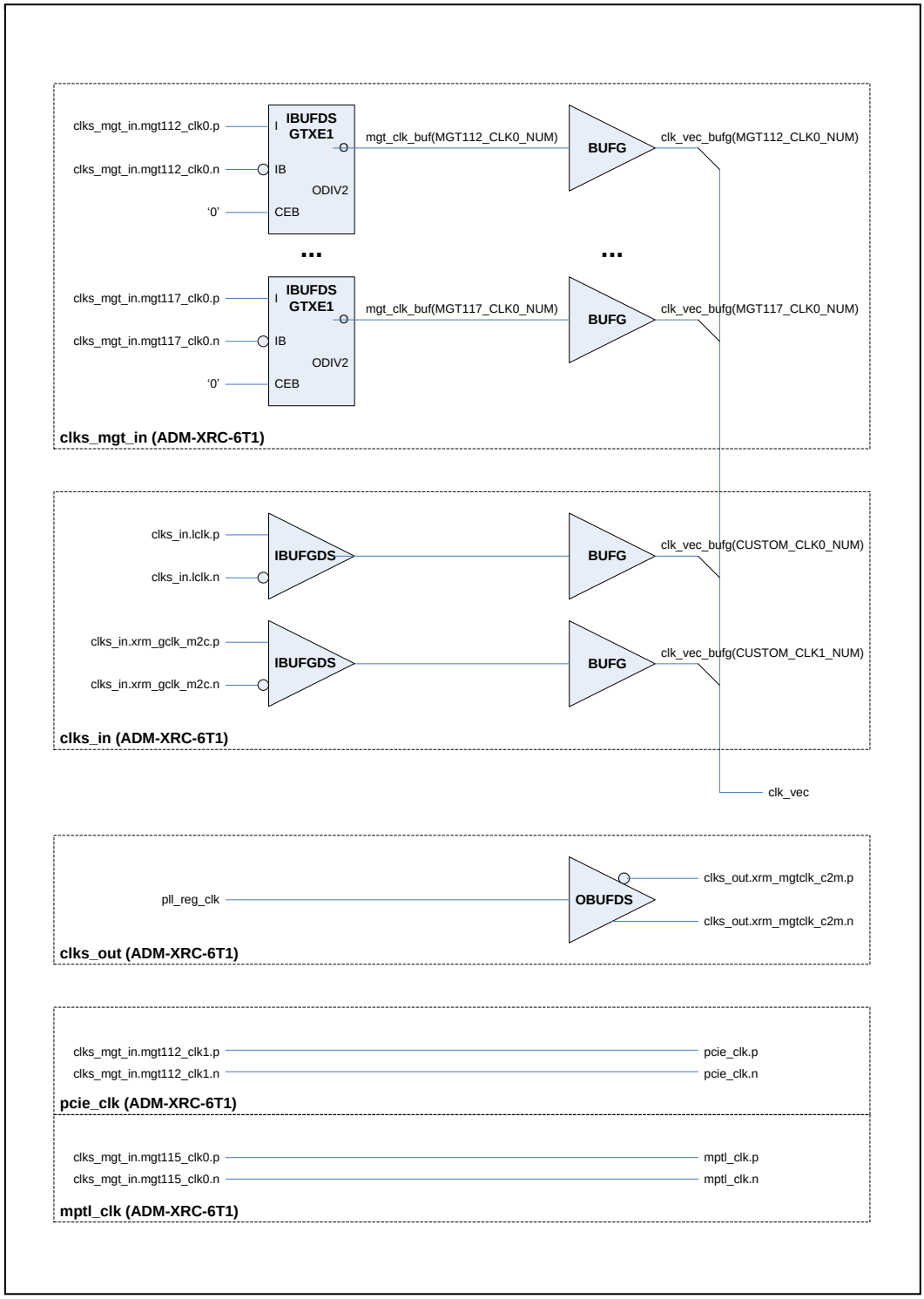

Figure 30 : Uber Design Clock Buffering/Extraction

5.9.5.2 Target MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the target MPTL interface core, instantiating an MPTL to OCP interface appropriate to the model in use. The purpose of the block is to connect the MPTL to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component mptl_if_target_wrap for details.

The **Uber** design output signal **mptl_sb_t2b.mptl_target_configured_l** indicates that the FPGA OCP based blocks are ready to communicate with the bridge via the MPTL interface. This output is generated using the mptl_if_target_wrap input **ocp_ready**. In the case of the **Uber** design, this **ocp_ready** input is driven by a signal derived from the **LOCKED** flag of the design's main MMCM (i.e. the one generating **pll_pri_clk** etc.). This holds off MPTL initialisation until after the MMCM is locked.

The reason for holding off MPTL initialisation is to prevent a race condition that might otherwise occur between (a) software attempting to read or write Target FPGA registers after configuration and (b) the main MMCM in the design achieving lock. Holding off MPTL initialisation between the Bridge and Target until the design's main MMCM has achieved lock causes any call made by software to the **ADMXRC3_ConfigureFromFile** API function to wait until MPTL initialization has been completed, thus guaranteeing that the Target FPGA is in the proper state for software on the host to communicate with it.

The **Uber** design input signal **mptl_sb_b2t.mptl_bridge_gtp_online_l** indicates that the bridge FPGA end of the MPTL interface is active. This input is used to generate the mptl_if_target_wrap output **mptl_ready**. In the case of the **Uber** design, this **mptl_ready** output is used as the asynchronous global reset.

> **Note**
> The Direct Slave and DMA address spaces supported by the Bridge FPGA are smaller than the full ADB3 OCP address space. For the model in use, they are indicated by the **DS_ADDR_WIDTH** and **DMA_ADDR_WIDTH** constants respectively, which are defined in the adb3_target_inc_pkg package.

5.9.5.3 Target PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the target PCIe interface core, instantiating a PCIe to OCP interface appropriate to the model in use. The purpose of the block is to connect the PCIe to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component pcie_if_target_wrap for details.

> **Note**
> The Direct Slave and DMA address spaces supported by the PCIe interface IP are smaller than the full ADB3 OCP address space. For the model in use, they are indicated by the **DS_ADDR_WIDTH** and **DMA_ADDR_WIDTH** constants respectively, which are defined in the adb3_target_inc_pkg package.

5.9.5.4 OCP Direct Slave Block

This block is implemented by **hdl/vhdl/examples/uber/common/blk_direct_slave.vhd**, and connects the Direct Slave OCP channel to various register blocks and a couple of memory access windows via OCP address space splitters. Most of the logic in this block is in the relatively low frequency (80 MHz) **pll_reg_clk** domain. Therefore, a secondary function of this block is to connect the high speed **pll_pri_clk** domain to the **pll_reg_clk** domain. The main elements are:

- Direct Slave address space splitter

- Direct Slave clock domain interface, between the **pll_pri_clk** domain and the relatively low frequency **pll_reg_clk** domain.
- Direct Slave register address space splitter
- Simple test register block (blk_ds_simple_test)
- Clock frequency measurement register block (blk_ds_clk_read)
- Interrupt test register block (blk_ds_int_test)
- Informational register block (blk_ds_info), including build datestamp and build timestamp
- GPIO test register block (blk_ds_io_test)
- On-board memory control and status register block (blk_ds_mem_reg)
- Direct Slave access to BRAM
- Direct Slave access to on-board memory

A block diagram of the OCP Direct Slave block is shown in Figure 31.

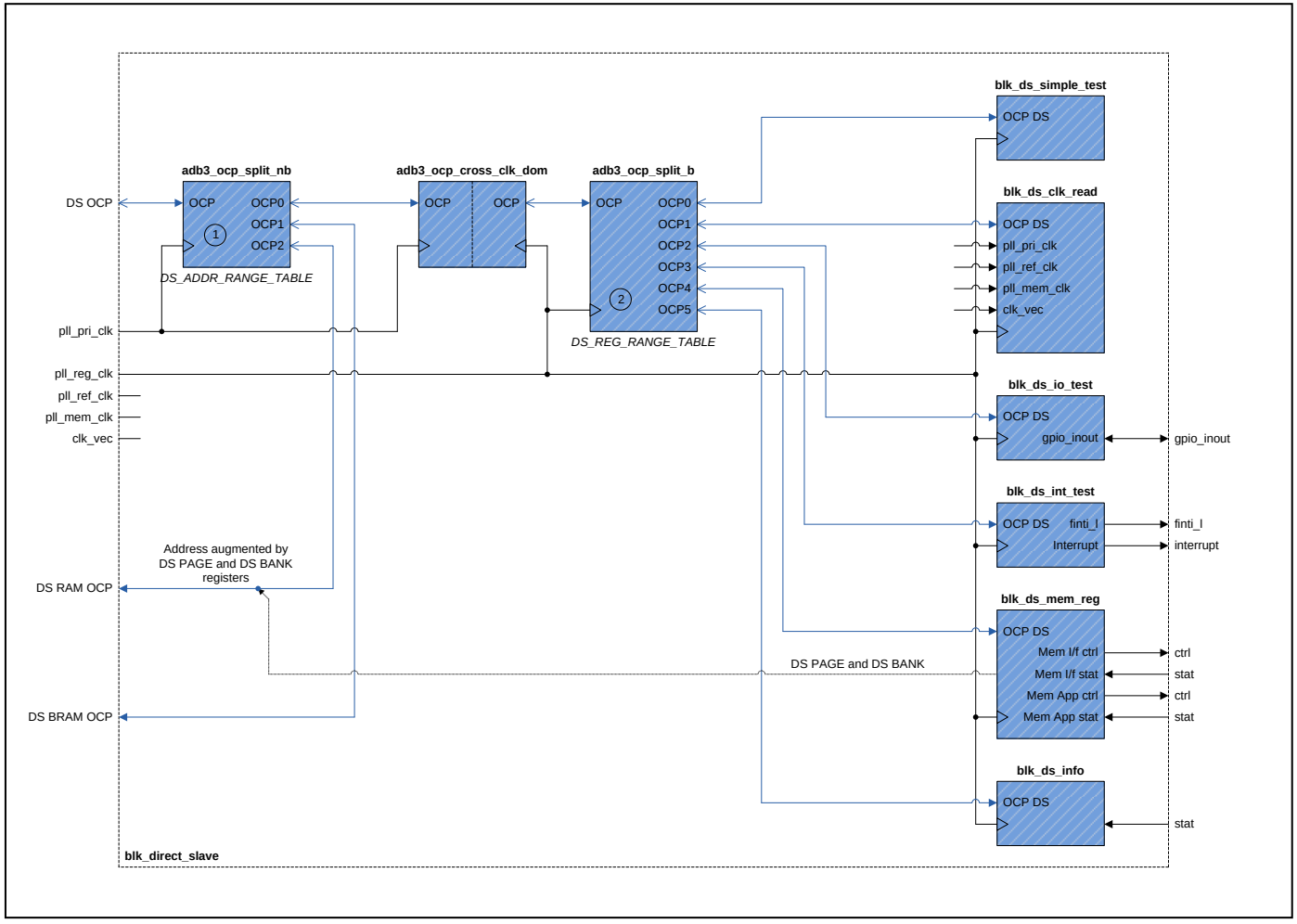

Figure 31 : Uber Direct Slave Block Diagram

5.9.5.4.1 Direct Slave Address Space Splitter

Referring to item 1 in Figure 31, this instance of adb3_ocp_split_nb splits the Direct Slave OCP channel into multiple secondary OCP channels, which are then routed to their appropriate blocks.

The split is defined by the Direct Slave address space ranges contained in the **DS_ADDR_RANGE_TABLE** constant in the uber_pkg package. It consists of {base address, mask} pairs for each address range that the splitter recognises. For each range, the lower address is identified by (base address), and the upper address is identified by (base address + mask).

Table Table 25 below shows the information in **DS_ADDR_RANGE_TABLE** and which functional area each index corresponds to:

Index	Address Range	Function
0	0x000000-0x0003FF	Direct Slave access to registers
1	0x080000-0x0FFFFF	Direct Slave access to BRAM
2	0x200000-0x3FFFFF	Direct Slave access to on-board memory

Table 25 : Uber Design Direct Slave Address Space

Note
Reads of undefined areas of the address space return data consisting of 0xDEADC0DE. Writes to undefined areas have no effect.

5.9.5.4.2 Direct Slave Register Address Space

The secondary OCP port 0 from the Direct Slave address space splitter is used to access direct slave register blocks in the **pll_reg_clk** clock domain. It is routed to the clock domain interface for clock domain transfer.

Note
Some registers in the relatively slow **pll_reg_clk** clock domain affect the operation of higher speed sections of the example FPGA design. To avoid out of sequence events, it is recommended that registers are read after they are written, before starting high speed events. An example of this is the DS_BANK/DS_PAGE registers which control on-board memory access.

5.9.5.4.2.1 Direct Slave Clock Domain Interface

This interfaces the Direct Slave register OCP channel in the higher speed clock domain (**pll_pri_clk**) to the lower speed register clock domain (**pll_reg_clk**). It uses an instance of the ADB3 OCP component adb3_ocp_cross_clk_dom.

5.9.5.4.2.2 Direct Slave Register Address Space Splitter

Referring to item 2 in Figure 31, this instance of adb3_ocp_split_b splits the Direct Slave register OCP channel into multiple secondary OCP channels, which are then routed to their appropriate blocks.

The split is defined by the Direct Slave register address space ranges contained in the **DS_REG_RANGE_TABLE** constant in the uber_pkg package. It consists of {base address, mask} pairs for each address range that the splitter recognises. For each range, the lower address is identified by (base address), and the upper address is identified by (base address + mask).

The **DS_REG_RANGE_TABLE** constant in the uber_pkg package uses the function **adb3_ocp_base** from the

adb3_ocp_comp package to extend the base address with '0's to width ADB3_OCP_ADDR_WIDTH.

In each mask value, a 1 bit causes the corresponding bit of the incoming OCP address to be ignored when the splitter determines which address range, if any, the incoming OCP address hits. The **DS_REG_RANGE_TABLE** constant in the uber_pkg package uses the function **adb3_ocp_mask** from the adb3_ocp_comp package to extend the mask address with '1's to width ADB3_OCP_ADDR_WIDTH, as these bits will never be anything but zero in incoming OCP addresses.

The following example illustrates how an address is determined to hit a given address range.

First, we note that address range 1 has the following base and mask information as defined in **DS_REG_RANGE_TABLE**:

- Address range 1 base = adb3_ocp_base(X"0000C0",DS_ADDR_WIDTH) = 0x00000000_000000C0.
- Address range 1 mask = adb3_ocp_mask(X"00003F",DS_ADDR_WIDTH) = 0xFFFFFFFF_FFC0003F.
- The address bits used in comparison are determined by the mask; in this case, they are bits (21:6).

Now consider an incoming OCP address of 0x00000000_000000D0, tested against address range 1:

- Incoming OCP address = 0x00000000_000000D0.
- So masked incoming OCP address = (NOT 0xFFFFFFFF_FFC0003F) AND 0x00000000_000000D0 = 0x00000000_000000C0.
- Since this masked address equals the base address for address range 1, the address is considered to hit address range 1.

Table Table 26 below shows the information in **DS_REG_RANGE_TABLE** and which functional area each index corresponds to:

Index	Address Range	Function
0	0x000000-0x00003F	Simple test registers
1	0x000040-0x00007F	Clock frequency measurement registers
2	0x0000C0-0x0000FF	Interrupt test registers
3	0x000140-0x00017F	Informational registers
4	0x000200-0x00027F	GPIO test registers
5	0x000300-0x0003FF	On-board memory control/status registers

Table 26 : Uber Design Direct Slave Register Address Space

Note
Reads of undefined areas of the address space return data consisting of 0xDEADC0DE. Writes to undefined areas have no effect.

5.9.5.4.2.3 Simple Test Register Block

5.9.5.4.2.3.1 Description

The Simple Test Register block contains a register that returns the nibble-reversed value of anything written to it. It is implemented by **hdl/vhdl/examples/uber/common/blk_ds_simple_test.vhd**. It consists of an instance of the ADB3 OCP component adb3_ocp_simple_bus_if connected to secondary port 0 of the Direct Slave register address space splitter, and a set of VHDL processes that implement the nibble-reversal register.

The adb3_ocp_simple_bus_if instance drives a simple parallel bus with the following signals:

1. **ds_a** - The register address, derived from some low order bits of the Direct Slave OCP address. This is used to select the correct register for writes, and to control a multiplexor that drives **ld_o** for reads.
2. **ds_w** - Indicates that a write cycle is taking place.
3. **ds_we** - Byte write enables. High when **ds_w** is high and bytes are enabled for writing.
4. **ds_d** - Write data bytes; qualified by **ds_we** bits.
5. **ds_r** - Indicates that a read cycle is taking place. Valid data must be present on **ds_q** after **read_latency** cycles.
6. **ds_q** - Driven with read data by a multiplexor controlled by **ds_a**. The registers of the FPGA design are inputs to the multiplexor.

5.9.5.4.2.3.2 Register Description

A set of VHDL processes in uses the signals **ds_a**, **ds_we** etc. described above to implement a single register. Although there is a single register in this example, in principle as many registers can be created as are required. The registers appear in the Direct Slave OCP address space as follows:

Name	Address
DATA	0x000000

Table 27 : Simple Test Register Block Address Map

Bits	Mnemonic	Type	Function
31:0	DATA	RW	Returns the nibble-reversed version of the last data written.

Table 28 : Simple Test Register Block, DATA Register (0x000000)

5.9.5.4.2.4 Clock Frequency Measurement Register Block

5.9.5.4.2.4.1 Description

The clock frequency measurement register block is implemented by the **blk_ds_clk_read** block which is model dependent.

Table 29 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-6TL	admxrc6tl/blk_ds_clk_read_6tl.vhd
ADM-XRC-6T1	admxrc6t1/blk_ds_clk_read_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/blk_ds_clk_read_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/blk_ds_clk_read_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/blk_ds_clk_read_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/blk_ds_clk_read_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/blk_ds_clk_read_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/blk_ds_clk_read_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/blk_ds_clk_read_admxrc7v1.vhd

Table 29 : Available Variants of blk_ds_clk_read Block

The **blk_ds_clk_read** block performs the following functions:

- Measurement of frequencies of internally generated (MMCM) clocks.
- Measurement of frequencies of externally sourced clocks (model dependent).

It consists of an instance of adb3_ocp_simple_bus_if connected to secondary port 1 of the Direct Slave register address space splitter, multiple instances of the clock frequency measurement block (blk_clock_freq), and a set of processes that implement the registers.

Clock frequency measurement components (blk_clock_freq) are instantiated for the main OCP clocks in the design, enabling them to be measured:

Clock	smp_clk_div_stages
pll_ref_clk	2 (0-400 MHz)
pll_pri_clk	2 (0-400 MHz)
pll_reg_clk	2 (0-400 MHz)
pll_mem_clk	3 (0-800 MHz) (Virtex-6 only)

Table 30 : Internally Generated Clock Frequency Measurement

Clock frequency measurement components blk_clock_freq are also instantiated for each model-dependent clock in the design, enabling them to be measured. For example, in the ADM-XRC-6T1:

Clock	smp_clk_div_stages
lclk	3 (0-800 MHz)
xrm_clkin	4 (0-1600 MHz)
MGT clocks	2 (0-400 MHz)

Table 31 : Externally Sourced Clock Frequency Measurement (ADM-XRC-6T1)

Within this block, the **REF_CLK_TCVAL** constant defines the measurement period in **pll_ref_clk** cycles. This constant is defined differently for simulation and synthesis as follows:

Simlation

- Period = (REF_CLK_FREQ_HZ/250000) ref_clk cycles = 4\ms.
- Resolution = 1MHz.

Synthesis

- Period = (REF_CLK_FREQ_HZ) ref_clk cycles = 1s.
- Resolution = 1Hz.

The differing definitions are controlled using synthesis directives. This enables a much shorted measurement period to be used during simulation.

If the clocking infrastructure of the **Uber** design as described in Clock and Reset Generation is modified to change the frequencies of **pll_pri_clk** and/or **pll_ref_clk**, the values mapped to the **smp_clk_div_stages** generics may need to be changed to ensure that the relationship defined in blk_clock_freq still holds for every instance.

5.9.5.4.2.4.2 Register Description

As in the simple test register block, an instance of adb3_ocp_simple_bus_if together with some VHDL processes implement the registers that control clock frequency measurement. These registers appear in the Direct Slave OCP address space as follows:

Name	Address
SEL	0x000040
CTRL/STAT	0x000044
FREQ	0x000048

Table 32 : Clock Frequency Measurement Register Block Address Map

Bits	Mnemonic	Type	Function
31:5			(Reserved) (Virtex-6 models)
4:0	SEL_CLK	M	Selects which clock's measured frequency and flags are available in the FREQ and STAT registers, respectively (Virtex-6 models). 00000 => pll_reg_clk (Internal clock)(0) 00001 => pll_pri_clk (Internal clock)(1) 00010 => pll_ref_clk (Internal clock)(2) 00011 => pll_mem_clk (Internal clock)(3) 00100 => Custom Clock 8 (External clock)(4) 00101 => Custom Clock 9 (External clock)(5) 00110 => Custom Clock 0 (External clock)(6) 00111 => Custom Clock 1 (External clock)(7) 01000 => Custom Clock 2 (External clock)(8) 01001 => Custom Clock 3 (External clock)(9) 01010 => Custom Clock 4 (External clock)(10) 01011 => Custom Clock 5 (External clock)(11) 01100 => Custom Clock 6 (External clock)(12) 01101 => Custom Clock 7 (External clock)(13) 01110 => mgt110_clk0 (External MGT clock)(14) 01111 => mgt110_clk1 (External MGT clock)(15) 10000 => mgt111_clk0 (External MGT clock)(16) 10001 => mgt111_clk1 (External MGT clock)(17) 10010 => mgt112_clk0 (External MGT clock)(18) 10011 => mgt112_clk1 (External MGT clock)(19) 10100 => mgt113_clk0 (External MGT clock)(20) 10101 => mgt113_clk1 (External MGT clock)(21) 10110 => mgt114_clk0 (External MGT clock)(22) 10111 => mgt114_clk1 (External MGT clock)(23) 11000 => mgt115_clk0 (External MGT clock)(24) 11001 => mgt115_clk1 (External MGT clock)(25) 11010 => mgt116_clk0 (External MGT clock)(26) 11011 => mgt116_clk1 (External MGT clock)(27) 11100 => mgt117_clk0 (External MGT clock)(28) 11101 => mgt117_clk1 (External MGT clock)(29) 11110 => mgt118_clk0 (External MGT clock)(30) 11111 => mgt118_clk1 (External MGT clock)(31)

Table 33 : Clock Frequency Measurement Register Block, SEL Register (0x000040) (Virtex-6 models)

Bits	Mnemonic	Type	Function
31:6			(Reserved)(7 Series models)
5:0	SEL_CLK	M	Selects which clock's measured frequency and flags are available in the FREQ and STAT registers, respectively (7 Series models). 000000 => pll_reg_clk (Internal clock)(0) 000001 => pll_pri_clk (Internal clock)(1) 000010 => pll_ref_clk (Internal clock)(2) 000011 => Unused 000100 => Custom Clock 8 (External clock)(4) 000101 => Custom Clock 9 (External clock)(5) 000110 => Custom Clock 0 (External clock)(6) 000111 => Custom Clock 1 (External clock)(7) 001000 => Custom Clock 2 (External clock)(8) 001001 => Custom Clock 3 (External clock)(9) 001010 => Custom Clock 4 (External clock)(10) 001011 => Custom Clock 5 (External clock)(11) 001100 => Custom Clock 6 (External clock)(12) 001101 => Custom Clock 7 (External clock)(13) 001110 => mgt110_clk0 (External MGT clock)(14) 001111 => mgt110_clk1 (External MGT clock)(15) 010000 => mgt111_clk0 (External MGT clock)(16) 010001 => mgt111_clk1 (External MGT clock)(17) 010010 => mgt112_clk0 (External MGT clock)(18) 010011 => mgt112_clk1 (External MGT clock)(19) 010100 => mgt113_clk0 (External MGT clock)(20) 010101 => mgt113_clk1 (External MGT clock)(21) 010110 => mgt114_clk0 (External MGT clock)(22) 010111 => mgt114_clk1 (External MGT clock)(23) 011000 => mgt115_clk0 (External MGT clock)(24) 011001 => mgt115_clk1 (External MGT clock)(25) 011010 => mgt116_clk0 (External MGT clock)(26) 011011 => mgt116_clk1 (External MGT clock)(27) 011100 => mgt117_clk0 (External MGT clock)(28) 011101 => mgt117_clk1 (External MGT clock)(29) 011110 => mgt118_clk0 (External MGT clock)(30) 011111 => mgt118_clk1 (External MGT clock)(31) 100000 => mgt119_clk0 (External MGT clock)(32) 100001 => mgt119_clk1 (External MGT clock)(33) 100010 => mgt120_clk0 (External MGT clock)(34) 100011 => mgt120_clk1 (External MGT clock)(35)

Table 34 : Clock Frequency Measurement Register Block, SEL Register (0x000040) (7 Series models)

Bits	Mnemonic	Type	Function
31	CLR_UPDATE	R/W1C	Write: controls frequency measurement updated flags: 1 = Clear all measurement updated flags. 0 = No action. Read: indicates selected frequency measurement update status: 1 = Measurement updated 0 = Measurement not updated.

Table 35 : Clock Frequency Measurement Register Block, CTRL/STAT Register (0x000044) (continued on

next page)

30	CLK_VALID	RO	Indicates selected board clock valid status: 1 = Clock valid on this board. 0 = Clock not valid on this board.
29	CLK_RUNNING	RO	Indicates selected clock running status: 1 = Clock running 0 = Clock not running.
28:0			(Reserved)

Table 35 : Clock Frequency Measurement Register Block, CTRL/STAT Register (0x000044)

Bits	Mnemonic	Type	Function
31:0	FREQ	RO	Indicates selected clock frequency measurement in Hz.

Table 36 : Clock Frequency Measurement Register Block, FREQ Register (0x000048)

5.9.5.4.2.5 Interrupt Test Register Block

5.9.5.4.2.5.1 Description

The interrupt test register block is implemented by **hdl/vhdl/examples/uber/common/blk_ds_int_test.vhd** and performs the following functions:

- Control of interrupt request generation using **finti_l** (MPTL) and **interrupt** (PCIe) outputs.

It consists of an instance of adb3_ocp_simple_bus_if connected to secondary port 3 of the Direct Slave register address space splitter, and a set of VHDL processes that implement the registers and interrupt generation.

5.9.5.4.2.5.2 Register Description

As in the simple test register block, an instance of adb3_ocp_simple_bus_if together with some VHDL processes implement a set of registers for generating interrupts on the host. These registers appear in the Direct Slave OCP address space as follows:

Name	Address
SET	0x0000C0
CLEAR/STAT	0x0000C4
MASK	0x0000C8
ARM	0x0000CC
COUNT	0x0000D0

Table 37 : Interrupt Test Register Block Address Map

Bits	Mnemonic	Type	Function
31:0	SET	W1S	Write: writing a 1 to a particular bit sets the corresponding bit in the STAT register. Read: returns undefined data.

Table 38 : Interrupt Test Register Block, SET Register (0x0000C0)

Bits	Mnemonic	Type	Function
31:0	CLEAR/STAT	R/W1C	The interrupt outputs are asserted whenever at least one bit in the STAT register is 1 and not masked by the MASK register. Write: writing a 1 to a particular bit clears the corresponding bit in the STAT register. Read: returns the current value of the STAT register.

Table 39 : Interrupt Test Register Block, CLEAR/STAT Register (0x0000C4)

Bits	Mnemonic	Type	Function
31:0	MASK	M	Controls/indicates the masking (1) or enabling (0) of individual bits in the STAT register. When a bit is 0, the corresponding bit in the STAT register is unmasked (i.e. allowed to assert the interrupt output).

Table 40 : Interrupt Test Register Block, MASK Register (0x0000C8)

Bits	Mnemonic	Type	Function
31:0	ARM	WO	A write to this register will force the FPGA interrupt outputs to their inactive state for one cycle of **pll_reg_clk**.

Table 41 : Interrupt Test Register Block, ARM Register (0x0000CC)

Bits	Mnemonic	Type	Function
31:0	COUNT	RW	Write: if the STAT register is zero, then the COUNT register is set to the value written. If the STAT register is non-zero, writes to the COUNT register have no effect. Read: indicates the number of clock cycles that have elapsed while the STAT register is non-zero.

Table 42 : Interrupt Test Register Block, COUNT Register (0x0000D0)

Since the COUNT register increments as long as at least one interrupt is active in the STAT register, the COUNT register can be used by host software to measure the time taken to respond to and clear an interrupt.

5.9.5.4.2.6 Informational Register Block

5.9.5.4.2.6.1 Description

The informational register block is implemented by **hdl/vhdl/examples/uber/common/blk_ds_info.vhd** and contains registers that indicate the following:

- The date and time at which design synthesis started.
- The status of Direct Slave OCP address splitter.
- The base address and size of the BRAM access window.

- The base address and size of the on-board memory access window.
- The number of banks of on-board memory.
- The version number of the SDK.

It consists of an instance of adb3_ocp_simple_bus_if connected to secondary port 5 of the Direct Slave register address space splitter, and a set of VHDL processes that implement the registers.

5.9.5.4.2.6.2 Register Description

As in the simple test register block, an instance of adb3_ocp_simple_bus_if together with some VHDL processes implement the informational registers. These registers appear in the Direct Slave OCP address space as follows:

Name	Address
DATE	0x000140
TIME	0x000144
SPLIT	0x000148
BRAM_BASE	0x00014C
BRAM_MASK	0x000150
MEM_BASE	0x000154
MEM_MASK	0x000158
MEM_BANKS	0x00015C
SDK_VER	0x000160

Table 43 : Informational Register Block Address Map

Bits	Mnemonic	Type	Function
31:0	DATE	RO	Indicates date of build (DD/MM/YYYY) in BCD format where: DD = Day of month MM = Month of year YYYY = Year. This information is obtained from the **TODAYS_DATE** constant in the today_pkg package (generated prior to synthesis).

Table 44 : Informational Register Block, DATE Register (0x000140)

Bits	Mnemonic	Type	Function
31:0	TIME	RO	Indicates time of build (HH/MM/SS/LL) in BCD format where: HH = Hour of day MM = Minute of hour SS = Second of minute LL = Millisecond of second. This information is obtained from the **TODAYS_TIME** constant in the today_pkg package (generated prior to synthesis).

Table 45 : Informational Register Block, TIME Register (0x000144)

Bits	Mnemonic	Type	Function
31:8			(Reserved).
7:0	SPLIT	RO	Indicates multiple split ports active error count.

Table 46 : Informational Register Block, SPLIT Register (0x000148)

Bits	Mnemonic	Type	Function
31:0	BRAM_BASE	RO	Indicates the base address of the BRAM access window in the Direct Slave OCP address space. This information is obtained from the **BRAM_ADDR_BASE** constant in the package **uber**.

Table 47 : Informational Register Block, BRAM_BASE Register (0x00014C)

Bits	Mnemonic	Type	Function
31:0	BRAM_MASK	RO	Indicates the address mask of the BRAM access window in the Direct Slave OCP address space. This information is obtained from the **BRAM_ADDR_MASK** constant in the package **uber**.

Table 48 : Informational Register Block, BRAM_MASK Register (0x000150)

Bits	Mnemonic	Type	Function
31:0	MEM_BASE	RO	Indicates the base address of the on-board memory access window in the Direct Slave OCP address space. This information is obtained from the **RAM_WIN_ADDR_BASE** constant in the package **uber**.

Table 49 : Informational Register Block, MEM_BASE Register (0x000154)

Bits	Mnemonic	Type	Function
31:0	MEM_MASK	RO	Indicates the address mask of the on-board memory access window in the Direct Slave OCP address space. This information is obtained from the **RAM_WIN_ADDR_MASK** constant in the package **uber**.

Table 50 : Informational Register Block, MEM_MASK Register (0x000158)

Bits	Mnemonic	Type	Function
31:4			(Reserved).
3:0	MEM_BANKS	RO	Indicates number of on-board memory bank interfaces present in the FPGA example design. This information is obtained from the **MEM_BANKS** constant in the adb3_target_inc_pkg package.

Table 51 : Informational Register Block, MEM_BANKS Register (0x00015C)

Bits	Mnemonic	Type	Function
31:24			(Reserved).
23:0	SDK_VER	RO	Indicates SDK version (AA/BB/CC) in BCD format where: AA = Major revision number BB = Minor revision number CC = Build revision number. This information is obtained from the **SDK_VERSION** constant in the today_pkg package.

Table 52 : Informational Register Block, SDK_VER Register (0x000160)

5.9.5.4.2.7 GPIO Test Register Block

5.9.5.4.2.7.1 Description

The GPIO test register block is implemented by the **blk_ds_io_test** block which is model dependent.

Table 53 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-6TL	admxrc6tl/blk_ds_io_test_6tl.vhd
ADM-XRC-6T1	admxrc6t1/blk_ds_io_test_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/blk_ds_io_test_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/blk_ds_io_test_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/blk_ds_io_test_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/blk_ds_io_test_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/blk_ds_io_test_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/blk_ds_io_test_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/blk_ds_io_test_admxrc7v1.vhd

Table 53 : Available Variants of blk_ds_io_test Component

The **blk_ds_io_test** block performs the following functions:

- Control of XRM GPIO bi-directional interface in example design (if present)
- Control of Pn4 GPIO bi-directional interface in example design (if present)
- Control of Pn6 GPIO bi-directional interface in example design (if present)
- Control of FMC GPIO bi-directional interface in example design (if present)
- Control of Secondary GPIO bi-directional interface in example design (if present)

It consists of an instance of adb3_ocp_simple_bus_if connected to secondary port 2 of the Direct Slave register address space splitter, and a set of processes that implement the registers that drive and return the logic levels on the GPIO pins.

Note
This block implements a general scheme for driving/accepting data on the GPIO interfaces using registers connected to the Direct Slave OCP channel. This scheme is known colloquially as "bit-banging", and is not suitable for high speed communication, as the block contains no logic for sequencing signals as required by a typical communications protocol. The user is encouraged to implement an I/O interface scheme appropriate to their own application.

5.9.5.4.2.7.2 Register Description

As in the simple test register block, an instance of adb3_ocp_simple_bus_if together with some VHDL processes implement the registers for the GPIO pins. These registers appear in the Direct Slave OCP address space as follows:

Name	Address
XRM_GPIO_DA_DATAO	0x000200
XRM_GPIO_DA_DATAI	0x000204
XRM_GPIO_DA_TRI	0x000208
XRM_GPIO_DB_DATAO	0x00020C
XRM_GPIO_DB_DATAI	0x000210
XRM_GPIO_DB_TRI	0x000214
XRM_GPIO_DC_DATAO	0x000218
XRM_GPIO_DC_DATAI	0x00021C
XRM_GPIO_DC_TRI	0x000220
XRM_GPIO_DD_DATAO	0x000224
XRM_GPIO_DD_DATAI	0x000228
XRM_GPIO_DD_TRI	0x00022C
XRM_GPIO_CS_DATAO	0x000230
XRM_GPIO_CS_DATAI	0x000234
XRM_GPIO_CS_TRI	0x000238
PN4_GPIO_P_DATAO	0x00023C
PN4_GPIO_P_DATAI	0x000240
PN4_GPIO_P_TRI	0x000244
PN4_GPIO_N_DATAO	0x000248
PN4_GPIO_N_DATAI	0x00024C
PN4_GPIO_N_TRI	0x000250
PN6_GPIO_MS_DATAO	0x000254
PN6_GPIO_MS_DATAI	0x000258
PN6_GPIO_MS_TRI	0x00025C
PN6_GPIO_LS_DATAO	0x000260
PN6_GPIO_LS_DATAI	0x000264
PN6_GPIO_LS_TRI	0x000268
FMC_GPIO_MS_LA_P_DATAO	0x000280
FMC_GPIO_MS_LA_P_DATAI	0x000284
FMC_GPIO_MS_LA_P_TRI	0x000288
FMC_GPIO_MS_LA_N_DATAO	0x00028C
FMC_GPIO_MS_LA_N_DATAI	0x000290

Table 54 : GPIO Test Register Block Address Map (continued on next page)

Name	Address
FMC_GPIO_MS_LA_N_TRI	0x000294
FMC_GPIO_LS_LA_P_DATAO	0x000298
FMC_GPIO_LS_LA_P_DATAI	0x00029C
FMC_GPIO_LS_LA_P_TRI	0x0002A0
FMC_GPIO_LS_LA_N_DATAO	0x0002A4
FMC_GPIO_LS_LA_N_DATAI	0x0002A8
FMC_GPIO_LS_LA_N_TRI	0x0002AC
FMC_GPIO_HA_P_DATAO	0x0002B0
FMC_GPIO_HA_P_DATAI	0x0002B4
FMC_GPIO_HA_P_TRI	0x0002B8
FMC_GPIO_HA_N_DATAO	0x0002BC
FMC_GPIO_HA_N_DATAI	0x0002C0
FMC_GPIO_HA_N_TRI	0x0002C4
FMC_GPIO_HB_P_DATAO	0x0002C8
FMC_GPIO_HB_P_DATAI	0x0002CC
FMC_GPIO_HB_P_TRI	0x0002D0
FMC_GPIO_HB_N_DATAO	0x0002D4
FMC_GPIO_HB_N_DATAI	0x0002D8
FMC_GPIO_HB_N_TRI	0x0002DC
SEC_GPIO_P_DATAO	0x0002E0
SEC_GPIO_P_DATAI	0x0002E4
SEC_GPIO_P_TRI	0x0002E8
SEC_GPIO_N_DATAO	0x0002EC
SEC_GPIO_N_DATAI	0x0002F0
SEC_GPIO_N_TRI	0x0002F4

Table 54 : GPIO Test Register Block Address Map

Bits	Mnemonic	Type	Function
31:16	DA_P_OUT	M	Controls/indicates logic levels driven on **da_p(15:0)** XRM GPIO pins.
15:0	DA_N_OUT	M	Controls/indicates logic levels driven on **da_n(15:0)** XRM GPIO pins.

Table 55 : GPIO Test Register Block, XRM_GPIO_DA_DATAO Register (0x000200)

Bits	Mnemonic	Type	Function
31:16	DA_P_IN	RO	Indicates actual logic levels on **da_p(15:0)** XRM GPIO pins.
15:0	DA_N_IN	RO	Indicates actual logic levels on **da_n(15:0)** XRM GPIO pins.

Table 56 : GPIO Test Register Block, XRM_GPIO_DA_DATAI Register (0x000204)

Bits	Mnemonic	Type	Function
31:16	DA_P_TRI	M	Controls/indicates tristate enables for the **da_p(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).
15:0	DA_N_TRI	M	Controls/indicates tristate enables for the **da_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 57 : GPIO Test Register Block, XRM_GPIO_DA_TRI Register (0x000208)

Bits	Mnemonic	Type	Function
31:16	DB_P_OUT	M	Controls/indicates logic levels driven on **db_p(15:0)** XRM GPIO pins.
15:0	DB_N_OUT	M	Controls/indicates logic levels driven on **db_n(15:0)** XRM GPIO pins.

Table 58 : GPIO Test Register Block, XRM_GPIO_DB_DATAO Register (0x00020C)

Bits	Mnemonic	Type	Function
31:16	DB_P_IN	RO	Indicates actual logic levels on **db_p(15:0)** XRM GPIO pins.
15:0	DB_N_IN	RO	Indicates actual logic levels on **db_n(15:0)** XRM GPIO pins.

Table 59 : GPIO Test Register Block, XRM_GPIO_DB_DATAI Register (0x000210)

Bits	Mnemonic	Type	Function
31:16	DB_P_TRI	M	Controls/indicates tristate enables for **db_p(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).
15:0	DB_N_TRI	M	Controls/indicates tristate enables for **db_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 60 : GPIO Test Register Block, XRM_GPIO_DB_TRI Register (0x000214)

Bits	Mnemonic	Type	Function
31:16	DC_P_OUT	M	Controls/indicates logic levels driven on **dc_p(15:0)** XRM GPIO pins.
15:0	DC_N_OUT	M	Controls/indicates logic levels driven on **dc_n(15:0)** XRM GPIO pins.

Table 61 : GPIO Test Register Block, XRM_GPIO_DC_DATAO Register (0x000218)

Bits	Mnemonic	Type	Function
31:16	DC_P_IN	RO	Indicates actual logic levels on **dc_p(15:0)** XRM GPIO pins.
15:0	DC_N_IN	RO	Indicates actual logic levels on **dc_n(15:0)** XRM GPIO pins.

Table 62 : GPIO Test Register Block, XRM_GPIO_DC_DATAI Register (0x00021C)

Bits	Mnemonic	Type	Function
31:16	DC_P_TRI	M	Controls/indicates tristate enables for **dc_p(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).
15:0	DC_N_TRI	M	Controls/indicates tristate enables for **dc_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 63 : GPIO Test Register Block, XRM_GPIO_DC_TRI Register (0x000220)

Bits	Mnemonic	Type	Function
31:16	DD_P_OUT	M	Controls/indicates logic levels driven on **dd_p(15:0)** XRM GPIO pins.
15:0	DD_N_OUT	M	Controls/indicates logic levels driven on **dd_n(15:0)** XRM GPIO pins.

Table 64 : GPIO Test Register Block, XRM_GPIO_DD_DATAO Register (0x000224)

Bits	Mnemonic	Type	Function
31:16	DD_P_IN	RO	Indicates actual logic levels on **dd_p(15:0)** XRM GPIO pins.
15:0	DD_N_IN	RO	Indicates actual logic levels on **dd_n(15:0)** XRM GPIO pins.

Table 65 : GPIO Test Register Block, XRM_GPIO_DD_DATAI Register (0x000228)

Bits	Mnemonic	Type	Function
31:16	DD_P_TRI	M	Controls/indicates tristate enables for **dd_p(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).
15:0	DD_N_TRI	M	Controls/indicates tristate enables for **dd_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 66 : GPIO Test Register Block, XRM_GPIO_DD_TRI Register (0x00022C)

Bits	Mnemonic	Type	Function
31:18			(Reserved)
17	DD_CC_P_OUT	M	Controls/indicates logic level driven on **dd_cc_p** XRM GPIO pin.
16	DD_CC_N_OUT	M	Controls/indicates logic level driven on **dd_cc_n** XRM GPIO pin.
15	DC_CC_P_OUT	M	Controls/indicates logic level driven on **dc_cc_p** XRM GPIO pin.
14	DC_CC_N_OUT	M	Controls/indicates logic level driven on **dc_cc_n** XRM GPIO pin.
13	DB_CC_P_OUT	M	Controls/indicates logic level driven on **db_cc_p** XRM GPIO pin.
12	DB_CC_N_OUT	M	Controls/indicates logic level driven on **db_cc_n** XRM GPIO pin.
11	DA_CC_P_OUT	M	Controls/indicates logic level driven on **da_cc_p** XRM GPIO pin.
10	DA_CC_N_OUT	M	Controls/indicates logic level driven on **da_cc_n** XRM GPIO pin.
9:6	SD_OUT	M	Controls/indicates logic levels driven on **sd(3:0)** XRM GPIO pins.
5:4	SC_OUT	M	Controls/indicates logic levels driven on **sc(1:0)** XRM GPIO pins.
3:2	SB_OUT	M	Controls/indicates logic levels driven on **sb(1:0)** XRM GPIO pins.
1:0	SA_OUT	M	Controls/indicates logic levels driven on **sa(1:0)** XRM GPIO pins.

Table 67 : GPIO Test Register Block, XRM_GPIO_CS_DATAO Register (0x000230)

Bits	Mnemonic	Type	Function
31:18			(Reserved)
17	DD_CC_P_IN	RO	Indicates actual logic level on **dd_cc_p** XRM GPIO pin.
16	DD_CC_N_IN	RO	Indicates actual logic level on **dd_cc_n** XRM GPIO pin.
15	DC_CC_P_IN	RO	Indicates actual logic level on **dc_cc_p** XRM GPIO pin.
14	DC_CC_N_IN	RO	Indicates actual logic level on **dc_cc_n** XRM GPIO pin.
13	DB_CC_P_IN	RO	Indicates actual logic level on **db_cc_p** XRM GPIO pin.
12	DB_CC_N_IN	RO	Indicates actual logic level on **db_cc_n** XRM GPIO pin.
11	DA_CC_P_IN	RO	Indicates actual logic level on **da_cc_p** XRM GPIO pin.
10	DA_CC_N_IN	RO	Indicates actual logic level on **da_cc_n** XRM GPIO pin.
9:6	SD_IN	RO	Indicates actual logic levels on **sd(3:0)** XRM GPIO pins.
5:4	SC_IN	RO	Indicates actual logic levels on **sc(1:0)** XRM GPIO pins.
3:2	SB_IN	RO	Indicates actual logic levels on **sb(1:0)** XRM GPIO pins.
1:0	SA_IN	RO	Indicates actual logic levels on **sa(1:0)** XRM GPIO pins.

Table 68 : GPIO Test Register Block, XRM_GPIO_CS_DATAI Register (0x000234)

Bits	Mnemonic	Type	Function
31:18			(Reserved)
17	DD_CC_P_TRI	M	Controls/indicates tristate enable for **dd_cc_p** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
16	DD_CC_N_TRI	M	Controls/indicates tristate enable for **dd_cc_n** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
15	DC_CC_P_TRI	M	Controls/indicates tristate enable for **dc_cc_p** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
14	DC_CC_N_TRI	M	Controls/indicates tristate enable for **dc_cc_n** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
13	DB_CC_P_TRI	M	Controls/indicates tristate enable for **db_cc_p** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
12	DB_CC_N_TRI	M	Controls/indicates tristate enable for **db_cc_n** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
11	DA_CC_P_TRI	M	Controls/indicates tristate enable for **da_cc_p** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
10	DA_CC_N_TRI	M	Controls/indicates tristate enable for **da_cc_n** XRM GPIO pin. If a bit is 1, the corresponding pin is tristated (high-impedance).
9:6	SD_TRI	M	Controls/indicates tristate enables for **sd(3:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).
5:4	SC_TRI	M	Controls/indicates tristate enables for **sc(1:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).
3:2	SB_TRI	M	Controls/indicates tristate enables for **sb(1:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 69 : GPIO Test Register Block, XRM_GPIO_CS_TRI Register (0x000238) (continued on next page)

Bits	Mnemonic	Type	Function
1:0	SA_TRI	M	Controls/indicates tristate enables for **sa(1:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 69 : GPIO Test Register Block, XRM_GPIO_CS_TRI Register (0x000238)

Bits	Mnemonic	Type	Function
31:0	P_DATAO	M	Controls/indicates logic levels driven on **gpio_p (PN4_GPIO_WIDTH:1)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** < 32, the top **32-PN4_GPIO_WIDTH** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in package adb3_target_inc_pkg.

Table 70 : GPIO Test Register Block, PN4_GPIO_P_DATAO Register (0x00023C)

Bits	Mnemonic	Type	Function
31:0	P_DATAI	RO	Indicates actual logic levels on **gpio_p(PN4_GPIO_WIDTH:1)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** < 32, the top **32-PN4_GPIO_WIDTH** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in package adb3_target_inc_pkg.

Table 71 : GPIO Test Register Block, PN4_GPIO_P_DATAI Register (0x000240)

Bits	Mnemonic	Type	Function
31:0	P_TRI	M	Controls/indicates tristate enables for **gpio_p(PN4_GPIO_WIDTH:1)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** < 32, the top **32-PN4_GPIO_WIDTH** bits of this register are unused. If a bit is 1, the corresponding pin is tristated (high-impedance). The constant **PN4_GPIO_WIDTH** is defined in package adb3_target_inc_pkg.

Table 72 : GPIO Test Register Block, PN4_GPIO_P_TRI Register (0x000244)

Bits	Mnemonic	Type	Function
31:0	N_DATAO	M	Controls/indicates logic levels driven on **gpio_n (PN4_GPIO_WIDTH:1)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** < 32, the top **32-PN4_GPIO_WIDTH** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in package adb3_target_inc_pkg.

Table 73 : GPIO Test Register Block, PN4_GPIO_N_DATAO Register (0x000248)

Bits	Mnemonic	Type	Function
31:0	N_DATAI	RO	Indicates actual logic levels on **gpio_n(PN4_GPIO_WIDTH:1)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** < 32, the top **32-PN4_GPIO_WIDTH** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in package adb3_target_inc_pkg.

Table 74 : GPIO Test Register Block, PN4_GPIO_N_DATAI Register (0x00024C)

Bits	Mnemonic	Type	Function
31:0	N_TRI	M	Controls/indicates tristate enables for **gpio_n(PN4_GPIO_WIDTH:1)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** < 32, the top **32-PN4_GPIO_WIDTH** bits of this register are unused. If a bit is 1, the corresponding pin is tristated (high-impedance). The constant **PN4_GPIO_WIDTH** is defined in package adb3_target_inc_pkg.

Table 75 : GPIO Test Register Block, PN4_GPIO_N_TRI Register (0x000250)

Bits	Mnemonic	Type	Function
31:0	MS_DATAO	M	**Single Ended Interface** If **PN6_GPIO_WIDTH** ≤ 32, this register is ignored. If **PN6_GPIO_WIDTH** > 32, this register controls/indicates logic levels driven on the **gpio(PN6_GPIO_WIDTH:33)** PN6 GPIO pins. If **PN6_GPIO_WIDTH** < 64, the top **64-PN6_GPIO_WIDTH** bits of this register are unused. **Double Ended Interface** Controls/indicates logic levels driven on **gpio_p (PN6_GPIO_WIDTH-1:0)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** < 32, the top **32-PN6_GPIO_WIDTH** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in package adb3_target_inc_pkg

Table 76 : GPIO Test Register Block, PN6_GPIO_MS_DATAO Register (0x000254)

Bits	Mnemonic	Type	Function
31:0	MS_DATAI	RO	**Single Ended Interface** If **PN6_GPIO_WIDTH** ≤ 32, this register is ignored. If **PN6_GPIO_WIDTH** > 32, this register indicates the actual logic levels on the **gpio(PN6_GPIO_WIDTH:33)** PN6 GPIO pins. If **PN6_GPIO_WIDTH** < 64, the top **64-PN6_GPIO_WIDTH** bits of this register are unused. **Double Ended Interface** Indicates actual logic levels on **gpio_p(PN6_GPIO_WIDTH-1:0)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** < 32, the top **32-PN6_GPIO_WIDTH** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in package adb3_target_inc_pkg

Table 77 : GPIO Test Register Block, PN6_GPIO_MS_DATAI Register (0x000258)

Bits	Mnemonic	Type	Function
31:0	MS_TRI	M	**Single Ended Interface** If **PN6_GPIO_WIDTH** ≤ 32, this register is ignored. If **PN6_GPIO_WIDTH** > 32, this register controls/indicates the tristate enables for the **gpio(PN6_GPIO_WIDTH:33)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** < 64, the top **64-PN6_GPIO_WIDTH** bits of this register are unused. **Double Ended Interface** Controls/indicates tristate enables for **gpio_p(PN6_GPIO_WIDTH:1)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** < 32, the top **32-PN6_GPIO_WIDTH** bits of this register are unused. If a bit is 1, the corresponding pin is tristated (high-impedance). The constant **PN6_GPIO_WIDTH** is defined in package adb3_target_inc_pkg

Table 78 : GPIO Test Register Block, PN6_GPIO_MS_TRI Register (0x00025C)

Bits	Mnemonic	Type	Function
31:0	LS_DATAO	M	**Single Ended Interface** If **PN6_GPIO_WIDTH** ≥ 32, this register controls/indicates logic levels driven on the **gpio(32:1)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** < 32, this register controls/indicates logic levels driven on the **gpio(PN6_GPIO_WIDTH:1)** Pn6 GPIO pins, and the top **32-PN6_GPIO_WIDTH** bits of this register are unused. **Double Ended Interface** Controls/indicates logic levels driven on **gpio_n (PN6_GPIO_WIDTH:1)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** is < 32, the top **32-PN6_GPIO_WIDTH** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in package adb3_target_inc_pkg

Table 79 : GPIO Test Register Block, PN6_GPIO_LS_DATAO Register (0x000260)

Bits	Mnemonic	Type	Function
31:0	LS_DATAI	RO	**Single Ended Interface** If **PN6_GPIO_WIDTH** ≥ 32, this register indicates the actual logic levels on the **gpio(32:1)** Pn6 GPIO pins If **PN6_GPIO_WIDTH** < 32, this register indicates the actual logic levels on the **gpio(PN6_GPIO_WIDTH:1)** Pn6 GPIO pins, and the top **32-PN6_GPIO_WIDTH** bits of this register are unused. **Double Ended Interface** Indicates actual logic levels on **gpio_n(PN6_GPIO_WIDTH:1)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** < 32, the top **32-PN6_GPIO_WIDTH** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in package adb3_target_inc_pkg

Table 80 : GPIO Test Register Block, PN6_GPIO_LS_DATAI Register (0x000264)

Bits	Mnemonic	Type	Function
31:0	LS_TRI	M	**Single Ended Interface** If **PN6_GPIO_WIDTH** ≥ 32, this register controls/indicates the tristate enables for the **gpio(32:1)** Pn6 GPIO pins If **PN6_GPIO_WIDTH** < 32, this register controls/indicates the tristate enables of the **gpio(PN6_GPIO_WIDTH:1)** Pn6 GPIO pins, and the top **32-PN6_GPIO_WIDTH** bits of this register are unused. **Double Ended Interface** Controls/indicates tristate enables for **gpio_n(PN6_GPIO_WIDTH:1)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** < 32, the top **32-PN6_GPIO_WIDTH** bits of this register are unused. If a bit is 1, the corresponding pin is tristated (high-impedance). The constant **PN6_GPIO_WIDTH** is defined in package adb3_target_inc_pkg

Table 81 : GPIO Test Register Block, PN6_GPIO_LS_TRI Register (0x000268)

Bits	Mnemonic	Type	Function
1:0	MS_P_DATAO	M	Controls/indicates logic levels driven on **fmc_la_p(33:32)** FMC GPIO pins.

Table 82 : GPIO Test Register Block, FMC_GPIO_MS_LA_P_DATAO Register (0x000280)

Bits	Mnemonic	Type	Function
1:0	MS_P_DATAI	M	Indicates actual logic levels on **fmc_la_p(33:32)** FMC GPIO pins.

Table 83 : GPIO Test Register Block, FMC_GPIO_MS_LA_P_DATAI Register (0x000284)

Bits	Mnemonic	Type	Function
1:0	MS_P_TRI	M	Controls/indicates tristate enables for the **fmc_la_p(33:32)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 84 : GPIO Test Register Block, FMC_GPIO_MS_LA_P_TRI Register (0x000288)

Bits	Mnemonic	Type	Function
1:0	MS_N_DATAO	M	Controls/indicates logic levels driven on **fmc_la_n(33:32)** FMC GPIO pins.

Table 85 : GPIO Test Register Block, FMC_GPIO_MS_LA_N_DATAO Register (0x00028C)

Bits	Mnemonic	Type	Function
1:0	MS_N_DATAI	M	Indicates actual logic levels on **fmc_la_n(33:32)** FMC GPIO pins.

Table 86 : GPIO Test Register Block, FMC_GPIO_MS_LA_N_DATAI Register (0x000290)

Bits	Mnemonic	Type	Function
1:0	MS_N_TRI	M	Controls/indicates tristate enables for the **fmc_la_n(33:32)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 87 : GPIO Test Register Block, FMC_GPIO_MS_LA_N_TRI Register (0x000294)

Bits	Mnemonic	Type	Function
31:0	LS_P_DATAO	M	Controls/indicates logic levels driven on **fmc_la_p(31:0)** FMC GPIO pins.

Table 88 : GPIO Test Register Block, FMC_GPIO_LS_LA_P_DATAO Register (0x000298)

Bits	Mnemonic	Type	Function
31:0	LS_P_DATAI	M	Indicates actual logic levels on **fmc_la_p(31:0)** FMC GPIO pins.

Table 89 : GPIO Test Register Block, FMC_GPIO_LS_LA_P_DATAI Register (0x00029C)

Bits	Mnemonic	Type	Function
31:0	LS_P_TRI	M	Controls/indicates tristate enables for the **fmc_la_p(31:0)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 90 : GPIO Test Register Block, FMC_GPIO_LS_LA_P_TRI Register (0x0002A0)

Bits	Mnemonic	Type	Function
31:0	LS_N_DATAO	M	Controls/indicates logic levels driven on **fmc_la_n(31:0)** FMC GPIO pins.

Table 91 : GPIO Test Register Block, FMC_GPIO_LS_LA_N_DATAO Register (0x0002A4)

Bits	Mnemonic	Type	Function
31:0	LS_N_DATAI	M	Indicates actual logic levels on **fmc_la_n(31:0)** FMC GPIO pins.

Table 92 : GPIO Test Register Block, FMC_GPIO_LS_LA_N_DATAI Register (0x0002A8)

Bits	Mnemonic	Type	Function
31:0	LS_N_TRI	M	Controls/indicates tristate enables for the **fmc_la_n(31:0)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 93 : GPIO Test Register Block, FMC_GPIO_LS_LA_N_TRI Register (0x0002AC)

Bits	Mnemonic	Type	Function
31:0	P_DATAO	M	Controls/indicates logic levels driven on **fmc_ha_p(31:0)** FMC GPIO pins.

Table 94 : GPIO Test Register Block, FMC_GPIO_HA_P_DATAO Register (0x0002B0)

Bits	Mnemonic	Type	Function
31:0	P_DATAI	M	Indicates actual logic levels on **fmc_ha_p(31:0)** FMC GPIO pins.

Table 95 : GPIO Test Register Block, FMC_GPIO_HA_P_DATAI Register (0x0002B4)

Bits	Mnemonic	Type	Function
31:0	P_TRI	M	Controls/indicates tristate enables for the **fmc_ha_p(31:0)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 96 : GPIO Test Register Block, FMC_GPIO_HA_P_TRI Register (0x0002B8)

Bits	Mnemonic	Type	Function
31:0	N_DATAO	M	Controls/indicates logic levels driven on **fmc_ha_n(31:0)** FMC GPIO pins.

Table 97 : GPIO Test Register Block, FMC_GPIO_HA_N_DATAO Register (0x0002BC)

Bits	Mnemonic	Type	Function
31:0	N_DATAI	M	Indicates actual logic levels on **fmc_ha_n(31:0)** FMC GPIO pins.

Table 98 : GPIO Test Register Block, FMC_GPIO_HA_N_DATAI Register (0x0002C0)

Bits	Mnemonic	Type	Function
31:0	N_TRI	M	Controls/indicates tristate enables for the **fmc_ha_n(31:0)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 99 : GPIO Test Register Block, FMC_GPIO_HA_N_TRI Register (0x0002C4)

Bits	Mnemonic	Type	Function
31:0	P_DATAO	M	Controls/indicates logic levels driven on **fmc_hb_p(31:0)** FMC GPIO pins.

Table 100 : GPIO Test Register Block, FMC_GPIO_HB_P_DATAO Register (0x0002C8)

Bits	Mnemonic	Type	Function
31:0	P_DATAI	M	Indicates actual logic levels on **fmc_hb_p(31:0)** FMC GPIO pins.

Table 101 : GPIO Test Register Block, FMC_GPIO_HB_P_DATAI Register (0x0002CC)

Bits	Mnemonic	Type	Function
31:0	P_TRI	M	Controls/indicates tristate enables for the **fmc_hb_p(31:0)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 102 : GPIO Test Register Block, FMC_GPIO_HB_P_TRI Register (0x0002D0)

Bits	Mnemonic	Type	Function
31:0	N_DATAO	M	Controls/indicates logic levels driven on **fmc_hb_n(31:0)** FMC GPIO pins.

Table 103 : GPIO Test Register Block, FMC_GPIO_HB_N_DATAO Register (0x0002D4)

Bits	Mnemonic	Type	Function
31:0	N_DATAI	M	Indicates actual logic levels on **fmc_hb_n(31:0)** FMC GPIO pins.

Table 104 : GPIO Test Register Block, FMC_GPIO_HB_N_DATAI Register (0x0002D8)

Bits	Mnemonic	Type	Function
31:0	N_TRI	M	Controls/indicates tristate enables for the **fmc_hb_n(31:0)** FMC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 105 : GPIO Test Register Block, FMC_GPIO_HB_N_TRI Register (0x0002DC)

Bits	Mnemonic	Type	Function
31:0	P_DATAO	M	Controls/indicates logic levels driven on **gpio_p(31:0)** SEC GPIO pins.

Table 106 : GPIO Test Register Block, SEC_GPIO_P_DATAO Register (0x0002E0)

Bits	Mnemonic	Type	Function
31:0	P_DATAI	M	Indicates actual logic levels on **gpio_p(31:0)** SEC GPIO pins.

Table 107 : GPIO Test Register Block, SEC_GPIO_P_DATAI Register (0x0002E4)

Bits	Mnemonic	Type	Function
31:0	P_TRI	M	Controls/indicates tristate enables for the **gpio_p(31:0)** SEC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 108 : GPIO Test Register Block, SEC_GPIO_P_TRI Register (0x0002E8)

Bits	Mnemonic	Type	Function
31:0	N_DATAO	M	Controls/indicates logic levels driven on **gpio_n(31:0)** SEC GPIO pins.

Table 109 : GPIO Test Register Block, SEC_GPIO_N_DATAO Register (0x0002EC)

Bits	Mnemonic	Type	Function
31:0	N_DATAI	M	Indicates actual logic levels on **gpio_n(31:0)** SEC GPIO pins.

Table 110 : GPIO Test Register Block, SEC_GPIO_N_DATAI Register (0x0002F0)

Bits	Mnemonic	Type	Function
31:0	N_TRI	M	Controls/indicates tristate enables for the **gpio_n(31:0)** SEC GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 111 : GPIO Test Register Block, SEC_GPIO_N_TRI Register (0x0002F4)

5.9.5.4.2.8 On-Board Memory Register Block

5.9.5.4.2.8.1 Description

The on-board Memory register block is implemented in **hdl/vhdl/examples/uber/common/blk_ds_mem_reg.vhd** and contains the following register groups:

- Control of paging for the Direct Slave on-board memory access window via the **DS_BANK** and **DS_PAGE** registers.
- Status of the on-board memory interfaces.
- Control and status of the on-board memory application block (FPGA-driven on-board memory test).

It consists of an instance of adb3_ocp_simple_bus_if connected to secondary port 4 of the Direct Slave register address space splitter, and a set of VHDL processes that implement the memory control and status registers.

5.9.5.4.2.8.2 Register Description

As in the simple test register block, an instance of adb3_ocp_simple_bus_if together with some VHDL processes implement the memory control and status registers. These registers appear in the Direct Slave OCP address space as follows:

Name	Address
DS_BANK	0x000300
DS_PAGE	0x000304
BANK0_CTRL BANK1_CTRL ...	0x000320 0x000340 ...
BANK0_OFFSET BANK1_OFFSET ...	0x000324 0x000344 ...
BANK0_LENGTH BANK1_LENGTH ...	0x000328 0x000348 ...
BANK0_INFO BANK1_INFO ...	0x00032C 0x00034C ...

Table 112 : On-Board Memory Register Block Address Map (continued on next page)

Name	Address
BANK0_STAT BANK1_STAT ...	0x000330 0x000350 ...
BANK0_APP_ERR_ADDR BANK1_APP_ERR_ADDR ...	0x000334 0x000354 ...
BANK0_MUX_ERR BANK1_MUX_ERR ...	0x000338 0x000358 ...
BANK0_IF_ERR BANK1_IF_ERR ...	0x00033C 0x00035C ...

Table 112 : On-Board Memory Register Block Address Map

Bits	Mnemonic	Type	Function
31:0	DS_BANK	M	Controls which on-board memory bank is accessed via the Direct Slave OCP address window. The number of bits of this field that are actually used is controlled by the **BANK_ADDR_WIDTH** constant defined in blk_direct_slave. Bits 31:BANK_ADDR_WIDTH are ignored. Refer to Direct Slave On-Board Memory Access Window for an explanation of how this register affects access to on-board memory.

Table 113 : On-Board Memory Register Block, DS_BANK Register (0x000300)

Bits	Mnemonic	Type	Function
31:0	DS_PAGE	M	Controls which page of the on-board memory bank selected by the **DS_BANK** register is accessed via the Direct Slave OCP address window. The number of bits of this field that are actually used is controlled by the **PAGE_ADDR_WIDTH** constant defined in blk_direct_slave. Bits 31:PAGE_ADDR_WIDTH are ignored. Refer to Direct Slave On-Board Memory Access Window for an explanation of how this register affects access to on-board memory.

Table 114 : On-Board Memory Register Block, DS_PAGE Register (0x000304)

Bits	Mnemonic	Type	Function
31:9			(Reserved)
8	START_TEST	WO	On-board memory application control: Write 1 to initiate the FPGA-driven on-board memory test for bank x; has no effect unless **BANKx_STAT. MEM_APP_DONE** is 1.

Table 115 : On-Board Memory Register Block, BANKx_CTRL Register (0x000320, 0x000340, ...) (continued

on next page)

Bits	Mnemonic	Type	Function
7:0			(Reserved)

Table 115 : On-Board Memory Register Block, BANKx_CTRL Register (0x000320, 0x000340, ...)

Bits	Mnemonic	Type	Function
31:0	MEM_APP_OFFSET	M	On-board memory application control: Determines the starting address (16-byte addressing) for the FPGA-driven on-board memory test for bank x.

Table 116 : On-Board Memory Register Block, BANKx_OFFSET Register (0x000324, 0x000344, ...)

Bits	Mnemonic	Type	Function
31:0	MEM_APP_LENGTH	M	On-board memory application control: Determines the number of 16-byte words that are tested by the FPGA-driven on-board memory test for bank x.

Table 117 : On-Board Memory Register Block, BANKx_LENGTH Register (0x000328, 0x000348, ...)

Bits	Mnemonic	Type	Function
31:28	DS_BANK_WIDTH	RO	Indicates the width in bits of the Direct Slave on-board Memory bank select register. The value of this register is determined by the constant **BANK_ADDR_WIDTH**. This is defined in blk_direct_slave.
27:24	DS_PAGE_WIDTH	RO	Indicates the width in bits of the Direct Slave on-board Memory page select register. The value of this register is determined by the constant **PAGE_ADDR_WIDTH**. This is defined in blk_direct_slave.
23:16	DATA_BYTES	RO	Indicates the number of bytes in the on-board Memory bank x OCP data word.
15:8	APP_ADDR_WIDTH	RO	Indicates the width in bits of the on-board memory bank x address space using 16-byte addressing. The value of this register is determined using the constant **MEM_BYTE_ADDR_WIDTH_ARRAY**-4. This is defined in the package adb3_target_inc_pkg.
7:0	BYTE_ADDR_WIDTH	RO	Indicates the width in bits of the on-board memory bank x address space using byte addressing. The value of this register is determined using the constant **MEM_BYTE_ADDR_WIDTH_ARRAY**. This is defined in the package adb3_target_inc_pkg.

Table 118 : On-Board Memory Register Block, BANKx_INFO Register (0x00032C, 0x00034C, ...)

Bits	Mnemonic	Type	Function
31:28	BANK_NUMBER	RO	The number of the bank this register applies to.
27:24			(Reserved)
23	MEM_APP_ERR	RO	On-board memory application status: 1 => An error occurred during the last FPGA-driven test of memory bank x; valid if and only if **MEM_APP_DONE** is 1.
22:20	MEM_APP_ERR_PH	RO	On-board memory application status: Indicates at which phase the last FPGA-driven test of memory bank x failed; valid if and only if both **MEM_APP_DONE** and **MEM_APP_ERR** are 1.
19:17			(Reserved)
16	MEM_APP_DONE	RO	On-board memory application status: 1 => The FPGA-driven test of memory bank x is idle/ done.
15:12			(Reserved)
11:8	MEM_IF_ERR	RO	On-board memory interface bank x initialisation error status: Bit (3): Reset (active high). Bit (2:1): Read leveling error. Bit (0): Write leveling error.
7:4			(Reserved)
3:0	MEM_IF_STAT	RO	On-board memory interface bank x initialisation status: Bit (3): Init complete. Bit (2:1): Read leveling complete. Bit (0): Write leveling complete.

Table 119 : On-Board Memory Register Block, BANKx_STAT Register (0x000330, 0x000350, ...)

Bits	Mnemonic	Type	Function
31:25			(Reserved)
24:0	MEM_APP_ERR_ADDR	RO	On-board memory application status: Returns the address (16-byte addressing) of the first error detected in the last FPGA-driven test of memory bank x; valid if and only if both **BANKx_STAT. MEM_APP_DONE** and **BANKx_STAT.MEM_APP_ERR** are 1.

Table 120 : On-Board Memory Register Block, BANKx_APP_ERR_ADDR Register (0x000334, 0x000354, ...)

Bits	Mnemonic	Type	Function
31:0	MUX_ERR	RO	OCP switching bank x **adb3_ocp_mux_nb** block error status.

Table 121 : On-Board Memory Register Block, BANKx_MUX_ERR Register (0x000338, 0x000358, ...)

Bits	Mnemonic	Type	Function
31:0	MEM_IF_ERR	RO	On-board memory interface bank x **adb3_ocp_ocp2ddr3_nb** block error status.

Table 122 : On-Board Memory Register Block, BANKx_IF_ERR Register (0x00033C, 0x00035C, ...)

5.9.5.4.3 Direct Slave BRAM Address Space

5.9.5.4.3.1 Description

The secondary OCP port 1 from the Direct Slave address space splitter is used to access the BRAM block. It is routed to the OCP switching block.

5.9.5.4.3.2 Direct Slave BRAM Access Window

As the BRAM requires an address space of 0.5 MiB, this can be accommodated within the Direct Slave OCP channel address space of 4 MiB.

The BRAM access window appears in the Direct Slave OCP address space as follows:

Name	Address
BRAM access window	0x080000-0x0FFFFF

Table 123 : Direct Slave BRAM Access Window

5.9.5.4.4 Direct Slave On-Board Memory Address Space

5.9.5.4.4.1 Description

The secondary OCP port 2 from the Direct Slave address space splitter is used to access the on-board memory interfaces. It is routed to the OCP switching block.

5.9.5.4.4.2 Direct Slave On-Board Memory Access Window

As the Direct Slave OCP channel has a useable address space of 4 MiB, this is not sufficient to access all on-board memory. The 2 MiB address window is used and augmented by the DS_BANK and DS_PAGE registers in order to access all on-board memory.

The on-board memory access window appears in the Direct Slave OCP address space as follows:

Name	Address
On-Board memory access window	0x200000-0x3FFFFF

Table 124 : Direct Slave On-Board Memory Access Window

The conversion from Direct Slave OCP addresses to augmented OCP memory addresses works as follows:

```
Augmented OCP memory address [20:0] = Direct Slave OCP address [20:0]
Augmented OCP memory address [DMA_ADDR_WIDTH-BANK_ADDR_WIDTH-1:21] = DS_PAGE
Augmented OCP memory address [DMA_ADDR_WIDTH-1:DMA_ADDR_WIDTH-BANK_ADDR_WIDTH] =
DS_BANK
Augmented OCP memory address [63:DMA_ADDR_WIDTH] = 0
```

where **DMA_ADDR_WIDTH** is defined in adb3_target_inc_pkg and **BANK_ADDR_WIDTH** is defined in blk_direct_slave.

For example, for the ADM-XRC-6T1, this yields:

```
Augmented OCP memory address [20:0] = Direct Slave OCP address [20:0]
Augmented OCP memory address [35:21] = DS_PAGE [14:0]
Augmented OCP memory address [38:36] = DS_BANK [2:0]
Augmented OCP memory address [63:39] = 0
```

This produces augmented OCP addresses which are compatible with the memory address decoding scheme defined in Table 125.

5.9.5.5 OCP Switching Block

This block is implemented by **hdl/vhdl/examples/uber/common/blk_dma_switch.vhd** and its purpose is to connect together the various OCP channels in the **Uber** design in a useful way. A block diagram of the OCP switching block is shown in Figure 32.

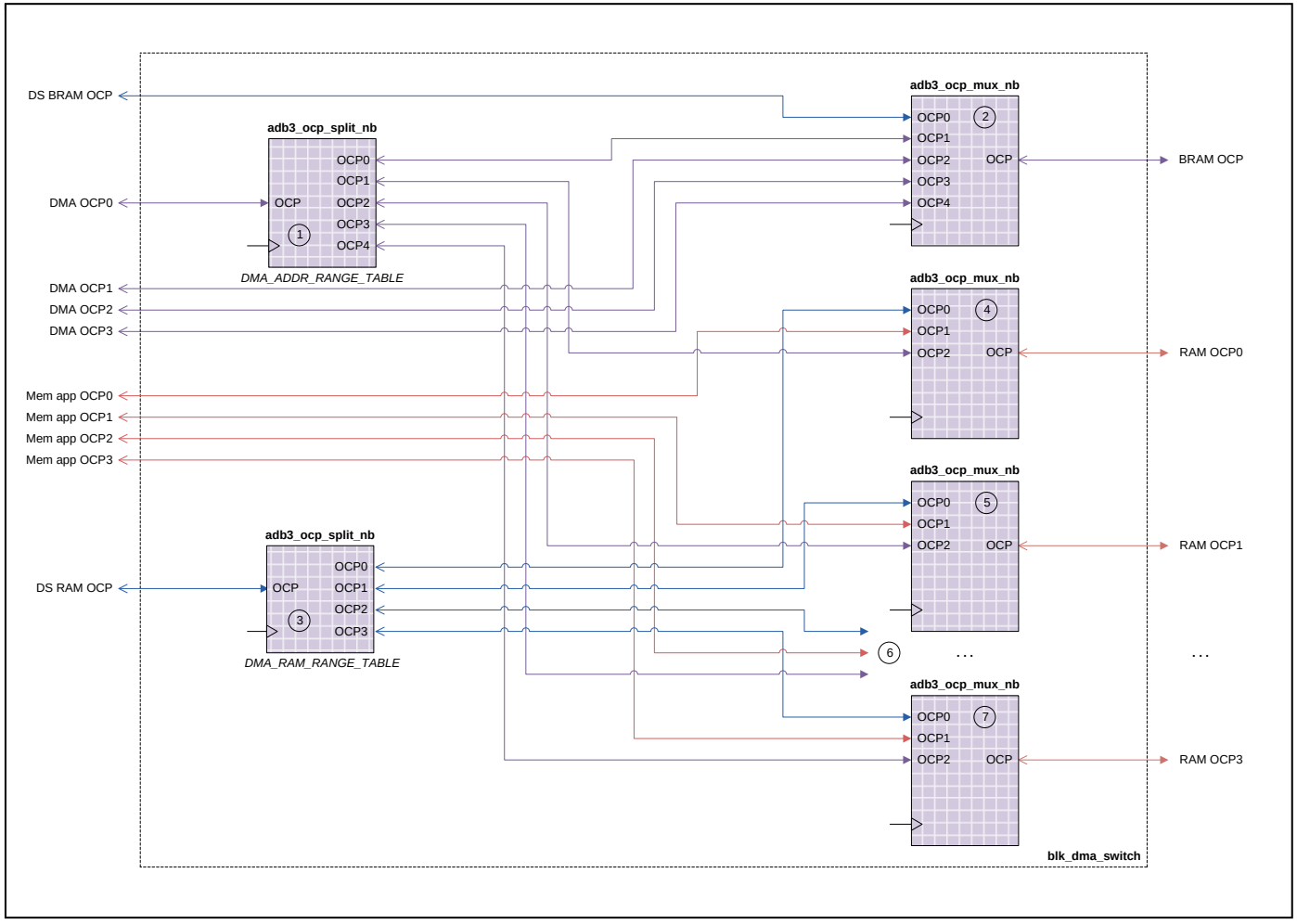

Figure 32 : Uber OCP Switching Block

The OCP switching block makes connections between the various OCP channels in the design as follows:

- Direct Slave on-board memory access OCP channel <=> On-board memory bank interface OCP channels
- Direct Slave BRAM access OCP channel <=> BRAM interface OCP channel
- Memory application <=> On-board memory bank interface OCP channels
- DMA OCP channel 0 <=> BRAM block OCP channel
- DMA OCP channel 0 <=> On-board memory bank interface OCP channels
- Other DMA OCP channels <=> BRAM block OCP channel

5.9.5.5.1 Direct Slave On-Board Memory OCP Address Space Splitter Block

Referring to item 1 in Figure 32, this instance of adb3_ocp_split_nb splits the Direct Slave on-board memory OCP channel into multiple secondary OCP channels, according to the address map in Table 125 below.

Index	Block	Type	Address Range
0	On-board memory bank 0	Memory	0x1000000000-0x1FFFFFFFFF
1	On-board memory bank 1	Memory	0x2000000000-0x2FFFFFFFFF
2	On-board memory bank 2	Memory	0x3000000000-0x3FFFFFFFFF
3	On-board memory bank 3	Memory	0x4000000000-0x4FFFFFFFFF
4	On-board memory bank 4	Memory	0x5000000000-0x5FFFFFFFFF
5	On-board memory bank 5	Memory	0x6000000000-0x6FFFFFFFFF
6	On-board memory bank 6	Memory	0x7000000000-0x7FFFFFFFFF

Table 125 : Uber Design Direct Slave On-Board Memory Address Map

The number of secondary OCP channels is defined by the constant **DMA_RAM_RANGE_TABLE** in the uber_pkg package. The range of this constant is controlled by the **MEM_BANKS** constant defined in the adb3_target_inc_pkg package.

> **Note**
> Reads of undefined areas of the address space return data consisting of 0xDEADC0DE. Writes to undefined areas have no effect.

5.9.5.5.2 BRAM OCP Multiplexor Block

Referring to item 2 in Figure 32, this instance of adb3_ocp_mux_nb multiplexes all OCP channels which require to be connected to the BRAM block:

- Direct Slave BRAM OCP channel (Direct Slave BRAM Address Space)
- DMA OCP channel 0 (DMA channel 0 splitter secondary OCP channel 0)
- Remaining DMA OCP channels

5.9.5.5.3 DMA Channel 0 OCP Address Space Splitter Block

Referring to item 3 in Figure 32, this instance of adb3_ocp_split_nb splits DMA OCP channel 0 into multiple secondary OCP channels according to the address map in Table 126.

Index	Block	Type	Address Range
0	BRAM	Memory	0x0000080000-0x00000FFFFF
1	On-board memory bank 0	Memory	0x1000000000-0x1FFFFFFFFF
2	On-board memory bank 1	Memory	0x2000000000-0x2FFFFFFFFF
3	On-board memory bank 2	Memory	0x3000000000-0x3FFFFFFFFF
4	On-board memory bank 3	Memory	0x4000000000-0x4FFFFFFFFF
5	On-board memory bank 4	Memory	0x5000000000-0x5FFFFFFFFF
6	On-board memory bank 5	Memory	0x6000000000-0x6FFFFFFFFF
7	On-board memory bank 6	Memory	0x7000000000-0x7FFFFFFFFF

Table 126 : Uber Design DMA Channel 0 Address Map

The number of secondary OCP channels is defined by the constant **DMA_ADDR_RANGE_TABLE** in the uber_pkg package. The range of this constant is controlled by the **MEM_BANKS** constant defined in the adb3_target_inc_pkg package.

Note
Reads of undefined areas of the address space return data consisting of 0xDEADC0DE. Writes to undefined areas have no effect.

5.9.5.5.4 On-Board Memory Bank OCP Multiplexors

Items 4, 5, 6 and 7 in Figure 32 are instances of adb3_ocp_mux_nb whose purpose is to enable multiple OCP channels to access the the on-board memory banks:

- Direct Slave on-board memory OCP channel (On-Board Memory splitter secondary OCP channels 0 to 3
- On-board memory application OCP channels (On-Board Memory Application Block)
- DMA OCP channel 0 (DMA channel 0 splitter secondary OCP channels 1 to 4)

5.9.5.6 BRAM Block

This block is implemented by **hdl/vhdl/examples/uber/common/blk_bram.vhd** and contains a RAM composed of BlockRAM primitives. The following agents can read and write BRAM via the OCP switching block:

- The Direct Slave OCP channel, via the BRAM access window.
- DMA channel 0, using the BRAM address range in Table 126.
- Any other DMA channel, where the BRAM block is aliased throughout the entire OCP address space.

Figure 33 shows the BRAM block and its associated part of the OCP switching block.

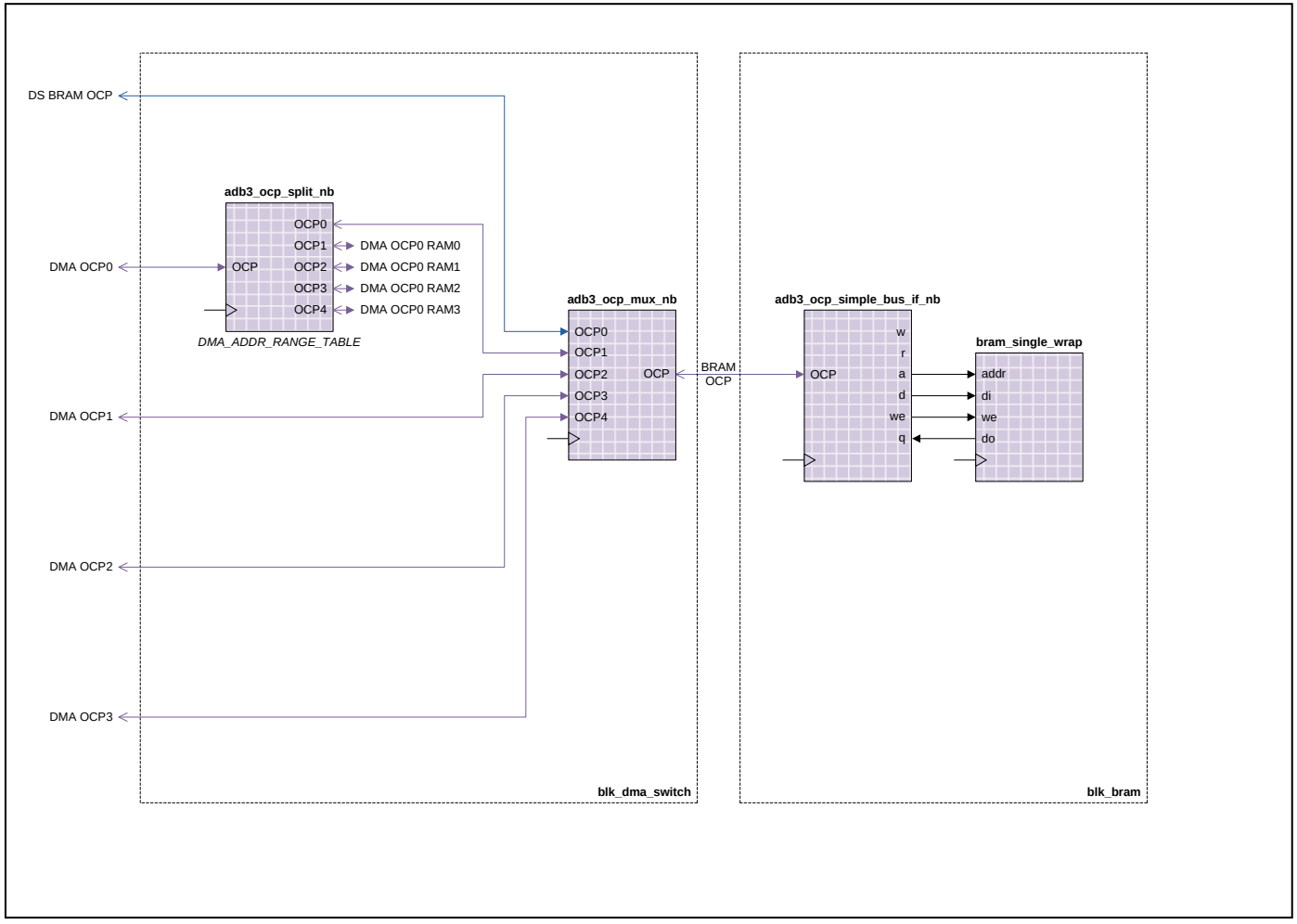

Figure 33 : Uber BRAM Block Diagram

An instance of adb3_ocp_simple_bus_if_nb is used, together with BlockRAM primitives, to implement this block.

A wrapper for a Virtex-6 BlockRAM called **bram_single_wrap**, implemented by **hdl/vhdl/examples/uber/common/bram_single_wrap.vhd** is instantiated multiple times to create a 512 KiB RAM.

The address to **bram_single_wrap** is replicated and re-timed to improve timing.

5.9.5.7 On-Board Memory Interface Block

The on-board memory interface block is implemented by the **blk_mem_if** block which is model dependent.

Table 127 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-6TL	admxrc6tl/blk_mem_if_6tl.vhd
ADM-XRC-6T1	admxrc6t1/blk_mem_if_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/blk_mem_if_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/blk_mem_if_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/blk_mem_if_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/blk_mem_if_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/blk_mem_if_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/blk_mem_if_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/blk_mem_if_admxrc7v1.vhd

Table 127 : Available Variants of blk_mem_if Block

The **blk_mem_if** block instantiates a memory interface for each bank of on-board memory. The following agents can read and write on-board memory banks via the OCP switching block:

- Direct Slave OCP channel, via the on-board memory access window.
- DMA channel 0, using the appropriate address range in Table 126.
- On-board memory application block.

The number of memory interface banks is defined by the **MEM_BANKS** constant in the adb3_target_inc_pkg package.

The number of DDR3 SDRAM interfaces is defined by the **DDR3_BANKS** constant in the adb3_target_inc_pkg package.

For models which are fitted only with DDR3 SDRAM, for example the **ADM-XRC-6T1**, **DDR3_BANKS** is equal to **MEM_BANKS**. This arrangement is subject to change, should support for models with on-board memory other than DDR3 SDRAM be added to the SDK.

Figure 34 shows the on-board memory interface block and its associated part of the OCP switching block.

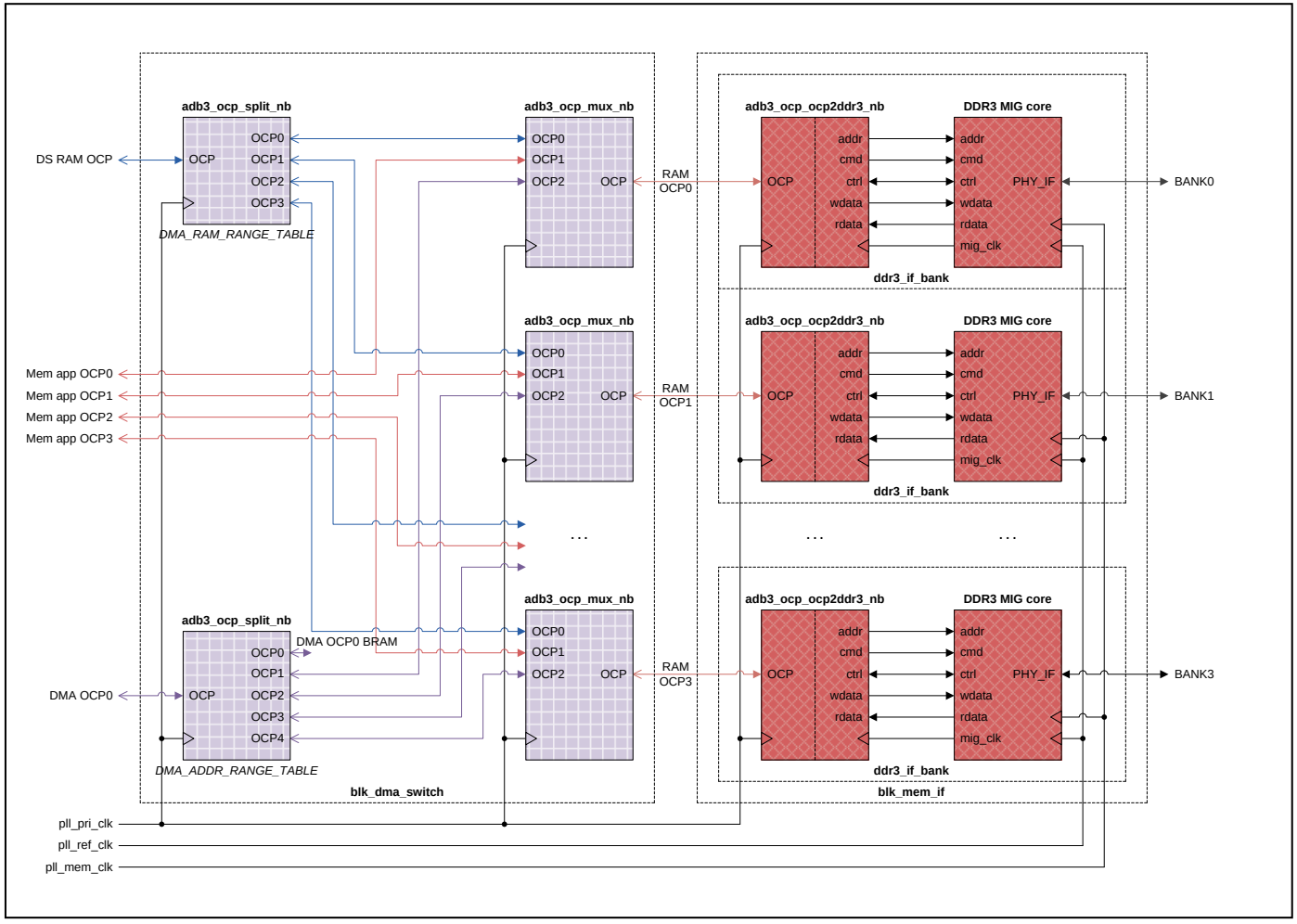

Figure 34 : Uber Memory Interface Block Diagram

For each bank of DDR3 SDRAM, this block instantiates a ddr3_if_bank component. In addition, this block contains logic common to all banks of DDR3 SDRAM such as reset logic and an **IDELAYCTRL** instance.

For each bank of DDR3 SDRAM running at 400MHz = 800MT/s, and 32-bits wide, the theoretical maximum transfer rate is 800MT/s x 4 = 3.2GB/s.

For each bank of DDR3 SDRAM running at 800MHz = 1600MT/s, and 16-bits wide, the theoretical maximum transfer rate is 1600MT/s x 2 = 3.2GB/s.

The actual transfer rate will be affected by DDR3 housekeeping and the efficiency of the Xilinx DDR3 MIG controller.

The status of the memory interfaces, which indicates whether or not training and initialisation was successful for each bank, can be determined via the BANKx_STAT registers in the on-board memory register block.

5.9.5.8 On-Board Memory Application Block

This block is implemented by **hdl/vhdl/examples/uber/common/blk_mem_app.vhd** and is intended to contain code that performs some useful function on the on-board memory banks.

In the **Uber** design as supplied by Alpha Data, the memory application is an FPGA-driven memory test. Therefore, it instantiates one memory test block per bank of on-board memory, allowing some or all of the on-board memory banks to be simultaneously tested. The advantage of the FPGA-driven memory test, over a host-driven memory test where test data is generated and verified on the host and transferred via the Bridge, is that the FPGA-driven memory test is faster and able to stress-test the memory subsystem by operating all banks simultaneously.

The memory test block is implemented by **hdl/vhdl/examples/common/mem_apps/blk_mem_test.vhd**.

> **Note**
> As this block has access to all banks of on-board memory, it is suitable for prototyping processing algorithms that operate on large amounts of data. Users are therefore encouraged to replace the logic in this block with their own application.

5.9.5.9 ChipScope Connection Block (optional)

This block optionally instantiates logic that enables several ADB3 OCP channels to be monitored using Xilinx ChipScope. When the **CHIPSCOPE_ON** constant in **hdl/vhdl/examples/uber/uber.vhd** is **true**, ChipScope logic is instantiated. Refer to blk_chipscope for a functional description.

> **Note**
> Before performing the first bitstream build of **Uber** with **CHIPSCOPE_ON** set to **true**, the ChipScope ILA core **chipscope_ila.ngc** and ICON core **chipscope_icon.ngc** must be generated using the script **gen_chipscope.tcl**. Refer to Xilinx ChipScope Core Generation (ICON/ILA) for details.

5.9.5.10 Design Package (uber_pkg)

The package **uber_pkg** defines types, constants, and functions which are used by the **Uber** example FPGA design. Table 128 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-6TL	common/uber_pkg.vhd
ADM-XRC-6T1	common/uber_pkg.vhd
ADM-XRC-6TGE	common/uber_pkg.vhd
ADM-XRC-6T-ADV8	common/uber_pkg.vhd
ADM-XRC-6T-DA1	common/uber_pkg.vhd
ADPE-XRC-6T	common/uber_pkg.vhd
ADPE-XRC-6T-L	common/uber_pkg.vhd
ADM-XRC-7K1	common/uber_pkg_s7.vhd
ADM-XRC-7V1	common/uber_pkg_s7.vhd

Table 128 : Available Variants of uber_pkg Package

Definitions are as follows:

Direct slave interface memory map constants

- Memory map sections base address constants (type **adb3_ocp_addr_s**).
- Memory map sections mask address constants (type **adb3_ocp_addr_s**).
- Memory map sections range constants (type **adb3_ocp_addr_range_t**).
- Memory map address range table constant **DS_ADDR_RANGE_TABLE** (type **adb3_ocp_addr_range_table_t).**).
- Register memory map sections base address constants (type **adb3_ocp_addr_s**).
- Register memory map sections mask address constants (type **adb3_ocp_addr_s**).
- Register memory map sections range constants (type **adb3_ocp_addr_range_t**).
- Direct slave memory map address range table constant **DS_REG_RANGE_TABLE** (type **adb3_ocp_addr_range_table_t**).
- Register memory map sections register offsets (type **natural**).
- Register memory map sections register addresses (type **adb3_ocp_addr_s**).

DMA interface memory map constants

- Memory map sections base address constants (type **adb3_ocp_addr_s**).
- Memory map sections mask address constants (type **adb3_ocp_addr_s**).
- Memory map sections range constants (type **adb3_ocp_addr_range_t**).
- Full memory map address range table constant **DMA_FULL_RANGE_TABLE** (type **adb3_ocp_addr_range_table_t**).
- Active memory map address range table constant **DMA_ADDR_RANGE_TABLE** (type **adb3_ocp_addr_range_table_t**).
- On-board RAM memory map address range table constant **DMA_RAM_RANGE_TABLE** (type **adb3_ocp_addr_range_table_t**).

Clock frequency measurement types

- **clk_vec_sel_t**. Type definition for clock select index vector.
- **clk_vec_range_t**. Type definition for clock select index number.
- **mgt_clk_pin_t**. Type definition for all MGT double ended clock inputs.
- **mgt_clk_buf_t**. Type definition for all MGT single ended buffered clock inputs.

- **clk_vec_t**. Type definition for all internal clocks/external clock inputs.
- **clk_vec_stat_t**. Type definition for measurement status for all internal clocks/external clock inputs.
- **clk_vec_freq_t**. Type definition for measurement frequency for all internal clocks/external clock inputs.

Clock frequency measurement constants

- Assignment of an index vector (type **clk_vec_sel_t**) to all internal/external clocks.
- Assignment of an index number (type **clk_vec_range_t**) to all internal/external clocks.

Memory interface array types

- **mem_clk_array_t**. Array of all memory interface bank clock signals (7 Series models).
- **mem_if_stat_array_t**. Array of all memory interface bank status vectors.
- **mem_if_err_array_t**. Array of all memory interface bank error vectors.
- **mem_if_rdy_array_t**. Array of all memory interface bank ready signals.
- **mem_if_debug_array_t**. Array of all memory interface bank debug vectors.

Memory application array types

- **mem_app_go_array_t**. Array of all memory application bank go signals.
- **mem_app_offset_array_t**. Array of all memory application bank test offset vectors.
- **mem_app_length_array_t**. Array of all memory application bank test length vectors.
- **mem_app_done_array_t**. Array of all memory application bank done signals.
- **mem_app_err_array_t**. Array of all memory application bank error signals.
- **mem_app_err_ph_array_t**. Array of all memory application bank error phase vectors.
- **mem_app_err_addr_array_t**. Array of all memory application bank error address vectors.

Component definitions

- blk_clocks
- blk_direct_slave
- blk_ds_simple_test
- blk_ds_clk_read
- blk_ds_io_test
- blk_ds_int_test
- blk_ds_mem_reg
- blk_ds_info
- blk_dma_switch
- blk_bram
- blk_mem_if
- blk_mem_app
- blk_chipscope
- blk_clock_freq

5.9.6 Testbench Description

The **uber** example FPGA design testbench tests operation of the **uber** example FPGA design.

It exists in two variants, one using Alpha Data MPTL interface IP (PCIe in bridge FPGA), the other using Alpha Data PCIe interface IP (PCIe in target FPGA). Table 129 lists the available variants:

Model	Interface	Filename relative to hdl/vhdl/examples/simple/common/
ADM-XRC-6TL	MPTL	test_uber.vhd
ADM-XRC-6T1	MPTL	test_uber.vhd
ADM-XRC-6TGE	MPTL	test_uber.vhd
ADM-XRC-6T-ADV8	PCIe	test_uber_pcie.vhd
ADM-XRC-6T-DA1	MPTL	test_uber.vhd
ADPE-XRC-6T	MPTL	test_uber.vhd
ADPE-XRC-6T-L	MPTL	test_uber.vhd
ADM-XRC-7K1	MPTL	test_uber.vhd
ADM-XRC-7V1	MPTL	test_uber.vhd

Table 129 : Available Variants of the Uber Example Design Testbench

It consists of the following functions:

- Clock generation and test for the testbench and the Unit Under Test (UUT).
- The Unit Under Test (UUT), which is the one and only instance of the top-level uber block.
- Bridge MPTL interface, using an instance of mptl_if_bridge_wrap or, host PCIe interface, using an instance of pcie_if_host_wrap.
- OCP channel probes, using instances of adb3_ocp_transaction_probe.
- Stimulus generation and verification.
- On-board memory simulation models.

Figure 35 shows the testbench and main elements of the uber FPGA design using MPTL interface IP.

Figure 36 shows the testbench and main elements of the uber FPGA design using PCIe interface IP.

Figure 37 shows the hierarchy of the uber testbench using MPTL interface IP.

Figure 38 shows the hierarchy of the uber testbench using PCIe interface IP.

The testbench includes the following packages:

- ADB3 OCP profile definition package (adb3_ocp)
- ADB3 OCP testbench package (adb3_ocp_tb_pkg)
- ADB3 target include package (adb3_target_inc_pkg)
- ADB3 target testbench include package (adb3_target_tb_inc_pkg)
- Testbench package (uber_tb_pkg)

Figure 28 shows the design package dependencies.

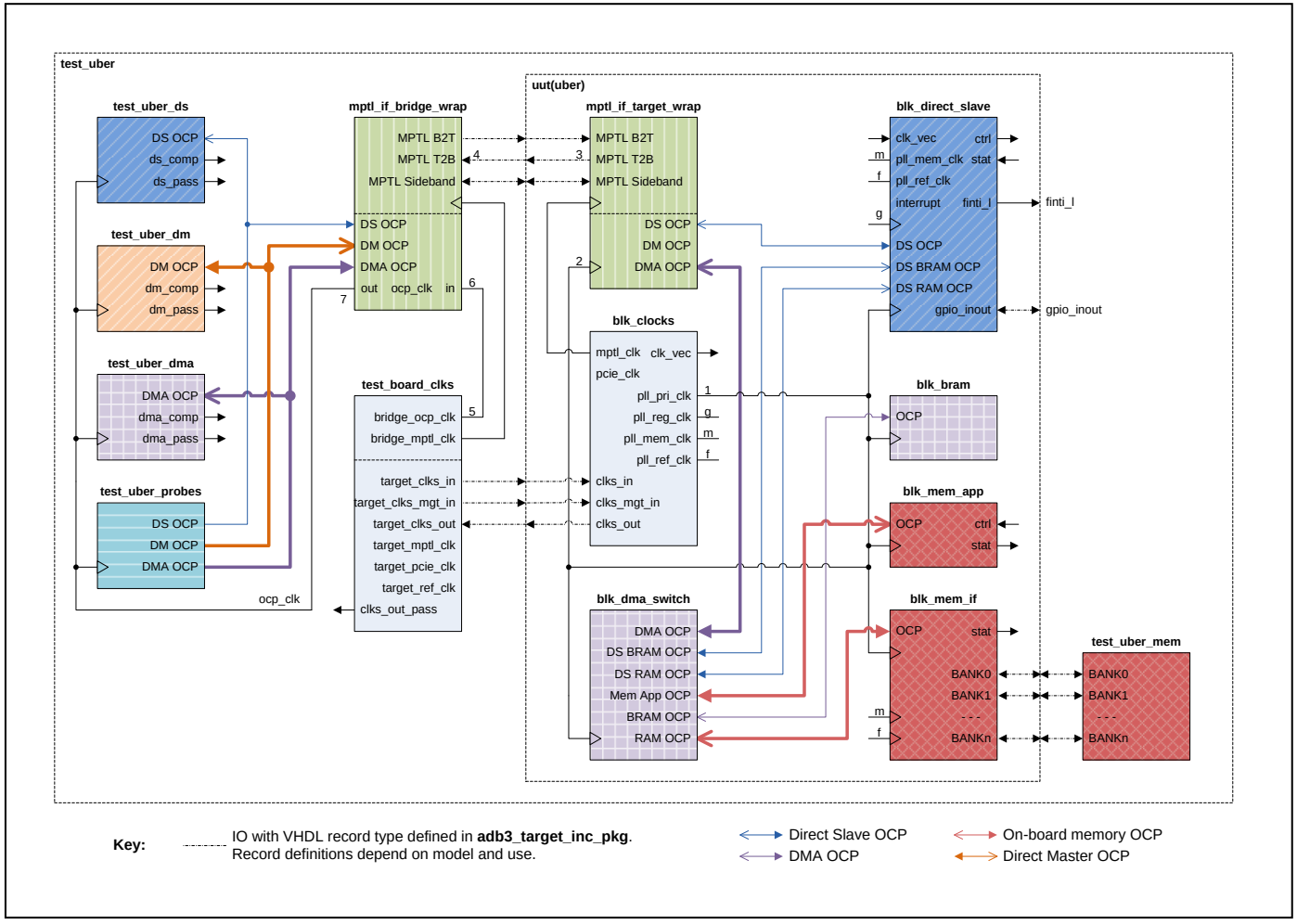

Figure 35 : Uber Design Testbench and Top Level Block Diagram (MPTL)

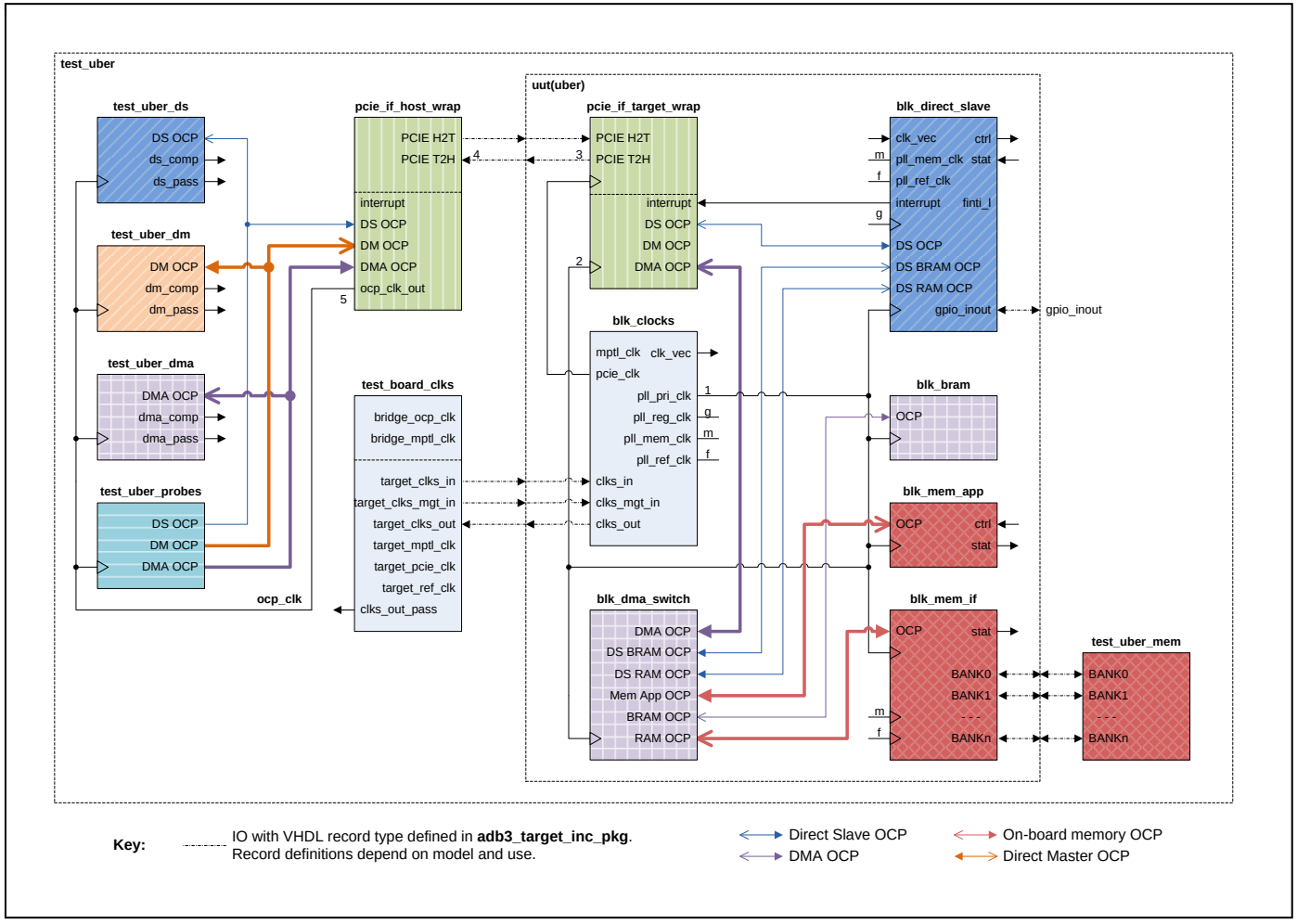

Figure 36 : Uber Design Testbench and Top Level Block Diagram (PCIe)

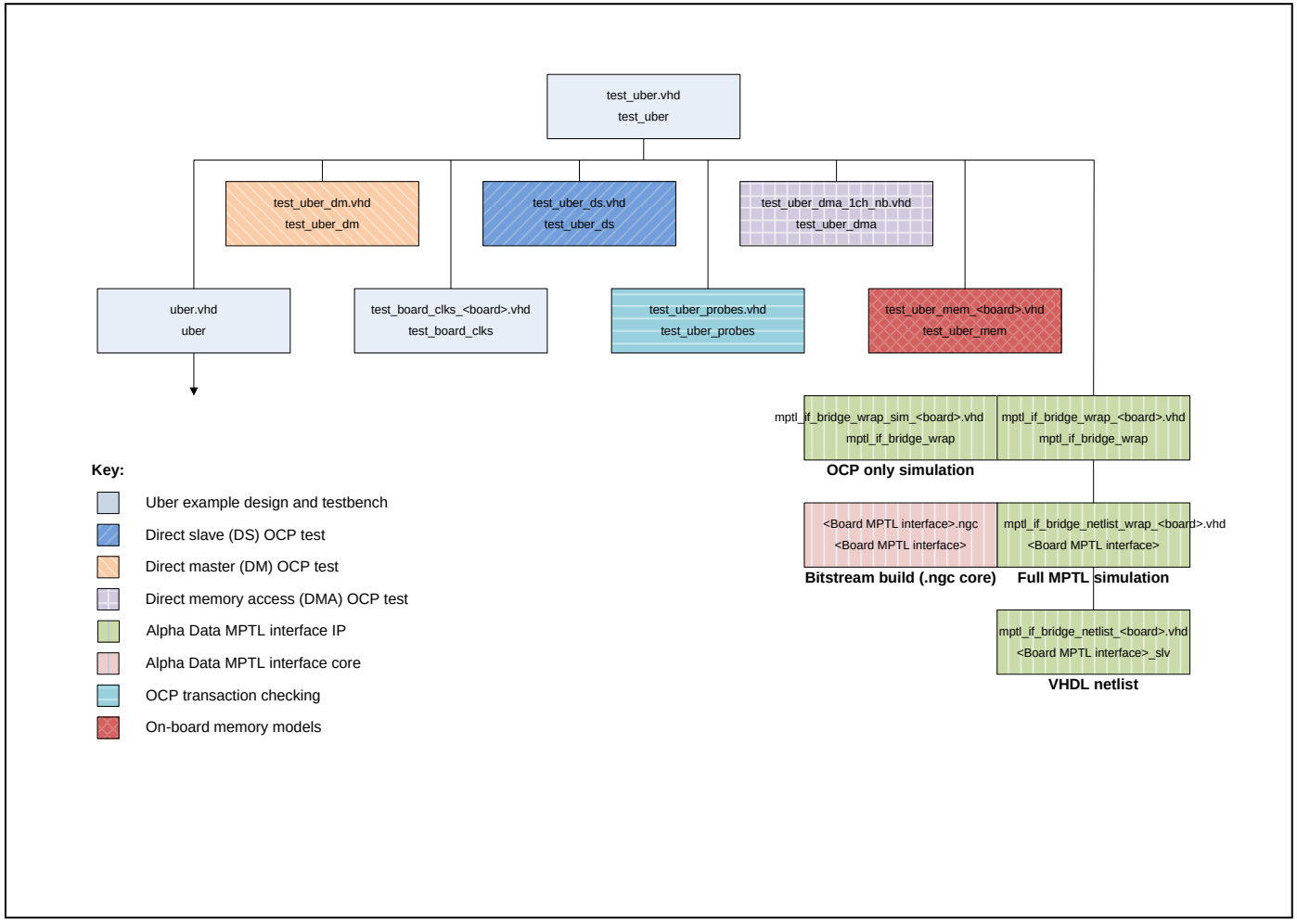

Figure 37 : Uber Design Testbench Hierarchy (MPTL)

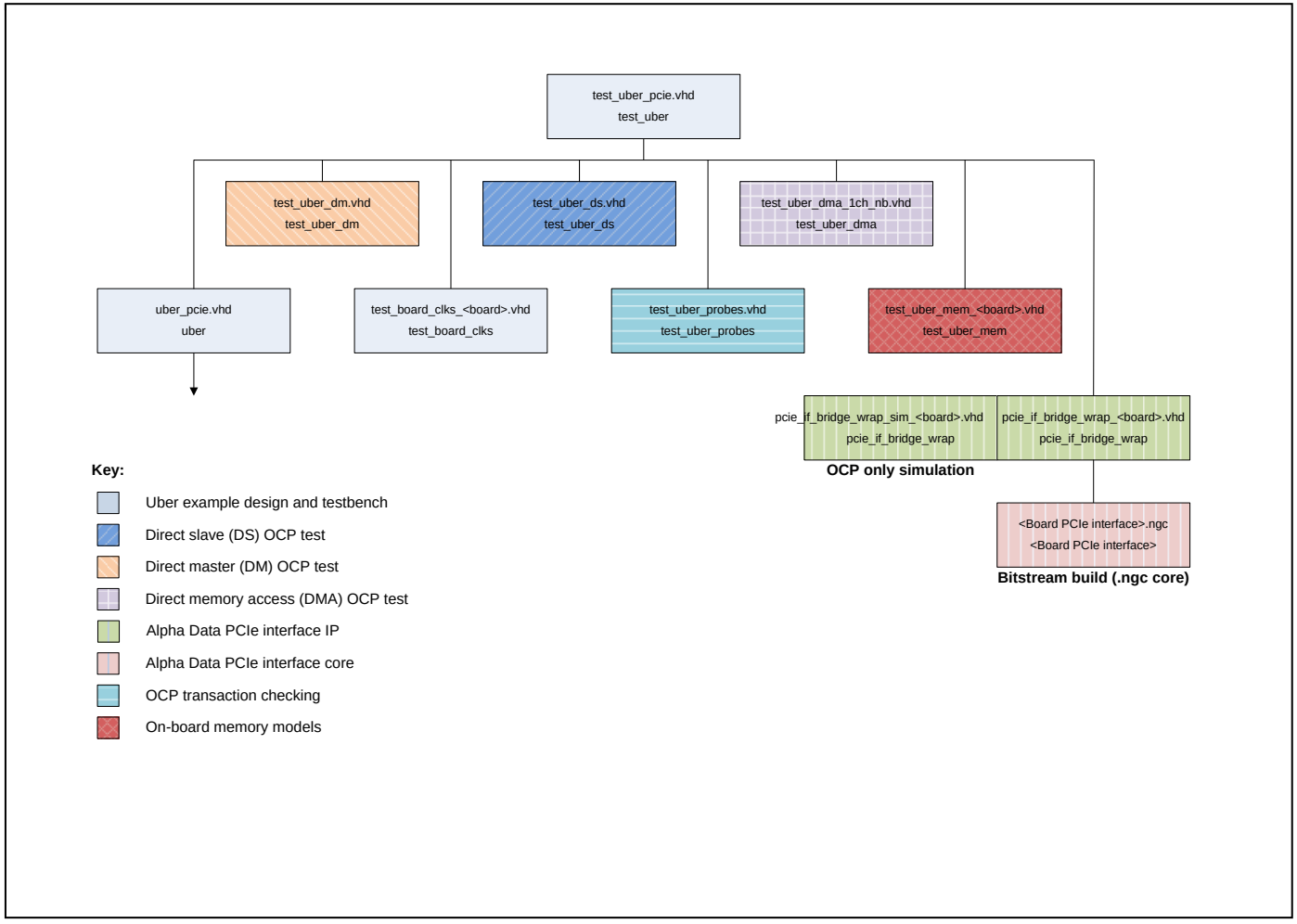

Figure 38 : Uber Design Testbench Hierarchy (PCIe)

5.9.6.1 Clock Generation and Test

The testbench uses the test_board_clks block to implement this function.

Target Clocks

- It generates the **clks_in** and **clks_mgt_in** clocks according to which model is selected. These clocks drive the unit under test (uber).
- **clks_in** is a signal of record type **clks_in_t** that drives the UUT's top-level **clks_in** port, and is a bundle of all of the non-MGT-related clock inputs. It is generated in a model-specific way, depending on the package adb3_target_inc_pkg. Among others, it contains the 200 MHz reference clock from which the main clocks in **Uber** are derived using its Clock and Reset Generation block.
- **clks_mgt_in** is a signal of record type **clks_mgt_in_t** that drives the UUT's top-level **clks_mgt_in** port, and is a bundle of all of the MGT-related clock inputs. It is generated in a model-specific way, depending on the package adb3_target_inc_pkg.
- It tests the **clks_out** clock frequency according to which model is selected. These clocks are generated by the unit under test (uber). The test result is indicated by the **clks_out_pass** output.

Bridge MPTL Interface Clock

- It generates the **bridge_mptl_clk** clock according to which model is selected. This clock drives the **mptl_clk** differential clock input on the bridge MPTL interface block.

Bridge OCP Clock (MPTL)

- It generates the **bridge_ocp_clk** clock according to which model is selected. This clock drives the **ocp_clk_in** clock input on the bridge MPTL interface block.
- This clock is only used during Full MPTL simulation. Refer to bridge MPTL interface for details.

5.9.6.2 Bridge MPTL Interface

The MPTL (Multiplexed Packet Transport Link) is the data channel which connects the Bridge and Target FPGAs.

This block wraps up the bridge MPTL interface core, instantiating an OCP to MPTL interface appropriate to the model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the MPTL. Refer to the component mptl_if_bridge_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the mptl_if_bridge_wrap **mptl_b2t** signals to the mptl_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the mptl_if_target_wrap **mptl_t2b** signals to the mptl_if_bridge_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 35 as the route consisting of points 1, 2, 3, 4 and 7.

Full MPTL simulation

- The testbench Direct Slave and DMA OCP m2s signals are input to the mptl_if_bridge_wrap.
- The UUT Direct Slave and DMA OCP m2s signals are output from the mptl_if_target_wrap.
- Apart from the packetisation, multiplexing and demultiplexing that occurs in the MPTL interfaces (both Bridge and Target), the arrangement is transparent. In other words, behaviour is as if the stimulus were applied directly to the Target FPGA's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 35 as the route consisting of points 5, 6

and 7.

The mptl_if_bridge_wrap output **mptl_online** indicates that the MPTL interface is active and stable. It is used by the testbench to generate the **mptl_online_long** signal which it monitors. Simulation will be terminated with an error message if it becomes inactive. This may occur if, for example, a protocol error arises on the MPTL signals.

The mptl_if_bridge_wrap output **dma_abort** indicates the status of the UUT's **dma_abort** signal.

5.9.6.3 Host PCIe Interface

The PCIe (PCI express) link is the data channel which connects the host and the target FPGA.

This block wraps up the host PCIe interface core, instantiating an OCP to PCIe interface appropriate to the model in use. The purpose of the block is to connect the Direct Slave and DMA OCP channels within the FPGA testbench to the PCIe. Refer to the component pcie_if_host_wrap for details.

OCP-only simulation

- The testbench Direct Slave and DMA OCP m2s signals are routed directly via the pcie_if_host_wrap **pcie_h2t** signals to the pcie_if_target_wrap UUT Direct Slave and DMA OCP m2s signals.
- The UUT Direct Slave and DMA OCP s2m signals are routed directly via the pcie_if_target_wrap **pcie_t2b** signals to the pcie_if_host_wrap testbench Direct Slave and DMA OCP s2m signals.
- In other words, the stimulus is applied directly to the Target FPGA's OCP channels, and the response is returned directly to the testbench's OCP channels.
- The testbench OCP clock **ocp_clk_out** path is shown in Figure 36 as the route consisting of points 1, 2, 3, 4 and 5.

The pcie_if_host_wrap output **dma_abort** indicates the status of the UUT's **dma_abort** signal.

5.9.6.4 OCP Channel Probes

This function monitors the Direct Slave and DMA OCP channels for addressing/transaction problems. It generates warnings/errors if it detects an illegal OCP operation. A probe error will result in a 'FAILED' **Uber** simulation result. It uses the component adb3_ocp_transaction_probe.

5.9.6.5 Stimulus Generation and Verification

This function consists of a set of processes that generate stimulus and verify the results of the simulation via the mptl_if_bridge_wrap instance.

5.9.6.5.1 Non-OCP Functions

The top level of the testbench verifies a few features of the UUT (the **Uber** design) that cannot be tested by application of OCP stimulus. These tests are explained in the next few subsections.

5.9.6.5.1.1 Clock Output Test

Note: all filenames mentioned in this section are relative to the path **hdl/vhdl/examples/uber/common/**.

The process **clk_out_p** continually monitors the **clks_out_pass** output from the test_board_clks block. This block measures the frequencies of the bundle of clocks **clks_out** driven by the UUT and compares them with expected frequencies defined by **CLKS_OUT_FREQ** in the uber_tb_pkg package.

Test complete and pass/fail indications are returned using the **top_comp.clk_out_complete** and **top_pass.clk_out_passed** signals respectively.

Results from this test are only reported on failure.

5.9.6.5.1.2 MPTL GPIO Bus Test (MPTL)

Note: all filenames mentioned in this section are relative to the path **hdl/vhdl/examples/uber/common/**.

The process **mptl_gpio_p** verifies that the general purpose I/O (GPIO) pins between the Bridge and Target FPGAs behave as expected. The UUT (top-level of **uber**) loops back these GPIO pins so that whatever value is driven into the top-level port **gpio_b2t** in **uber.vhd** is driven out of the **gpio_t2b** port.

The testbench drives the constant value X"F1D0" onto the **gpio_b2t** port of the UUT, so the process **mptl_gpio_p** verifies that the UUT drives the same value out of its **gpio_t2b** port.

Test complete and pass/fail indications are returned using the **top_comp.mptl_gpio_complete** and **top_pass.mptl_gpio_passed** signals respectively.

Example results from this test are documented in MPTL GPIO bus test results.

5.9.6.5.1.3 DMA Abort Bus Test

Note: all filenames mentioned in this section are relative to the path **hdl/vhdl/examples/uber/common/**.

The process **dma_abort_p** verifies that the Target FPGA never attempts to abort a DMA transfer. If any bit of the signal **dma_abort** driven by the mptl_if_bridge_wrap/pcie_if_host_wrap is asserted, it indicates that the UUT is attempting to abort a DMA transfer. This should never happen by design. The process therefore verifies that all bits of the **dma_abort** signal are always zero.

Test complete and pass/fail indications are returned using the **top_comp.dma_abort_complete** and **top_pass.dma_abort_passed** signals respectively.

Results from this test are only reported on failure.

5.9.6.5.2 Direct Slave OCP Channel

Testing of the direct slave OCP channel is implemented by the test_uber_ds block which is model dependent. Table 130 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/common/
ADM-XRC-6TL	common/test_uber_ds.vhd
ADM-XRC-6T1	common/test_uber_ds.vhd
ADM-XRC-6TGE	common/test_uber_ds.vhd
ADM-XRC-6T-ADV8	common/test_uber_ds.vhd
ADM-XRC-6T-DA1	common/test_uber_ds.vhd
ADPE-XRC-6T	common/test_uber_ds_adpexrc6t.vhd
ADPE-XRC-6T-L	common/test_uber_ds_adpexrc6tl.vhd
ADM-XRC-7K1	common/test_uber_ds.vhd
ADM-XRC-7V1	common/test_uber_ds.vhd

Table 130 : Available Variants of test_uber_ds Package

The component provides test stimulus to, and verifies test results from, the UUT's OCP Direct Slave channel.

It uses the **adb3_ocp_sim_write_reg32** and **adb3_ocp_sim_read_reg32** procedures to perform 32-bit register writes and reads using ADB3 OCP. These procedures are defined in the ADB3 OCP testbench package

(adb3_ocp_tb_pkg). The **adb3_ocp_sim_read_reg32** procedure is blocking, so all OCP response data must be returned before it completes.

> **Note**
> 32-bit Register addresses used by the testbench are byte addresses which should be 4-byte aligned. ADB3 OCP protocol addresses are also byte addresses, but as the data is 128-bits wide, they are 16-byte aligned.

test_uber_ds performs several tests, which are detailed in the following subsections.

5.9.6.5.2.1 Simple Test

This test exercises the Simple Test Register Block as follows:

1. Writes the 32-bit value 0xCAFEFACE to the DATA register.
2. Reads back the DATA register and compares it with the expected value 0xECAFEFAC. If the expected and actual values do not match, the test is considered a failure.

Test complete and pass/fail indications are returned using the **ds_comp.simple_complete** and **ds_pass.simple_passed** signals respectively.

Example results from this test are documented in simple test results.

5.9.6.5.2.2 Clock Frequency Measurement Test

This test exercises the Clock Frequency Measurement Register Block as follows:

1. Clears the "measurement valid" flags for all clocks whose frequencies can be measured, by writing 0x80000000 to the CTRL/STAT register.
2. Selects **pll_reg_clk** by writing 0 (corresponding to **PLL_REG_CLK_SEL**) to the SEL register.
3. Waits for a measurement to be completed, by polling until bit 31 of the CTRL/STAT register is 1.
4. Reads the FREQ register and compares it with the expected frequency for **pll_reg_clk** of 80 MHz.

The test then performs similar steps for **pll_pri_clk**, which is the main OCP clock in **Uber**:

1. Selects **pll_pri_clk** by writing 1 (corresponding to **PLL_PRI_CLK_SEL**) to the SEL register.
2. Waits for a measurement to be completed, by polling until bit 31 of the CTRL/STAT register is 1.
3. Reads the FREQ register and compares it with the expected frequency for **pll_pri_clk** of 200 MHz.

The test then performs similar steps for **TEST_MGT_CLK**, which is defined in the package uber_tb_pkg):

1. Selects the **TEST_MGT_CLK** clock by writing **TEST_MGT_CLK_SEL** to the SEL register.
2. Waits for a measurement to be completed, by polling until bit 31 of the CTRL/STAT register.
3. Reads the FREQ register (see Table 36) and compares it with the expected frequency **TEST_MGT_CLK_FREQ**.

Lastly, the test checks the frequency of the **TEST_CUS_CLK** clock, which is defined in the package uber_tb_pkg:

1. Selects the **TEST_CUS_CLK** clock by writing **TEST_CUS_CLK_SEL** to the SEL register.
2. Waits for a measurement to be completed, by polling until bit 31 of the CTRL/STAT register.
3. Reads the FREQ register (see Table 36) and compares it with the expected frequency **TEST_CUS_CLK_FREQ**.

Note: When measured frequencies are compared with expected frequencies, they are permitted a small margin of error, since they are subject to quantization error due to the small number of reference clock cycles over which the measurement is performed (so that the simulation does not take excessive real time to complete). If the expected and actual values do not match to within the error margin, the test is considered a failure.

Test complete and pass/fail indications are returned using the **ds_comp.clock_complete** and **ds_pass.clock_passed** signals respectively.

Example results from this test are documented in clock frequency measurement test results.

5.9.6.5.2.3 XRM GPIO Test

This test is model dependent, and exercises with the XRM-related registers of the GPIO Test Register Block. This test section is enabled by the **XRM_GPIO_WIDTH** constant defined in the package adb3_target_inc_pkg. The test procedure is as follows:

1. Writes the 32-bit value 0x76543210 to the XRM_GPIO_DD_DATAO register. This sets the value to be driven onto the **dd_p(15:0)** and **dd_n(15:0)** XRM GPIO pins, but at this point these pins are still tristated (high-impedance).
2. Writes the 32-bit value 0x00000000 to the XRM_GPIO_DD_TRI register. This allows the value written in the previous step to be driven onto the **dd_p(15:0)** and **dd_n(15:0)** XRM GPIO pins.
3. Reads the XRM_GPIO_DD_DATAI register, to get the actual logic levels on the **dd_p(15:0)** and **dd_n (15:0)** XRM GPIO pins. It then compares the actual value with the expected value of 0x76543210. If the expected and actual values do not match, the test is considered a failure.
4. Writes the 32-bit value 0xFFFFFFFF to the XRM_GPIO_DD_TRI register in order to stop driving the **dd_p (15:0)** and **dd_n(15:0)** XRM GPIO pins.

Section complete and pass/fail indications are returned using the **ds_comp.frontio_complete** and **ds_pass.frontio_passed** signals respectively.

Example results from this test are documented in XRM GPIO test results.

5.9.6.5.2.4 Pn4/Pn6 GPIO Test

This test is model dependent, and exercises with the Pn4-related and Pn6-related registers of the GPIO Test Register Block. This test section is enabled by the **PN4_GPIO_WIDTH** and **PN6_GPIO_WIDTH** constants defined in the package adb3_target_inc_pkg. First, the Pn4-related registers are exercised as follows:

1. Writes the 32-bit values 0xAABBCCDD and 0x55443322 to the PN4_GPIO_P_DATAO and PN4_GPIO_N_DATAO registers, respectively. This sets the values to be driven onto the **gpio_p** and **gpio_n** Pn4 GPIO pins, but at this point these pins are still tristated (high-impedance).
2. Writes the 32-bit value 0x00000000 to both the PN4_GPIO_P_TRI and PN4_GPIO_N_TRI registers. This allows the values written in the previous step to be driven onto the **gpio_p** and **gpio_n** Pn4 GPIO pins.
3. Reads the PN4_GPIO_P_DATAI and PN4_GPIO_N_DATAI registers, to get the actual logic levels on the **gpio_p** and **gpio_n** Pn4 GPIO pins. It then compares the actual values with the expected values of 0xAABBCCDD and 0x55443322 respectively. If the expected and actual values do not match, the test is considered a failure.

   Note: If the constant **PN4_GPIO_WIDTH** from the package adb3_target_inc_pkg is less than 32, the top **32-PN4_GPIO_WIDTH** bits of each value are not used in the comparison.
4. Writes the 32-bit value 0xFFFFFFFF to both the PN4_GPIO_P_TRI and PN4_GPIO_N_TRI registers in order to stop driving **gpio_p** and **gpio_n** Pn4 GPIO pins.

The second part exercises with the Pn6-related registers of the GPIO Test Register Block as follows:

1. Writes the 32-bit values 0xAAAABBBB and 0xCCCCDDDD to the PN6_GPIO_MS_DATAO and PN6_GPIO_LS_DATAO registers, respectively. This sets the values to be driven onto the Pn6 GPIO pins, but at this point these pins are still tristated (high-impedance).
2. Writes the 32-bit value 0x00000000 to both the PN6_GPIO_MS_TRI and PN6_GPIO_LS_TRI registers. This allows the values written in the previous step to be driven onto the Pn6 GPIO pins.
3. Reads the PN6_GPIO_MS_DATAI and PN6_GPIO_LS_DATAI registers, to get the actual logic levels on the Pn6 GPIO pins. It then compares the actual values with the expected values of 0xAAAABBBB and 0xCCCCDDDD respectively. If the expected and actual values do not match, the test is considered a failure.

   Note: Depending on the constant **PN6_GPIO_WIDTH** from the package adb3_target_inc_pkg some of the bits of the actual and expected values may be unused in the comparison. For example, if **PN6_GPIO_WIDTH** is 46, the top 18 bits of the value read from PN6_GPIO_MS_DATAI are unused.
4. Writes the 32-bit value 0xFFFFFFFF to both the PN6_GPIO_MS_TRI and PN6_GPIO_LS_TRI registers in order to stop driving the Pn6 GPIO pins.

Section complete and pass/fail indications are returned using the **ds_comp.reario_complete** and **ds_pass.reario_passed** signals respectively.

Example results from this test are documented in Pn4/Pn6 GPIO test results.

5.9.6.5.2.5 FMC GPIO Test

This test is model dependent, and exercises with the FMC-related registers of the GPIO Test Register Block. This test section is enabled by the **FMC_LA_GPIO_WIDTH**, **FMC_HA_GPIO_WIDTH**, and **FMC_HB_GPIO_WIDTH** constants defined in the package adb3_target_inc_pkg. The test procedure is as follows:

1. Writes the 32-bit value 0x76543210 to the FMC_GPIO_LS_LA_P_DATAO register. This sets the value to be driven onto the **fmc_la_p(31:0)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).
2. Writes the 32-bit value 0x89ABCDEF to the FMC_GPIO_LS_LA_N_DATAO register. This sets the value to be driven onto the **fmc_la_n(31:0)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).
3. Writes the 32-bit value 0x00000000 to the FMC_GPIO_LS_LA_P_TRI register. This allows the value written in the previous step to be driven onto the **fmc_la_p(31:0)** FMC GPIO pins.
4. Writes the 32-bit value 0x00000000 to the FMC_GPIO_LS_LA_N_TRI register. This allows the value written in the previous step to be driven onto the **fmc_la_n(31:0)** FMC GPIO pins.
5. Reads the FMC_GPIO_LS_LA_P_DATAI register, to get the actual logic levels on the **fmc_la_p(31:0)** FMC GPIO pins. It then compares the actual value with the expected value of 0x76543210. If the expected and actual values do not match, the test is considered a failure.
6. Reads the FMC_GPIO_LS_LA_N_DATAI register, to get the actual logic levels on the **fmc_la_n(31:0)** FMC GPIO pins. It then compares the actual value with the expected value of 0x89ABCDEF. If the expected and actual values do not match, the test is considered a failure.
7. Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_LS_LA_P_TRI register in order to stop driving the **fmc_la_p(31:0)** FMC GPIO pins.
8. Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_LS_LA_N_TRI register in order to stop driving the **fmc_la_n(31:0)** FMC GPIO pins.

The test then performs similar steps for **fmc_la_p(33:32)** and **fmc_la_n(33:32)**:

1. Writes the 32-bit value 0x87654321 to the FMC_GPIO_MS_LA_P_DATAO register. This sets the value to

be driven onto the **fmc_la_p(33:32)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).

2. Writes the 32-bit value 0x789ABCDE to the FMC_GPIO_MS_LA_N_DATAO register. This sets the value to be driven onto the **fmc_la_n(33:32)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).
3. Writes the 32-bit value 0x00000000 to the FMC_GPIO_MS_LA_P_TRI register. This allows the value written in the previous step to be driven onto the **fmc_la_p(33:32)** FMC GPIO pins.
4. Writes the 32-bit value 0x00000000 to the FMC_GPIO_MS_LA_N_TRI register. This allows the value written in the previous step to be driven onto the **fmc_la_n(33:32)** FMC GPIO pins.
5. Reads the FMC_GPIO_MS_LA_P_DATAI register, to get the actual logic levels on the **fmc_la_p(33:32)** FMC GPIO pins. It then compares the actual value with the expected value of 01. If the expected and actual values do not match, the test is considered a failure.
6. Reads the FMC_GPIO_MS_LA_N_DATAI register, to get the actual logic levels on the **fmc_la_n(33:32)** FMC GPIO pins. It then compares the actual value with the expected value of 10. If the expected and actual values do not match, the test is considered a failure.
7. Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_MS_LA_P_TRI register in order to stop driving the **fmc_la_p(33:32)** FMC GPIO pins.
8. Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_MS_LA_N_TRI register in order to stop driving the **fmc_la_n(33:32)** FMC GPIO pins.

The test then performs similar steps for **ha_p(31:0)** and **ha_n(31:0)**:

1. Writes the 32-bit value **FMC_HA_DATA** to the FMC_GPIO_HA_P_DATAO register. This sets the value to be driven onto the **fmc_ha_p(31:0)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).
2. Writes the 32-bit value 0x89ABCDEF to the FMC_GPIO_HA_N_DATAO register. This sets the value to be driven onto the **fmc_ha_n(31:0)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).
3. Writes the 32-bit value 0x00000000 to the FMC_GPIO_HA_P_TRI register. This allows the value written in the previous step to be driven onto the **fmc_ha_p(31:0)** FMC GPIO pins.
4. Writes the 32-bit value 0x00000000 to the FMC_GPIO_HA_N_TRI register. This allows the value written in the previous step to be driven onto the **fmc_ha_n(31:0)** FMC GPIO pins.
5. Reads the FMC_GPIO_HA_P_DATAI register, to get the actual logic levels on the **fmc_ha_p(31:0)** FMC GPIO pins. It then compares the actual value with the expected value of 0x76543210. If the expected and actual values do not match, the test is considered a failure.
6. Reads the FMC_GPIO_HA_N_DATAI register, to get the actual logic levels on the **fmc_ha_n(31:0)** FMC GPIO pins. It then compares the actual value with the expected value of 0x89ABCDEF. If the expected and actual values do not match, the test is considered a failure.
7. Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_HA_P_TRI register in order to stop driving the **fmc_ha_p(31:0)** FMC GPIO pins.
8. Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_HA_N_TRI register in order to stop driving the **fmc_ha_n(31:0)** FMC GPIO pins.

The test then performs similar steps for **fmc_hb_p(31:0)** and **fmc_hb_n(31:0)**:

1. Writes the 32-bit value **FMC_HB_DATA** to the FMC_GPIO_HB_P_DATAO register. This sets the value to be driven onto the **fmc_hb_p(31:0)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).
2. Writes the 32-bit value 0x89ABCDEF to the FMC_GPIO_HB_N_DATAO register. This sets the value to be

driven onto the **fmc_hb_n(31:0)** FMC GPIO pins, but at this point these pins are still tristated (high-impedance).

3 Writes the 32-bit value 0x00000000 to the FMC_GPIO_HB_P_TRI register. This allows the value written in the previous step to be driven onto the **fmc_hb_p(31:0)** FMC GPIO pins.

4 Writes the 32-bit value 0x00000000 to the FMC_GPIO_HB_N_TRI register. This allows the value written in the previous step to be driven onto the **fmc_hb_n(31:0)** FMC GPIO pins.

5 Reads the FMC_GPIO_HB_P_DATAI register, to get the actual logic levels on the **fmc_hb_p(31:0)** FMC GPIO pins. It then compares the actual value with the expected value of 0x76543210. If the expected and actual values do not match, the test is considered a failure.

6 Reads the FMC_GPIO_HB_N_DATAI register, to get the actual logic levels on the **fmc_hb_n(31:0)** FMC GPIO pins. It then compares the actual value with the expected value of 0x89ABCDEF. If the expected and actual values do not match, the test is considered a failure.

7 Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_HB_P_TRI register in order to stop driving the **fmc_hb_p(31:0)** FMC GPIO pins.

8 Writes the 32-bit value 0xFFFFFFFF to the FMC_GPIO_HB_N_TRI register in order to stop driving the **fmc_hb_n(31:0)** FMC GPIO pins.

Section complete and pass/fail indications are returned using the **ds_comp.frontio_complete** and **ds_pass.frontio_passed** signals respectively.

Example results from this test are documented in FMC GPIO test results.

5.9.6.5.2.6 Secondary GPIO Test

This test is model dependent, and exercises with the secondary GPIO related registers of the GPIO Test Register Block. This test section is enabled by the **SEC_GPIO_WIDTH** constants defined in the package adb3_target_inc_pkg. The test procedure is as follows:

1 Writes the 32-bit value **SEC_DATA** to the SEC_GPIO_P_DATAO register. This sets the value to be driven onto the **gpio_p(31:0)** Secondary GPIO pins, but at this point these pins are still tristated (high-impedance).

2 Writes the 32-bit value **not(SEC_DATA)** to the SEC_GPIO_N_DATAO register. This sets the value to be driven onto the **gpio_n(31:0)** Secondary GPIO pins, but at this point these pins are still tristated (high-impedance).

3 Writes the 32-bit value 0x00000000 to the SEC_GPIO_P_TRI register. This allows the value written in the previous step to be driven onto the **gpio_p(31:0)** Secondary GPIO pins.

4 Writes the 32-bit value 0x00000000 to the SEC_GPIO_N_TRI register. This allows the value written in the previous step to be driven onto the **gpio_n(31:0)** Secondary GPIO pins.

5 Reads the SEC_GPIO_P_DATAI register, to get the actual logic levels on the **gpio_p(31:0)** Secondary GPIO pins. It then compares the actual value with the expected value of **SEC_DATA**. If the expected and actual values do not match, the test is considered a failure.

6 Reads the SEC_GPIO_N_DATAI register, to get the actual logic levels on the **gpio_n(31:0)** Secondary GPIO pins. It then compares the actual value with the expected value of **not(SEC_DATA)**. If the expected and actual values do not match, the test is considered a failure.

7 Writes the 32-bit value 0xFFFFFFFF to the SEC_GPIO_P_TRI register in order to stop driving the **gpio_p (31:0)** Secondary GPIO pins.

8 Writes the 32-bit value 0xFFFFFFFF to the SEC_GPIO_N_TRI register in order to stop driving the **gpio_n (31:0)** Secondary GPIO pins.

Section complete and pass/fail indications are returned using the **ds_comp.reario_complete** and

ds_pass.reario_passed signals respectively.

Example results from this test are documented in Secondary GPIO test results.

5.9.6.5.2.7 Interrupt Test

This test exercises the Interrupt Test Register Block. The **DO_INTERRUPT_NUM** constant is defined in **test_uber_ds.vhd**. Test operation can be expressed in pseudocode as the following algorithm:

1. Unmask all interrupts by writing 0 to the MASK register.
2. Read back the MASK register and verify that it has the expected value of 0. If the expected and actual values do not match, the test is considered a failure.
3. Write the value 0xFFFFFFF to the COUNT register.
4. Verify that the COUNT register has the expected value 0xFFFFFFF.
5. For n in 0 to **DO_INTERRUPT_NUM**-1 do
    a. Generate interrupt n, by writing the value 2^n to the SET register.
    b. Wait for the interrupt signal **linti_l** (MPTL)/**interrupt** (PCIe) to be asserted. This is a falling-edge sensitive signal in the testbench that is driven low by the top-level port of the UUT whenever at least one interrupt is active in the CLEAR/STAT register and also unmasked by the MASK register.
    c. Sample the CLEAR/STAT register to determine which interrupt bit/bits is/are active.
    d. Clear the active interrupt(s) by writing the sampled value back to the CLEAR/STAT register.
    e. Force the **linti_l** (MPTL)/**interrupt** (PCIe) signal high (deasserted) for a clock cycle by writing a dummy value to the ARM register.
6. end do
7. Verify that the CLEAR/STAT register now has a value of 0, since all interrupts should have been cleared. If the value is non-zero, the test is considered a failure.

Steps c,d, and e model what an interrupt service routine (ISR) in a device driver might do. Step e is not strictly necessary in this case, because this test exercises only one interrupt source at a time, but it is included to model what an ISR would do. In a real application, multiple interrupt sources might become active at any time, including during the time taken for an ISR to service an interrupt. Forcing **linti_l** (MPTL)/**interrupt** (PCIe) high for one cycle ensures that the newly active interrupt source results in another falling edge.

Test complete and pass/fail indications are returned using the **ds_comp.int_complete** and **ds_pass.int_passed** signals respectively.

Example results from this test are documented in Interrupt test results.

5.9.6.5.2.8 Informational Register Test

This test verifies that the Informational Register Block returns the expected values when read:

1. Reads the DATE register and verifies that it is equal to the constant **TODAYS_DATE** from the autogenerated package today_pkg. If not, the test is considered a failure.
2. Reads the TIME register and verifies that it is equal to the constant **TODAYS_TIME** from the autogenerated package today_pkg. If not, the test is considered a failure.
3. Reads the BRAM_BASE register and verifies that it is equal to the constant **BRAM_ADDR_BASE** from the package **uber**. If not, the test is considered a failure.
4. Reads the BRAM_MASK register and verifies that it is equal to the constant **BRAM_ADDR_MASK** from

the package **uber**. If not, the test is considered a failure.

5 Reads the MEM_BASE register and verifies that it is equal to the constant **RAM_WIN_ADDR_BASE** from the package **uber**. If not, the test is considered a failure.

6 Reads the MEM_MASK register and verifies that it is equal to the constant **RAM_WIN_ADDR_MASK** from the package **uber**. If not, the test is considered a failure.

7 Reads the MEM_BANKS register and verifies that it is equal to the constant **MEM_BANKS** from the package adb3_target_inc_pkg. If not, the test is considered a failure.

8 Reads the SDK_VER register and verifies that it is equal to the constant **SDK_VERSION** from the autogenerated package today_pkg. If not, the test is considered a failure.

Test complete and pass/fail indications are returned using the **ds_comp.info_complete** and **ds_pass.info_passed** signals respectively.

Example results from this test are documented in informational register test results.

5.9.6.5.2.9 BRAM Test

This section exercises the BRAM Block by writing various values to it and reading them back. In the following test cases, if the actual value read back is not equal to the expected value, the test is considered a failure:

1 Writes the 32-bit word 0x2389EF45 to the lowest address in the BRAM Block. This address is the value of the constant **BRAM_ADDR_BASE** in the uber_pkg package. This value is then read back and compared with the expected value (the same data that was written).

2 Writes 16 bytes consisting of the 32-bit words { 0xEF123456, ... etc. ..., 0x56789ABC } to the lowest address in the BRAM Block, i.e. **BRAM_ADDR_BASE**. This value is then read back and compared with the expected value (the same data that was written).

3 Writes 32 bytes consisting of the 32-bit words { 0xABCDEF12, ... etc. ..., 0xFEDCBA98 } to the lowest address in the BRAM Block, i.e. **BRAM_ADDR_BASE**. This value is then read back and compared with the expected value (the same data that was written).

4 Writes the 32-bit word 0x369CF258 to an address that is 4 bytes below the lowest address in the BRAM Block, i.e. **BRAM_ADDR_BASE-4**. This value is then read back and compared with the expected value, which is 0xDEADC0DE (since the address used does not belong to any Direct Slave address range decoded by the **Uber** design). This verifies that the lower address boundary of the BRAM Block is as expected.

5 Writes the 32-bit word 0x258BE147 to an address that is just above the highest address in the BRAM Block, i.e. **BRAM_ADDR_BASE+BRAM_ADDR_MASK+1**. This value is then read back and compared with the expected value, which is 0xDEADC0DE (since the address used does not belong to any Direct Slave address range decoded by the **Uber** design). This verifies that the upper address boundary of the BRAM Block is as expected.

6 Writes 32 bytes consisting of the 32-bit words { 0xABCDEF12, ... etc. ..., 0xFEDCBA98 } to an address that is just above the highest address in the BRAM Block, i.e. **BRAM_ADDR_BASE+ BRAM_ADDR_MASK+1**. This value is then read back and compared with the expected value, which is 8 32-bit words of 0xDEADC0DE (since the address used does not belong to any Direct Slave address range decoded by the **Uber** design). This verifies that the upper address boundary of the BRAM Block is as expected.

7 Writes the 32-bit word 0x147AD036 to an address that is 4 bytes below the highest address in the BRAM Block, i.e. **BRAM_ADDR_BASE+BRAM_ADDR_MASK-3**. This value is then read back and compared with the expected value (the same data that was written).

Test complete and pass/fail indications are returned using the **ds_comp.bram_complete** and **ds_pass.bram_passed** signals respectively.

Example results from this test are documented in BRAM test results.

5.9.6.5.2.10 On-Board Memory Test

This test exercises several subsystems of the **Uber** design, including Direct Slave on-board memory access, the memory application and on-board memory. To exercise the on-board memory bank OCP multiplexors, the test programs the memory application to perform a short test of memory bank 1, while the test itself concurrently reads and writes memory locations in a different region of bank 1.

This test section is enabled by the **DO_RAM_TEST** constant defined in the package uber_tb_pkg.

The FPGA-driven memory test is enabled by the **DO_INT_RAM_TEST** constant defined in the package uber_tb_pkg. This applies to steps marked with **.

The steps performed by this test can be expressed in pseudocode as the following algorithm:

1. Poll the BANK1_STAT register until it indicates (via bit 3) that initialisation of memory bank 1 is complete.
2. Display the value of the BANK1_STAT register on the simulator console.
3. ** Set the BANK1_OFFSET register to 0x00FFFEFF, which is the value of the constant **RAM_TEST_START** in **test_uber_ds.vhd**. This is the address in bank 1 (as a 16-byte word index) at which the FPGA-driven memory test will begin testing.
4. ** Display the value of the BANK1_OFFSET register on the simulator console.
5. ** Set the BANK1_LENGTH register to 0x0000FF, which is the value of the constant **RAM_TEST_LEN** in **test_uber_ds.vhd**. This is the number of 16-byte words, beginning at the **BANK1_OFFSET** address in bank 1, that the FPGA-driven memory test will test during each phase, minus 1. The value 0x0000FF therefore results in 256 16-byte words being tested.
6. ** Display the value of the BANK1_LENGTH register on the simulator console.
7. ** Write 0x00000100 to the BANK1_CTRL, which initiates the FPGA-driven memory test for bank 1.
8. Set the memory access window for accessing memory bank 1, by writing 1 to the DS_BANK register.
9. Display the value of the DS_BANK register on the simulator console.
10. Set the memory access window for accessing the bottom 2 MiB page of memory bank 1, by writing 0 to the DS_PAGE register.
11. Display the value of the DS_PAGE register on the simulator console.
12. Write the 32-bit word 0x349AF056 to the bottom of the memory access window (the constant **RAM_WIN_ADDR_BASE** in the uber_pkg package).
13. Read back the value just written, and compare it to the expected value of 0x349AF056. If the expected and actual values are not equal, the test is considered a failure.
14. Set the memory access window for accessing page 127, by writing 0x0000007F to the DS_PAGE register. 0x0000007F is the value of the constant **DS_WIN_RAM_PAGE_TOP** in **test_uber_ds.vhd**.
15. Display the value of the DS_PAGE register on the simulator console.
16. Write the 32-bit word 0x47AD0369 to the top of the memory access window (**RAM_WIN_ADDR_BASE+ RAM_WIN_ADDR_MASK-3**).
17. Read back the value just written, and compare it to the expected value of 0x47AD0369. If the expected and actual values are not equal, the test is considered a failure.
18. Set the memory access window for accessing the bottom 2 MiB page, by writing 0 to the DS_PAGE register.
19. Display the value of the DS_PAGE register on the simulator console.

20 Write 96 bytes consisting of the 32-bit words \{ 0x12345678, ... etc. ..., 0x4321FEDC \} to the bottom of the memory access window (**RAM_WIN_ADDR_BASE**). This is the case of an even-length burst (6 16-byte words) at an even address (bit 4 of the OCP address is 0).

21 Read back the data just written, and compare it to the expected value (the same data that was written). If the expected and actual values are not equal, the test is considered a failure.

22 Write 80 bytes consisting of the 32-bit words \{ 0x456789AB, ... etc. ..., 0xF1234567 \} to the bottom of the memory access window (**RAM_WIN_ADDR_BASE**). This is the case of an odd-length burst (5 16-byte words) at an even address (bit 4 of the OCP address is 0).

23 Read back the data just written, and compare it to the expected value (the same data that was written). If the expected and actual values are not equal, the test is considered a failure.

24 Write 32 bytes consisting of the 32-bit words \{ 0x789ABCDE, ... etc. ..., 0x1FEDCBA9 \} to 16 bytes above the bottom of the memory access window (**RAM_WIN_ADDR_BASE+16**). This is the case of an even-length burst (2 16-byte words) at an odd address (bit 4 of the OCP address is 1).

25 Read back the data just written, and compare it to the expected value (the same data that was written). If the expected and actual values are not equal, the test is considered a failure.

26 Write 64 bytes consisting of the 32-bit words \{ 0x89ABCDEF, ... etc. ..., 0xEDCBA987 \} to 16 bytes above the bottom of the memory access window (**RAM_WIN_ADDR_BASE+16**). This is the case of an even-length burst (4 16-byte words) at an odd address (bit 4 of the OCP address is 1).

27 Read back the data just written, and compare it to the expected value (the same data that was written). If the expected and actual values are not equal, the test is considered a failure.

28 Write 80 bytes consisting of the 32-bit words \{ 0xABCDEF12, ... etc. ..., 0x6789ABCD \} to 16 bytes above the bottom of the memory access window (**RAM_WIN_ADDR_BASE+16**). This is the case of an odd-length burst (5 16-byte words) at an odd address (bit 4 of the OCP address is 1).

29 Read back the data just written, and compare it to the expected value (the same data that was written). If the expected and actual values are not equal, the test is considered a failure.

30 Write the 32-bit word 0x45000000 to the bottom of the memory access window with byte enables 1000. This exercises writing data on byte lane 3 (only) of the memory bank.

31 Write the 32-bit word 0x00AB0000 to the bottom of the memory access window with byte enables 0100. This exercises writing data on byte lane 2 (only) of the memory bank.

32 Write the 32-bit word 0x00000100 to the bottom of the memory access window with byte enables 0010. This exercises writing data on byte lane 1 (only) of the memory bank.

33 Write the 32-bit word 0x00000067 to the bottom of the memory access window with byte enables 0001. This exercises writing data on byte lane 0 (only) of the memory bank.

34 Read back the 32-bit word just written, and compare it to the expected value of 0x45AB0167. If the expected and actual values are not equal, the test is considered a failure.

35 ** Poll the BANK1_STAT register until it indicates (via bit 16) that the FPGA-driven memory test of bank 1 is complete. If the last value read from BANK1_STAT indicates (via bit 23) that the FPGA-driven memory test encountered an error, the test is considered a failure.

Test complete and pass/fail indications are returned using the **ds_comp.ram_complete** and **ds_pass.ram_passed** signals respectively.

Example results from this test are documented in on-board memory test results.

5.9.6.5.3 DMA OCP Channels

Note: all filenames mentioned in this section are relative to the path **hdl/vhdl/examples/uber/common/**.

An instance of the **test_uber_dma** component, implemented in **test_uber_dma_1ch_nb.vhd**, provides test

stimulus to and verifies test results from the UUT's DMA OCP channels. The stimulus is actually applied in the form of OCP commands and data to the Bridge MPTL interface, but apart from the packetisation, multiplexing and demultiplexing that occurs in the MPTL interfaces (both Bridge and Target), the arrangement is transparent. In other words, it behaves as if the stimulus were applied directly to the Target FPGA's DMA OCP channels.

test_uber_dma uses the **adb3_ocp_sim_write**, **adb3_ocp_sim_read_cmd** and **adb3_ocp_sim_read_resp** procedures to perform DMA writes and reads using ADB3 OCP. These procedures are defined in the ADB3 OCP testbench package (adb3_ocp_tb_pkg). The **adb3_ocp_sim_read_cmd** procedure is non-blocking, so it does not wait for all OCP response data to be returned before it completes.

Note
DMA addresses used by the testbench are byte addresses which should be 16-byte aligned. ADB3 OCP protocol addresses are also byte addresses which are 16-byte aligned.

In this testbench, DMA channel 0 (the value of the constant **DMA_SINGLE_CHANNEL** in the package uber_tb_pkg) is tested. Changing this constant is not recommended as in the **Uber** design, only DMA channel 0 has access to both the BRAM Block and the On-Board Memory Interface Block.

The DMA write and read addresses (16-byte aligned) are controlled by the **DMA_ADDR_WR** and **DMA_ADDR_RD** constants defined in the package uber_tb_pkg. The DMA size (multiple of 16-bytes) is controlled by the **DMA_SIZE** constant also defined in the package uber_tb_pkg.

The entity **test_uber_dma** contains two processes that (i) generate OCP commands & OCP write data, and (ii) accept OCP responses (read data). The following subsections describe these processes.

5.9.6.5.3.1 DMA OCP Command and Write Data Process

The process **dma_channel_cmd_p** in **test_uber_dma_1ch_nb.vhd** exercises DMA_SINGLE_CHANNEL in the UUT as described by the following pseudocode:

1. Set address := **DMA_ADDR_WR**, remaining := **DMA_SIZE**, tag := 0, index := 0
2. while remaining != 0 do
   - Set chunk := min(remaining, 128)
   - Generate 'chunk' bytes of data consisting of 32-bit words equal to (0xBEEF0000 + index), incrementing 'index' by one with each 32-bit word generated
   - Issue an OCP write command on DMA_SINGLE_CHANNEL with 'address' as the address, 'tag' as the tag, and length equal to 'chunk', using the data from step 4. Wait until the command has been accepted and all of the data for the command has been transferred (**adb3_ocp_sim_write**)
   - Set remaining := remaining - chunk, address := address + chunk, tag := tag + 1
3. end while
4. Set address := **DMA_ADDR_RD**, remaining := **DMA_SIZE**, tag := 0
5. while remaining != 0 do
   - Set chunk := min(remaining, 128)
   - Issue an OCP read command on DMA_SINGLE_CHANNEL with 'address' as the address, 'tag' as the tag, and length equal to 'chunk'. Wait until the command has been accepted (**adb3_ocp_sim_read_cmd**)
   - Set remaining := remaining - chunk, address := address + chunk, tag := tag + 1
6. end while

The values of **DMA_ADDR_WR** and **DMA_ADDR_RD** correspond to byte offset 0x0200 into on-board memory bank 1.

Test complete and pass/fail indications for steps 1 to 3 are returned using the **dma_comp.dma_write_cmd_complete** and **dma_pass.dma_write_cmd_passed** signals respectively. Test complete and pass/fail indications for steps 4 to 6 are returned using the **dma_comp.dma_read_cmd_complete** and **dma_pass.dma_read_cmd_passed** signals respectively.

Example results from this test are documented in DMA OCP channels results.

5.9.6.5.3.2 DMA OCP Response Process

The process **dma_channel_resp_p** in **test_uber_dma_1ch_nb.vhd** exercises DMA_SINGLE_CHANNEL in the UUT as described by the following pseudocode:

1. Set remaining := **DMA_SIZE**, index := 0, expected_tag := 0
2. while remaining != 0 do
   - Set chunk := min(remaining, 128)
   - Wait for 'chunk' bytes of response data to be received from DMA_SINGLE_CHANNEL (**adb3_ocp_sim_read_resp**)
   - Verify that the received data, considered as 32-bit words, equals the incrementing pattern 0xBEEF0000 + index, where index is incremented by 1 with each word checked. If a received 32-bit word does not equal the expected pattern, the test is considered a failure
   - Verify that the received OCP response tag for each 16-byte OCP word received equals 'expected_tag'. If it does not, the test is considered a failure
   - Set remaining := remaining - chunk, expected_tag := expected_tag + 1
3. end while

Test complete and pass/fail indications are returned using the **dma_comp.dma_read_resp_complete** and **dma_pass.dma_read_resp_passed** signals respectively.

Example results from this test are documented in DMA OCP channels results.

5.9.6.6 On-Board Memory Simulation Models

The on-board memory model block is implemented by the **test_uber_mem** block which is model dependent.

Table 131 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-6TL	admxrc6tl/test_uber_mem_6tl.vhd
ADM-XRC-6T1	admxrc6t1/test_uber_mem_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/test_uber_mem_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/test_uber_mem_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/test_uber_mem_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/test_uber_mem_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/test_uber_mem_adpexrc6tl.vhd

Table 131 : Available Variants of test_uber_mem Component (continued on next page)

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-7K1	admxrc7k1/test_uber_mem_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/test_uber_mem_admxrc7v1.vhd

Table 131 : Available Variants of test_uber_mem Component

The testbench instantiates a simulation model for each memory device in each bank of on-board memory.

Table 132 lists the available variants:

Model	Simulation Model/
DDR3 SDRAM	DDR3 SDRAM Model (ddr3_sdram)

Table 132 : Available Variants of On-Board Memory Models

The number of banks or DDR3 SDRAM to be instantiated in the testbench is controlled by the **DDR3_BANKS** constant in the adb3_target_inc_pkg package.

The size of the DDR3 SDRAM part to be simulated is defined by selecting either **1**, **2**, or **4** for the value of the build option m **OPTION_M** constant in the adb3_target_tb_inc_pkg package.

Note
The default size of the DDR3 SDRAM on-board memory part used for simulation is 1 Gib. The user should verify that this matches the size of the memory parts on the model in use and selected by the package adb3_target_inc_pkg.

Initialisation of DDR3 memory contents is activated by the **init_start** input changing to a high state. Logging of DDR3 memory contents is activated by the **log_start** input changing to a high state.

5.9.6.7 Testbench Package (uber_tb_pkg)

The package **uber_tb_pkg** defines types, constants, and functions which are used by the **Uber** example FPGA testbench. Table 133 lists the available variants:

Model	Filename relative to hdl/vhdl/examples/uber/
ADM-XRC-6TL	admxrc6tl/uber_tb_pkg_6tl.vhd
ADM-XRC-6T1	admxrc6t1/uber_tb_pkg_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/uber_tb_pkg_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/uber_tb_pkg_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/uber_tb_pkg_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/uber_tb_pkg_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/uber_tb_pkg_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/uber_tb_pkg_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/uber_tb_pkg_admxrc7v1.vhd

Table 133 : Available Variants of uber_tb_pkg Package

Definitions are as follows:

General test constants

- **MPTL_GPIO**. The value used for the MPTL GPIO loopback test on this model (MPTL).

DS clock test constants

- **CLK_TEST_DIFF**. The acceptable difference for clock frequency measurement results on this model.
- **PLL_PRI_CLK_FREQ**. The **pll_pri_clk** expected clock frequency measurement result on this model.
- **PLL_REG_CLK_FREQ**. The **pll_reg_clk** expected clock frequency measurement result on this model.
- **PLL_MEM_CLK_FREQ**. The **pll_mem_clk** expected clock frequency measurement result on this model.
- **PLL_REF_CLK_FREQ**. The **pll_ref_clk** expected clock frequency measurement result on this model.
- **CLKS_OUT_FREQ**. Defines **clks_out** output clocks expected frequencies (MHz).
- **BRIDGE_LCLK_FREQ**. Defines frequency (Hz) of lclk (generated by bridge).
- **TEST_MGT_CLK_NUM**. The **TEST_MGT_CLK** clock frequency measurement clock select index.
- **TEST_CUS_CLK_NUM**. The **TEST_CUS_CLK** clock frequency measurement clock select index.
- **TEST_MGT_CLK_FREQ**. The **TEST_MGT_CLK** expected clock frequency measurement result (MHz) on this model.
- **TEST_CUS_CLK_FREQ**. The **TEST_CUS_CLK** expected clock frequency measurement result (MHz) on this model.

DS GPIO test constants (Model Dependent)

- **PN4_DATA**. Data used during Pn4 GPIO test on this model.
- **PN6_DATA**. Data used during Pn6 GPIO test on this model.
- **FMC_HA_DATA**. Data used during FMC GPIO test on this model.
- **FMC_HB_DATA**. Data used during FMC GPIO test on this model.
- **SEC_DATA**. Data used during Secondary GPIO test on this model.

DS on-board RAM test constants

- **DS_RAM_TEST_BANK**. On-board memory bank used during direct slave test.
- **DS_WIN_RAM_PAGES**. Number of direct slave window pages in bank of on-board memory.
- **DS_WIN_RAM_BANK**. 32-bit register value containing DS_RAM_TEST_BANK.
- **DS_WIN_RAM_PAGE_BOT**. 32-bit register value containing direct slave window page 0.
- **DS_WIN_RAM_PAGE_TOP**. 32-bit register value containing direct slave window page **DS_WIN_RAM_PAGES**-1.

DS internal on-board RAM test constants

- **INT_RAM_TEST_BANK**. On-board memory bank used during internal memory test.
- **INT_RAM_TEST_LENGTH**. Length of phase of internal memory test (16-byte words).
- **INT_RAM_TEST_LEN**. 32-bit register value containing INT_RAM_TEST_LENGTH.
- **INT_RAM_TEST_TOP**. Top address of internal memory test (16-byte words).
- **INT_RAM_TEST_START**. Start address of internal memory test (16-byte words).

DS interrupt test constants and variables

- **MASK_EN_ALL**. 32-bit register value containing interrupt mask register enable all.
- **int_edge**. Shared variable which indicates detection of active edge on **linti_l** (MPTL) or **interrupt** (PCIe).

DS test control constants

- **DO_RAM_TEST**. Perform direct slave test of on-board memory.
- **DO_INT_RAM_TEST**. Perform internal test of on-board memory.
- **DO_CLOCK_TEST**. Perform direct slave clock frequency measurement test.

- **DO_INTERRUPT_NUM**. Set length of direct slave interrupt test.

DMA test constants

- **DMA_WRITE_CHANNEL**. The DMA channel used by writes (Host to FPGA) during 2 channel DMA test.
- **DMA_READ_CHANNEL**. The DMA channel used by reads (FPGA to Host) during 2 channel DMA test.
- **DMA_SINGLE_CHANNEL**. The DMA channel used by writes and reads during 1 channel DMA test.
- **DMA_ADDR_WR**. The start address used by writes (Host to FPGA) during DMA test.
- **DMA_ADDR_RD**. The start address used by reads (FPGA to Host) during DMA test.
- **DMA_SIZE**. The size of the DMA transfer in bytes (multiple of 16 bytes).
- **DMA_BL_WRITE**. The OCP burst length used by writes (Host to FPGA) during DMA test (16-byte aligned) .
- **DMA_BL_READ**. The OCP burst length used by reads (FPGA to Host) during DMA test (16-byte aligned).

Test status types

- **top_comp_t**. A record type containing non-OCP test completion elements.
- **ds_comp_t**. A record type containing direct slave OCP test completion elements.
- **dma_comp_t**. A record type containing DMA OCP test completion elements.
- **top_pass_t**. A record type containing non-OCP test pass elements.
- **ds_pass_t**. A record type containing direct slave OCP test pass elements.
- **dma_pass_t**. A record type containing DMA OCP test pass elements.

Component definitions

- test_uber_ds
- test_uber_dma
- test_uber_probes
- test_uber_mem

Function definitions

- **conv_ddr3_part**. A function converting OPTION_M integer values into DDR3 SDRAM parts.

5.9.7 Design Simulation

5.9.7.1 Simulation Using ModelSim

For a complete list of the source files used during simulation, refer to the appropriate **ModelSim** macro file (**.do**). These are located in each of the model-specific design directories. The macro file that should be used depends upon the type of simulation required:

- OCP-Only: **hdl/vhdl/examples/uber/*<model>*/uber-*<model>*.do**
- Full MPTL: **hdl/vhdl/examples/uber/*<model>*/uber-*<model>*-mptl.do**

where ***<model>*** corresponds to the model in use; for example **admxrc6t1** for the ADM-XRC-6T1.

The **ModelSim** macro file (**.do**) **VAR_MIG** variable should be set to **LWB** to perform an OCP-Only simulation using the light weight behavioural (LWB) variant of the DDR3 SDRAM Interface.

The **ModelSim** macro file (**.do**) **VAR_MIG** variable should be set to the MIG version number to perform an OCP-Only/Full MPTL simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

ModelSim simulation is initiated using the **vsim** command with the appropriate macro file. For example, to perform an OCP-Only simulation in Windows for the ADM-XRC-6T1, start a shell and issue the following

commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber\admxrc6t1
vsim -do "uber-admxrc6t1.do"
```

In Linux, the commands are:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber/admxrc6t1
vsim -do "uber-admxrc6t1.do"
```

Note
Before performing the first simulation of the **Uber** design, HDL files for the Xilinx Memory Interface Generator (MIG) DDR3 SDRAM interface must be generated using the script **gen_mem_if.tcl**. Refer to DDR3 SDRAM MIG Cores for details.

Note: The value of the **DDR3_BANK_ROW_WIDTH** constant determines the maximum size of DDR3 SDRAM parts supported by the target FPGA design. Currently, valid vaues are 13 for 1Gib parts only, 14 for 1Gib/2Gib parts, or 15 for 1Gib/2Gib/4Gib parts. The simulation model for the appropriate memory part will also need to be selected in the example design testbench. This is achieved by selecting either **1**, **2**, or **4** for the value of the build option m **OPTION_M** constant in the adb3_target_tb_inc_pkg.

Note
The user should verify that the value of the **DDR3_BANK_ROW_WIDTH** (default = 14) and **OPTION_M** (default = 1) constants are appropriate for the size of the memory parts on the model in use.

Note
The **ModelSim** macro files always delete any previously compiled data before compiling the **Uber** design.

5.9.7.2 Simulation Using PlanAhead

In order to perform an **ISIM** simulation in **PlanAhead**, it is first necessary to generate **PlanAhead** project files as described in Section 5.9.3 - "Generating PlanAhead Projects". Then, use the **PlanAhead** GUI to perform a behavioural simulation as normal.

The **ISIM** simulation set **sim_lwb** should be selected to perform an OCP-Only simulation using the light weight behavioural (LWB) variant of the DDR3 SDRAM Interface.

The **ISIM** simulation set **sim_ocp** should be selected to perform an OCP-Only simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

The **ISIM** simulation set **sim_mptl** should be selected to perform a Full MPTL simulation using the Xilinx MIG variant of the DDR3 SDRAM Interface.

5.9.7.3 Date/Time Package Generation

In the **ModelSim** design flow, before compiling the **Uber** example design HDL and initiating simulation, the macro file runs the TCL script **hdl/vhdl/examples/uber/gen_today_pkg.tcl** to generate a file containing the **today_pkg** package. This package defines HDL constants containing the SDK version and date/time at which the script was run. The name of the generated file depends upon the model selected and it is located in the model design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/today_pkg_admxrc6t1_sim.vhd** for the ADM-XRC-6T1.

In the **PlanAhead** design flow, the PlanAhead project file generation scripts will run the TCL script

hdl/vhdl/examples/uber/gen_today_pkg.tcl to generate a file containing the **today_pkg** package. This package defines HDL constants containing the SDK version and date/time at which the script was run. The name of the generated file depends upon the model selected and it is located in the **Uber** design **planahead** directory; for example, **hdl/vhdl/examples/uber/planahead/today_pkg_admxrc6t1_sim.vhd** for the ADM-XRC-6T1.

Transcript output is of the form:

```
--
-- This file was generated automatically by gen_today_pkg.tcl
--
-- SDK:  01.05.00 (Maj/Min/Bld)
-- Date: 08/10/2011 (dd/mm/YYYY)
-- Time: 15:26:46 (HH/MM/SS)
--

library ieee;
use ieee.std_logic_1164.all;

package today_pkg is

  constant SDK_VERSION : std_logic_vector(31 downto 0) := X"00010500";
  constant TODAYS_DATE : std_logic_vector(31 downto 0) := X"08102011";
  constant TODAYS_TIME : std_logic_vector(31 downto 0) := X"15264600";

end package today_pkg;
```

Note
The Modelsim script runs the **gen_today_pkg.tcl** script using the Xilinx **xtclsh** tool. The path to this shell must be defined before initiating simulation.

5.9.7.4 Simulation Results

5.9.7.4.1 Initialisation Results

ModelSim transcript output during initialisation of the simulation is of the form described in the following subsections.

5.9.7.4.1.1 DDR3 SDRAM MIG Core MMCM Status

Each instantiation of the DDR3 SDRAM MIG core produces a summary of its MMCM clocking parameters:

```
############## Write Clocks MMCM_ADV Parameters ##############
# nCK_PER_CLK      = 2
# CLK_PERIOD       = 5000
# CLKIN1_PERIOD    = 2.500000e+000
# DIVCLK_DIVIDE    = 2
# CLKFBOUT_MULT_F  = 6
# VCO_PERIOD       = 833
# CLKOUT0_DIVIDE_F = 3
# CLKOUT1_DIVIDE   = 6
# CLKOUT2_DIVIDE   = 3
# CLKOUT0_PERIOD   = 2499
# CLKOUT1_PERIOD   = 4998
# CLKOUT2_PERIOD   = 2499
##############################################################
```

5.9.7.4.1.2 Testbench Status (MPTL)

The testbench produces a summary of the model and simulation type, and then waits for the MPTL interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_uber
# ** Note: Waiting for MPTL online...
#    Time: 0 ps  Iteration: 0  Instance: /test_uber
# ** Note: MPTL link online
#    Time: 23056 ns  Iteration: 2  Instance: /test_uber
```

5.9.7.4.1.3 Testbench Status (PCIe)

The testbench produces a summary of the model and simulation type, and then waits for the PCIe interface to complete its initialisation:

```
# ** Note: Model Name : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_uber
# ** Note: Waiting for PCIe online...
#    Time: 0 ps  Iteration: 0  Instance: /test_uber
# ** Note: PCIe link online
#    Time: 2985 ns  Iteration: 13  Instance: /test_uber
```

5.9.7.4.1.4 DDR3 SDRAM Initialisation

Each instantiated DDR3 SDRAM MIG core produces a truncated initialisation sequence during simulation. This is detected by the DDR3 SDRAM models and warnings are issued by each instantiated part:

```
# ** Warning: DDR3 SDRAM Init FSM (3) : Deviation from recommended initialisation
sequence:
violation of 200us delay before RESET_L de-assertion
#    Time: 971771500 fs  Iteration: 4  Instance:
/test_uber/ddr3_model_g__0/ddr3_sdram_bank_ls_i/ddr3_sdram_init_fsm_i
# ** Warning: DDR3 SDRAM Init FSM (4) : Deviation from recommended initialisation
sequence:
violation of 10ns CKE delay before RESET_L de-assertion
#    Time: 971771500 fs  Iteration: 4  Instance:
/test_uber/ddr3_model_g__0/ddr3_sdram_bank_ls_i/ddr3_sdram_init_fsm_i
```

Each instantiated DDR3 SDRAM MIG6 core produces status information during its initialisation sequence:

```
# PHY_INIT: Memory Initialization completed at 5063.017 ns
# PHY_INIT: Write Leveling completed at 22183.017 ns
# PHY_INIT: Read Leveling Stage 1 completed at 30373.017 ns
# PHY_INIT: Read Leveling CLKDIV cal completed at 43313.017 ns
# PHY_INIT: Read Leveling Stage 2 completed at 50638.017 ns
# PHY_INIT: Phase Detector Initial Cal completed at 55618.017 ns
```

Each instantiated DDR3 SDRAM MIG7 core produces status information during its initialisation sequence:

```
# PHY_INIT: Memory Initialization completed at          5655100000
# PHY_INIT: Write Leveling completed at          6680100000
# PHY_INIT: Phaser_In Phase Locked at          8945100000
# PHY_INIT: Phaser_In DQSFOUND completed at         20530100000
# PHY_INIT: Read Leveling Stage 1 completed at         26540100000
# PHY_INIT: Write Calibration completed at         31075100000
```

5.9.7.4.2 Non-OCP Functions Results

5.9.7.4.2.1 MPTL GPIO Bus Test Results (MPTL)

ModelSim transcript output during simulation is of the form:

```
# ** Note: Test mptl_gpio completed: PASSED.
#    Time: 3028750 ps  Iteration: 13  Instance: /test_uber
```

5.9.7.4.3 Direct Slave OCP Channel Results

5.9.7.4.3.1 Simple Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote simple WDATA 4 bytes 0xCAFEFACE with enable 0b1111 to byte address
0x000000
#    Time: 2038750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read simple RDATA 4 bytes 0xECAFEFAC from byte address 0x000000
#    Time: 2351250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Simple completed: PASSED.
#    Time: 2351250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

5.9.7.4.3.2 Clock Frequency Measurement Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote Clear All CTRL 4 bytes 0x80000000 with enable 0b1111 to byte
address 0x000044
#    Time: 2530 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote PLL_REG_CLK_SEL 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000040
#    Time: 2540 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PLL_REG_CLK_FREQ 4 bytes 0x00000140 from byte address 0x000048
#    Time: 5482500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 8.000000e+001 MHz ±2.000000e+000 MHz
#    Time: 5482500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 8.000000e+001 MHz
#    Time: 5482500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote PLL_PRI_CLK_SEL 4 bytes 0x00000001 with enable 0b1111 to byte
address 0x000040
#    Time: 5490 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PLL_PRI_CLK_FREQ 4 bytes 0x00000320 from byte address 0x000048
#    Time: 6182500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 2.000000e+002 MHz ±2.000000e+000 MHz
#    Time: 6182500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 2.000000e+002 MHz
#    Time: 6182500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote TEST_MGT_CLK_SEL 4 bytes 0x00000014 with enable 0b1111 to byte
address 0x000040
#    Time: 6190 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read TEST_MGT_CLK_FREQ 4 bytes 0x000003E8 from byte address 0x000048
#    Time: 6882500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 2.500000e+002 MHz ±2.000000e+000 MHz
#    Time: 6882500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 2.500000e+002 MHz
#    Time: 6882500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

```
# ** Note: Wrote TEST_CUS_CLK_SEL 4 bytes 0x00000006 with enable 0b1111 to byte
address 0x000040
#    Time: 6890 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read TEST_CUS_CLK_FREQ 4 bytes 0x00000A68 from byte address 0x000048
#    Time: 7582500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 6.660000e+002 MHz ±2.000000e+000 MHz
#    Time: 7582500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 6.660000e+002 MHz
#    Time: 7582500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Clock Read completed: PASSED.
#    Time: 7582500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

Results will be model dependent. Results are shown for ADM-XRC-6T1.

5.9.7.4.3.3 XRM GPIO Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote XRM_GPIO_DD DATAO 4 bytes 0x76543210 with enable 0b1111 to byte
address 0x000224
#    Time: 5508750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote XRM_GPIO_DD TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x00022C
#    Time: 5518750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read XRM_GPIO_DD DATAI 4 bytes 0x76543210 from byte address 0x000228
#    Time: 6026250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote XRM_GPIO_DD TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x00022C
#    Time: 6033750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Front IO completed: PASSED.
#    Time: 6033750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

Results will be model dependent. Results are shown for ADM-XRC-6T1.

5.9.7.4.3.4 Pn4/Pn6 GPIO Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote PN4_GPIO_P DATAO 4 bytes 0xAABBCCDD with enable 0b1111 to byte
address 0x00023C
#    Time: 6043750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_N DATAO 4 bytes 0x55443322 with enable 0b1111 to byte
address 0x000248
#    Time: 6053750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_P TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000244
#    Time: 6063750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_N TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000250
#    Time: 6073750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN4_GPIO_P DATAI 4 bytes 0xAABBCCDD from byte address 0x000240
#    Time: 6651250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN4_GPIO_N DATAI 4 bytes 0x55443322 from byte address 0x00024C
#    Time: 6901250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_P TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x000244
#    Time: 6908750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_N TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x000250
#    Time: 6918750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

```
# ** Note: Wrote PN6_GPIO_MS DATAO 4 bytes 0xAAAABBBB with enable 0b1111 to byte
address 0x000254
#    Time: 6928750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_LS DATAO 4 bytes 0xCCCCDDDD with enable 0b1111 to byte
address 0x000260
#    Time: 6938750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_MS TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x00025C
#    Time: 6948750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_LS TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000268
#    Time: 6958750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN6_GPIO_MS DATAI 4 bytes 0x0000003B from byte address 0x000258
#    Time: 7601250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN6_GPIO_LS DATAI 4 bytes 0xCCCCDDDD from byte address 0x000264
#    Time: 7851250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_MS TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x00025C
#    Time: 7858750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_LS TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x000268
#    Time: 7868750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Rear IO completed: PASSED.
#    Time: 7868750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

Results will be model dependent. Results are shown for ADM-XRC-6T1.

5.9.7.4.3.5 FMC GPIO Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote FMC_GPIO_LS_LA_P DATAO 4 bytes 0x76543210 with enable 0b1111 to
byte address 0x000298
#    Time: 7940 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_LS_LA_N DATAO 4 bytes 0x89ABCDEF with enable 0b1111 to
byte address 0x0002A4
#    Time: 7950 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_LS_LA_P TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002A0
#    Time: 7960 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_LS_LA_N TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002AC
#    Time: 7970 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_LS_LA_P DATAI 4 bytes 0x76543210 from byte address
0x00029C
#    Time: 8650 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_LS_LA_N DATAI 4 bytes 0x89ABCDEF from byte address
0x0002A8
#    Time: 9025 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_LS_LA_P TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002A0
#    Time: 9035 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_LS_LA_N TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002AC
#    Time: 9045 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote FMC_GPIO_MS_LA_P DATAO 4 bytes 0x87654321 with enable 0b1111 to
byte address 0x000280
#    Time: 9055 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_MS_LA_N DATAO 4 bytes 0x789ABCDE with enable 0b1111 to
byte address 0x00028C
#    Time: 9065 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

```
# ** Note: Wrote FMC_GPIO_MS_LA_P TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000288
#    Time: 9075 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_MS_LA_N TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000294
#    Time: 9085 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_MS_LA_P DATAI 4 bytes 0xXXXXXXXX from byte address
0x000284
#    Time: 9925 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_MS_LA_N DATAI 4 bytes 0xXXXXXXXX from byte address
0x000290
#    Time: 10300 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_MS_LA_P TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x000288
#    Time: 10310 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_MS_LA_N TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x000294
#    Time: 10320 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote FMC_GPIO_HA_P DATAO 4 bytes 0xBCDEF012 with enable 0b1111 to byte
address 0x0002B0
#    Time: 10330 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HA_N DATAO 4 bytes 0x43210FED with enable 0b1111 to byte
address 0x0002BC
#    Time: 10340 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HA_P TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002B8
#    Time: 10350 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HA_N TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002C4
#    Time: 10360 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_HA_P DATAI 4 bytes 0xXXDEF012 from byte address 0x0002B4
#    Time: 11200 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_HA_N DATAI 4 bytes 0xXX210FED from byte address 0x0002C0
#    Time: 11575 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HA_P TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002B8
#    Time: 11585 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HA_N TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002C4
#    Time: 11595 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote FMC_GPIO_HB_P DATAO 4 bytes 0xCDEF0123 with enable 0b1111 to byte
address 0x0002C8
#    Time: 11605 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HB_N DATAO 4 bytes 0x3210FEDC with enable 0b1111 to byte
address 0x0002D4
#    Time: 11615 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HB_P TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002D0
#    Time: 11625 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HB_N TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002DC
#    Time: 11635 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_HB_P DATAI 4 bytes 0xXXXF0123 from byte address 0x0002CC
#    Time: 12475 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read FMC_GPIO_HB_N DATAI 4 bytes 0xXXX0FEDC from byte address 0x0002D8
#    Time: 12850 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HB_P TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002D0
#    Time: 12860 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote FMC_GPIO_HB_N TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002DC
#    Time: 12870 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

```
# ** Note: Test Front IO completed: PASSED.
#    Time: 12870 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

Results will be model dependent. Results are shown for ADPE-XRC-6T.

5.9.7.4.3.6 Secondary GPIO Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote SEC_GPIO_P DATAO 4 bytes 0xABCDEF01 with enable 0b1111 to byte
address 0x0002E0
#    Time: 12880 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote SEC_GPIO_N DATAO 4 bytes 0x543210FE with enable 0b1111 to byte
address 0x0002EC
#    Time: 12890 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote SEC_GPIO_P TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002E8
#    Time: 12900 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote SEC_GPIO_N TRI 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0002F4
#    Time: 12910 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read SEC_GPIO_P DATAI 4 bytes 0xABCDEF01 from byte address 0x0002E4
#    Time: 13750 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read SEC_GPIO_N DATAI 4 bytes 0x543210FE from byte address 0x0002F0
#    Time: 14125 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote SEC_GPIO_P TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002E8
#    Time: 14135 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote SEC_GPIO_N TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0002F4
#    Time: 14145 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Rear IO completed: PASSED.
#    Time: 14145 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

Results will be model dependent. Results are shown for ADPE-XRC-6T.

5.9.7.4.3.7 Interrupt Test Results (MPTL)

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote Interrupt MASK 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x0000C8
#    Time: 8173750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Interrupt MASK 4 bytes 0x00000000 from byte address 0x0000C8
#    Time: 8491250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote Interrupt COUNT 4 bytes 0xFFFFFFFF with enable 0b1111 to byte
address 0x0000D0
#    Time: 8498750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Interrupt COUNT 4 bytes 0xFFFFFFFF from byte address 0x0000D0
#    Time: 8816250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Interrupt Monitor: Detected falling edge on linti_l
#    Time: 8958750 ps  Iteration: 13  Instance: /test_uber
# ** Note: Interrupt Handler: Cleared interrupt(s), masked STAT = 0x00000001
#    Time: 9298750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Interrupt Monitor: Detected falling edge on linti_l
#    Time: 9583750 ps  Iteration: 13  Instance: /test_uber
# ** Note: Interrupt Handler: Cleared interrupt(s), masked STAT = 0x00000002
#    Time: 9923750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

```
...

# ** Note: Interrupt Monitor: Detected falling edge on linti_l
#    Time: 28333750 ps  Iteration: 13  Instance: /test_uber
# ** Note: Interrupt Handler: Cleared interrupt(s), masked STAT = 0x80000000
#    Time: 28673750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Read Interrupt STAT 4 bytes 0x00000000 from byte address 0x0000C4
#    Time: 29066250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Interrupt completed: PASSED.
#    Time: 29066250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

5.9.7.4.3.8 Informational Register Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Read Info DATE 4 bytes 0x05092011 from byte address 0x000140
#    Time: 32382500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info TIME 4 bytes 0x11431100 from byte address 0x000144
#    Time: 32697500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info BRAM BASE 4 bytes 0x00080000 from byte address 0x00014C
#    Time: 33007500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info BRAM MASK 4 bytes 0x0007FFFF from byte address 0x000150
#    Time: 33322500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info RAM BASE 4 bytes 0x00200000 from byte address 0x000154
#    Time: 33632500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info RAM MASK 4 bytes 0x001FFFFF from byte address 0x000158
#    Time: 33947500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info RAM INFO 4 bytes 0xXXXXXXX4 from byte address 0x00015C
#    Time: 34257500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info SDK VERSION 4 bytes 0x00010500 from byte address 0x000160
#    Time: 34572500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Info completed: PASSED.
#    Time: 34572500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

5.9.7.4.3.9 BRAM Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Wrote BRAM Addr base 4 bytes 0x2389EF45 with enable 0b1111 to byte
address 0x080000
#    Time: 30498750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read BRAM Addr base 4 bytes 0x2389EF45 from byte address 0x080000
#    Time: 30876250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote BRAM Addr base 16 bytes 0x56789ABCDEF123456789ABCDEF123456
with enable 0b1111111111111111 to byte address 0x080000
#    Time: 30883750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read BRAM Addr base 16 bytes 0x56789ABCDEF123456789ABCDEF123456
from byte address 0x080000
#    Time: 31266250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote BRAM Addr base 32 bytes
0xFEDCBA987654321FEDCBA987654321FE123456789ABCDEF123456789ABCDEF12
with enable 0b11111111111111111111111111111111 to byte address 0x080000
#    Time: 31278750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read BRAM Addr base 32 bytes
0xFEDCBA987654321FEDCBA987654321FE123456789ABCDEF123456789ABCDEF12
from byte address 0x080000
#    Time: 31671250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

```
# ** Note: Wrote OOR Addr base-4 4 bytes 0x369CF258 with enable 0b1111 to byte
address 0x07FFFC
#    Time: 31678750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read OOR Addr base-4 4 bytes 0xDEADC0DE from byte address 0x07FFFC
#    Time: 31916250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote OOR Addr top+1 4 bytes 0x258BE147 with enable 0b1111 to byte
address 0x100000
#    Time: 31923750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read OOR Addr top+1 4 bytes 0xDEADC0DE from byte address 0x100000
#    Time: 32151250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote OOR Addr top+1 32 bytes
0xFEDCBA987654321FEDCBA987654321FE123456789ABCDEF123456789ABCDEF12
with enable 0b11111111111111111111111111111111 to byte address 0x100000
#    Time: 32163750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read OOR Addr top+1 32 bytes
0xDEADC0DEDEADC0DEDEADC0DEDEADC0DEDEADC0DEDEADC0DEDEADC0DEDEADC0DE
from byte address 0x100000
#    Time: 32416250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote BRAM Addr top 4 bytes 0x147AD036 with enable 0b1111 to byte
address 0x0FFFFC
#    Time: 32423750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read BRAM Addr top 4 bytes 0x147AD036 from byte address 0x0FFFFC
#    Time: 32801250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test BRAM completed: PASSED.
#    Time: 32801250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

5.9.7.4.3.10 On-Board Memory Test Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: Waiting for on-board RAM bank 1 to initialise...
#    Time: 35777500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: On-board RAM bank 1 initialised
#    Time: 58582500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Read RAM Bank Info Reg 4 bytes 0x3F10181C from byte address 0x00034C
#    Time: 58897500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Bank Offset Reg 4 bytes 0x00FFFEFF with enable 0b1111 to byte
address 0x000344
#    Time: 58905 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Bank Offset Reg 4 bytes 0x00FFFEFF from byte address 0x000344
#    Time: 59282500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Bank Length Reg 4 bytes 0x000000FF with enable 0b1111 to byte
address 0x000348
#    Time: 59290 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Bank Length Reg 4 bytes 0x000000FF from byte address 0x000348
#    Time: 59672500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Bank Ctrl Reg 4 bytes 0x00000100 with enable 0b1111 to byte
address 0x000340
#    Time: 59680 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM DS Bank Reg Addr 4 bytes 0x00000001 with enable 0b1111 to byte
address 0x000300
#    Time: 59690 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM DS Bank Reg Addr 4 bytes 0x00000001 from byte address 0x000300
#    Time: 60132500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM DS Page Reg Addr 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000304
```

```
#    Time: 60140 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM DS Page Reg Addr 4 bytes 0x00000000 from byte address 0x000304
#    Time: 60522500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 4 bytes 0x349AF056 with enable 0b1111 to byte
address 0x200000
#    Time: 60530 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 4 bytes 0x349AF056 from byte address 0x200000
#    Time: 61917500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM DS Page Reg Addr 4 bytes 0x0000007F with enable 0b1111 to byte
address 0x000304
#    Time: 61925 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM DS Page Reg Addr 4 bytes 0x0000007F from byte address 0x000304
#    Time: 62307500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Win Addr top 4 bytes 0x47AD0369 with enable 0b1111 to byte
address 0x3FFFFC
#    Time: 62315 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr top 4 bytes 0x47AD0369 from byte address 0x3FFFFC
#    Time: 63147500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM DS Page Reg Addr 4 bytes 0x00000000 with enable 0b1111 to byte
address 0x000304
#    Time: 63155 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM DS Page Reg Addr 4 bytes 0x00000000 from byte address 0x000304
#    Time: 63532500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 96 bytes
0xDCBA987654321FEDCBA987654321FEDC3456789ABCDEF123456789ABCDEF1234
  EDCBA987654321FEDCBA987654321FED23456789ABCDEF123456789ABCDEF123
  FEDCBA987654321FEDCBA987654321FE123456789ABCDEF123456789ABCDEF12
with enable
0b111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
to byte address 0x200000
#    Time: 63565 ns  Iteration: 14  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 96 bytes
0xDCBA987654321FEDCBA987654321FEDC3456789ABCDEF123456789ABCDEF1234
  EDCBA987654321FEDCBA987654321FED23456789ABCDEF123456789ABCDEF123
  FEDCBA987654321FEDCBA987654321FE123456789ABCDEF123456789ABCDEF12
from byte address 0x200000
#    Time: 64537500 ps  Iteration: 14  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 80 bytes
0x6789ABCDEF123456789ABCDEF1234567BA987654321FEDCBA987654321FEDCBA
  56789ABCDEF123456789ABCDEF123456CBA987654321FEDCBA987654321FEDCB
  456789ABCDEF123456789ABCDEF12345
with enable
0b111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
to byte address 0x200000
#    Time: 64565 ns  Iteration: 14  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 80 bytes
0x6789ABCDEF123456789ABCDEF1234567BA987654321FEDCBA987654321FEDCBA
  56789ABCDEF123456789ABCDEF123456CBA987654321FEDCBA987654321FEDCB
  456789ABCDEF123456789ABCDEF12345
from byte address 0x200000
#    Time: 65727500 ps  Iteration: 14  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 32 bytes
0xA987654321FEDCBA987654321FEDCBA9789ABCDEF123456789ABCDEF12345678
with enable 0b11111111111111111111111111111111
to byte address 0x200010
#    Time: 65740 ns  Iteration: 14  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 32 bytes
0xA987654321FEDCBA987654321FEDCBA9789ABCDEF123456789ABCDEF12345678
```

```
from byte address 0x200010
#    Time: 66937500 ps  Iteration: 14  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 64 bytes
0x87654321FEDCBA987654321FEDCBA9879ABCDEF123456789ABCDEF123456789A
  987654321FEDCBA987654321FEDCBA9889ABCDEF123456789ABCDEF123456789
with enable 0b11111111111111111111111111111111111111111111111111111111111111111111111111111111
to byte address 0x200010
#    Time: 66960 ns  Iteration: 14  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 64 bytes
0x87654321FEDCBA987654321FEDCBA9879ABCDEF123456789ABCDEF123456789A
  987654321FEDCBA987654321FEDCBA9889ABCDEF123456789ABCDEF123456789
from byte address 0x200010
#    Time: 67547500 ps  Iteration: 14  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 80 bytes
0xCDEF123456789ABCDEF123456789ABCD654321FEDCBA987654321FEDCBA98765
  BCDEF123456789ABCDEF123456789ABC7654321FEDCBA987654321FEDCBA9876
  ABCDEF123456789ABCDEF123456789AB
with enable
0b11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
to byte address 0x200010
#    Time: 67575 ns  Iteration: 14  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 80 bytes
0xCDEF123456789ABCDEF123456789ABCD654321FEDCBA987654321FEDCBA98765
  BCDEF123456789ABCDEF123456789ABC7654321FEDCBA987654321FEDCBA9876
  ABCDEF123456789ABCDEF123456789AB
from byte address 0x200010
#    Time: 68162500 ps  Iteration: 14  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 4 bytes 0x45000000 with enable 0b1000 to byte
address 0x200000
#    Time: 68170 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Win Addr base 4 bytes 0x00AB0000 with enable 0b0100 to byte
address 0x200000
#    Time: 68180 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Win Addr base 4 bytes 0x00000100 with enable 0b0010 to byte
address 0x200000
#    Time: 68190 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Win Addr base 4 bytes 0x00000067 with enable 0b0001 to byte
address 0x200000
#    Time: 68200 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 4 bytes 0x45AB0167 from byte address 0x200000
#    Time: 68757500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Waiting for internal test of on-board RAM bank 1 to complete...
#    Time: 68757500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Read RAM Bank Stat Reg 4 bytes 0x1XXXX0XF from byte address 0x000350
#    Time: 80947500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Internal test of on-board RAM bank 1 complete
#    Time: 80947500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test RAM completed: PASSED.
#    Time: 80947500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

5.9.7.4.4 DMA OCP Channels Results

ModelSim transcript output during simulation is of the form:

```
# ** Note: DMA read response data process started
#    Time: 2028750 ps  Iteration: 14  Instance: /test_uber/test_uber_dma_i
```

```
# ** Note: DMA write process started (Base address = 0x2000007F00)
#    Time: 2028750 ps  Iteration: 14  Instance: /test_uber/test_uber_dma_i
# ** Note: DMA write process completed
#    Time: 63493750 ps  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: 4032 bytes transferred.
#    Time: 63493750 ps  Iteration: 13  Instance: /test_uber/test_uber_dma_i

# ** Note: DMA read command process started (Base address = 0x2000007F00)
#    Time: 63493750 ps  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: DMA read command process completed
#    Time: 63576250 ps  Iteration: 13  Instance: /test_uber/test_uber_dma_i

# ** Note: DMA read response data process completed
#    Time: 67456250 ps  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: 4032 bytes transferred with 0 data error(s)
#    Time: 67456250 ps  Iteration: 13  Instance: /test_uber/test_uber_dma_i

# ** Note: Test DMA completed: PASSED.
#    Time: 67456250 ps  Iteration: 13  Instance: /test_uber/test_uber_dma_i
```

5.9.7.4.5 Completion Results

Assuming that all tests passed, **ModelSim** transcript output on successful completion of simulation is of the form:

```
# ** Failure: Test of design UBER completed: PASSED.
#    Time: 82126250 ps  Iteration: 15  Process: /test_uber/test_results_p File:
../common/test_uber.vhd
# Break in Process test_results_p at ../common/test_uber.vhd line 407
# Simulation Breakpoint: Break in Process test_results_p at ../common/test_uber.vhd
line 407
# MACRO ./uber-admxrc6t1.do PAUSED at line 216
```

6 Common HDL Components

The ADM-XRC Gen 3 SDK provides a number of HDL components that are used in the example FPGA designs and testbenches. These components may also be used in customer FPGA designs. This section provides details of their interfaces and structure.

The components are divided into groups as follows:

- ADB3 OCP
- ADB3 Target
- ADB3 Probe
- Memory Interface (Deprecated)
- DDR3 SDRAM Interface
- Memory Application
- Memory Model
- Clock Frequency Measurement
- ChipScope

6.1 ADB3 OCP

The ADB3 OCP library VHDL source code is located in **hdl/vhdl/common/adb3_ocp** and contains the following packages:

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)
- ADB3 OCP Testbench Package (adb3_ocp_tb_pkg)

6.1.1 ADB3 OCP Profile Definition Package (adb3_ocp)

The package **adb3_ocp** defines constants and types which relate to the ADB3 OCP profiles. These OCP profiles are used by many of the reuseable VHDL modules in this SDK, and to connect together the various blocks in the example FPGA designs.

ADB3 OCP Constants

- **ADB3_OCP_ADDR_WIDTH**. Transaction byte address vector width.
- **ADB3_OCP_BURST_WIDTH**. Transaction burst length vector width.
- **ADB3_OCP_TAG_WIDTH**. Transaction identifier vector width.
- **ADB3_OCP_DATA_WIDTH**. Transfer data vector width.
- **ADB3_OCP_DATA_WIDTH_LOG2**. Log base 2 of **ADB3_OCP_DATA_WIDTH**.
- **ADB3_OCP_BE_WIDTH**. Transfer data byte enables vector width.
- **ADB3_OCP_BE_WIDTH_LOG2**. Log base 2 of **ADB3_OCP_BE_WIDTH**.

ADB3 OCP Lite Constants

- **ADB3_OCP_L_ADDR_WIDTH**. Transaction byte address vector width.
- **ADB3_OCP_L_DATA_WIDTH**. Transfer data vector width.
- **ADB3_OCP_L_DATA_WIDTH_LOG2**. Log base 2 of **ADB3_OCP_L_DATA_WIDTH**.
- **ADB3_OCP_L_BE_WIDTH**. Transfer data byte enables vector width.
- **ADB3_OCP_L_BE_WIDTH_LOG2**. Log base 2 of **ADB3_OCP_L_BE_WIDTH**.

Command Types/Constants

- **ADB3_OCP_CMD_WIDTH**. Transaction command vector width.
- **ocp_CmdT**. Transaction command type vector of width **ADB3_OCP_CMD_WIDTH**.
- **OCP_CMD_IDLE**. Master to slave invalid command.
- **OCP_CMD_WRITE**. Master to slave valid write command.
- **OCP_CMD_READ**. Master to slave valid read command.

Response Types/Constants

- **ADB3_OCP_RESP_WIDTH**. Transaction command vector width.
- **ocp_RespT**. Transaction command type vector of width **ADB3_OCP_RESP_WIDTH**.
- **OCP_RESP_NONE**. Slave to master invalid response.
- **OCP_RESP_VALID**. Slave to master valid response.

ADB3 OCP Types

- **adb3_ocp_addr_s**. Transaction address vector of width **ADB3_OCP_ADDR_WIDTH**.
- **adb3_ocp_burst_s**. Transaction burst length vector of width **ADB3_OCP_BURST_WIDTH**.

- **adb3_ocp_tag_s**. Transaction identifier vector of width **ADB3_OCP_TAG_WIDTH**.
- **adb3_ocp_data_s**. Transfer data vector of width **ADB3_OCP_DATA_WIDTH**.
- **adb3_ocp_be_s**. Transfer data byte enable vector of width **ADB3_OCP_BE_WIDTH**.

ADB3 OCP Lite Types

- **adb3_ocp_l_addr_s**. Transaction address vector of width **ADB3_OCP_L_ADDR_WIDTH**.
- **adb3_ocp_l_data_s**. Transfer data vector of width **ADB3_OCP_L_DATA_WIDTH**.
- **adb3_ocp_l_be_s**. Transfer data byte enable vector of width **ADB3_OCP_L_BE_WIDTH**.

adb3_ocp_m2sT ADB3 OCP Interface Record

- **Cmd**. Transaction type and validity of type **ocp_CmdT**.
- **Addr**. Transaction byte address (16-byte aligned) of type **adb3_ocp_addr_s**.
- **BurstLength**. Transaction burst length of type **adb3_ocp_burst_s**.
- **Tag**. Transaction identifier of type **adb3_ocp_tag_s**.
- **Data**. Transfer data of type **adb3_ocp_data_s**.
- **DataByteEn**. Transfer data byte enable of type **adb3_ocp_be_s**.
- **DataValid**. Transfer valid of type **std_logic**.
- **RespAccept**. Transfer ready of type **std_logic**.

adb3_ocp_s2mT ADB3 OCP Interface Record

- **CmdAccept**. Transaction ready of type **std_logic**.
- **DataAccept**. Transfer ready of type **std_logic**.
- **Resp**. Transfer type and validity of type **ocp_RespT**.
- **Data**. Transfer data of type **adb3_ocp_data_s**.
- **Tag**. Transaction identifier of type **adb3_ocp_tag_s**.

adb3_ocp_l_m2sT ADB3 OCP Lite Interface Record

- **Cmd**. Transaction type and validity of type **ocp_CmdT**.
- **Addr**. Transaction byte address (16-byte aligned) of type **adb3_ocp_l_addr_s**.
- **Data**. Transfer data of type **adb3_ocp_l_data_s**.
- **DataByteEn**. Transfer data byte enable of type **adb3_ocp_l_be_s**.

adb3_ocp_l_s2mT ADB3 OCP Lite Interface Record

- **CmdAccept**. Transaction ready of type **std_logic**.
- **Resp**. Transfer type and validity of type **ocp_RespT**.
- **Data**. Transfer data of type **adb3_ocp_l_data_s**.

ADB3 OCP Interface Constants

- **ADB3_OCP_DEFAULT_M2S**. Default values for type **adb3_ocp_m2sT**.
- **ADB3_OCP_DEFAULT_S2M**. Default values for type **adb3_ocp_s2mT**.
- **ADB3_OCP_L_DEFAULT_M2S**. Default values for type **adb3_ocp_l_m2sT**.
- **ADB3_OCP_L_DEFAULT_S2M**. Default values for type **adb3_ocp_l_s2mT**.
- **ADB3_OCP_FLOAT_M2S**. Float values to allow multi-channel drive.
- **ADB3_OCP_Z_M2S**. Tri-state values to allow multi-channel drive.
- **ADB3_OCP_Z_S2M**. Tri-state values to allow multi-channel drive.

Procedures and Functions

- **adb3_ocp_l_wr**. Helper procedure for register write using ADB3 OCP lite. Uses the **adb3_ocp_m2sT Data** and **DataByteEn** signals together with a data bit offset to return a register value.
- **adb3_ocp_cmd_is_wr**. Returns true if command is write.
- **adb3_ocp_cmd_is_rd**. Returns true if command is read.
- **adb3_ocp_cmd_is_idle**. Returns true if command is idle.

For descriptions of the OCP protocols and example transactions, refer to ADB3 OCP Protocol Reference and ADB3 OCP Lite Protocol Reference.

6.1.2 ADB3 OCP Component Declaration Package (adb3_ocp_comp)

The package **adb3_ocp_comp** defines types, functions and components which use the ADB3 OCP profile.

Type Definitions

- **adb3_ocp_addr_range_t**. Address range table entry type used by **adb3_ocp_addr_range_table_t**.
- **adb3_ocp_l_addr_range_t**. Address range table entry type used by **adb3_ocp_l_addr_range_table_t**.
- **adb3_ocp_addr_range_table_t**. Address range table type used by adb3_ocp_split_b component.
- **adb3_ocp_l_addr_range_table_t**. Address range table type used by adb3_ocp_l_split component.
- **adb3_ocp_reg32_array_t**. 32-bit register array type used by adb3_ocp_reg32_b component.

Function Definitions

- **adb3_ocp_base**. Extend (with '0's) a base address vector of width **w** to an ADB3 OCP base address vector of width **ADB3_OCP_ADDR_WIDTH**.
- **adb3_ocp_mask**. Extend (with '1's) a mask address vector of width **w** to an ADB3 OCP mask address vector of width **ADB3_OCP_ADDR_WIDTH**.
- **adb3_ocp_off**. Convert an integer address offset to an ADB3 OCP address vector of width **ADB3_OCP_ADDR_WIDTH**.
- **adb3_ocp_mask_width**. Return the integer width of an ADB3 OCP mask address vector.

Components that require the data for the current OCP command to be fully read or written before the next OCP command is accepted are categorised as 'blocking'. Blocking components have a lower data throughput, but require less FPGA resources. An example of their use would be register access. Blocking components in the ADB3 OCP group are as follows:

Blocking Component Definitions

- adb3_ocp_mux_b
- adb3_ocp_split_b
- adb3_ocp_simple_bus_if
- adb3_ocp_reg32_b
- adb3_ocp_full2lite_b
- adb3_ocp_l_split
- adb3_ocp_l_spliteq
- adb3_ocp_l_ictrl

Components that can accept further OCP commands before the data for the current OCP command has been fully read or written are categorised as 'non-blocking'. Non-blocking components have a higher data throughput, but require more FPGA resources. An example of their use would be on-board memory access. Non-blocking components in the ADB3 OCP group are as follows:

Non-Blocking Component Definitions

- adb3_ocp_cross_clk_dom
- adb3_ocp_mux_nb
- adb3_ocp_split_nb
- adb3_ocp_ocp2ddr3_nb
- adb3_ocp_retime_nb
- adb3_ocp_simple_bus_if_nb

6.1.2.1 adb3_ocp_mux_b

6.1.2.1.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to multiplex multiple primary ADB3 OCP channels onto a single secondary ADB3 OCP channel. The multiplex is controlled by round-robin arbitration of OCP commands.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.1.2 Interface

The **adb3_ocp_mux_b** component interface is shown in Figure 39 below and described in Table 134.

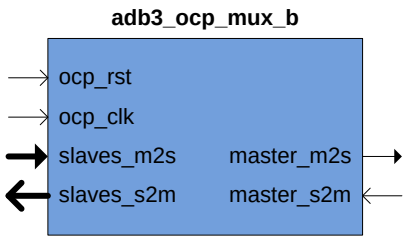

Figure 39 : adb3_ocp_mux_b component interface

Signal	Type	Description
mux_inputs	Generic	Number of primary OCP channels to be multiplexed.

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
		OCP Primary Ports
slaves_m2s	Input	OCP Primary (slave) ports M2S connection.
slaves_s2m	Output	OCP Primary (slave) ports S2M connection.
		OCP Secondary Port
master_m2s	Output	OCP Secondary (master) port M2S connection.
master_s2m	Input	OCP Secondary (master) port S2M connection.

Table 134 : adb3_ocp_mux_b component interface

6.1.2.1.3 Description

The **adb3_ocp_mux_b** component block diagram is shown in Figure 40 below.

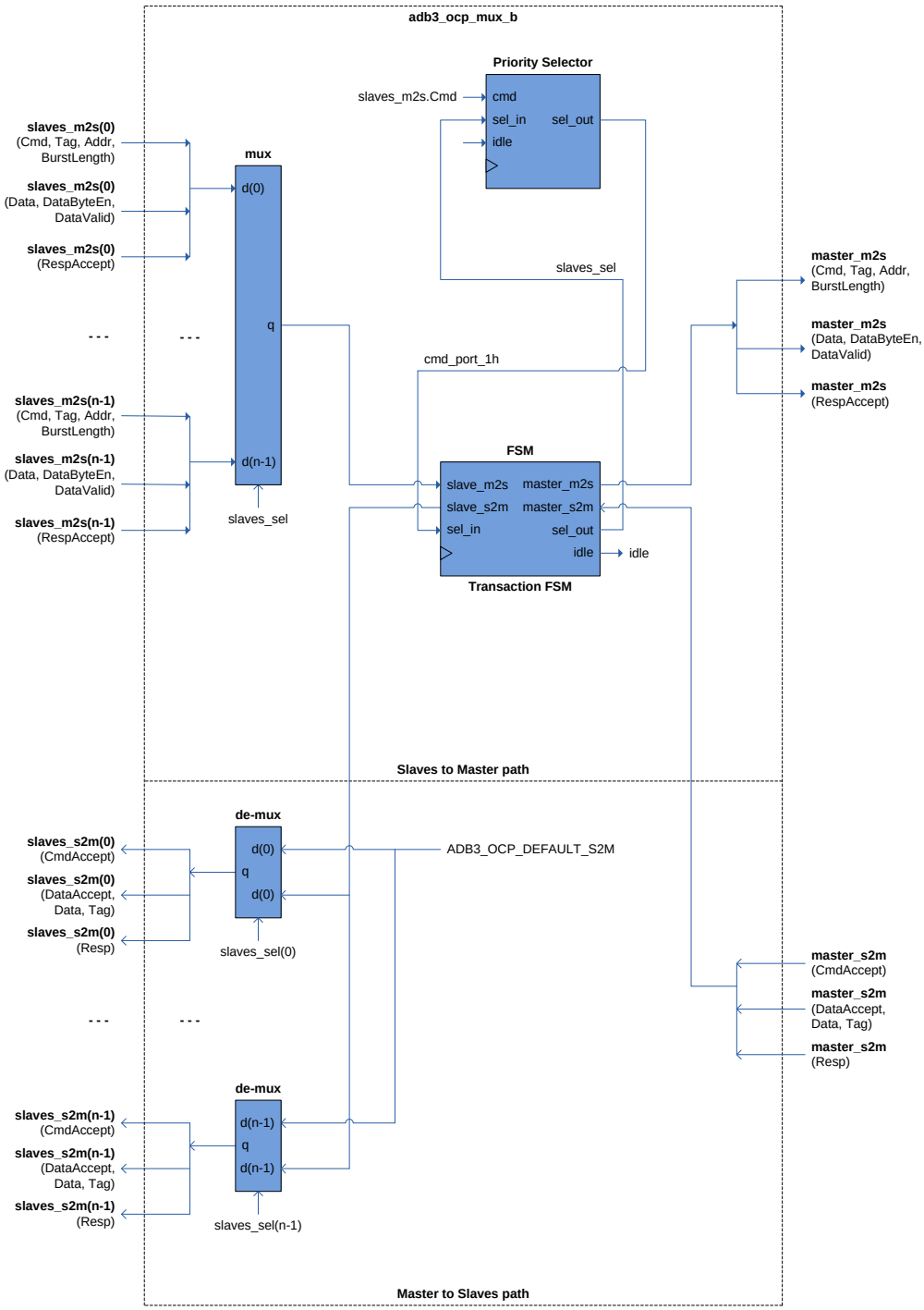

Figure 40 : adb3_ocp_mux_b block diagram

The component consists of a priority selector, an OCP transaction state machine, and OCP channel multiplexors.

The slaves to master path consists of the **Cmd**, **Tag**, **BurstLength**, **Addr**, **DataValid**, **DataByteEn**, **Data**, and **RespAccept** elements of the **slaves_m2s**/**master_m2s** signals.

The master to slaves path consists of the **CmdAccept**, **DataAccept**, **Resp**, **Tag**, and **Data** elements of the **master_s2m**/**slaves_s2m** signals.

Priority Selector

- The highest priority OCP channel with active slaves_m2s.Cmd is selected when the transaction state machine is idle.
- Priority is assigned on a round-robin basis .
- Selection is made using a 1-hot vector **cmd_port_1h**.

OCP Transaction State Machine

- The state machine remains idle until a non-zero value appears on **cmd_port_1h**.
- The new active OCP channel is selected by re-timing **cmd_port_1h** to produce **slaves_sel**.
- The OCP transaction on the **slave_m2s**/**slave_s2m** signals is re-timed and connected to the **master_m2s**/**master_s2m** signals.

OCP Channel Multiplexors

- The active **slaves_m2s** signals are selected using the **slaves_sel** signal.
- The active **slaves_s2m** signals are driven by the state machine. The inactive **slaves_s2m** signals are tied to the inactive state (**ADB3_OCP_DEFAULT_S2M**).

6.1.2.2 adb3_ocp_split_b

6.1.2.2.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to de-multiplex a single primary ADB3 OCP channel into multiple secondary ADB3 OCP channels. Selection of the active secondary channel is controlled by the primary channel command address.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.2.2 Interface

The **adb3_ocp_split_b** component interface is shown in Figure 41 below and described in Table 135.

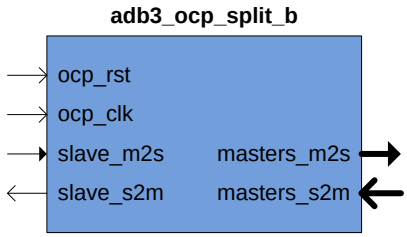

Figure 41 : adb3_ocp_split_b component interface

Signal	Type	Description
addr_range_table	Generic	Table defining the address ranges to be used to control the split operation.
error_data	Generic	OCP Response Data to be returned if address is out of range (default = "DEADC0DEDEADC0DEDEADC0DEDEADC0DE").
full_addr_out	Generic	Select full or partial OCP address output (default = true).
unused_addr_bit	Generic	Select value of unused OCP address output bits (default = '0').
retime_slave	Generic	Unused.
retime_masters	Generic	Unused.

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP port clock.
		OCP Primary Port
slave_m2s	Input	OCP Primary (slave) port M2S connection.
slave_s2m	Output	OCP Primary (slave) port S2M connection.
		OCP Secondary Ports
masters_m2s	Output	OCP Secondary (master) ports M2S connection.
masters_s2m	Input	OCP Secondary (master) ports S2M connection.

Table 135 : adb3_ocp_split_b component interface

6.1.2.2.3 Description

The **adb3_ocp_split_b** component block diagram is shown in Figure 42 below.

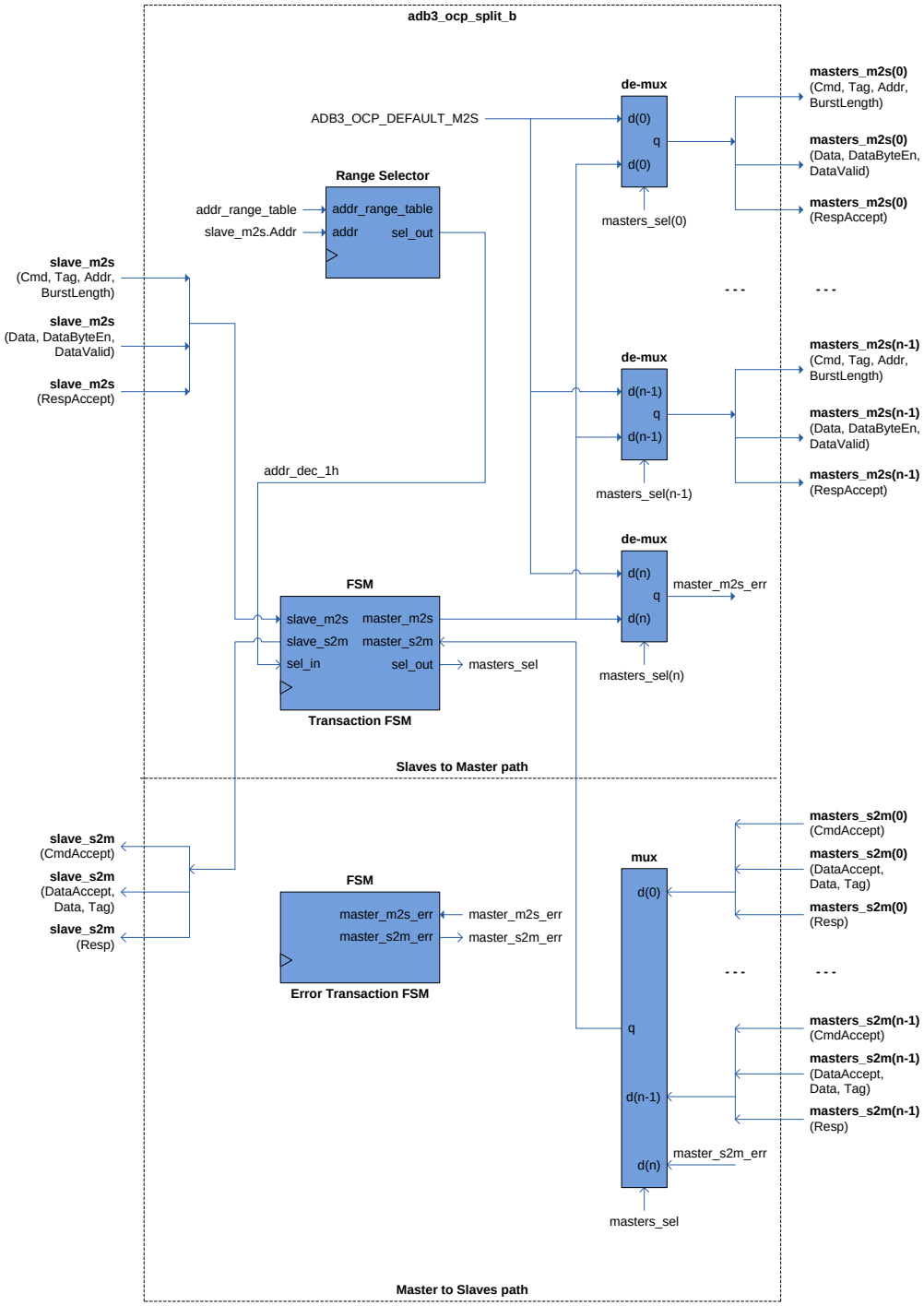

Figure 42 : adb3_ocp_split_b block diagram

The component consists of an address range selector, an OCP transaction state machine, an OCP error transaction state machine, and OCP channel multiplexors.

The slave to masters path consists of the **Cmd**, **Tag**, **BurstLength**, **Addr**, **DataValid**, **DataByteEn**, **Data**, and **RespAccept** elements of the **slave_m2s**/**masters_m2s** signals.

The masters to slave path consists of the **CmdAccept**, **DataAccept**, **Resp**, **Tag**, and **Data** elements of the **masters_s2m**/**slave_s2m** signals.

Address Range Table

- This consists of a number of **addr_mask**, **addr_base** pairs. Each pair specifies the active address range for its associated **masters_m2s**/**masters_s2m** channel.
- The **addr_mask** vector is the address range active bit mask (active low). That is, bits that are '0' will enable, bits that are '1' will mask.
- The **addr_base** vector is the base address of the range.
- For example, **addr_base** = 0x0..00000400, **addr_mask** = 0xF..FFC003FF, specifies an address range 0x0..00000400 to 0x0..000007FF with address bits (21:10) used for address range decoding.

Address Range Selector

- Selection is made using an (n+1) bit 1-hot vector **addr_dec_1h(n:0)**, where n is the number of entries in the **addr_range_table**.
- The slave_m2s.Addr is compared with the address ranges specified by the **addr_range_table** generic. If the address is in a valid range, then the appropriate bit of the **addr_dec_1h** signal is set. If the address is not in any range, the top bit of the **addr_dec_1h** signal is set.

OCP Transaction State Machine

- The state machine remains idle until a command on **slave_m2s.Cmd** is accepted.
- The new active OCP channel is selected by re-timing **addr_dec_1h** to produce **masters_sel**.
- The OCP transaction on the **slave_m2s**/**slave_s2m** signals is re-timed and connected to the **master_m2s**/**master_s2m** signals.

OCP Error Transaction State Machine

- The state machine remains idle until a command on **master_m2s_err** is accepted.
- The OCP transaction on the **master_m2s_err** signals is re-timed and connected to the **master_s2m_err** signals.
- OCP writes on **master_m2s_err** signals are ignored.
- OCP reads on **master_m2s_err** signals will return the **error_data** generic value.

OCP Channel Multiplexors

- The active **masters_s2m** signals are selected using the **masters_sel** signal.
- The active **masters_m2s** signals are driven by the state machine. The inactive **masters_m2s** signals are tied to the inactive state (**ADB3_OCP_DEFAULT_M2S**).
- The OCP address output on the active **masters_m2s.Addr** signal is controlled by the **full_addr_out** and **unused_addr_bit** generics. If **full_addr_out** is true, then the full OCP address is used. If **full_addr_out** is false, then only the addresss bits within the range are used, with the higher bits set to **unused_addr_bit**.

6.1.2.3 adb3_ocp_simple_bus_if

6.1.2.3.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to convert a single ADB3 OCP channel to a simple parallel interface.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.3.2 Interface

The **adb3_ocp_simple_bus_if** component interface is shown in Figure 43 below and described in Table 136.

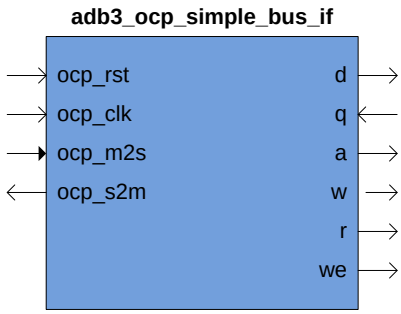

Figure 43 : adb3_ocp_simple_bus_if component interface

Signal	Type	Description
addr_width	Generic	Width of the a address output (byte address).
data_width	Generic	Width of the d/q data input/output.
read_latency	Generic	Number of cycles delay before q data input is available (default = 1).

Signal	Type	Description
		OCP Interface
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP M2S connection.
ocp_s2m	Output	OCP S2M connection.
		Simple Bus Interface
d	Output	Write data of width data_width.
q	Input	Read data of width data_width.
a	Output	Write/Read address (byte address) of width addr_width.
w	Output	Write enable.
r	Output	Read enable.

Table 136 : adb3_ocp_simple_bus_if component interface (continued on next page)

Signal	Type	Description
we	Output	Write data byte enable of width data_width/8.

Table 136 : adb3_ocp_simple_bus_if component interface

6.1.2.3.3 Description

An OCP burst write command on the **ocp_m2s** input is accepted if the simple bus interface is inactive. Each 16-byte burst of write data is converted into 1 or more writes on the simple bus interface, depending on the generic **data_width**. Each write consists of data **d** (width **data_width**), byte address **a** (width **addr_width**), write active bit **w**, and write enable bits **we** (width **data_width/8**).

An OCP burst read command on the **ocp_m2s** input is accepted if the simple bus interface is inactive. Each 16-byte read burst is converted into 1 or more reads on the simple bus interface, depending on the generic **data_width**. Each read consists of data **q** (width **data_width**), byte address **a** (width **addr_width**), and read active bit **r**. The data **q** is sampled **read_latency** cycles after the read active bit **r** is active.

6.1.2.3.3.1 Example Waveforms

In the following example, we are performing 2 OCP writes with burst length of 1, to a simple bus with 32 bit data. OCP data consists of 16 bytes, and so this results in 4 writes to the simple bus, each of 4 bytes. Each simple bus write is indicated by the **w** signal. The simple bus 4-byte write data on **d** is enabled by the 4-bit **we** bus. OCP addresses are always 16-byte aligned. The write address **a** ls nibble will therefore always sequence through values 0x00, 0x04, 0x08, 0x0C.

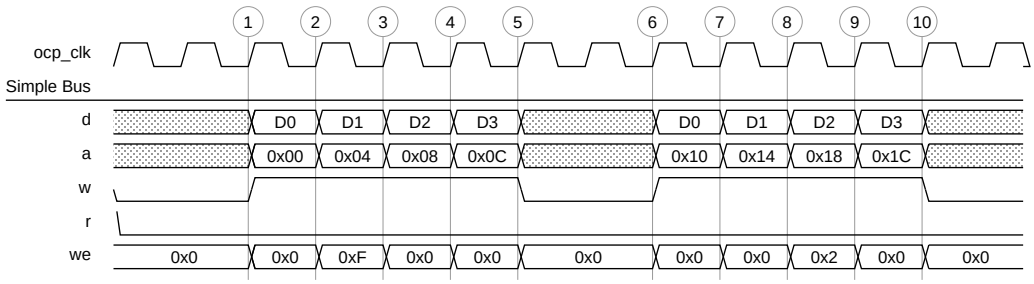

Figure 44 : OCP Writes (Burst Length = 1, d = 32 bits) To Simple Bus

The first OCP write is bytes 0-3 of D1 to byte addresses 0x04, 0x05, 0x06, 0x07.

The second OCP write is byte 1 of D2 to byte address 0x19.

In the following example, we are performing 1 OCP read with burst length of 1, from a simple bus with **read_latency** of 1 and 32 bit data. OCP responses consists of 16 bytes, and so this results in 4 reads from the simple bus, each of 4 bytes. Each simple bus read is indicated by the **r** signal. The simple bus 4-byte read data on **q** is expected 1 cycle after **r** is valid (read_latency = 1). OCP addresses are always 16-byte aligned. The read address **a** ls nibble will therefore always sequence through values 0x00, 0x04, 0x08, 0x0C.

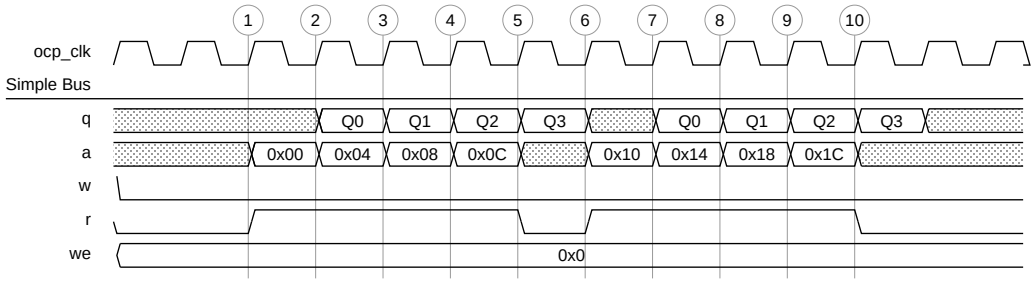

Figure 45 : OCP Read (Burst Length = 1, Read Latency = 1, d = 32 bits) From Simple Bus

The OCP read is 16 bytes from byte address starting at 0x00.

In the following example, we are performing 2 OCP writes with burst length of 1, to a simple bus with 128 bit data. OCP data consists of 16 bytes, and so this results in 1 write to the simple bus of 16 bytes. Each simple bus write is indicated by the **w** signal. The simple bus 16-byte write data on **d** is enabled by the 16-bit **we** bus. OCP addresses are always 16-byte aligned. The write address **a** ls nibble will therefore always have value 0x00.

In the following example, we are performing 2 OCP reads with burst length of 1, from a simple bus with **read_latency** of 1 and 128 bit data. OCP responses consists of 16 bytes, and so this results in 1 read from the simple bus of 16 bytes. Each simple bus read is indicated by the **r** signal. The simple bus 16-byte read data on **q** is expected 1 cycle after **r** is valid (read_latency = 1). OCP addresses are always 16-byte aligned. The read address **a** ls nibble will therefore always have value 0x00.

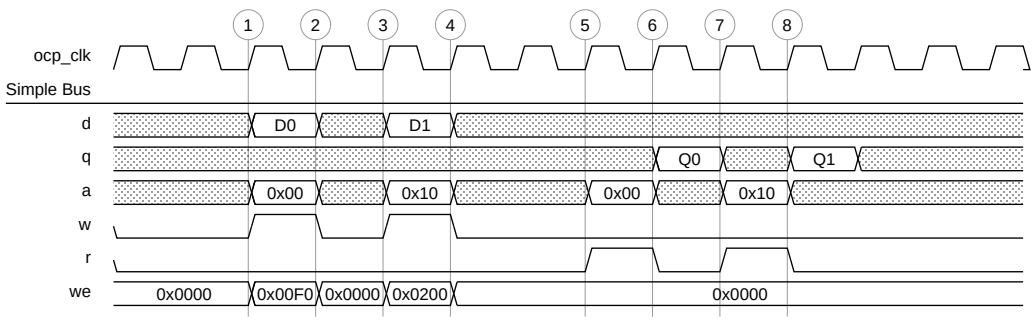

Figure 46 : OCP Writes/Reads (Burst Length = 1, Read Latency = 1, d = 128 bits) To/From Simple Bus

The first OCP write is bytes 4-7 of D0 to byte addresses 0x04, 0x05 0x06 0x07.

The second OCP write is byte 9 of D1 to byte address 0x19.

The first OCP read is 16 bytes from byte address starting at 0x00.

The second OCP read is 16 bytes from byte address starting at 0x10.

6.1.2.4 adb3_ocp_reg32_b

6.1.2.4.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to implement a register bank connected to a single ADB3 OCP channel. The number of register in the bank is controlled by the number of elements in the register initialisation generic **reg_init**.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.4.2 Interface

The **adb3_ocp_reg32_b** component interface is shown in Figure 47 below and described in Table 137.

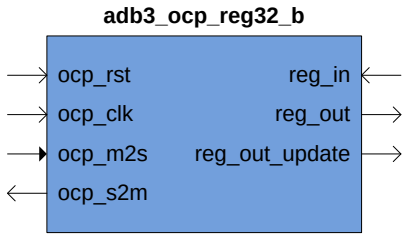

Figure 47 : adb3_ocp_reg32_b component interface

Signal	Type	Description
reg_init	Generic	Array defining the number and size of registers and their initial values.

Signal	Type	Description
		OCP Interface
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP port clock.
ocp_m2s	Input	OCP port M2S connection.
ocp_s2m	Output	OCP port S2M connection.
		Register Interface
reg_in	Input	Array of register data to be read by OCP interface.
reg_in_rd	Output	Vector of register read active flags.
reg_out	Output	Array of register data written by OCP interface.
reg_out_wr	Output	Vector of register write active flags.

Table 137 : adb3_ocp_reg32_b component interface

6.1.2.4.3 Description

The **adb3_ocp_reg32_b** component block diagram is shown in Figure 48 below.

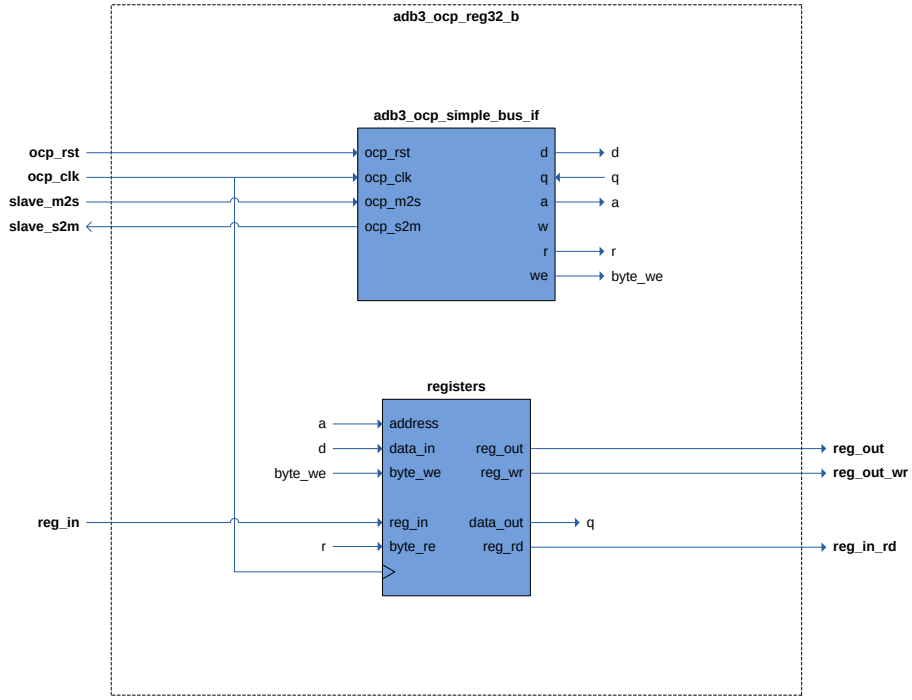

Figure 48 : adb3_ocp_reg32_b block diagram

The **adb3_ocp_reg32_b** consists of an instance of the adb3_ocp_simple_bus_if component connected to a programmable number of 32-bit read/write registers.

Write commands from the adb3_ocp_simple_bus_if result in **d** data appearing on the **reg_out** 32-bit register array output. The register which is written is decoded using the address **a**. The bytes of the register that are written are controlled by the byte write enables **byte_we**. The appropriate bit of the **reg_out_wr** output vector indicates that the output register has been written.

Read commands from the adb3_ocp_simple_bus_if result in **reg_in** 32-bit register array input data appearing on **q**. The register which is read is decoded using the address **a**. All bytes of the register are read. The appropriate bit of the **reg_in_rd** output vector indicates that the input register has been read.

The number of registers, and their contents on reset, are defined by the **reg_init** generic.

The **ocp_m2s.Addr** address field is truncated to width **REG_ADDR_WIDTH** by the adb3_ocp_simple_bus_if block to produce its **a** address output. The registers decode the **a** address output to determine the register to be accessed.

6.1.2.4.3.1 Example Waveforms

In the following example, we are performing 2 OCP writes with burst length of 1, to a simple bus with 32 bit registers. OCP data consists of 16 bytes, and so this results in 4 writes to the simple bus, each of 4 bytes. Each simple bus write is indicated by the **w** signal. The simple bus 4-byte write data on **d** is enabled by the 4-bit **we** bus. OCP addresses are always 16-byte aligned. The write address **a** ls nibble will therefore always sequence through values 0x00, 0x04, 0x08, 0x0C.

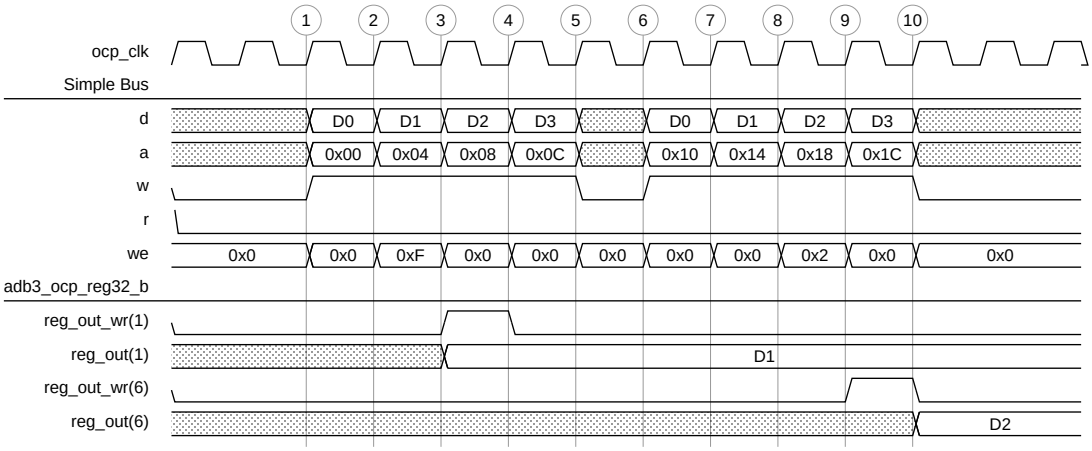

Figure 49 : OCP Writes (Burst Length = 1, d = 32 bits) To Registers

The first OCP write is bytes 0-3 of D1 to register index 1.

The second OCP write is byte 1 of D2 to register index 6.

In the following example, we are performing 1 OCP read with burst length of 1, from a simple bus with **read_latency** of 1 and 32 bit registers. OCP responses consists of 16 bytes, and so this results in 4 reads from the simple bus, each of 4 bytes. Each simple bus read is indicated by the **r** signal. The simple bus 4-byte read data on **q** is expected 1 cycle after **r** is valid (read_latency = 1). OCP addresses are always 16-byte aligned. The read address **a** ls nibble will therefore always sequence through values 0x00, 0x04, 0x08, 0x0C.

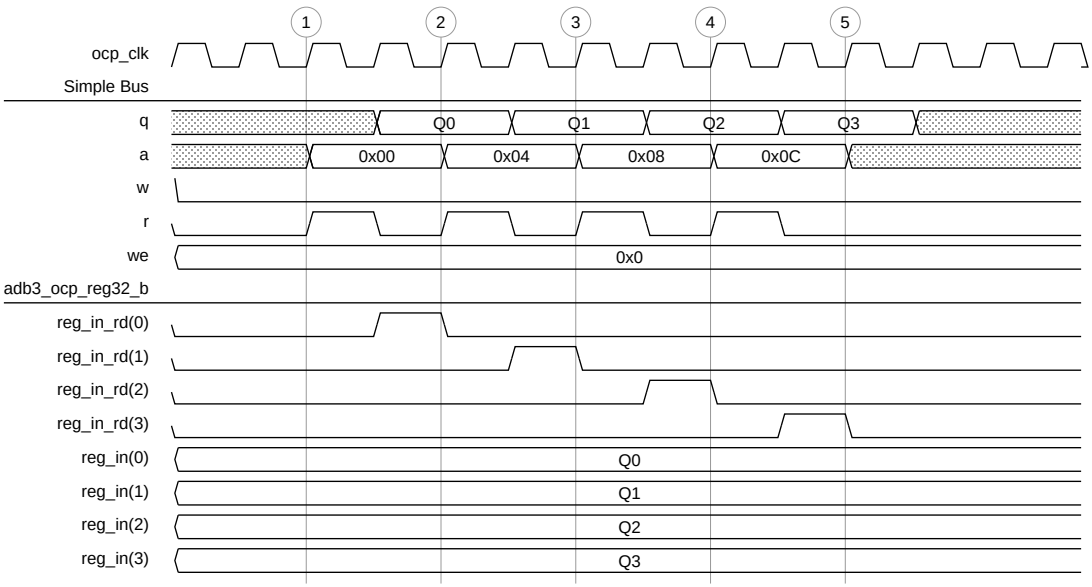

Figure 50 : OCP Read (Burst Length = 1, Read Latency = 1, d = 32 bits) From Registers

The OCP read is 16 bytes from register indices 0..3.

6.1.2.5 adb3_ocp_full2lite_b

6.1.2.5.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to interface a single primary ADB3 OCP full channel to a single secondary ADB3 OCP lite channel.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.5.2 Interface

The **adb3_ocp_full2lite_b** component interface is shown in Figure 51 below and described in Table 138.

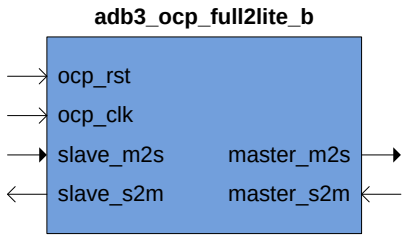

Figure 51 : adb3_ocp_full2lite_b component interface

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
		OCP Primary Ports
slave_m2s	Input	OCP full (slave) port M2S connection.
slave_s2m	Output	OCP full (slave) port S2M connection.
		OCP Secondary Port
master_m2s	Output	OCP lite (master) port M2S connection.
master_s2m	Input	OCP lite (master) port S2M connection.

Table 138 : adb3_ocp_full2lite_b component interface

6.1.2.5.3 Description

The **adb3_ocp_full2lite_b** component block diagram is shown in Figure 52 below.

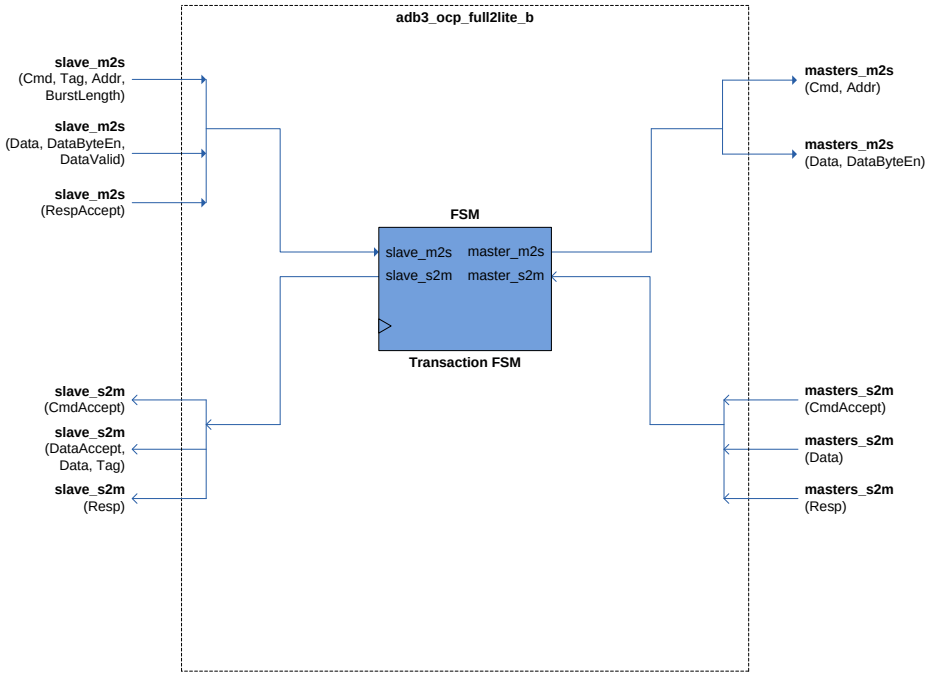

Figure 52 : adb3_ocp_full2lite_b block diagram

The component consists of an OCP transaction state machine.

The slave to master path consists of the **Cmd**, **Addr**, **DataByteEn**, and **Data** elements of the **slaves_m2s**/**master_m2s** signals.

The master to slave path consists of the **CmdAccept**, **Resp**, and **Data** elements of the **master_s2m**/**slaves_s2m** signals.

OCP Transaction State Machine

- The state machine remains idle until a valid command appears on the OCP full slave_m2s **Cmd** signal.
- Following a write command on the OCP full slave_m2s **Cmd** input, every write data burst is converted to OCP lite write commands on the master_m2s output. An OCP lite write command is issued for each 4-bytes of the **Data** signal which has non-zero **ByteEnable** bits.
- Following a read command on the OCP full slave_m2s **Cmd** input, four OCP lite read commands are issued on the master_m2s output. The Four OCP lite read data responses produced on the master_s2m **Data** signal are then converted back to a single OCP full read response on the slave_s2m **Data** output.

6.1.2.5.3.1 Example Waveforms

In the following example, a single OCP full write with burst length of 1 is converted to 4 OCP lite writes. OCP full addresses are always 16-byte aligned, OCP lite addresses are always 4-byte aligned.

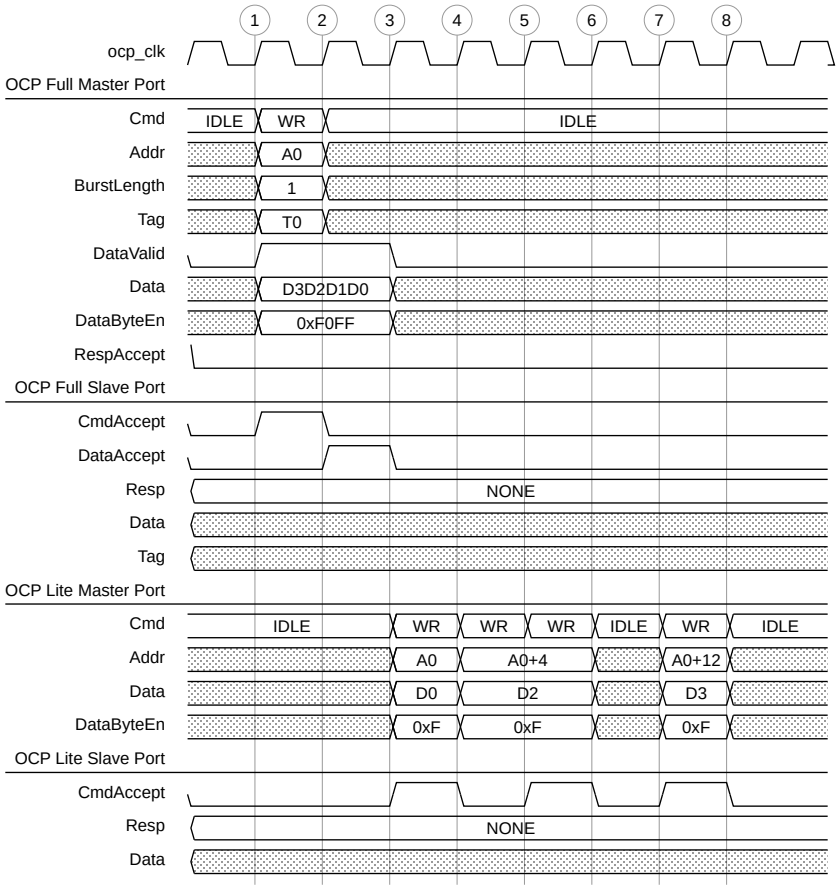

Figure 53 : OCP Full To OCP Lite Write Conversion

There is no OCP lite write to A0+8 as all BE2 bits are zero.

In the following example, a single OCP full read with burst length of 1 is converted to 4 OCP lite reads. The 4 OCP lite read responses are then converted to a single OCP full read response. OCP full addresses are always 16-byte aligned. OCP lite addresses are always 4-byte aligned.

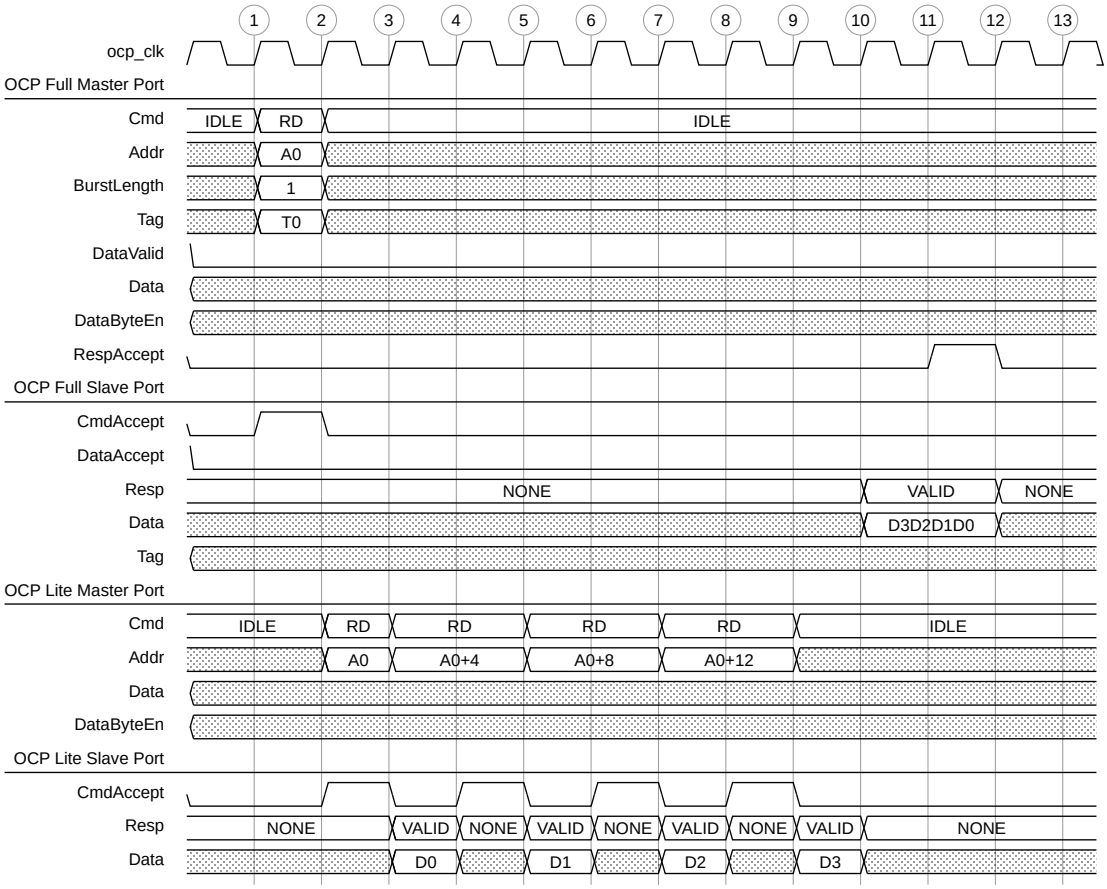

Figure 54 : OCP Full To OCP Lite Read Conversion

The first OCP lite read is 4 bytes from byte address A0.

6.1.2.6 adb3_ocp_l_split

6.1.2.6.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to de-multiplex a single primary ADB3 OCP lite channel into multiple secondary ADB3 OCP lite channels. The de-multiplex is controlled by the **addr_range_table** generic input. Selection of the active secondary channel is controlled by the primary channel command address.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.6.2 Interface

The **adb3_ocp_l_split** component interface is shown in Figure 55 below and described in Table 139.

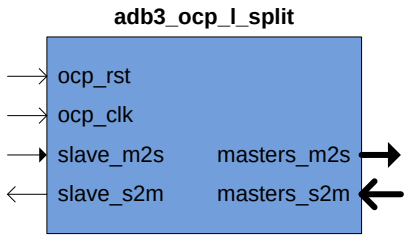

Figure 55 : adb3_ocp_l_split component interface

Signal	Type	Description
addr_range_table	Generic	Table defining the address ranges to be used to control the split operation.
error_data	Generic	OCP Response Data to be returned if address is out of range (default = "DEADC0DE").
full_addr_out	Generic	Select full or partial OCP address output (default = true).
unused_addr_bit	Generic	Select value of unused OCP address output bits (default = '0').

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP port clock.
		OCP Primary Port
slave_m2s	Input	OCP Primary (slave) port M2S connection.
slave_s2m	Output	OCP Primary (slave) port S2M connection.
		OCP Secondary Ports
masters_m2s	Output	OCP Secondary (master) ports M2S connection.
masters_s2m	Input	OCP Secondary (master) ports S2M connection.

Table 139 : adb3_ocp_l_split component interface

6.1.2.6.3 Description

The **adb3_ocp_l_split** component block diagram is shown in Figure 56 below.

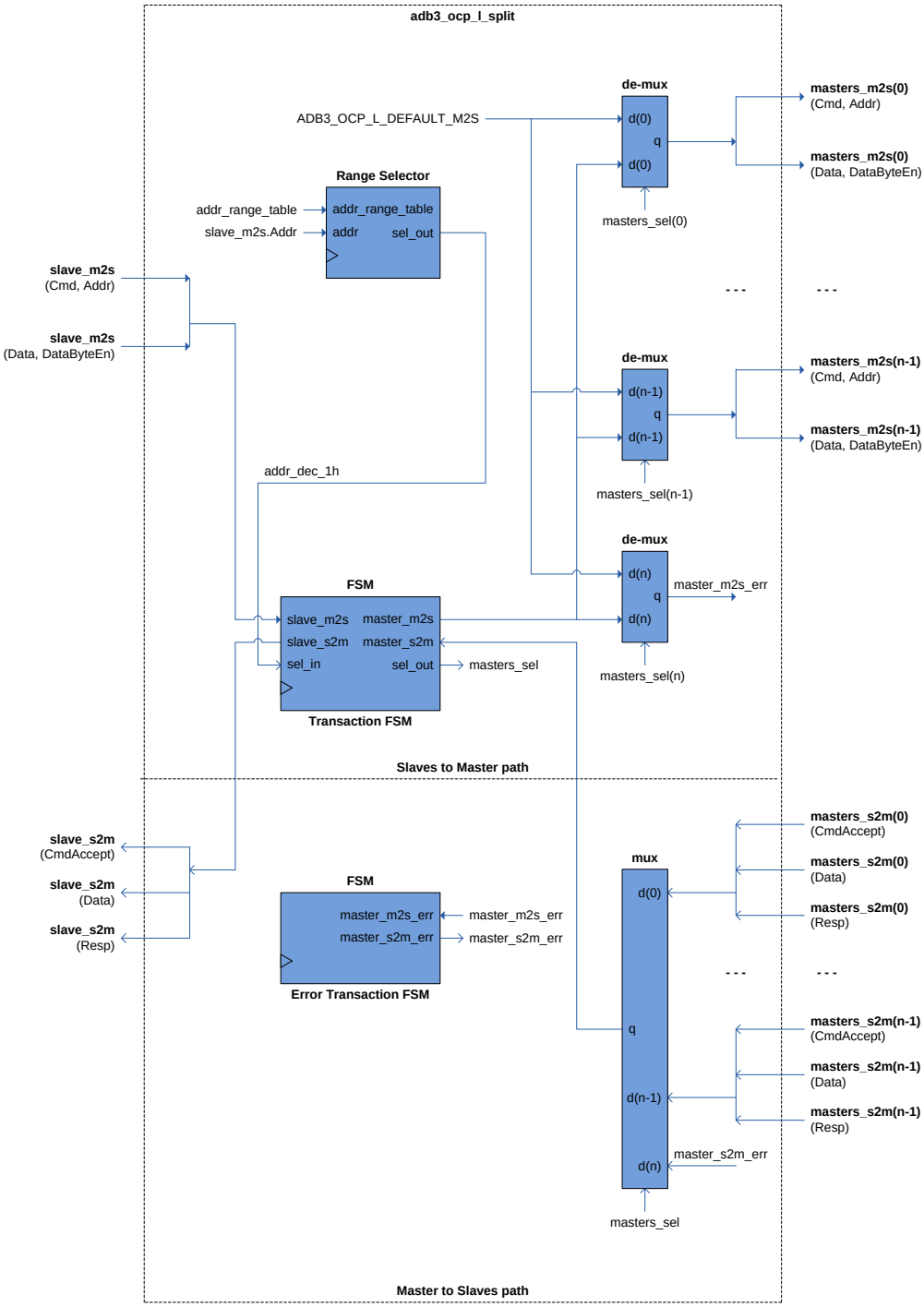

Figure 56 : adb3_ocp_l_split block diagram

The component consists of an address range selector, an OCP transaction state machine, an OCP error transaction state machine, and OCP channel multiplexors.

The slave to masters path consists of the **Cmd**, **Addr**, **DataByteEn**, and **Data** elements of the **slave_m2s**/**masters_m2s** signals.

The masters to slave path consists of the **CmdAccept**, **Resp**, and **Data** elements of the **masters_s2m**/**slave_s2m** signals.

Address Range Table

- This consists of a number of **addr_mask**, **addr_base** pairs. Each pair specifies the active address range for its associated **masters_m2s**/**masters_s2m** channel.
- The **addr_mask** vector is the address range active bit mask (active low). That is, bits that are '0' will enable, bits that are '1' will mask.
- The **addr_base** vector is the base address of the range.
- For example, **addr_base** = 0x00000400, **addr_mask** = 0xFFC003FF, specifies an address range 0x000400 to 0x000007FF with address bits (21:10) used for address range decoding.

Address Range Selector

- Selection is made using an (n+1) bit 1-hot vector **addr_dec_1h(n:0)**, where n is the number of entries in the **addr_range_table**.
- The slave_m2s.Addr is compared with the address ranges specified by the **addr_range_table** generic. If the address is in a valid range, then the appropriate bit of the **addr_dec_1h** signal is set. If the address is not in any range, the top bit of the **addr_dec_1h** signal is set.

OCP Transaction State Machine

- The state machine remains idle until a command on **slave_m2s.Cmd** is accepted.
- The new active OCP channel is selected by re-timing **addr_dec_1h** to produce **masters_sel**.
- The OCP transaction on the **slave_m2s**/**slave_s2m** signals is re-timed and connected to the **master_m2s**/**master_s2m** signals.

OCP Error Transaction State Machine

- The state machine remains idle until a command on **master_m2s_err** is accepted.
- The OCP transaction on the **master_m2s_err** signals is re-timed and connected to the **master_s2m_err** signals.
- OCP writes on **master_m2s_err** signals are ignored.
- OCP reads on **master_m2s_err** signals will return the **error_data** generic value.

OCP Channel Multiplexors

- The active **masters_s2m** signals are selected using the **masters_sel** signal.
- The active **masters_m2s** signals are driven by the state machine. The inactive **masters_m2s** signals are tied to the inactive state (**ADB3_OCP_L_DEFAULT_M2S**).
- The OCP address output on the active **masters_m2s.Addr** signal is controlled by the **full_addr_out** and **unused_addr_bit** generics. If **full_addr_out** is true, then the full OCP lite address is used. If **full_addr_out** is false, then only the addresss bits within the range are used, with the higher bits set to **unused_addr_bit**.

6.1.2.7 adb3_ocp_l_spliteq

6.1.2.7.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to de-multiplex a single primary ADB3 OCP lite channel into multiple secondary ADB3 OCP lite channels. The de-multiplex is controlled by the **masters** and **addr_lsb** generic inputs. Selection of the active secondary channel is controlled by the primary channel command address.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.7.2 Interface

The **adb3_ocp_l_spliteq** component interface is shown in Figure 57 below and described in Table 140.

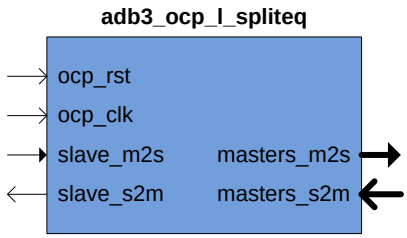

Figure 57 : adb3_ocp_l_spliteq component interface

Signal	Type	Description
masters	Generic	Number of equally sized address regions (0..masters-1).
addr_lsb	Generic	Size of equally sized address regions (addr_lsb-1..0).
full_addr_dec	Generic	Select full or partial decoding of master command address (default = true).
full_addr_out	Generic	Select full or partial OCP address output (default = true).
unused_addr_bit	Generic	Select value of unused OCP address output bits (default = '0').

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP port clock.
		OCP Primary Port
slave_m2s	Input	OCP Lite Primary (slave) port M2S connection.
slave_s2m	Output	OCP Lite Primary (slave) port S2M connection.
		OCP Secondary Ports
masters_m2s	Output	OCP Lite Secondary (master) ports M2S connection.
masters_s2m	Input	OCP Lite Secondary (master) ports S2M connection.

Table 140 : adb3_ocp_l_spliteq component interface

6.1.2.7.3 Description

The **adb3_ocp_l_spliteq** component is a wrapper for an instance of the adb3_ocp_l_split component. The **adb3_ocp_l_spliteq** generic inputs are used to generate the adb3_ocp_l_split generic inputs.

The adb3_ocp_l_split **addr_range_table** generic is generated using the **gen_table** function. The function uses the **masters**, **addr_lsb**, and **full_addr_dec** generics to generate the table.

For example, a 3 way split on address bits above 15 is produced using **masters** = 3, and **addr_lsb** = 16. The **addr_range_table** generic that is passed to the underlying instance of adb3_ocp_l_split is generated as follows:

- addr_range_table(0).addr_base = X"00000000"
- addr_range_table(1).addr_base = X"00010000"
- addr_range_table(2).addr_base = X"00020000"
- If **full_addr_dec** is true, then address (31:16) is used for matching, so that addr_range_table(n) .addr_mask = X"0000FFFF"
- If **full_addr_dec** is false, then address (17:16) is used for matching, so that addr_range_table(n) .addr_mask = X"FFFCFFFF"

The **full_addr_out** and **unused_addr_bit** generics are passed directly to the underlying adb3_ocp_l_split instance.

6.1.2.8 adb3_ocp_l_ictrl

6.1.2.8.1 Introduction

This is a blocking component in the **ADB3 OCP** group. Its function is to use its 32-bit interrupt request register input and OCP control registers to control an active low interrupt request output.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.8.2 Interface

The **adb3_ocp_l_ictrl** component interface is shown in Figure 58 below and described in Table 141.

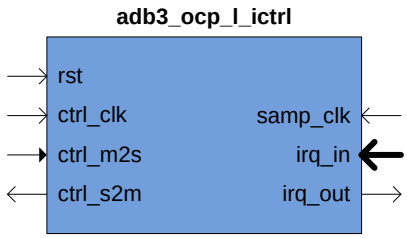

Figure 58 : adb3_ocp_l_ictrl component interface

Signal	Type	Description
width	Generic	Defines the width of the **irq_in** port, i.e. the number of independent interrupt sources (default = 32).
level	Generic	Defines which of the interrupt sources are level-sensitive (default = X "00000000"). A '1' in a particular bit position means the corresponding bit of **irq_in** is level-sensitive.
sample	Generic	For non-level-sensitive bits of **irq_in**, each bit of this vector determines whether a true rising-edge-detection mechanism is used ('0'), or detection of a low-to-high transition in retimed version of **irq_in** ('1') (default = X"FFFFFFFF").

Signal	Type	Description
rst	Input	OCP asynchronous reset.
		OCP Lite Slave Port
ctrl_clk	Input	OCP lite control port clock.
ctrl_m2s	Input	OCP lite control port M2S connection.
ctrl_s2m	Output	OCP lite control port S2M connection.
		Interrupt interface
samp_clk	Input	Sampling clock for low-to-high transition detection.
irq_in	Input	Interrupt request vector (active high).
irq_out	Output	Interrupt request (active high).

Table 141 : adb3_ocp_l_ictrl component interface

6.1.2.8.3 Description

This component encapsulates the logic for capturing events (interrupts) and presenting them via a register interface. Software running on the host CPU can interact with these registers to determine what events have occurred (if any) and dismiss events for which it has taken (or will take) the appropriate action.

There can be up to 32 interrupt sources, input via the **irq_in** port, which can be individually configured to be (a) rising-edge-sensitive, or (b) low-to-high-sensitive or (c) level-sensitive. This is configured via the **LVLCTL** register and the **sample** generic as indicated in Table 142 - "Configuring sensitivity mode in adb3_ocp_l_ictrl" below:

Bit i of LVLCTL	Bit i of sample	Sensitivity mode of bit i of irq_in
0	0	Sensitivity mode 00, rising-edge-sensitive
0	1	Sensitivity mode 01, sensitive to low-to-high transition
1	X	Sensitivity mode 1, level-sensitive

Table 142 : Configuring sensitivity mode in adb3_ocp_l_ictrl

These three modes work as follows:

- **Sensivitity mode 00 (rising edge)**

  When a particular bit of **irq_in** is configured for this mode, it is fed into the clock input of a flip-flop (whose output is visible via the **STAT** register) whose D input is hardwired to 1. The flip-flop is cleared when a '1' is written to the corresponding bit of the **CLEAR** register. This mode is useful when a particular bit of **irq_in** carries pulses of unknown width, but has the disadvantage that synthesis tools will implicitly generate another clock (with a single load) that is routed via general routing. Rigorous design practices may dictate that additional timing constraints be created to prevent this clock being flagged as unconstrained path.

- **Sensivitity mode 01 (sampled low-to-high transition)**

  When a particular bit of **irq_in** is configured for this mode, it is retimed and sampled using **samp_clk**. When a low-to-high transition is detected, a flip-flop (whose output is visible via the **STAT** register) is set to 1. The flip-flop is cleared when a '1' is written to the corresponding bit of the **CLEAR** register. This mode is preferable to sensitivity mode 00 when **irq_in** is synchronous to samp_clk or the pulse width is known to be longer than the period of **samp_clk**, because it avoids the implicit creation of another clock signal. If at least one bit of **irq_in** is configured for this mode, **samp_clk** must be connected to a valid sampling clock, which is permitted to be the same signal as **ctrl_clk**.

- **Sensitivity mode 1 (level)**

  When a particular bit of **irq_in** is configured for this mode, it is retimed and visible in the corresponding bit of the **STAT** register. Such an interrupt cannot be cleared by writing to the corresponding bit of the **CLEAR** register. Instead, software running on the host must take whatever action is necessary to deassert the aforementioned bit of **irq_in**.

The default values of the generics **level** and **sample** and default value of the the **LVLCTL** register mean that all bits of **irq_in** function in sensitivity mode 01 by default, and the **samp_clk** port must be mapped to the clock used to sample **irq_in**. On the other hand, if the generics or the **LVLCTL** register are overridden such that none of the bits of **irq_in** operate in sensitivity mode 01, the **samp_clk** input may be wired to an arbitrary constant logic level.

The **irq_out** signal is asserted whenever there is at least one active and enabled interrupt. In other words, if at least one bit of the **STAT** register is 1 and the corresponding bit of the **ENABLE** register is 1, **irq_out** is asserted.

Note that disabling interrupt sources via **ENABLE** has no effect on whether or not they are reported via the **STAT** register; this permits software to operate in a polled mode, where all interrupts are disabled and software

periodically reads the **STAT** register to determine if any action need be taken.

Writing a '1' to a particular bit of the **CLEAR** register will clear the corresponding bit in the **STAT** register if it is configured for sensitivity mode 00 or 01. By constrast, a level-sensitive interrupt (sensitivity mode 1) can only be cleared by taking some action that causes the appropriate bit of **irq_in** to be deasserted.

A side-effect of writing to the **CLEAR** register is that it forces the **irq_out** signal to be deasserted for a few clock cycles. This is known as "rearming" the **irq_out** signal, and avoids a race condition that may otherwise occur if the **irq_out** signal itself is treated as being edge-sensitive by whatever it is connected to, and multiple interrupt sources become active at around the same time.

6.1.2.8.3.1 Register Description

The OCP lite slave port is interfaced to a a set of registers to control interrupt generation on the host. These registers appear at address offsets as follows:

Name	Address
ENABLE	0x80
CLEAR	0x84
COUNT	0x88
STAT	0x8C
LVLCTL	0x90

Table 143 : adb3_ocp_l_ictrl Register Address Map

The register address space used by this component is defined by the **REG_ADDR_WIDTH** constant. This is set to 8 giving a decoded address region of 256 bytes. The register offsets in this space are defined using constants, for example **REG_ENABLE_ADDR** for the ENABLE register.

Once reset is complete, the OCP lite **ctrl_s2m.CmdAccept** input is permanently set to 1.

Any write command on **ctrl_m2s.Cmd** has its address on **ctrl_m2s.Addr** compared to the register address constants. If there is a match, then the **ctrl_m2s.Data** is written to the register using the **adb3_ocp_l_wr** helper procedure defined in ADB3 OCP Profile Definition Package (adb3_ocp). This procedure updates a register signal using **ctrl_m2s.Data**, **ctrl_m2s.DataByteEn**, and a data offset value.

Any read command on **ctrl_m2s.Cmd** has its address on **ctrl_m2s.Addr** compared to the register address constants. If there is a match, then **ctrl_m2s.Resp** is set to valid, and the register value is returned on **ctrl_s2m.Data**. If there is no match, then X"BAADC0DE" is returned on **ctrl_s2m.Data**

Bits	Mnemonic	Type	Function
31:0	ENABLE	M	Controls/indicates the masking (0) or enabling (1) of individual bits in the STAT register. When a bit is 1, the corresponding bit in the STAT register is enabled (i.e. allowed to assert the interrupt output). This register has a default value of X "00000000".

Table 144 : adb3_ocp_l_ictrl, ENABLE Register (0x000080)

Bits	Mnemonic	Type	Function
31:0	CLEAR	WO	Writing a 1 to a particular bit clears the corresponding bit in the STAT register.

Table 145 : adb3_ocp_l_ictrl, CLEAR Register (0x000084)

Bits	Mnemonic	Type	Function
31:0	COUNT	RW	Write: if the **irq_out** signal is asserted on a given clock cycle, the COUNT register is incremented. It is possible to write to this register at any time, which can be used to initialize or reset it (even when incrementing). This register has a default value of X "00000000".

Table 146 : adb3_ocp_l_ictrl, COUNT Register (0x000088)

Bits	Mnemonic	Type	Function
31:0	STAT	RO	Returns the current value of the STAT register.

Table 147 : adb3_ocp_l_ictrl, STAT Register (0x00008C)

Bits	Mnemonic	Type	Function
31:0	LEVEL	M	Determines whether a particular bit of **irq_in** is rising-edge-sensitive or level-sensitive. When a bit is 1, the corresponding bit of **irq_in** is level-sensitive; otherwise it is rising-edge-sensitive or sampled low-to-high-sensitive depending on the **sample** generic. This register defaults to the value of the **level** generic.

Table 148 : adb3_ocp_l_ictrl, LVLCTL Register (0x000090)

6.1.2.9 adb3_ocp_cross_clk_dom

6.1.2.9.1 Introduction

This is a non-blocking component in the **ADB3 OCP** group. Its function is to connect a single primary ADB3 OCP channel in the primary clock domain to a single secondary ADB3 OCP channel in the secondary clock domain.

Dependencies

- The command path is independent from the write data path. Data acceptance does not block command acceptance.
- The command path is independent from the read response path. Response acceptance does not block command acceptance.
- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.9.2 Interface

The **adb3_ocp_cross_clk_dom** component interface is shown in Figure 59 below and described in Table 149.

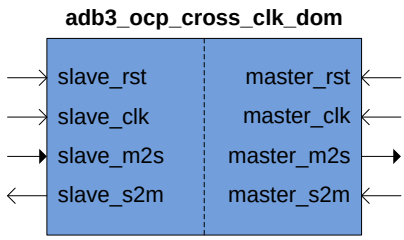

Figure 59 : adb3_ocp_cross_clk_dom component interface

Signal	Type	Description
		OCP Primary Port
slave_rst	Input	OCP Primary (slave) port asynchronous reset.
slave_clk	Input	OCP Primary (slave) port clock.
slave_m2s	Input	OCP Primary (slave) port M2S connection.
slave_s2m	Output	OCP Primary (slave) port S2M connection.
		OCP Secondary Port
master_rst	Input	OCP Secondary (master) port asynchronous reset.
master_clk	Input	OCP Secondary (master) port clock.
master_m2s	Output	OCP Secondary (master) port M2S connection.
master_s2m	Input	OCP Secondary (master) port S2M connection.

Table 149 : adb3_ocp_cross_clk_dom component interface

6.1.2.9.3 Description

The **adb3_ocp_cross_clk_dom** component block diagram is shown in Figure 60 below.

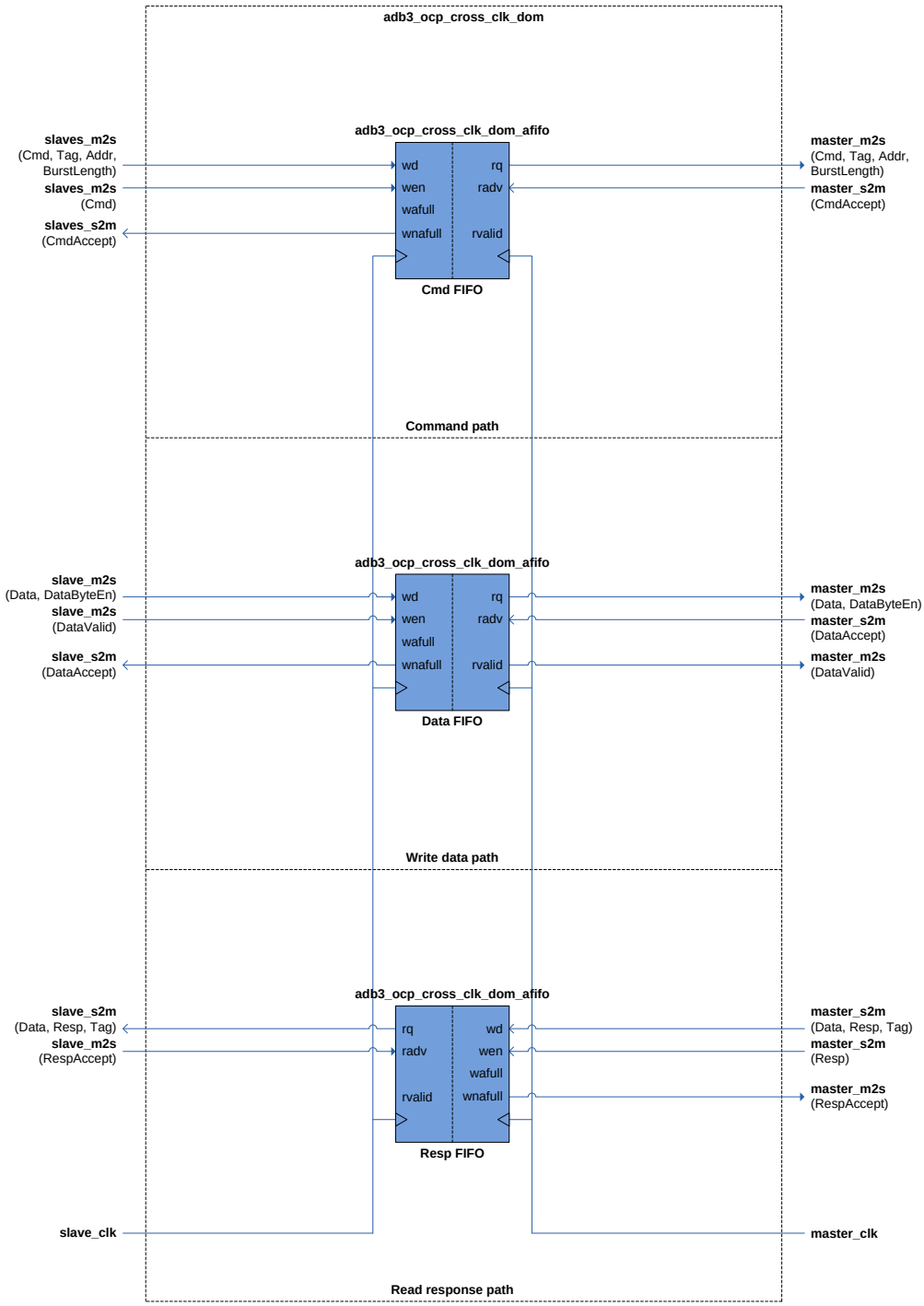

Figure 60 : adb3_ocp_cross_clk_dom block diagram

The component consists of three instances of the Asychronous FIFO block **adb3_ocp_cross_clk_dom_afifo**. One for command signals, one for data signals, and the third for response signals as follows:

6.1.2.9.3.1 Command Path

This consists of the **Cmd**, **Tag**, **BurstLength**, and **Addr** elements of the **slave_m2s**/**master_m2s** signals, and the **CmdAccept** element of the **slave_s2m**/**master_s2m** signals.

Command FIFO

- The **slave_m2s** port command elements are interfaced to the **master_m2s** port command elements via the command FIFO.
- The **slave_s2m** port **CmdAccept** element is generated from the command FIFO full flag.
- The command FIFO write advance is generated from the **slave_m2s** port **Cmd** element and the command FIFO full flag.
- The command FIFO read advance is generated from the **master_s2m** port **CmdAccept** element and the command FIFO valid flag.

6.1.2.9.3.2 Write Data Path

This consists of the **DataValid**, **DataByteEn**, and **Data** elements of the **slave_m2s**/**master_m2s** signals, and the **DataAccept** element of the **slave_s2m**/**master_s2m** signals.

Write Data FIFO

- **slave_m2s** port write data elements are interfaced to the **master_m2s** port write data elements via the write data FIFO.
- The **slave_s2m** port **DataAccept** element is generated from the data FIFO full flag.
- The write data FIFO write advance is generated from the **slave_m2s** port **DataValid** element and the write data FIFO full flag.
- The write data FIFO read advance is generated from the **master_s2m** port **DataAccept** element and the write data FIFO valid flag.

6.1.2.9.3.3 Read Response Path

This consists of the **Resp**, **Tag**, and **Data** elements of the **master_s2m**/**slave_s2m** signals, and the **RespAccept** element of the **master_m2s**/**slave_m2s** signals.

Read Response FIFO

- **master_s2m** port read response elements are interfaced to the **slave_s2m** port read response elements via the read response FIFO.
- The **master_m2s** port **RespAccept** element is generated from the read response FIFO full flag.
- The read response FIFO write advance is generated from the **master_s2m** port **Resp** element and the read response FIFO full flag.
- The read response FIFO read advance is generated from the **slave_m2s** port **RespAccept** element and the read response FIFO valid flag.

6.1.2.10 adb3_ocp_mux_nb

6.1.2.10.1 Introduction

This is a non-blocking component in the **ADB3 OCP** group. Its function is to multiplex multiple primary ADB3 OCP channels onto a single secondary ADB3 OCP channel. The multiplex is controlled by round-robin arbitration of OCP commands.

Dependencies

- The command path is independent from the write data path. Data acceptance does not block command acceptance.
- The command path is independent from the read response path. Response acceptance does not block command acceptance.
- Transactions on multiple primary ADB3 OCP channels may be accepted simultaneously.
- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.10.2 Interface

The **adb3_ocp_mux_nb** component interface is shown in Figure 61 below and described in Table 150.

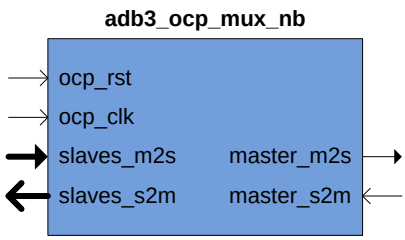

Figure 61 : adb3_ocp_mux_nb component interface

Signal	Type	Description
mux_inputs	Generic	Number of primary OCP channels to be multiplexed.

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
		OCP Primary Ports
slaves_m2s	Input	OCP Primary (slave) ports M2S connection.
slaves_s2m	Output	OCP Primary (slave) ports S2M connection.
		OCP Secondary Port
master_m2s	Output	OCP Secondary (master) port M2S connection.
master_s2m	Input	OCP Secondary (master) port S2M connection.

Table 150 : adb3_ocp_mux_nb component interface

6.1.2.10.3 Description

The **adb3_ocp_mux_nb** component block diagram is shown in Figure 62 below.

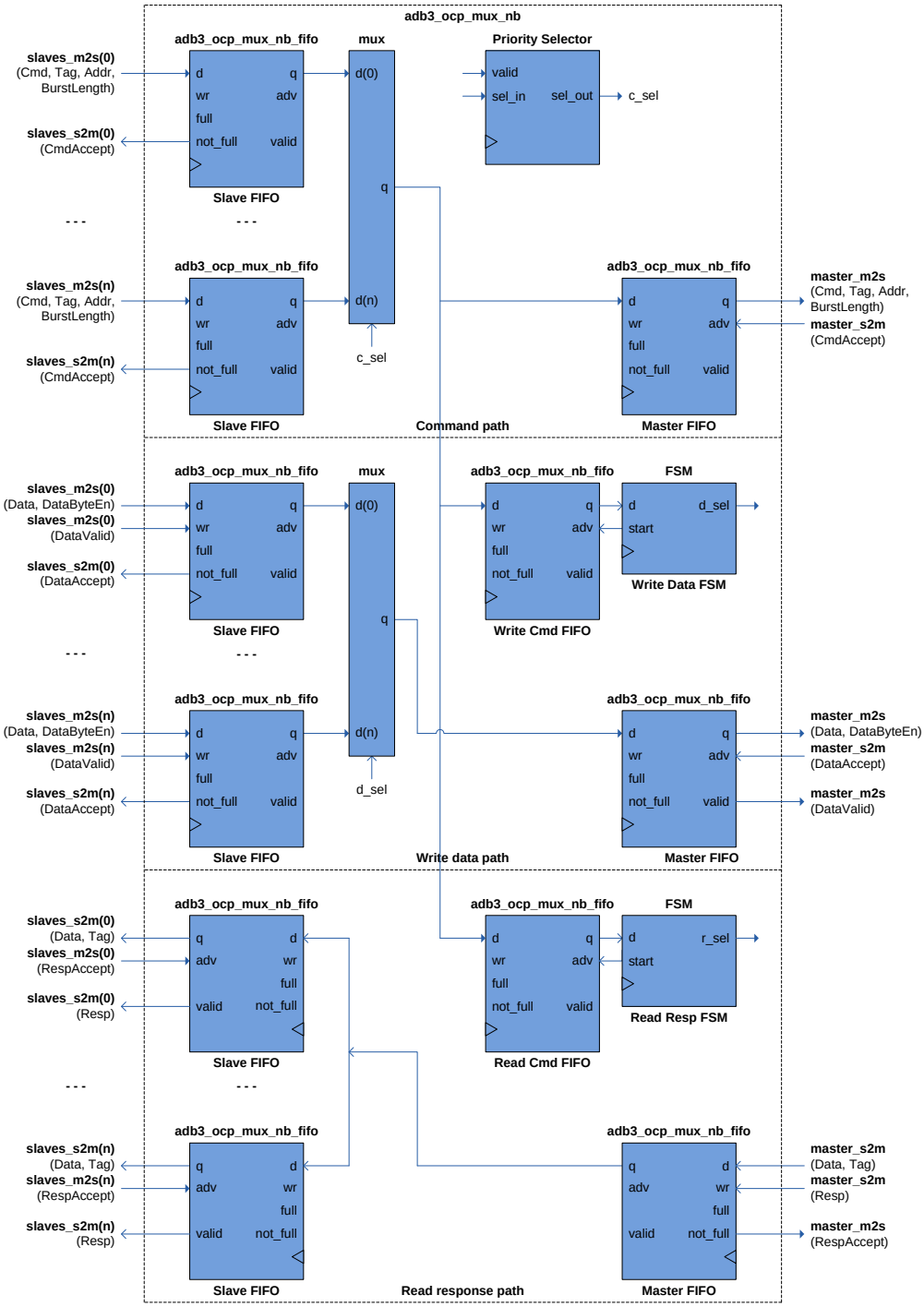

Figure 62 : adb3_ocp_mux_nb block diagram

6.1.2.10.3.1 Command Path

This consists of the **Cmd**, **Tag**, **BurstLength**, and **Addr** elements of the **slaves_m2s/master_m2s** signals, and the **CmdAccept** element of the **slaves_s2m/master_s2m** signals.

Slave Command FIFOs

- The **slaves_m2s** ports command elements are interfaced to the slave command mux inputs via the slave command FIFOs.
- The **slaves_s2m** ports **CmdAccept** elements are generated from the slave command FIFOs not full flags.
- The slave command FIFOs write advances are generated from the **slaves_m2s** ports **Cmd** elements and the slave command FIFOs not full flags.
- The slave command FIFOs read advances are generated from the slave command select and the master, write, and read command FIFO not full flags.

Priority Selector

- Priority is assigned on a round-robin basis.
- The slave command select is generated from the highest priority non-empty slave command FIFO.

Slave Command Mux

- The slave command mux select is generated by the priority selector.
- The slave command mux routes the selected slave command FIFO to the master command FIFO.

Master Command FIFO

- The slave command mux output is interfaced to the **master_m2s** port command elements via the master command FIFO.
- The master command FIFO write advance is generated from the slave command select and the master, write, and read command FIFO not full flags.
- The master command FIFO read advance is generated from the **master_s2m** port **CmdAccept** element and the master command FIFO not empty flag.

Write Command FIFO

- The slave command select and slave command FIFO output **BurstLength** element are interfaced to the write data FSM via the write command FIFO.
- The write command FIFO write advance is generated from the master command FIFO write advance and master command FIFO **Cmd** element.
- The write command FIFO read advance is generated from the write data FSM.

Read Command FIFO

- The slave command select and slave command FIFO output **BurstLength** element are interfaced to the read data FSM via the read command FIFO.
- The read command FIFO write advance is generated from the master command FIFO write advance and master command FIFO **Cmd** element.
- The read command FIFO read advance is generated from the read data FSM.

6.1.2.10.3.2 Write Data Path

This consists of the **DataValid**, **DataByteEn**, and **Data** elements of the **slaves_m2s/master_m2s** signals, and the **DataAccept** element of the **slaves_s2m/master_s2m** signals.

Slave Write Data FIFOs

- The **slaves_m2s** ports write data elements are interfaced to the slave write data mux inputs via the slave write data FIFOs.

- The **slaves_s2m** ports **DataAccept** elements are generated from the slave write data FIFOs not full flags.
- The slave write data FIFOs write advances are generated from the **slaves_m2s** ports **DataValid** elements and the slave write data FIFOs not full flags.
- The slave write data FIFOs read advances are generated from the write data select, the slave write data FIFOs not empty flags, and the master write data FIFO not full flag.

Slave Write Data Mux

- The slave write data mux select is generated by the write data FSM.
- The slave write data mux routes the selected slave write data FIFO to the master write data FIFO.

Master Write Data FIFO

- The slave write data mux output is interfaced to the **master_m2s** port write data elements via the master write data FIFO.
- The master write data FIFO write advance is generated from the write data select, the slave write data FIFO not empty flags, and the master write data FIFO not full flag.
- The master write data FIFO read advance is generated from the **master_s2m** port **DataAccept** element and the master write data FIFO not empty flag.

Write Data FSM

- Counts write data bursts for current entry in the write command FIFO.
- The write data select is generated from the FSM state and write command FIFO output.
- The write command FIFO read advance is generated from the FSM state.

6.1.2.10.3.3 Read Response Path

This consists of the **Resp**, **Tag**, and **Data** elements of the **master_s2m**/**slaves_s2m** signals, and the **RespAccept** element of the **master_m2s**/**slaves_m2s** signals.

Master Read Response FIFO

- The **master_s2m** port read response elements are interfaced to the slave read response FIFOs via the master read response FIFO.
- The master read response FIFO write advance is generated from the **master_s2m** port **Resp** element and the master read response FIFO not full flag.
- The master read response FIFO read advance is generated from the read response select, slave read response FIFOs not full flags, and the master read response FIFO not empty flag.

Slave Read Response FIFOs

- The master read response FIFO is interfaced to the **slaves_s2m** ports read response elements via the slave read response FIFOs.
- The slave read response FIFOs write advances are generated from the read response select, the slave read response FIFOs not full flags, and the master read response FIFO not full flag.
- The slave read response FIFOs read advances are generated from the **slaves_m2s** ports **RespAccept** elements and the slave read response FIFOs not empty flags.

Read Response FSM

- Counts read response bursts for current entry in the read command FIFO.
- The read response select is generated from the FSM state and read command FIFO output.
- The read command FIFO read advance is generated from the FSM state.

6.1.2.11 adb3_ocp_split_nb

6.1.2.11.1 Introduction

This is a non-blocking component in the **ADB3 OCP** group. Its function is to de-multiplex a single primary ADB3 OCP channel into multiple secondary ADB3 OCP channels. Selection of the active secondary channel is controlled by the primary channel command address.

Dependencies

- The command path is independent from the write data path. Data acceptance does not block command acceptance.
- The command path is independent from the read response path. Response acceptance does not block command acceptance.
- Transactions on multiple secondary ADB3 OCP channels may be initiated simultaneously.
- Transaction response order always matches transaction command acceptance order.
- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.11.2 Interface

The **adb3_ocp_split_nb** component interface is shown in Figure 63 below and described in Table 151.

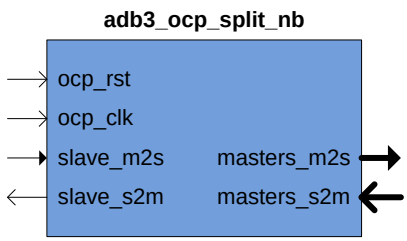

Figure 63 : adb3_ocp_split_nb component interface

Signal	Type	Description
addr_range_table	Generic	Table defining the address ranges to be used to control the split operation.
error_data	Generic	OCP Response Data to be returned if address is out of range (default = "DEADC0DEDEADC0DEDEADC0DEDEADC0DE").
full_addr_out	Generic	Select full or partial OCP address output (default = true).
unused_addr_bit	Generic	Select value of unused OCP address output bits (default = '0').

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP port clock.
		OCP Primary Port
slave_m2s	Input	OCP Primary (slave) port M2S connection.

Table 151 : adb3_ocp_split_nb component interface (continued on next page)

Signal	Type	Description
slave_s2m	Output	OCP Primary (slave) port S2M connection.
		OCP Secondary Ports
masters_m2s	Output	OCP Secondary (master) ports M2S connection.
masters_s2m	Input	OCP Secondary (master) ports S2M connection.

Table 151 : adb3_ocp_split_nb component interface

6.1.2.11.3 Description

The **adb3_ocp_split_nb** component block diagram is shown in Figure 64 below.

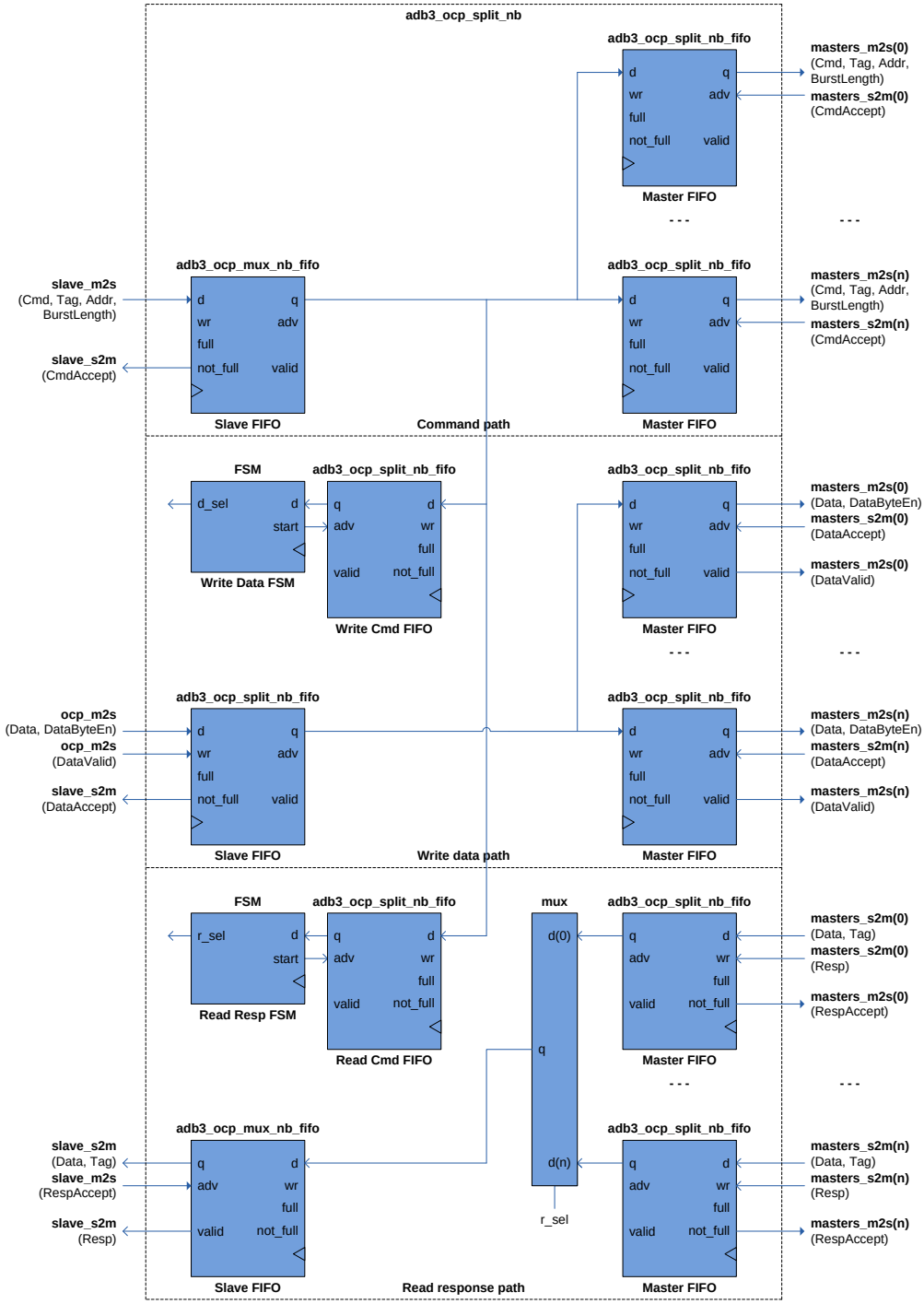

Figure 64 : adb3_ocp_split_nb block diagram

6.1.2.11.3.1 Command Path

This consists of the **Cmd**, **Tag**, **BurstLength**, and **Addr** elements of the **slaves_m2s**/**master_m2s** signals, and the **CmdAccept** element of the **slaves_s2m**/**master_s2m** signals.

Slave Command FIFO

- The **slave_m2s** port command elements are interfaced to the master command FIFOs via the slave command FIFO.
- The **slave_s2m** port **CmdAccept** element is generated from the slave command FIFO not full flag.
- The slave command FIFO write advance is generated from the **slave_m2s** port **Cmd** element and the slave command FIFO not full flag.
- The slave command FIFO read advance is generated from the slave command FIFO not empty, slave command select, and the master, write, and read command FIFO not full flags.

Address Range Table

- This consists of a number of **addr_mask**, **addr_base** pairs. Each pair specifies the active address range for its associated **masters_m2s**/**masters_s2m** channel.
- The **addr_mask** vector is the address range active bit mask (active low). That is, bits that are '0' will enable, bits that are '1' will mask.
- The **addr_base** vector is the base address of the range.
- For example, **addr_base** = 0x00000400, **addr_mask** = 0xFFC003FF, specifies an address range 0x000400 to 0x000007FF with address bits (21:10) used for address range decoding.

Address Range Selector

- Selection is made using an (n+1) bit 1-hot vector **sec_decode_1h(n:0)**, where n is the number of entries in the **addr_range_table**.
- The slave command FIFO **Addr** element is compared with the address ranges specified by the **addr_range_table** generic. If the address is in a valid range, then the appropriate bit of the **sec_decode_1h** signal is set. If the address is not in any range, the top bit of the **sec_decode_1h** signal is set.

Master Command FIFOs

- The slave command FIFO is interfaced to the **masters_m2s** ports command elements via the master command FIFOs.
- The master command FIFOs write advances are generated from the slave command FIFO not empty, slave command select, and the master, write, and read command FIFO not full flags.
- The master command FIFOs read advances are generated from the **master_s2m** port **CmdAccept** element and the master command FIFOs not empty flags.
- The OCP address output on the active **masters_m2s.Addr** signal is controlled by the **full_addr_out** and **unused_addr_bit** generics. If **full_addr_out** is true, then the full OCP address is used. If **full_addr_out** is false, then only the addresss bits within the range are used, with the higher bits set to **unused_addr_bit**.

Write Command FIFO

- The slave command select and slave command FIFO output **BurstLength** element are interfaced to the write data FSM via the write command FIFO.
- The write command FIFO write advance is generated from the slave command FIFO write advance and slave command FIFO **Cmd** element.
- The write command FIFO read advance is generated from the write data FSM.

Read Command FIFO

- The slave command select and slave command FIFO output **BurstLength** and **Tag** elements are

interfaced to the read data FSM via the read command FIFO.

- The read command FIFO write advance is generated from the slave command FIFO write advance and slave command FIFO **Cmd** element.
- The read command FIFO read advance is generated from the read data FSM.

6.1.2.11.3.2 Write Data Path

This consists of the **DataValid**, **DataByteEn**, and **Data** elements of the **slaves_m2s**/**master_m2s** signals, and the **DataAccept** element of the **slaves_s2m**/**master_s2m** signals.

Slave Write Data FIFO

- The **slave_m2s** port write data elements are interfaced to the master write data FIFOs via the slave write data FIFO.
- The **slave_s2m** port **DataAccept** element is generated from the slave write data FIFO not full flag.
- The slave write data FIFO write advance is generated from the **slave_m2s** port **DataValid** element and the slave write data FIFO not full flag.
- The slave write data FIFO read advance is generated from the write data select, the slave write data FIFO not empty flag, and the master write data FIFOs not full flags.

Master Write Data FIFOs

- The slave write data FIFO is interfaced to the **masters_m2s** ports write data elements via the master write data FIFOs.
- The master write data FIFOs write advances are generated from the write data select, the slave write data FIFO not empty flag, and the master write data FIFOs not full flags.
- The master write data FIFOs read advances are generated from the **masters_s2m** ports **DataAccept** elements and the master write data FIFOs not empty flags.

Write Data FSM

- Counts write data bursts for current entry in the write command FIFO.
- The write data select is generated from the FSM state and write command FIFO output.
- The write command FIFO read advance is generated from the FSM state.

6.1.2.11.3.3 Read Response Path

This consists of the **Resp**, **Tag**, and **Data** elements of the **master_s2m**/**slaves_s2m** signals, and the **RespAccept** element of the **master_m2s**/**slaves_m2s** signals.

Master Read Response FIFOs

- The **masters_s2m** ports read response elements are interfaced to the slave read response mux inputs via the master read response FIFOs.
- The **masters_s2m** ports **CmdAccept** elements are generated from the master read response FIFOs not full flags.
- The master read response FIFOs write advances are generated from the **masters_s2m** ports **Resp** elements and the master read response FIFOs not full flags.
- The master read response FIFOs read advances are generated from the read response select, slave read response FIFO not full flag, and the master read response FIFOs not empty flags.

Master Read Response Mux

- The master read response mux select is generated from the master read response select.
- The master read response mux routes the selected master read response FIFO to the slave read response FIFO.

Slave Read Response FIFO

- The master read response mux is interfaced to the **slave_s2m** port read response elements via the slave read response FIFO.
- The slave read response FIFO write advance is generated from the read response select, the slave read response FIFO not full flag, and the master read response FIFOs not empty flags.
- The slave read response FIFO read advance is generated from the **slave_m2s** port **RespAccept** element and the slave read response FIFO not empty flag.

Read Response FSM

- Counts read response bursts for current entry in the read command FIFO.
- The read response select is generated from the FSM state and read command FIFO output.
- The read command FIFO read advance is generated from the FSM state.

6.1.2.12 adb3_ocp_ocp2ddr3_nb

6.1.2.12.1 Introduction

This is a non-blocking component in the **ADB3 OCP** group. Its function is to interface a single ADB3 OCP channel to the Xilinx DDR3 SDRAM MIG core user interface.

Dependencies

- The command path is independent from the write data path. Data acceptance does not block command acceptance.
- The command path is independent from the read response path. Response acceptance does not block command acceptance.
- Transaction response order always matches transaction command acceptance order.
- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.12.2 Interface

The **adb3_ocp_ocp2ddr3_nb** component interface is shown in Figure 65 below and described in Table 152.

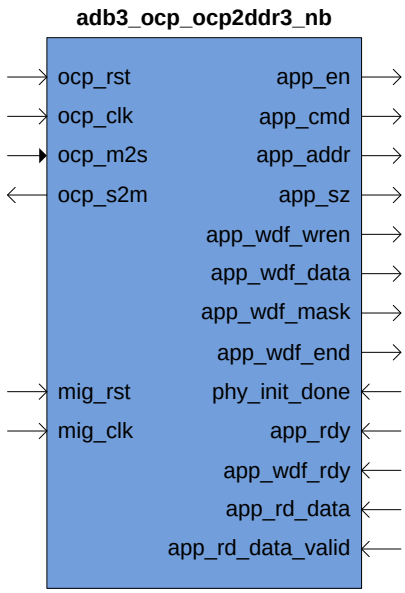

Figure 65 : adb3_ocp_ocp2ddr3_nb component interface

Signal	Type	Description
app_row_width	Generic	Width of the row part of the app_addr output.
app_col_width	Generic	Width of the col part of the app_addr output.
app_bank_width	Generic	Width of the bank part of the app_addr output.
app_addr_width	Generic	Width of the app_addr output (aligned to bank data width).
app_dqs_cnt_width	Generic	Log2 of DDR3 bank DQ bytes (default is log2(4) = 2).

Signal	Type	Description
		OCP Interface
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP M2S connection.
ocp_s2m	Output	OCP S2M connection.
		DDR3 SDRAM MIG Core User Interface
mig_rst	Input	User interface reset.
mig_clk	Input	User interface clock.
phy_init_done	Input	User interface phy calibration complete.
app_rdy	Input	User interface command ready.
app_wdf_rdy	Input	User interface write data ready.
app_rd_data	Input	User interface read command data.
app_rd_data_valid	Input	User interface read command data valid.
app_en	Output	User interface command enable.
app_cmd	Output	User interface command.
app_addr	Output	User interface command address (aligned to bank data width).
app_sz	Output	User interface command 1/2 cycle select.
app_wdf_wren	Output	User interface write command data enable.
app_wdf_data	Output	User interface write command data.
app_wdf_mask	Output	User interface write command data mask (active low).
app_wdf_end	Output	User interface write command data end.

Table 152 : adb3_ocp_ocp2ddr3_nb component interface

6.1.2.12.3 Description

The **adb3_ocp_ocp2ddr3_nb** component block diagram is shown in Figure 66 below.

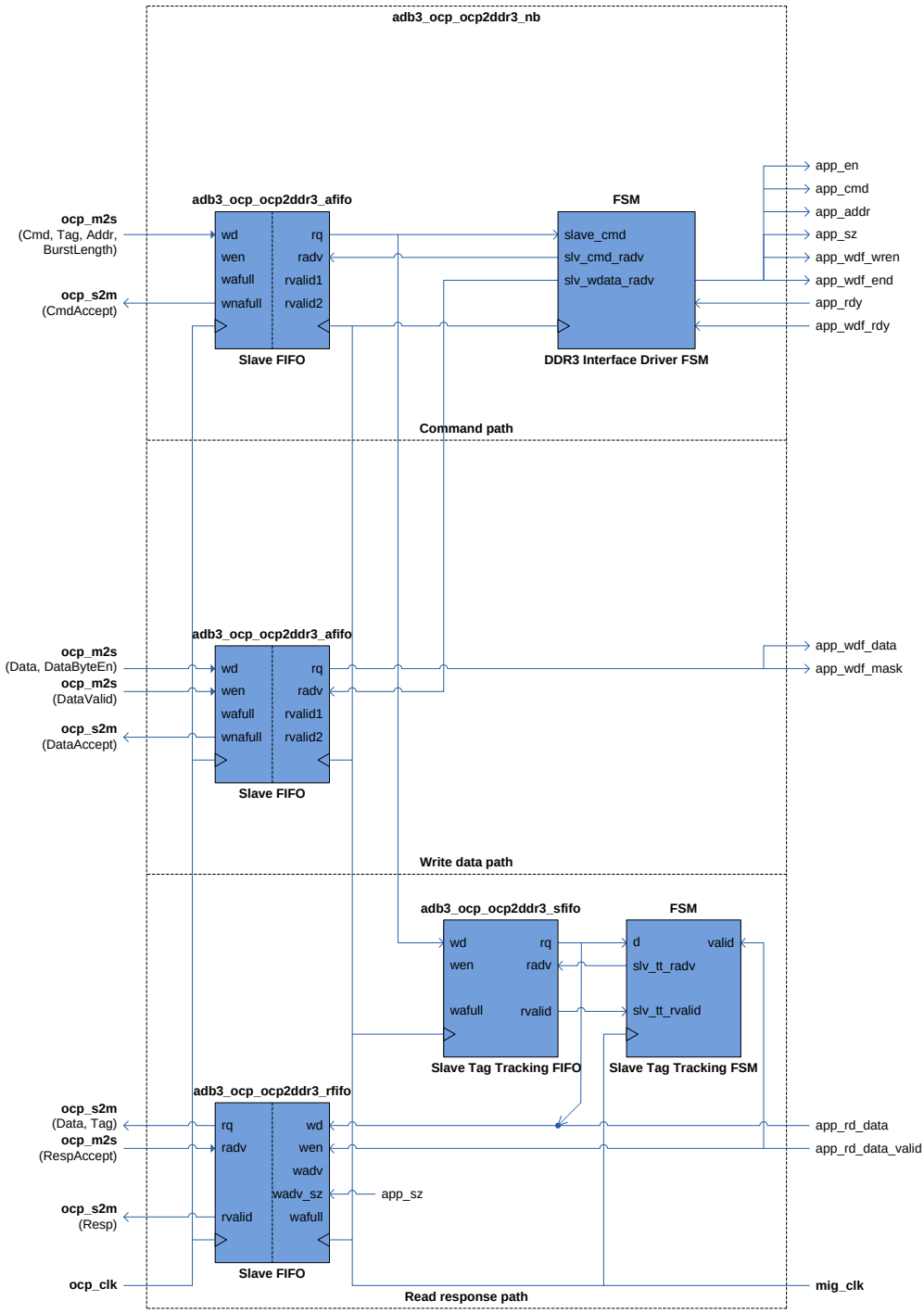

Figure 66 : adb3_ocp_ocp2ddr3_nb block diagram

6.1.2.12.3.1 Command Path

This consists of the **Cmd**, **Tag**, **BurstLength**, and **Addr** elements of the **ocp_m2s** signal, and the **app_rdy**, **app_en**, **app_cmd**, **app_addr**, and **app_sz** MIG core user interface signals.

Slave Command FIFO

- The **ocp_m2s** port command elements are interfaced to the DDR3 interface driver FSM via the slave command FIFO.
- This is an instance of the **adb3_ocp_ocp2ddr3_afifo** asynchronous FIFO. The write side is in the **ocp_clk** clock domain. The read side is in the **mig_clk** domain.
- The **ocp_s2m** port **CmdAccept** element is generated from the slave command FIFO not full flag **slv_cmd_wnafull**.
- The slave command FIFO write enable is generated from the **ocp_m2s** port **Cmd** element and the slave command FIFO not full flag **slv_cmd_wnafull**.
- The slave command FIFO read advance **slv_cmd_radv** is generated from the DDR3 interface driver FSM.

DDR3 Interface Driver FSM

- This FSM operates in the **mig_clk** domain.
- Slave command FIFO data is converted into MIG core user interface commands.
- The FSM output **slv_cmd_radv** is used to generate the slave command FIFO read advance.
- The FSM output **slv_wdata_radv** is used to generate the slave write data FIFO read advance.
- The **app_addr** output to the MIG DDR3 SDRAM interface is produced by reordering the address component of the slave command FIFO read data from logical (Row/Bank/Col) to MIG (Bank/Row/Col). This ensures that the MIG core is compatible with DDR3 SDRAM devices with differing Row sizes.
- The **app_addr** output to the MIG DDR3 SDRAM interface is n-byte aligned, where n is the width in bytes of the DDR3 bank DQ data bus. Byte aligned OCP address are converted using the **app_dqs_cnt_width** generic.

Slave Tag Tracking FIFO

- The slave command FIFO **slv_cmd** output fields **BurstLength** and **Tag** are written to the slave tag tracking FIFO using the **slv_tt_wen** signal. This is active
- This is an instance of the **adb3_ocp_ocp2ddr3_sfifo** synchronous FIFO. It operates in the **mig_clk** domain.
- The slave tag tracking FIFO output **slv_tt_tag** is used as the tag value for OCP read responses written into the slave read data FIFO.
- The slave tag tracking FIFO outputs **slv_tt_blen** and **slv_tt_rvalid** are used by the slave tag tracking FSM.
- The slave tag tracking FIFO read advance **slv_tt_radv** is generated by slave tag tracking FSM.

Slave Tag Tracking FSM

- This FSM operates in the **mig_clk** domain.
- The FSM uses the **slv_tt_blen**, **slv_tt_rvalid** and **mig_app_rd_data_valid** signals to generate the slave tag tracking FIFO read advance **slv_tt_radv**.

6.1.2.12.3.2 Write Data Path

This consists of the **DataValid**, **DataByteEn**, and **Data** elements of the **ocp_m2s** signal, and the **app_wdf_rdy**, **app_wdf_wren**, **app_wdf_data**, **app_wdf_mask**, and **app_wdf_end** MIG core user interface signals.

Slave Write Data FIFO

- The **ocp_m2s** port write data elements are interfaced to the MIG core user interface write data signals via the slave write data FIFO.
- This is an instance of the **adb3_ocp_ocp2ddr3_afifo** asynchronous FIFO. The write side is in the **ocp_clk** clock domain. The read side is in the **mig_clk** domain.
- The **ocp_s2m** port **DataAccept** element is generated from the slave write data FIFO not full flag **slv_wdata_wnafull**.
- The slave write data FIFO write advance is generated from the **ocp_m2s** port **DataValid** element and the slave write data FIFO not full flag **slv_wdata_wnafull**.
- The slave write data FIFO read advance **slv_wdata_radv** is generated from the DDR3 interface driver FSM.

6.1.2.12.3.3 Read Data Path

This consists of the **Resp**, **Tag**, and **Data** elements of the **ocp_s2m** signal, and the **app_rd_data**, and **app_rd_data_valid** MIG core user interface signals.

Slave Read Data FIFO

- The MIG core user interface read data signals are interfaced to the **ocp_s2m** port read response elements via the slave read response FIFO.
- This is an instance of the **adb3_ocp_ocp2ddr3_rfifo** asynchronous FIFO. The write side is in the **mig_clk** clock domain. The read side is in the **ocp_clk** domain.
- The **ocp_s2m** port **Resp** element is generated from the slave read data FIFO valid flag **slv_rdata_rvalid**.
- The slave read data FIFO write enable is generated from the MIG core user interface read data signal **app_rd_data_valid**. Valid data must always be accepted.
- The slave read data FIFO accounting write pointer advance inputs (**wadv**, **wadv_sz**) are active when a read command is output on the MIG core user interface. They are generated using the **mig_app_en**, **mig_app_cmd**, and **mig_app_sz** signals. They are used to reserve space in the slave read data FIFO for the data which results from read commands issued to the MIG core user interface.
- The slave read data FIFO read advance **slv_rdata_radv** is generated from the slave read data FIFO valid flag **slv_rdata_rvalid**, and the **ocp_m2s** port **RespAccept** element.

6.1.2.13 adb3_ocp_retime_nb

6.1.2.13.1 Introduction

This is a non-blocking component in the **ADB3 OCP** group. Its function is to re-time a single primary ADB3 OCP channel, producing a single secondary ADB3 OCP channel. This will improve system timing at the expense of latency.

Dependencies

- The command path is independent from the write data path. Data acceptance does not block command acceptance.
- The command path is independent from the read response path. Response acceptance does not block command acceptance.
- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.13.2 Interface

The **adb3_ocp_retime_nb** component interface is shown in Figure 67 below and described in Table 153.

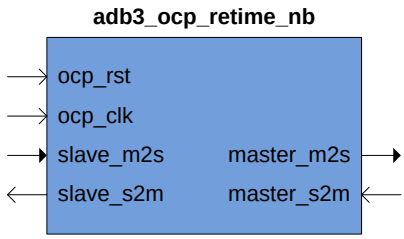

Figure 67 : adb3_ocp_retime_nb component interface

Signal	Type	Description
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
		OCP Primary Port
slave_m2s	Input	OCP Primary (slave) port M2S connection.
slave_s2m	Output	OCP Primary (slave) port S2M connection.
		OCP Secondary Port
master_m2s	Output	OCP Secondary (master) port M2S connection.
master_s2m	Input	OCP Secondary (master) port S2M connection.

Table 153 : adb3_ocp_retime_nb component interface

6.1.2.13.3 Description

The **adb3_ocp_retime_nb** component block diagram is shown in Figure 68 below.

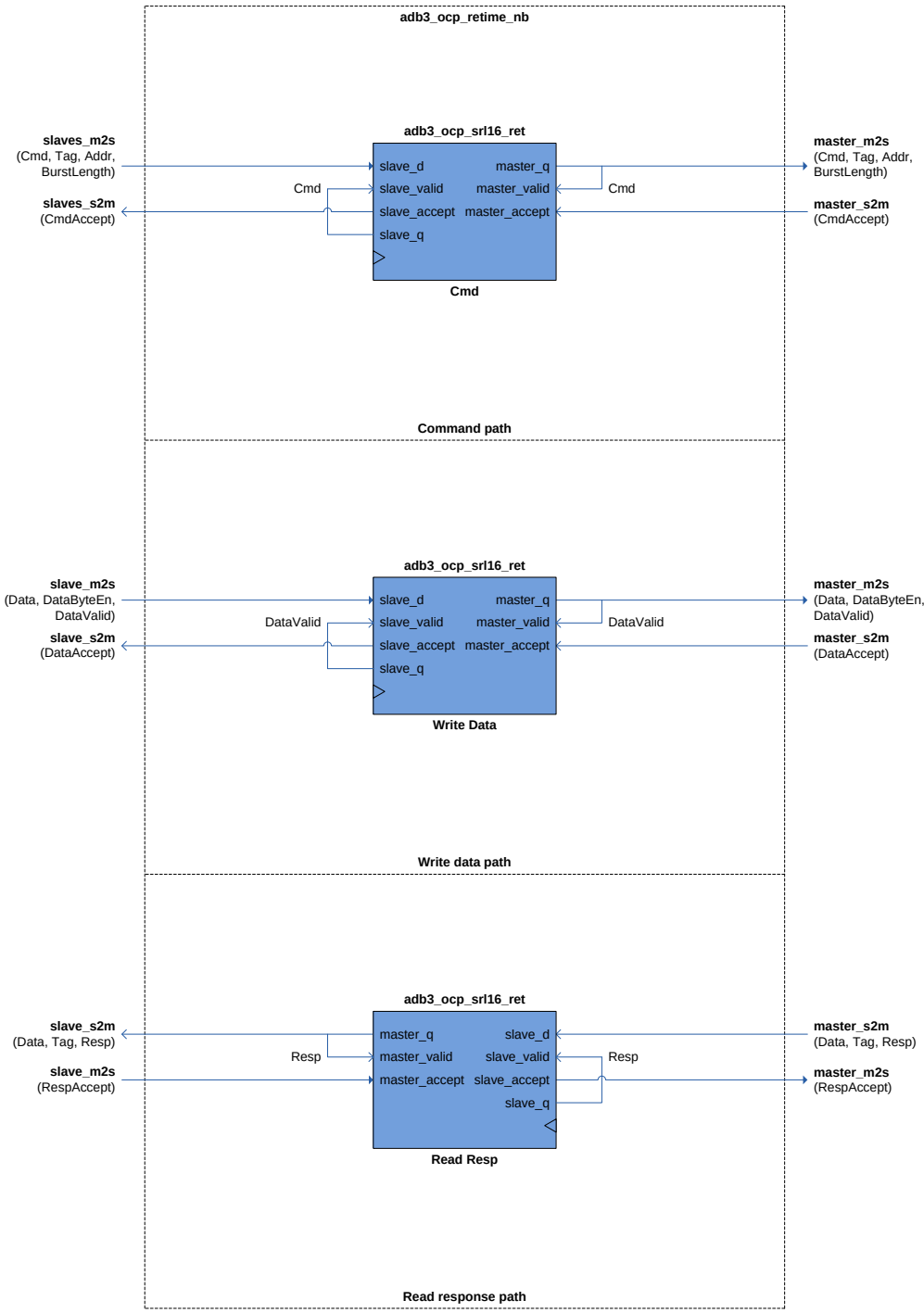

Figure 68 : adb3_ocp_retime_nb block diagram

6.1.2.13.3.1 Command Path

This consists of the **Cmd**, **Tag**, **BurstLength**, and **Addr** elements of the **slave_m2s/master_m2s** signals, and the **CmdAccept** element of the **slave_s2m/master_s2m** signals.

- The **slave_m2s** port command elements are interfaced to the **master_m2s** port command elements via the command **adb3_ocp_srl16_ret** component.
- The command **adb3_ocp_srl16_ret** slave valid is generated from the **slave_q** port **Cmd** element.
- The command **adb3_ocp_srl16_ret** master valid is generated from the **master_m2s** port **Cmd** element.

6.1.2.13.3.2 Write Data Path

This consists of the **DataValid**, **DataByteEn**, and **Data** elements of the **slave_m2s/master_m2s** signals, and the **DataAccept** element of the **slave_s2m/master_s2m** signals.

- **slave_m2s** port write data elements are interfaced to the **master_m2s** port write data elements via the write data **adb3_ocp_srl16_ret** component.
- The write data **adb3_ocp_srl16_ret** slave valid is generated from the **slave_q** port **DataValid** element.
- The write data **adb3_ocp_srl16_ret** master valid is generated from the **master_m2s** port **DataValid** element.

6.1.2.13.3.3 Read Response Path

This consists of the **Resp**, **Tag**, and **Data** elements of the **master_s2m/slave_s2m** signals, and the **RespAccept** element of the **master_m2s/slave_m2s** signals.

- **master_s2m** port read response elements are interfaced to the **slave_s2m** port read response elements via the read response **adb3_ocp_srl16_ret** component.
- The read response **adb3_ocp_srl16_ret** slave valid is generated from the **slave_q** port **Resp** element.
- The read response **adb3_ocp_srl16_ret** master valid is generated from the **slave_s2m** port **Resp** element.

6.1.2.13.3.4 SRL16E Retime Block (adb3_ocp_srl16_ret)

The **adb3_ocp_srl16_ret** component block diagram is shown in Figure 69 below.

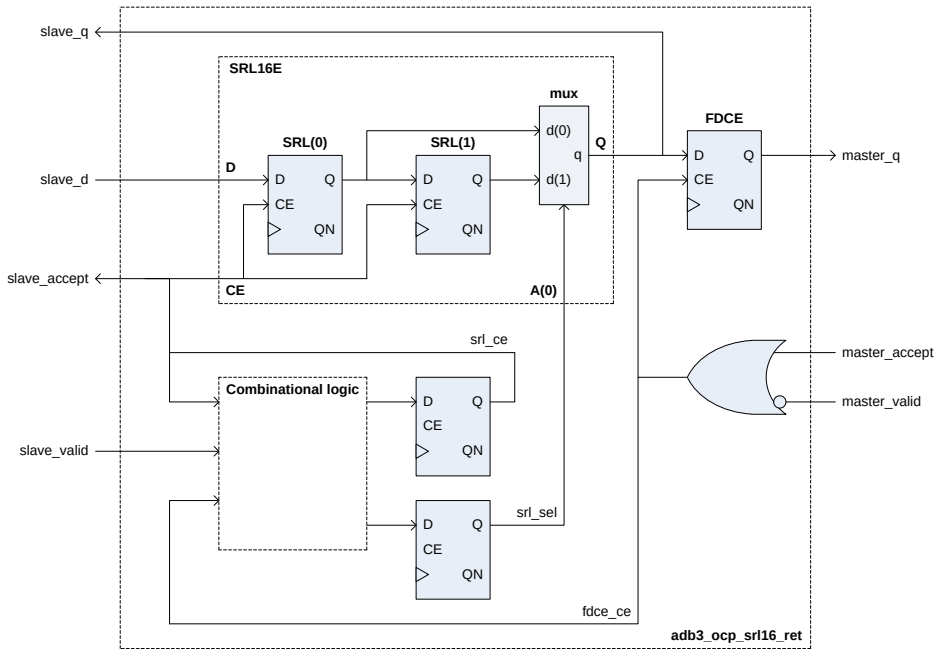

Figure 69 : adb3_ocp_srl16_ret block diagram

The component has two modes of operation, shift, and hold. SRL16E shifting is controlled by **srl_ce**. FDCE shifting is controlled by **fdce_ce**.

Shift Mode

- Slave data **slave_d** is shifted through SR(0) and mux **d(0)** to master data **master_d**.
- SRL16E Shifting continues until the FDCE is not able to accept valid data (**fdce_ce**='0' and **slave_valid**= '1')

Hold Mode

- SRL16E holds slave data in **SR(0)** and **SR(1)**. **SR(1)** is selected by mux **srl_sel**.
- SRL16E returns to shifting when the the FDCE is enabled (**fdce_ce**='1'). **SR(0)** is selected by mux **srl_sel**.

6.1.2.14 adb3_ocp_simple_bus_if_nb

6.1.2.14.1 Introduction

This is a non-blocking component in the **ADB3 OCP** group. Its function is to convert a single ADB3 OCP channel to a simple parallel interface.

Dependencies

- The command path is independent from the write data path. Data acceptance does not block command acceptance.
- The command path is independent from the read response path. Response acceptance does not block command acceptance.
- Transaction response order always matches transaction command acceptance order.
- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)

6.1.2.14.2 Interface

The **adb3_ocp_simple_bus_if_nb** component interface is shown in Figure 70 below and described in Table 154.

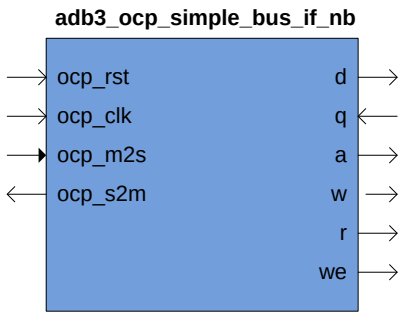

Figure 70 : adb3_ocp_simple_bus_if_nb component interface

Signal	Type	Description
addr_width	Generic	Width of the a address output (byte address).
data_width	Generic	Width of the d/q data input/output.
read_latency	Generic	Number of cycles delay before q data input is available.

Signal	Type	Description
		OCP Interface
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP M2S connection.
ocp_s2m	Output	OCP S2M connection.
		Simple Bus Interface

Table 154 : adb3_ocp_simple_bus_if_nb component interface (continued on next page)

Signal	Type	Description
d	Output	Write data of width data_width.
q	Input	Read data of width data_width.
a	Output	Write/Read address (byte address) of width addr_width.
w	Output	Write enable.
r	Output	Read enable.
we	Output	Write data byte enable of width data_width/8.

Table 154 : adb3_ocp_simple_bus_if_nb component interface

6.1.2.14.3 Description

The **adb3_ocp_simple_bus_if_nb** component block diagram is shown in Figure 71 below.

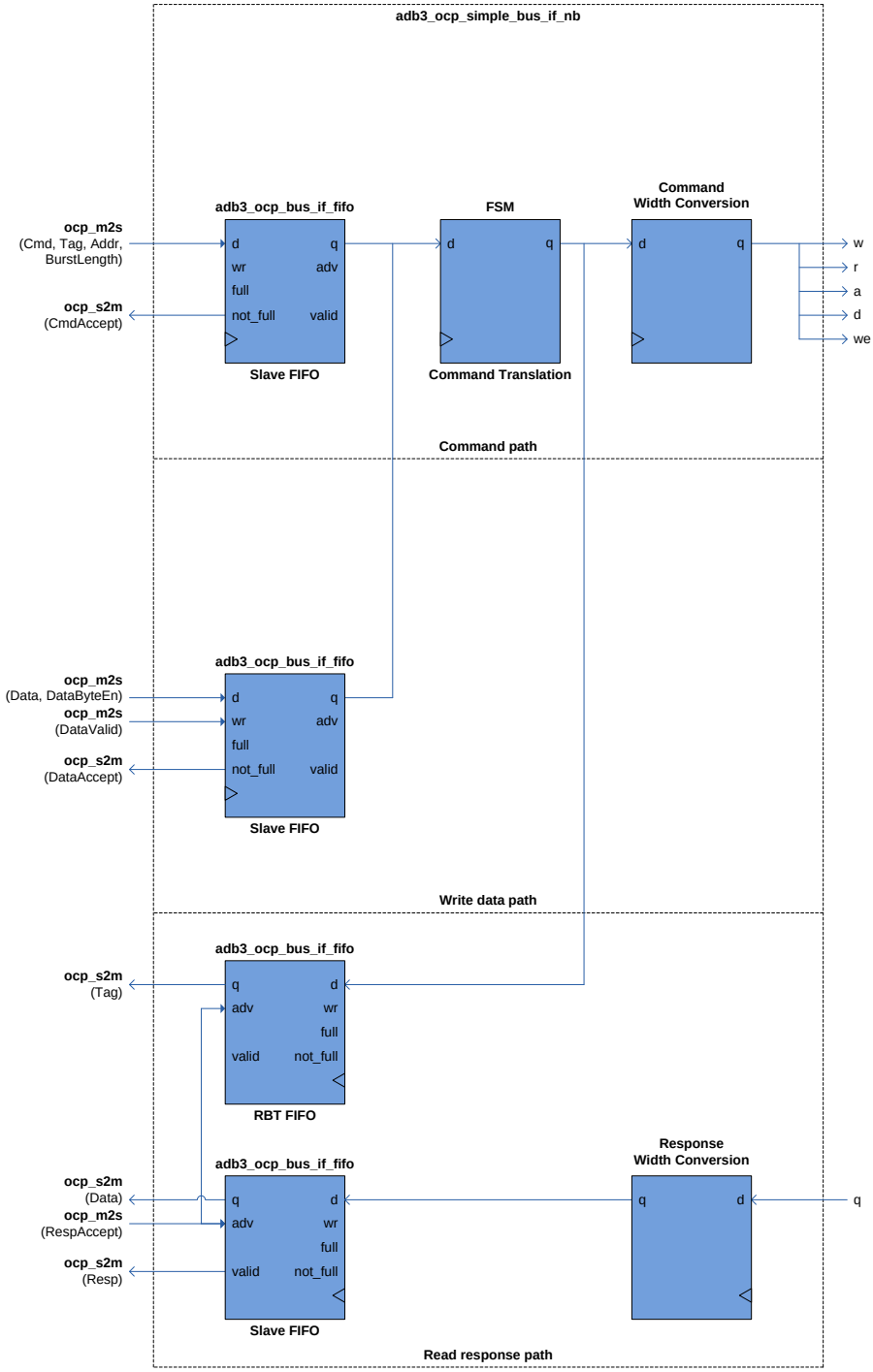

Figure 71 : adb3_ocp_simple_bus_if_nb block diagram

6.1.2.14.3.1 Command Path

This consists of the **Cmd**, **Tag**, **BurstLength**, and **Addr** elements of the **ocp_m2s** signal, and the **a**, **w**, and **r** simple bus interface signals.

Slave Command FIFO

- The **ocp_m2s** port command elements are interfaced to the command translation FSM via the slave command FIFO.
- The **ocp_s2m** port **CmdAccept** element is generated from the slave command FIFO not full flag.
- The slave command FIFO write advance is generated from the **ocp_m2s** port **Cmd** element and the slave command FIFO not full flag.
- The slave command FIFO read advance is generated from the slave command FIFO not empty, and the FSM burst start output **n_mcx_bstart**.

Command Translation FSM

- Slave command/write data FIFO data is converted into ADB3 OCP data width simple bus interface commands which are then written to the command conversion function.
- The FSM will pause if the slave write data FIFO data is not valid, or **mst_cmd_busy** is active during an OCP write command.
- The FSM will pause if the read burst tracking FIFO is full, or **mst_cmd_busy** is active during an OCP read command.
- The FSM burst start output **n_mcx_bstart** is used to generate the slave command FIFO read advance.
- The FSM master write and write command outputs **n_mcx_mst_wr**, and **n_mcx_cmd_wr** are used to generate the slave write data FIFO read advance.

Command Conversion

- ADB3 OCP data width simple bus interface commands are converted into **data_width** data width simple bus interface commands by the command conversion function.
- The signal **mst_cmd_busy** is used to pause the command translation FSM while command conversion is ongoing.

Read Burst Tracking FIFO

- The command translation FSM output **n_mcx_tag** is written to the read burst tracking FIFO on every ADB3 OCP data width simple bus interface read command.
- The read burst tracking FIFO output **rbt_q_tag** is used as the tag value for OCP read response data.
- The read burst tracking FIFO write advance is generated from the command translation FSM outputs **n_mcx_mst_wr** and **n_mcx_cmd_rd**.
- The read burst tracking FIFO read advance is generated from the read burst tracking FIFO not empty, and the slave read response FIFO write advance.

6.1.2.14.3.2 Write Data Path

This consists of the **DataValid**, **DataByteEn**, and **Data** elements of the **ocp_m2s** signal, and the **d**, and **we** simple bus interface signals.

Slave Write Data FIFO

- The **ocp_m2s** port write data elements are interfaced to the command translation FSM via the slave write data FIFO.
- The **ocp_s2m** port **DataAccept** element is generated from the slave write data FIFO not full flag.
- The slave write data FIFO write advance is generated from the **ocp_m2s** port **DataValid** element and the slave write data FIFO not full flag.

- The slave write data FIFO read advance is generated from the slave write data FIFO not empty, and the command translation FSM master write and write command outputs **n_mcx_mst_wr** and **n_mcx_cmd_wr**.

6.1.2.14.3.3 Read Response Path

This consists of the **Resp**, **Tag**, and **Data** elements of the **ocp_s2m** signal, and the **q** simple bus interface signal.

Slave Read Response FIFO

- The ADB3 OCP width simple bus interface read data is interfaced to the **ocp_s2m** port read response elements via the slave read response FIFO.
- The **ocp_s2m** port **Resp** element is generated from the slave read response FIFO not empty.
- The slave read response FIFO write advance is generated from the response conversion **slv_resp_val** response valid signal. Valid data must always be accepted.
- The slave read response FIFO read advance is generated from the slave read response FIFO not empty, and the **ocp_m2s** port **RespAccept** element.

Response Conversion

- **data_width** data width simple bus interface commands are converted into ADB3 OCP data width simple bus interface commands by the response conversion function.
- The signal **slv_resp_val** is used to generate the slave read response FIFO write advance.

6.1.2.14.3.4 Example Waveforms

In the following example, we are performing 2 OCP writes with burst length of 1, to a simple bus with 32 bit data. OCP data consists of 16 bytes, and so this results in 4 writes to the simple bus, each of 4 bytes. Each simple bus write is indicated by the **w** signal. The simple bus 4-byte write data on **d** is enabled by the 4-bit **we** bus. OCP addresses are always 16-byte aligned. The write address **a** ls nibble will therefore always sequence through values 0x00, 0x04, 0x08, 0x0C.

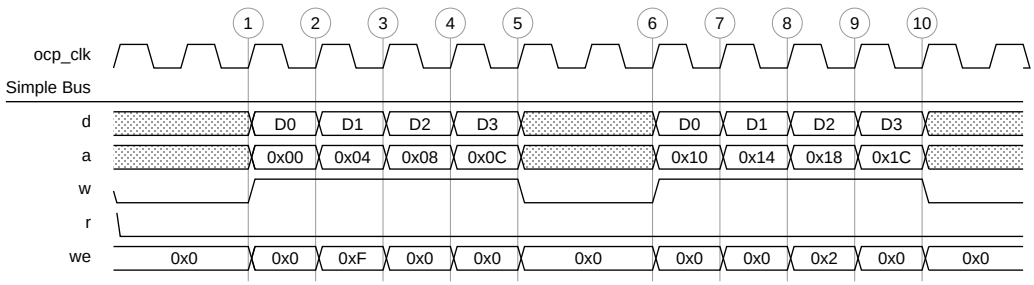

Figure 72 : OCP Writes (Burst Length = 1, d = 32 bits) To Simple Bus

The first OCP write is bytes 0-3 of D1 to byte addresses 0x04, 0x05, 0x06, 0x07.

The second OCP write is byte 1 of D2 to byte address 0x19.

In the following example, we are performing 2 OCP reads with burst length of 1, from a simple bus with **read_latency** of 1 and 32 bit data. OCP responses consists of 16 bytes, and so this results in 4 reads from the simple bus, each of 4 bytes. Each simple bus read is indicated by the **r** signal. The simple bus 4-byte read data on **q** is expected 1 cycle after **r** is valid (**read_latency** = 1). OCP addresses are always 16-byte aligned. The read address **a** ls nibble will therefore always sequence through values 0x00, 0x04, 0x08, 0x0C.

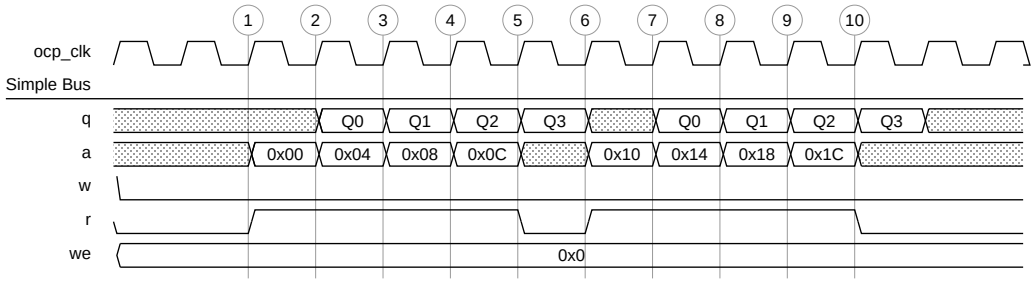

Figure 73 : OCP Read (Burst Length = 1, Read Latency = 1, d = 32 bits) From Simple Bus

The OCP read is 16-bytes from byte address starting at 0x00.

In the following example, we are performing 2 OCP writes with burst length of 1, to a simple bus with 128 bit data. OCP data consists of 16 bytes, and so this results in 1 write to the simple bus of 16 bytes. Each simple bus write is indicated by the **w** signal. The simple bus 16-byte write data on **d** is enabled by the 16-bit **we** bus. OCP addresses are always 16-byte aligned. The write address **a** ls nibble will therefore always have value 0x00.

In the following example, we are performing 2 OCP reads with burst length of 1, from a simple bus with **read_latency** of 1 and 128 bit data. OCP responses consists of 16 bytes, and so this results in 1 read from the simple bus of 16 bytes. Each simple bus read is indicated by the **r** signal. The simple bus 16-byte read data on **q** is expected 1 cycle after **r** is valid (**read_latency** = 1). OCP addresses are always 16-byte aligned. The read address **a** ls nibble will therefore always have value 0x00.

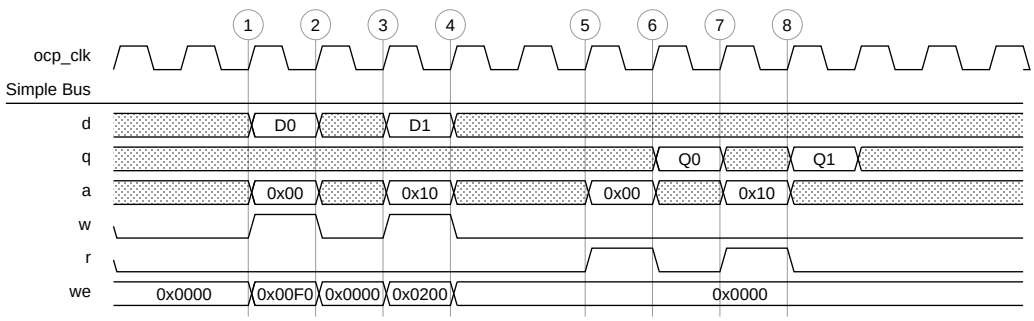

Figure 74 : OCP Writes/Reads (Burst Length = 1, Read Latency = 1, d = 128 bits) To/From Simple Bus

The first OCP write is bytes 4-7 of D0 to byte addresses 0x04, 0x05, 0x06, 0x07.

The second OCP write is byte 9 of D1 to byte address 0x19.

The first OCP read is 16 bytes from byte address starting at 0x00.

The second OCP read is 16 bytes from byte address starting at 0x10.

6.1.3 ADB3 OCP Testbench Package (adb3_ocp_tb_pkg)

The package **adb3_ocp_tb_pkg** defines functions and procedures relating to the ADB3 OCP profile which are used in target example FPGA testbenches.

Type Definitions

- **byte_t**. A byte datatype
- **byte_vector_t**. Variable width array of **byte_t** data bytes.
- **byte_enable_t**. Variable width array of **std_logic** data byte enables.

Function Definitions

- **conv_byte_vector**. Convert type **std_logic_vector** to type **byte_vector_t**
- **conv_byte_enable**. Convert type **std_logic_vector** to type **byte_enable_t**
- **conv_vector**. Convert type **byte_vector_t** to type **std_logic_vector**
- **conv_string_hex**. Convert types **byte_vector_t/std_logic_vector** to type **string**
- **conv_string**. Convert types **byte_enable_t/std_logic_vector** to type **string**

Procedure Definitions

- **adb3_ocp_sim_read_reg32**. ADB3 OCP read procedure reading 4-byte data using **adb3_ocp_sim_read**. Procedure is blocking and will not complete until all ADB3 OCP response data has been returned. Address input is byte aligned and is converted to a 16-byte aligned ADB3 OCP read command address. 4-byte data from the correct offset is returned from the ADB3 OCP 16-byte response data.
- **adb3_ocp_sim_read**. ADB3 OCP read procedure. Procedure is blocking and will not complete until all ADB3 OCP response data has been returned. Address input is 16-byte aligned and is used as the first ADB3 OCP read command address. Read data is returned from the ADB3 OCP response data.
- **adb3_ocp_sim_read_nb**. ADB3 OCP read command procedure. Procedure is non-blocking and will complete after an ADB3 OCP read command has been issued, or read data has been returned from ADB3 OCP response data. Address input is 16-byte aligned and is used as the first ADB3 OCP read command address.
- **adb3_ocp_sim_read_cmd**. ADB3 OCP read command procedure. Procedure is non-blocking and will complete after all ADB3 OCP read commands have been issued. Address input is 16-byte aligned and is used as the first ADB3 OCP read command address.
- **adb3_ocp_sim_read_resp**. ADB3 OCP read response procedure. Read data is returned from the ADB3 OCP response data.
- **adb3_ocp_sim_write_reg32**. ADB3 OCP write procedure writing 4-byte data using **adb3_ocp_sim_write**. Data input is 4-bytes, Byte enable input is 4 bits. Address input is byte aligned and is converted to a 16-byte aligned ADB3 OCP write command address. The 4-byte data with the correct offset will be inserted in the ADB3 OCP 16-byte write data. The 4-bit byte enables with the correct offset will be inserted in the ADB3 OCP 16-bit byte enables.
- **adb3_ocp_sim_write**. ADB3 OCP write procedure. Data input is n-bytes, Byte enable input is n bits. Address input is 16-byte aligned and is used as the ADB3 OCP write command address. The n-byte data is used as ADB3 OCP 16-byte write data. The n-bit byte enables is used as ADB3 OCP 16-bit byte enables.
- **adb3_ocp_sim_wait_cycles**. Clock cycle wait procedure.

6.2 ADB3 Target

The ADB3 target group is located in **hdl/vhdl/common/adb3_target/** and contains the following elements:

- ADB3 Target Include Package (adb3_target_inc_pkg)
- ADB3 Target Components
- ADB3 Target Testbench Include Package (adb3_target_tb_inc_pkg)
- ADB3 Target Testbench Components

6.2.1 ADB3 Target Include Package (adb3_target_inc_pkg)

The **adb3_target_inc_pkg** package defines constants and types which characterise the target example FPGA design on the model selected. The package exists in two variants for each supported model. This enables a simulation to perform "lightweight" versions of certain lengthy initialisation sequences. Without these aids, rapid development of code would be unfeasible due to the length of real time required for simulations.

Table 155 lists the available variants of the **adb3_target_inc_pkg** package used during OCP-Only simulation:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/adb3_target_inc_pkg_sim_ocp_6tl.vhd
ADM-XRC-6T1	admxrc6t1/adb3_target_inc_pkg_sim_ocp_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/adb3_target_inc_pkg_sim_ocp_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/adb3_target_inc_pkg_sim_ocp_6tadv8_pcie.vhd
ADM-XRC-6T-DA1	admxrc6tda1/adb3_target_inc_pkg_sim_ocp_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/adb3_target_inc_pkg_sim_ocp_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/adb3_target_inc_pkg_sim_ocp_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/adb3_target_inc_pkg_sim_ocp_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/adb3_target_inc_pkg_sim_ocp_admxrc7v1.vhd

Table 155 : OCP-only simulation variants of the adb3_target_inc_pkg package

Table 156 lists the available variants of the **adb3_target_inc_pkg** package used during Full MPTL simulation and synthesis:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/adb3_target_inc_pkg_syn_ngc_6tl.vhd
ADM-XRC-6T1	admxrc6t1/adb3_target_inc_pkg_syn_ngc_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/adb3_target_inc_pkg_syn_ngc_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/adb3_target_inc_pkg_syn_ngc_6tadv8_pcie.vhd
ADM-XRC-6T-DA1	admxrc6tda1/adb3_target_inc_pkg_syn_ngc_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/adb3_target_inc_pkg_syn_ngc_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/adb3_target_inc_pkg_syn_ngc_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/adb3_target_inc_pkg_syn_ngc_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/adb3_target_inc_pkg_syn_ngc_admxrc7v1.vhd

Table 156 : Full MPTL simulation/synthesis variants of the adb3_target_inc_pkg package

The following definitions are available in this package:

General Definitions

- **std_logic_dbl_t**. Type defining a general-purpose differential std_logic signal.

Clock Definitions (Varies depending on model)

- **REF_CLK_FREQ_HZ**. The frequency in Hz of the reference clock input used by the target FPGA design on this model.
- **clks_in_t**. Record defining target FPGA clock inputs on this model.
- **MGT_CLKS_VALID** Vector defining target FPGA MGT clock inputs which require to be buffered.
- **clks_mgt_in_t**. Record defining target FPGA MGT clock inputs on this model.
- **clks_out_t**. Record defining target FPGA clock outputs on this model.

GPIO Definitions (Varies depending on model)

- **XRM_GPIO_WIDTH**. Indicates width of XRM GPIO interface on this model.
- **PN4_GPIO_WIDTH**. Indicates width of Pn4 GPIO interface on this model.
- **PN6_GPIO_WIDTH**. Indicates width of Pn6 GPIO interface on this model.
- **xrm_gpio_t**. Record defining target FPGA XRM GPIO interface on this model.
- **XRM_GPIO_DEFAULT**. Record defining default value for xrm_gpio_t on this model.
- **pn4_gpio_t**. Record defining target FPGA Pn4 GPIO interface on this model.
- **PN4_GPIO_DEFAULT**. Record defining default value for pn4_gpio_t on this model.
- **pn6_gpio_t**. Record defining target FPGA Pn6 GPIO interface on this model.
- **PN6_GPIO_DEFAULT**. Record defining default value for pn6_gpio_t on this model.
- **gpio_inout_t**. Record defining target FPGA GPIO interface on this model.
- **GPIO_DEFAULT**. Record defining default value for gpio_inout_t on this model.

Custom IO Definitions (Varies depending on model)

- **custom_in_t**. Record defining target FPGA custom interface (inputs) on this model.
- **custom_out_t**. Record defining target FPGA custom interface (outputs) on this model.
- **custom_inout_t**. Record defining target FPGA custom interface (bidirs) on this model.

On-Board Memory Interface Definitions (Varies depending on model)

- **DDR3_BANKS**. Indicates the number of target FPGA DDR3 SDRAM bank interfaces on this model.
- **DDR3_BANK_ROW_WIDTH***. Indicates the maximum width of the target FPGA DDR3 SDRAM row address interface on this model.
- **DDR3_BANK_DATA_WIDTH**. Indicates the width of the target FPGA DDR3 SDRAM data interface on this model
- **DDR3_BYTE_ADDR_WIDTH**. Indicates the maximum width of the target FPGA DDR3 SDRAM byte address interface on this model.
- **ddr3_addr_out_t**. Record defining target FPGA DDR3 SDRAM bank address interface on this model.
- **ddr3_ctrl_out_t**. Record defining target FPGA DDR3 SDRAM bank control interface on this model.
- **ddr3_data_inout_t**. Record defining target FPGA DDR3 SDRAM bank data interface on this model.
- **ddr3_clk_out_t**. Record defining target FPGA DDR3 SDRAM bank clock interface on this model.
- **DDR3_ADDR_DEFAULT_OUT**. Record defining default value for ddr3_addr_out_t on this model.
- **DDR3_CTRL_DEFAULT_OUT**. Record defining default value for ddr3_ctrl_out_t on this model.
- **DDR3_DATA_DEFAULT_INOUT**. Record defining default value for ddr3_data_inout_t on this model.

- **DDR3_CLK_DEFAULT_OUT**. Record defining default value for ddr3_clk_out_t on this model.
- **ddr3_addr_out_array_t**. Array defining target FPGA DDR3 SDRAM address interface on this model.
- **ddr3_ctrl_out_array_t**. Array defining target FPGA DDR3 SDRAM control interface on this model.
- **ddr3_data_inout_array_t**. Array defining target FPGA DDR3 SDRAM data interface on this model.
- **ddr3_clk_out_array_t**. Array defining target FPGA DDR3 SDRAM clock interface on this model.
- **MEM_BANKS**. Indicates the number of target FPGA on-board memory interfaces on this model.
- **DDR3_BANK0**. Indicates the bank number of the first target FPGA DDR3 SDRAM bank interface on this model.
- **mem_byte_addr_width_array_t**. Array defining target FPGA on-board memory bank address widths on this model.
- **MEM_BYTE_ADDR_WIDTH_ARRAY**. Indicates the address width of each bank of on-board memory on this model.
- **mem_addr_out_t**. Record defining target FPGA address interface on this model.
- **mem_ctrl_out_t**. Record defining target FPGA control interface on this model.
- **mem_data_inout_t**. Record defining target FPGA data interface on this model.
- **mem_clk_out_t**. Record defining target FPGA clock interface on this model.

* Note: The value of the **DDR3_BANK_ROW_WIDTH** constant determines the maximum size of DDR3 SDRAM parts supported by the target FPGA design. Currently, valid vaues are 13 for 1Gib parts only, 14 for 1Gib/2Gib parts, or 15 for 1Gib/2Gib/4Gib parts. The simulation model for the appropriate memory part will also need to be selected in the example design testbench. This is achieved by selecting either **1**, **2**, or **4** for the value of the build option m **OPTION_M** constant in the adb3_target_tb_inc_pkg.

Note
The user should verify that the value of the **DDR3_BANK_ROW_WIDTH** (default = 14) and **OPTION_M** (default = 1) constants are appropriate for the size of the memory parts on the model in use.

OCP Interface Definitions

- **DS_CHANNELS**. Indicates the number of direct slave OCP channels on this model.
- **DMA_CHANNELS**. Indicates the number of dma OCP channels on this model.
- **DM_CHANNELS**. Indicates the number of direct master OCP channels on this model.
- **DS_ADDR_WIDTH**. Indicates the address space size for a direct slave OCP channel on this model.
- **DMA_ADDR_WIDTH**. Indicates the address space size for a dma OCP channel on this model.
- **DM_ADDR_WIDTH**. Indicates the address space size for a direct master OCP channel on this model.

MPTL Interface Definitions (Varies depending on model)

- **MPTL_SER_WIDTH**. Indicates the width of the MPTL serial data interface that exists on this model.
- **mptl_t2b_t**. Type defining the MPTL interface signals between the target and bridge FPGAs.
- **mptl_b2t_t**. Type defining the MPTL interface signals between the bridge and target FPGAs.
- **mptl_sb_b2t_t**. Type defining the MPTL sideband interface signals from the bridge to the target.
- **mptl_sb_t2b_t**. Type defining the MPTL sideband interface signals from the target to the bridge.

PCIe Interface Definitions (Varies depending on model)

- **PCIE_SER_WIDTH**. Indicates the width of the PCIe serial data interface that exists on this model.
- **pcie_t2h_t**. Type defining the PCIe interface signals between the target FPGA and host.

- **pcie_h2t_t**. Type defining the PCIe interface signals between the host and target FPGA.

Component Definitions (Varies depending on model)

- Target MPTL Interface Wrapper (mptl_if_target_wrap). MPTL interface component on models with Bridge FPGA.
- Target PCIe Interface Wrapper (pcie_if_target_wrap). PCIe interface component on models with no Bridge FPGA.

6.2.2 ADB3 Target Components

6.2.2.1 Target MPTL Interface Wrapper (mptl_if_target_wrap)

6.2.2.1.1 Introduction

This is a component in the **ADB3_Target** group. It is used by the example FPGA designs as the target FPGA end of the MPTL interface.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 Target Include Package (adb3_target_inc_pkg)

6.2.2.1.2 Interface

The **mptl_if_target_wrap** component interface is shown in Figure 75 below and described in Table 157.

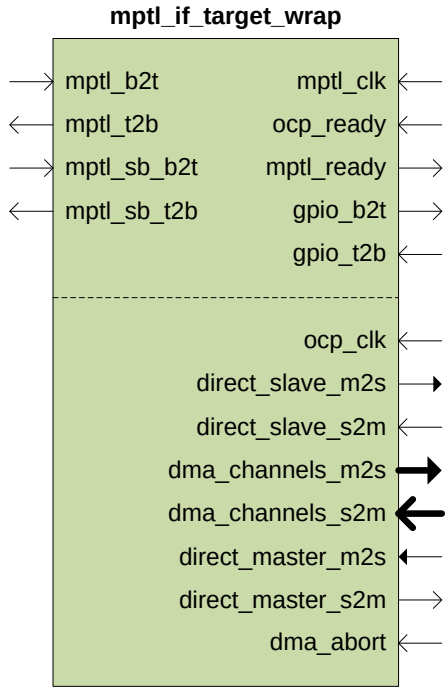

Figure 75 : mptl_if_target_wrap component interface

Signal	Type	Description
enable_dm	Generic	Enable use of MPTL interface with Direct Master.

Signal	Type	Description
		OCP Interface
ocp_clk	Input	OCP clock (from target).
direct_slave_m2s	Output	Direct slave OCP channel master (from bridge via MPTL interface).
direct_slave_s2m	Input	Direct slave OCP channel slave (to bridge via MPTL interface).
dma_channels_m2s	Output	DMA OCP channels master (from bridge via MPTL interface).
dma_channels_s2m	Input	DMA OCP channels slave (to bridge via MPTL interface).
dma_abort	Input	DMA abort request (from target).
direct_master_m2s	Input	Direct master OCP channel slave (to bridge via MPTL interface).
direct_master_s2m	Output	Direct master OCP channel master (from bridge via MPTL interface).
		MPTL Interface
mptl_t2b	Output	MPTL serial interface signals (to bridge).
mptl_b2t	Input	MPTL serial interface signals (from bridge).
mptl_clk	Input	MPTL clock (from target).
ocp_ready	Input	OCP channels ready (from target).
mptl_ready	Output	MPTL interface ready (to target).
mptl_sb_t2b	Output	MPTL interface sideband signals (to bridge).
mptl_sb_b2t	Input	MPTL interface sideband signals (from bridge).
gpio_b2t	Output	General purpose i/o (from bridge via MPTL interface).
gpio_t2b	Input	General purpose i/o (to bridge via MPTL interface).

Table 157 : mptl_if_target_wrap component interface

6.2.2.1.3 Description

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interfaces. They are of types **mptl_t2b_t/mptl_b2t_t** which are defined in the adb3_target_inc_pkg package. During OCP-only simulation, these signals transfer OCP transactions directly between the bridge and target MPTL interfaces. During full MPTL simulation and synthesis, these signals transfer MPTL serial data between the bridge and target MPTL interfaces.

The MPTL interface sideband signals **mptl_sb_t2b** and **mptl_sb_b2t** connect the bridge and target MPTL interfaces. They are of types **mptl_sb_t2b_t/mptl_sb_b2t_t** which are also defined in the adb3_target_inc_pkg package. These signals transfer MPTL sideband information directly between the bridge and target MPTL interfaces.

The type of Target MPTL interface that is instantiated depends upon whether simulation or synthesis is required as follows:

6.2.2.1.3.1 OCP-Only Simulation

During OCP-only simulation, a simulation only version of the **mptl_if_target_wrap** component is instantiated. Table 158 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_target/mptl_if_target_wrap_sim_6tl.vhd
ADM-XRC-6T1	admxrc6t1/mptl_target/mptl_if_target_wrap_sim_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/mptl_target/mptl_if_target_wrap_sim_6tge.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mptl_target/mptl_if_target_wrap_sim_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/mptl_target/mptl_if_target_wrap_sim_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/mptl_target/mptl_if_target_wrap_sim_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/mptl_target/mptl_if_target_wrap_sim_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/mptl_target/mptl_if_target_wrap_sim_admxrc7v1.vhd

Table 158 : Available variants of simulation only version of mptl_if_target_wrap component

Clock Generation

- During OCP-only simulation, the bridge MPTL interface OCP clock must be the same as the target MPTL interface OCP clock. This is accomplished by connecting the target clock input **ocp_clk** to the bridge clock via the **mptl_t2b.target_ocp_clk** signal.

Initialisation

- The **mptl_sb_t2b.mptl_target_configured_l** output is generated using the OCP channels ready **ocp_ready** input.
- The **mptl_sb_t2b.mptl_target_gtp_online_l** output is generated using an online delay counter and the **mptl_sb_b2t.mptl_bridge_gtp_online_l** input.
- The **mptl_ready** output is generated using the bridge online **mptl_sb_b2t.mptl_bridge_gtp_online_l** input.

MPTL Interface

- The direct slave OCP channel master output **direct_slave_m2s** is driven by the **mptl_b2t.direct_slave_m2s** input from the bridge MPTL interface. The **mptl_t2b.direct_slave_s2m** output to the bridge MPTL interface is driven by the direct slave OCP channel slave input **direct_slave_s2m**.
- The DMA OCP channels master output **dma_channels_m2s** is driven by the **mptl_b2t.dma_channels_m2s** input from the bridge MPTL interface. The **mptl_t2b.dma_channels_s2m** output to the bridge MPTL interface is driven by the DMA OCP channels slave input **dma_channels_s2m**.
- The direct master OCP channel slave input **direct_master_m2s** drives the **mptl_t2b.direct_masters_m2s** channel 0 input to the bridge MPTL interface when the **enable_dm** generic is true. The **mptl_b2t.direct_masters_s2m** output from the bridge MPTL interface drives the direct master OCP channel master output **direct_master_s2m** when the **enable_dm** generic is true.
- The general purpose i/o bus **gpio_t2b** input drives the **mptl_t2b.gpio_t2b** output to the bridge MPTL interface. The **mptl_b2t.gpio_b2t** input from the bridge MPTL interface drives the general purpose i/o bus output **gpio_b2t**.

DMA Abort

- On the ADM-XRC-6TL, the inverted **dma_abort** input from the target FPGA drives the DMA abort request output **mptl_sb_t2b.mptl_dma_abort_l**.
- On all other models, the **dma_abort** input from the target FPGA drives the DMA abort request output **mptl_t2b.dma_abort**.

6.2.2.1.3.2 Full MPTL Simulation and Synthesis

During full MPTL simulation and synthesis, the **mptl_if_target_wrap** component is instantiated. Table 159 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_target/mptl_if_target_wrap_6tl.vhd
ADM-XRC-6T1	admxrc6t1/mptl_target/mptl_if_target_wrap_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/mptl_target/mptl_if_target_wrap_6tge.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mptl_target/mptl_if_target_wrap_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/mptl_target/mptl_if_target_wrap_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/mptl_target/mptl_if_target_wrap_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/mptl_target/mptl_if_target_wrap_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/mptl_target/mptl_if_target_wrap_admxrc7v1.vhd

Table 159 : Available variants of mptl_if_target_wrap component

Initialisation

- The **mptl_sb_t2b.mptl_target_configured_l** output is generated using the OCP channels ready **ocp_ready** input.
- The **mptl_ready** output is generated using the bridge online **mptl_sb_b2t.mptl_bridge_gtp_online_l** input.

During full MPTL simulation, the **mptl_if_target_wrap** component instantiates the MPTL interface HDL netlist appropriate to the model in use. Table 160 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_target/mptl_if_target_netlist_wrap_6tl.vhd
ADM-XRC-6T1	admxrc6t1/mptl_target/mptl_if_target_netlist_wrap_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/mptl_target/mptl_if_target_netlist_wrap_6tge.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mptl_target/mptl_if_target_netlist_wrap_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/mptl_target/mptl_if_target_netlist_wrap_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/mptl_target/mptl_if_target_netlist_wrap_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/mptl_target/mptl_if_target_netlist_wrap_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/mptl_target/mptl_if_target_netlist_wrap_admxrc7v1.vhd

Table 160 : Available variants of target MPTL interface netlist

The **mptl_if_target_wrap** component signals are connected to their equivalents on the MPTL interface HDL netlist.

During synthesis, the **mptl_if_target_wrap** component instantiates the MPTL interface core (**.ngc**) appropriate to the model in use. Table 161 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_target/mptl64par_interface_target_6tl.ngc
ADM-XRC-6T1	admxrc6t1/mptl_target/mptl128_interface_target_6t1.ngc
ADM-XRC-6TGE	admxrc6tge/mptl_target/mptl128_interface_target_6t1.ngc
ADM-XRC-6T-DA1	admxrc6tda1/mptl_target/mptl128_interface_target_6t1.ngc
ADPE-XRC-6T	adpexrc6t/mptl_target/mptl128_interface_target_6t1.ngc
ADPE-XRC-6T-L	adpexrc6tl/mptl_target/mptl128_interface_target_v6x1.ngc
ADM-XRC-7K1	admxrc7k1/mptl_target/mptl128_interface_target_7k1.ngc
ADM-XRC-7V1	admxrc7v1/mptl_target/mptl128_interface_target_7k1.ngc

Table 161 : Available variants of MPTL interface core

The **mptl_if_target_wrap** component signals are connected to their equivalents on the MPTL interface core.

6.2.2.2 Target PCIe Interface Wrapper (pcie_if_target_wrap)

6.2.2.2.1 Introduction

This is a component in the **ADB3_Target** group. It is used by the example FPGA designs as the target FPGA end of the PCIe interface.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 Target Include Package (adb3_target_inc_pkg)

6.2.2.2.2 Interface

The **pcie_if_target_wrap** component interface is shown in Figure 76 below and described in Table 162.

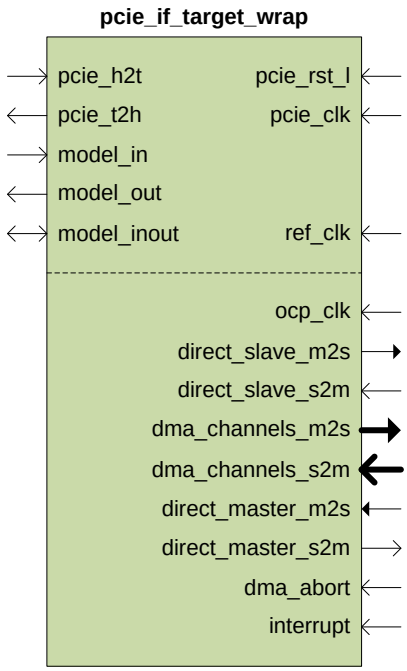

Figure 76 : pcie_if_target_wrap component interface

Signal	Type	Description
enable_dm	Generic	Enable use of PCIe interface with Direct Master.

Signal	Type	Description
		OCP Interface
ocp_clk	Input	OCP clock (from target).
direct_slave_m2s	Output	Direct slave OCP channel master (from host via PCIe interface).
direct_slave_s2m	Input	Direct slave OCP channel slave (to host via PCIe interface).
dma_channels_m2s	Output	DMA OCP channels master (from host via PCIe interface).

Table 162 : pcie_if_target_wrap component interface (continued on next page)

Signal	Type	Description
dma_channels_s2m	Input	DMA OCP channels slave (to host via PCIe interface).
dma_abort	Input	DMA abort request (from target).
direct_master_m2s	Input	Direct master OCP channel slave (to host via PCIe interface).
direct_master_s2m	Output	Direct master OCP channel master (from host via PCIe interface).
interrupt	Input	Interrupt request (from target).
		PCIe Interface
pcie_t2h	Output	PCIe serial interface signals (to host).
pcie_h2t	Input	PCIe serial interface signals (from host).
pcie_clk	Input	PCIe clock (from target).
pcie_rst_l	Input	PCIe reset (from target).
		Model Interface
ref_clk	Input	Model interface clock (from target).
model_in	Input	Model interface signals (from board).
model_out	Output	Model interface signals (to board).
model_inout	Output	Model interface signals (from/to board).

Table 162 : pcie_if_target_wrap component interface

6.2.2.2.3 Description

The PCIe interface signals **pcie_t2h** and **pcie_h2t** connect the host and target PCIe interfaces. They are of types **pcie_t2h_t/pcie_h2t_t** which are defined in the adb3_target_inc_pkg package. During OCP-only simulation, these signals transfer OCP transactions directly between the host and target PCIe interfaces. During synthesis, these signals transfer PCIe serial data between the host and target PCIe interface.

The Model interface signals **model_in**, **model_out** and **model_inout** connect the board and target PCIe interface. They are of types **model_in_t/model_out_t/model_inout_t** which are also defined in the adb3_target_inc_pkg package. These signals implement model specific interfaces on models without a bridge FPGA.

The type of Target PCIe interface that is instantiated depends upon whether simulation or synthesis is required as follows:

6.2.2.2.3.1 OCP-Only Simulation

During OCP-only simulation, a simulation only version of the **pcie_if_target_wrap** component is instantiated. Table 163 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6T-ADV8	admxrc6tadv8/pcie_target/pcie_if_target_wrap_sim_6tadv8.vhd

Table 163 : Available variants of simulation only version of pcie_if_target_wrap component

Clock Generation

- During OCP-only simulation, the host PCIe interface OCP clock must be the same as the target PCIe interface OCP clock. This is accomplished by connecting the target clock to the host clock via the **pcie_t2h.target_ocp_clk** signal.

- The **ocp_clk** input drives the **pcie_t2h.target_ocp_clk** signal.

PCIe Interface

- The direct slave OCP channel master output **direct_slave_m2s** is driven by the **pcie_h2t.direct_slave_m2s** input from the host PCIe interface. The **pcie_t2h.direct_slave_s2m** output to the host PCIe interface is driven by the direct slave OCP channel slave input **direct_slave_s2m**.
- The DMA OCP channels master output **dma_channels_m2s** is driven by the **pcie_h2t.dma_channels_m2s** input from the host PCIe interface. The **pcie_t2h.dma_channels_s2m** output to the host PCIe interface is driven by the DMA OCP channels slave input **dma_channels_s2m**.
- The direct master OCP channel slave input **direct_master_m2s** drives the **pcie_t2h.direct_masters_m2s** channel 0 input to the host PCIe interface when the **enable_dm** generic is true. The **pcie_h2t.direct_masters_s2m** output from the host PCIe interface drives the direct master OCP channel master output **direct_master_s2m** when the **enable_dm** generic is true.

DMA Abort

- The **dma_abort** input from the target FPGA drives the DMA abort request output **pcie_t2h.dma_abort**.

Interrupt

- The **interrupt** input from the target FPGA drives the interrupt request output **pcie_t2h.interrupt**.

6.2.2.2.3.2 Synthesis

During synthesis, the **pcie_if_target_wrap** component is instantiated. Table 164 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6T-ADV8	admxrc6tadv8/pcie_target/pcie_if_target_wrap_6tadv8.vhd

Table 164 : Available variants of pcie_if_target_wrap component

During synthesis, the **pcie_if_target_wrap** component instantiates the PCIe interface core (**.ngc**) appropriate to the model in use. Table 165 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6T-ADV8	admxrc6tadv8/pcie_target/admxrc6tadv8_pcie_x4.ngc

Table 165 : Available variants of PCIe interface core

The **pcie_if_target_wrap** component signals are connected to their equivalents on the PCIe interface core.

6.2.3 ADB3 Target Testbench Include Package (adb3_target_tb_inc_pkg)

The **adb3_target_tb_inc_pkg** package defines constants and types which characterise the model selected. The package exists in several variants, one for each supported model. Table 166 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/adb3_target_tb_inc_pkg_6tl.vhd
ADM-XRC-6T1	admxrc6t1/adb3_target_tb_inc_pkg_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/adb3_target_tb_inc_pkg_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/adb3_target_tb_inc_pkg_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/adb3_target_tb_inc_pkg_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/adb3_target_tb_inc_pkg_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6t1/adb3_target_tb_inc_pkg_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/adb3_target_tb_inc_pkg_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/adb3_target_tb_inc_pkg_admxrc7v1.vhd

Table 166 : Available variants of the adb3_target_tb_inc_pkg package

The following definitions are available in this package:

General Definitions

- **MODEL_NAME**. Defines a string containing the model name.
- **clks_out_exp_t**. Record defining expected frequency types for target FPGA clock outputs on this model.

Build Options

- **OPTION_M**. Defines DDR3 SDRAM part size option if relevant on this model.
- **OPTION_P**. Defines Pn4 connector fit option if relevant on this model.
- **OPTION_E**. Defines Ethernet link fit option if relevant on this model.
- **OPTION_G**. Defines Pn6 clock option if relevant on this model.

Clock Generation

- Defines board clock period constants for this model. These are used by the test_board_clks block to generate board clocks.

Component Definitions

- mptl_if_bridge_wrap
- pcie_if_host_wrap
- test_board_clks

6.2.4 ADB3 Target Testbench Components

6.2.4.1 Bridge MPTL Interface Wrapper (mptl_if_bridge_wrap)

6.2.4.1.1 Introduction

This is a component in the **ADB3_Target** group. It is used by the example FPGA testbenches as the bridge FPGA end of the MPTL interface.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 Target Include Package (adb3_target_inc_pkg)

6.2.4.1.2 Interface

The **mptl_if_bridge_wrap** component interface is shown in Figure 77 below and described in Table 167.

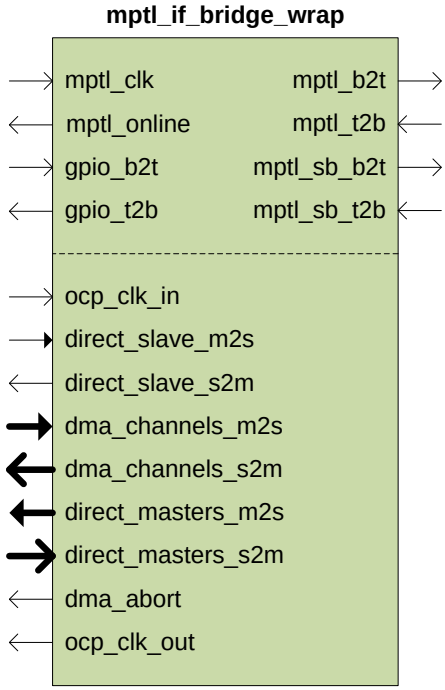

Figure 77 : mptl_if_bridge_wrap component interface

Signal	Type	Description
		OCP Interface
ocp_clk_in	Input	Independent OCP clock source (from testbench).
direct_slave_m2s	Input	Direct slave OCP channel master (to target via MPTL interface).
direct_slave_s2m	Output	Direct slave OCP channel slave (from target via MPTL interface.
dma_channels_m2s	Input	DMA OCP channels master (to target via MPTL interface).
dma_channels_s2m	Output	DMA OCP channels slave (from target via MPTL interface).

Table 167 : mptl_if_bridge_wrap component interface (continued on next page)

Signal	Type	Description
direct_masters_m2s	Output	Direct master OCP channels master (from target via MPTL interface).
direct_masters_s2m	Input	Direct master OCP channels slave (to target via MPTL interface).
dma_abort	Output	DMA abort request (to testbench).
ocp_clk_out	Output	OCP clock (to testbench).
		MPTL Interface
mptl_t2b	Input	MPTL interface signals (from target).
mptl_b2t	Output	MPTL interface signals (to target).
mptl_clk	Input	MPTL interface clock (from testbench).
mptl_online	Output	MPTL interface is online (to testbench).
mptl_sb_t2b	Input	MPTL interface sideband signals (from target).
mptl_sb_b2t	Output	MPTL interface sideband signals (to target).
gpio_b2t	Input	General purpose i/o (to target via MPTL interface).
gpio_t2b	Output	General purpose i/o (from target via MPTL interface).

Table 167 : mptl_if_bridge_wrap component interface

6.2.4.1.3 Description

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interfaces. They are of types **mptl_t2b_t/mptl_b2t_t** which are defined in the adb3_target_inc_pkg package. During OCP-only simulation, these signals transfer OCP transactions directly between the bridge and target MPTL interfaces. During full MPTL simulation and synthesis, these signals transfer MPTL data between the bridge and target MPTL interfaces.

The MPTL interface sideband signals **mptl_sb_t2b** and **mptl_sb_b2t** connect the bridge and target MPTL interfaces. They are of types **mptl_sb_t2b_t/mptl_sb_b2t_t** which are also defined in the adb3_target_inc_pkg package. These signals transfer MPTL sideband information directly between the bridge and target MPTL interfaces.

The type of Bridge MPTL interface that is instantiated depends upon whether simulation or synthesis is required as follows:

6.2.4.1.3.1 OCP-Only Simulation

During OCP-only simulation, a simulation only version of the **mptl_if_bridge_wrap** component is instantiated. Table 168 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_bridge/mptl_if_bridge_wrap_sim_6tl.vhd
ADM-XRC-6T1	admxrc6t1/mptl_bridge/mptl_if_bridge_wrap_sim_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/mptl_bridge/mptl_if_bridge_wrap_sim_6tge.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mptl_bridge/mptl_if_bridge_wrap_sim_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/mptl_bridge/mptl_if_bridge_wrap_sim_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/mptl_bridge/mptl_if_bridge_wrap_sim_adpexrc6tl.vhd

Table 168 : Available variants of simulation only version of mptl_if_bridge_wrap component (continued

on next page)

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-7K1	admxrc7k1/mptl_bridge/mptl_if_bridge_wrap_sim_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/mptl_bridge/mptl_if_bridge_wrap_sim_admxrc7v1.vhd

Table 168 : Available variants of simulation only version of mptl_if_bridge_wrap component

Clock Generation

- During OCP-only simulation, the bridge MPTL interface OCP clock must be the same as the target MPTL interface OCP clock. This is accomplished by connecting the target clock to the bridge clock via the **mptl_t2b.target_ocp_clk** signal.
- The **ocp_clk_in** input is unused.
- The **ocp_clk_out** output is driven by **mptl_t2b.target_ocp_clk**.

Initialisation

- The **mptl_sb_b2t.mptl_bridge_gtp_online_l** output is generated using an online delay counter and the **mptl_sb_t2b.mptl_target_configured_l** input.
- The **mptl_online** output is produced from the **mptl_sb_b2t.mptl_bridge_gtp_online_l**, **mptl_sb_t2b.mptl_target_configured_l**, and **mptl_sb_t2b.mptl_target_gtp_online_l** signals.

MPTL Interface

- The direct slave OCP channel master input **direct_slave_m2s** drives the **mptl_b2t.direct_slave_m2s** output to the target MPTL interface. The **mptl_t2b.direct_slave_s2m** input from the target MPTL interface drives the direct slave OCP channel slave output **direct_slave_s2m**.
- The DMA OCP channels master input **dma_channels_m2s** drives the **mptl_b2t.dma_channels_m2s** output to the target MPTL interface. The **mptl_t2b.dma_channels_s2m** input from the target MPTL interface drives the DMA OCP channels slave output **dma_channels_s2m**.
- The direct master OCP channels slave input **direct_masters_s2m** drives the **mptl_b2t.direct_masters_s2m** output to the target MPTL interface. The **mptl_t2b.direct_masters_m2s** input from the target MPTL interface drives the direct master OCP channels master output **direct_masters_m2s**.
- The general purpose i/o bus **gpio_b2t** input drives the **mptl_b2t.gpio_b2t** output to the target MPTL interface. The **mptl_t2b.gpio_t2b** input from the target MPTL interface drives the general purpose i/o bus output **gpio_t2b**.

DMA Abort

- On the ADM-XRC-6TL, the inverted **mptl_sb_t2b.mptl_dma_abort_l** input from the target MPTL interface drives the DMA abort request output **dma_abort**.
- On all other models, the **mptl_t2b.dma_abort** input drives the DMA abort request output **dma_abort**.

6.2.4.1.3.2 Full MPTL Simulation

During full MPTL simulation, the **mptl_if_bridge_wrap** component is instantiated. Table 169 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_bridge/mptl_if_bridge_wrap_6tl.vhd
ADM-XRC-6T1	admxrc6t1/mptl_bridge/mptl_if_bridge_wrap_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/mptl_bridge/mptl_if_bridge_wrap_6tge.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mptl_bridge/mptl_if_bridge_wrap_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/mptl_bridge/mptl_if_bridge_wrap_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/mptl_bridge/mptl_if_bridge_wrap_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/mptl_bridge/mptl_if_bridge_wrap_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/mptl_bridge/mptl_if_bridge_wrap_admxrc7v1.vhd

Table 169 : Available variants of mptl_if_bridge_wrap component

During full MPTL simulation, the **mptl_if_bridge_wrap** component instantiates the MPTL interface HDL netlist appropriate to the model in use. Table 170 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_bridge/mptl_if_bridge_netlist_wrap_6tl.vhd
ADM-XRC-6T1	admxrc6t1/mptl_bridge/mptl_if_bridge_netlist_wrap_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/mptl_bridge/mptl_if_bridge_netlist_wrap_6tge.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mptl_bridge/mptl_if_bridge_netlist_wrap_admxrc6tda1.vhd
ADPE-XRC-6T	adpexrc6t/mptl_bridge/mptl_if_bridge_netlist_wrap_adpexrc6t.vhd
ADPE-XRC-6T-L	adpexrc6tl/mptl_bridge/mptl_if_bridge_netlist_wrap_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/mptl_bridge/mptl_if_bridge_netlist_wrap_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/mptl_bridge/mptl_if_bridge_netlist_wrap_admxrc7v1.vhd

Table 170 : Available variants of bridge MPTL interface netlist

Clock Generation

- During full MPTL simulation, the bridge MPTL interface OCP clock may be independent of the target MPTL interface OCP clock.
- The **ocp_clk_in** input provides the independent OCP clock generated by the testbench.
- The **ocp_clk_out** output is driven by the **ocp_clk_in** signal.

Initialisation

- The **mptl_online** output is produced from the **mptl_sb_b2t.mptl_bridge_gtp_online_l**, **mptl_sb_t2b.mptl_target_configured_l**, and **mptl_sb_t2b.mptl_target_gtp_online_l** signals.

The **mptl_if_bridge_wrap** component signals are connected to their equivalents on the MPTL interface HDL netlist.

6.2.4.2 Host PCIe Interface Wrapper (pcie_if_host_wrap)

6.2.4.2.1 Introduction

This is a component in the **ADB3_Target** group. It is used by the example FPGA testbenches as the host end of the PCIe interface.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 Target Include Package (adb3_target_inc_pkg)

6.2.4.2.2 Interface

The **pcie_if_host_wrap** component interface is shown in Figure 78 below and described in Table 171.

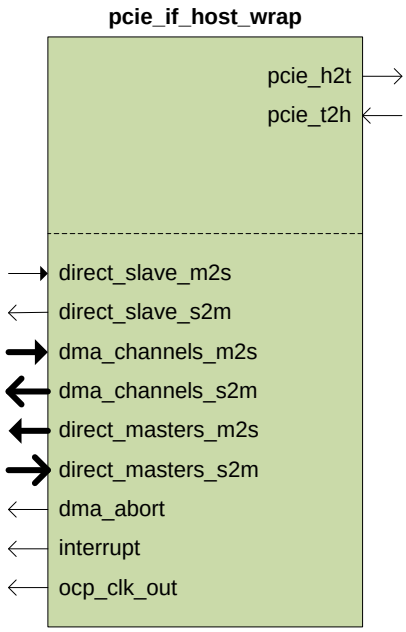

Figure 78 : pcie_if_host_wrap component interface

Signal	Type	Description
		OCP Interface
direct_slave_m2s	Input	Direct slave OCP channel master (to target via PCIe interface).
direct_slave_s2m	Output	Direct slave OCP channel slave (from target via PCIe interface.
dma_channels_m2s	Input	DMA OCP channels master (to target via PCIe interface).
dma_channels_s2m	Output	DMA OCP channels slave (from target via PCIe interface).
direct_masters_m2s	Output	Direct master OCP channels master (from target via PCIe interface).
direct_masters_s2m	Input	Direct master OCP channels slave (to target via PCIe interface).
dma_abort	Output	DMA abort request (to testbench).
interrupt	Output	Interrupt request (to testbench).

Table 171 : pcie_if_host_wrap component interface (continued on next page)

Signal	Type	Description
ocp_clk_out	Output	OCP clock (to testbench).
		PCIe Interface
pcie_t2h	Input	PCIe serial interface signals (from target).
pcie_h2t	Output	PCIe serial interface signals (to target).

Table 171 : pcie_if_host_wrap component interface

6.2.4.2.3 Description

The PCIe interface signals **pcie_t2h** and **pcie_h2t** connect the host and target PCIe interfaces. They are of types **pcie_t2h_t**/**pcie_h2t_t** which are defined in the adb3_target_inc_pkg package. During OCP-only simulation, these signals transfer OCP transactions directly between the host and target PCIe interfaces. During synthesis, these signals transfer PCIe serial data between the host and target PCIe interface.

The type of host PCIe interface that is instantiated depends upon whether simulation or synthesis is required as follows:

6.2.4.2.3.1 OCP-Only Simulation

During OCP-only simulation, a simulation only version of the **pcie_if_host_wrap** component is instantiated. Table 172 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6T-ADV8	admxrc6tadv8/pcie_host/pcie_if_host_wrap_sim_6tadv8.vhd

Table 172 : Available variants of simulation only version of pcie_if_host_wrap component

Clock Generation

- During OCP-only simulation, the host OCP clock must be the same as the target PCIe interface OCP clock. This is accomplished by connecting the target clock to the host clock via the **pcie_t2h.target_ocp_clk** signal.
- The **ocp_clk_out** output is driven by **pcie_t2h.target_ocp_clk**.

PCIe Interface

- The direct slave OCP channel master input **direct_slave_m2s** drives the **pcie_b2t.direct_slave_m2s** output to the target PCIe interface. The **pcie_t2h.direct_slave_s2m** input from the target PCIe interface drives the direct slave OCP channel slave output **direct_slave_s2m**.
- The DMA OCP channels master input **dma_channels_m2s** drives the **pcie_b2t.dma_channels_m2s** output to the target PCIe interface. The **pcie_t2h.dma_channels_s2m** input from the target PCIe interface drives the DMA OCP channels slave output **dma_channels_s2m**.
- The direct master OCP channels slave input **direct_masters_s2m** drives the **pcie_b2t.direct_masters_s2m** output to the target PCIe interface. The **pcie_t2h.direct_masters_m2s** input from the target PCIe interface drives the direct master OCP channels master output **direct_masters_m2s**.
- The general purpose i/o bus **gpio_b2t** input drives the **pcie_b2t.gpio_b2t** output to the target PCIe interface. The **pcie_t2h.gpio_t2b** input from the target PCIe interface drives the general purpose i/o bus output **gpio_t2b**.

DMA Abort

- The **pcie_t2h.dma_abort** input from the target PCIe interface drives the DMA abort request output

dma_abort.

Interrupt

- The **pcie_t2h.interrupt** input from the target PCIe interface drives the interrupt request output **interrupt**.

6.2.4.3 Board Clock Generation and Test (test_board_clks)

6.2.4.3.1 Introduction

This is a component in the **ADB3_Target** group. It is used by the example FPGA testbenches for board clock generation and test.

Dependencies

- ADB3 Target Include Package (adb3_target_inc_pkg)
- ADB3 Target Testbench Include Package (adb3_target_tb_inc_pkg)

6.2.4.3.2 Interface

The **test_board_clks** component interface is shown in Figure 79 below and described in Table 173.

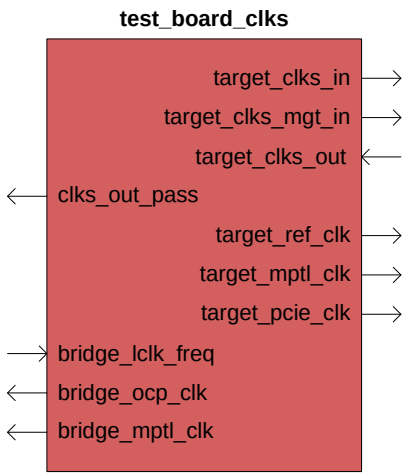

Figure 79 : test_board_clks component interface

Signal	Type	Description
clks_out_freq	Generic	Target clks_out expected frequencies.

Signal	Type	Description
		Target Interface
target_clks_in	Output	Target clks_in (to target)
target_clks_mgt_in	Output	Target clks_mgt_in (to target).
target_clks_out	Input	Target clks_out (from target).
clks_out_pass	Output	Target clks_out test result (to testbench).
target_ref_clk	Output	Target reference clock (to target).
target_mptl_clk	Output	Target MPTL clock (to target).
target_pcie_clk	Output	Target PCIe clock (to target).
		Bridge Interface

Table 173 : test_board_clks component interface (continued on next page)

Signal	Type	Description
bridge_lclk_freq	Input	Bridge lclk frequency (real) (from testbench).
bridge_ocp_clk	Output	Bridge OCP clock (to testbench).
bridge_mptl_clk	Output	Bridge MPTL clock (to testbench).

Table 173 : test_board_clks component interface

6.2.4.3.3 Description

This component is used by example target FPGA testbenches for the following functions:

- Generation of target FPGA clock inputs **clks_in** and **clks_mgt_in** from clocks present on the model in use.
- Test of target FPGA clock outputs **clks_out** using generic input **clks_out_freq** for the model in use.
- Generation of target FPGA reference, MPTL and PCIe clock inputs for the model in use. These are used by designs that do not use **clks_in** and **clks_mgt_in** as their clock sources.
- Generation of Bridge (testbench) OCP clock for the model in use.
- Generation of Bridge (testbench) MPTL clock for the model in use.

Table 174 lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	admxrc6tl/mptl_bridge/test_board_clks_6tl.vhd
ADM-XRC-6T1	admxrc6t1/mptl_bridge/test_board_clks_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/mptl_bridge/test_board_clks_6tge.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/pcie_host/test_board_clks_6tadv8.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mptl_bridge/test_board_clks_admxrc6tda1.vhd
ADPE-XRC-6T	admxrc6t1/mptl_bridge/test_board_clks_adpexrc6t.vhd
ADPE-XRC-6T-L	admxrc6t1/mptl_bridge/test_board_clks_adpexrc6tl.vhd
ADM-XRC-7K1	admxrc7k1/mptl_bridge/test_board_clks_admxrc7k1.vhd
ADM-XRC-7V1	admxrc7v1/mptl_bridge/test_board_clks_admxrc7v1.vhd

Table 174 : Available variants of test_board_clks component

6.3 ADB3 Probe

The ADB3 Probe group is located in the **hdl/vhdl/common/adb3_probe/** directory and contains the following elements:

- ADB3 Probe Package (adb3_probe_pkg)
- ADB3 Probe Components

6.3.1 ADB3 Probe Package (adb3_probe_pkg)

The package **adb3_probe_pkg** defines constants and types which are used by the ADB3 probe components.

Definitions are as follows:

- **adb3_ocp_probe_status_r**. A record type containing probe status elements.
- **ADB3_OCP_PROBE_STATUS_OK**. A constant for probe status with no errors.
- adb3_ocp_transaction_probe component definition.

6.3.2 ADB3 Probe Components

6.3.2.1 adb3_ocp_transaction_probe

6.3.2.1.1 Introduction

This is a component in the ADB3 probe group. Its function is to monitor an OCP channel and produce warnings/errors if specific conditions occur. It is used by target example FPGA testbenches.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 Probe Package (adb3_probe_pkg)

6.3.2.1.2 Interface

The **adb3_ocp_transaction_probe** component interface is shown in Figure 80 below and described in Table 175.

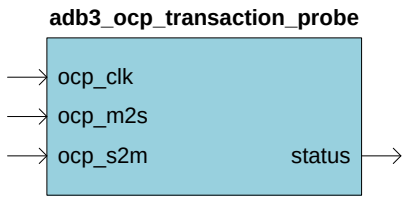

Figure 80 : adb3_ocp_transaction_probe component interface

Signal	Type	Description
enable_logging	Generic	Enable use of log file for info/warnings/errors.
sel_int_log_file	Generic	Select between internal name and external name for log file.
int_log_filename	Generic	Internal filename for log file if selected and enabled.
addr_align_bits	Generic	Set number of unused address LSBs for checking.
addr_width_max	Generic	Set maximum address width for checking.
data_burst_max	Generic	Set maximum burst length for checking.
enable_tag_check	Generic	Enable checking of OCP_CMD_READ tag with read data tag.

Signal	Type	Description
		OCP Port
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP port M2S monitor connection.
ocp_s2m	Input	OCP port S2M monitor connection.
		Status
status	Output	Probe status.

Table 175 : adb3_ocp_transaction_probe component interface

6.3.2.1.3 Description

This component checks for the following conditions:

- Read data with incorrect tag for active read command (**enable_tag_check** generic).
- Read data for read command which has completed.
- Write data for write command which has completed.
- Write data with invalid (all zero) DataByteEn value.
- Invalid command detection.
- Invalid address alignment detection (**addr_align_bits** generic).
- Invalid address detection (**addr_width_max** generic).
- Invalid burst length detection (**data_burst_max** generic).
- Invalid response detection.

The above conditions are flagged using the **status** output of type **adb3_ocp_probe_status_r**.

6.4 Memory Interface (Deprecated)

The memory interface group is located in the **hdl/vhdl/common/mem_if/** directory and contains the following elements:

- Xilinx DDR3 SDRAM MIG Cores (Deprecated)
- Memory Interface Package (mem_if_pkg) (Deprecated)
- Memory Interface Components (Deprecated)

6.4.1 DDR3 SDRAM MIG Cores (Deprecated)

These cores are generated by the Xilinx Core Generator software and are instantiated by the Memory Interface Components (Deprecated). The user is required to generate the appropriate DDR3 SDRAM MIG core files, using a TCL script, prior to simulation or synthesis. The core versions supported by the Memory Interface group and their availability in Xilinx ISE is as follows:

MIG Version	ISE Versions
MIG v3.6	ISE 12.3 to ISE 13.2.

Table 176 : DDR3 SDRAM MIG Core And ISE Version Compatibility

The core version selected for simulation is defined in the appropriate simulation script for the model in use.

The core version selected for synthesis is defined in the appropriate synthesis .prj file for the model in use.

Note
This version of the SDK uses MIG6 v3.9 DDR3 SDRAM MIG Cores for simulation and synthesis.

The files for each supported core version are generated using a different TCL generation script. Table 177 lists the scripts available:

MIG Version	Generation Script
MIG v3.6	%ADMXRC3_SDK%/hdl/vhdl/common/mem_if/gen_mem_if.tcl

Table 177 : DDR3 SDRAM MIG core generation scripts (deprecated)

Examples are as follows:

To generate MIG v3.6 HDL files for an ADM-XRC-6T1 using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\commonµem_if
xtclsh gen_mem_if.tcl admxrc6t1
```

To generate MIG v3.6 HDL files for an ADM-XRC-6T1 using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/common/mem_if
xtclsh ./gen_mem_if.tcl admxrc6t1
```

Xilinx documentation is included with the generated Xilinx DDR3 SDRAM MIG core. For example, after generation of the MIG v3.6 core for ADM-XRC-6T1 models, its documentation can be found in **hdl/vhdl/common/mem_if/ddr3_sdram/admxrc6t1/mig_temp/mig_v3_6/docs/**.

Similarly its VHDL source files can be found in

hdl/vhdl/common/mem_if/ddr3_sdram/admxrc6t1/rtl/mig_v3_6/.

> **Note**
> The TCL script is run using the Xilinx customized TCL distribution TCL shell **xtclsh**. The path to this shell must be defined for successful script execution.

6.4.2 Memory Interface Package (mem_if_pkg)(Deprecated)

The package **mem_if_pkg** contains definitions which are used by the DDR3 SDRAM Interface Bank (ddr3_if_bank)(Deprecated) components. The package is located at **%ADMXRC3_SDK%\hdl\vhdl\common\mem_if\mem_if_pkg.vhd**

The package **mem_if_pkg** defines types, constants, functions and components as follows:

Memory interface functions

- **conv_t_rfc_option**. Returns the value of **t_rfc** in ps that is appropriate for the **DDR3_BANK_ROW_WIDTH** value in the variant of the adb3_target_inc_pkg that has been selected.
- **conv_sim_bypass_init_cal**. Selects simulation init option "FAST" or "OFF".
- **conv_sim_init_option**. Selects simulation init option "SKIP_PU_DLY" or "NONE".
- **conv_sim_cal_option**. Selects simulation init option "FAST_CAL" or "NONE".

MIG DDR3 SDRAM core types

- **mig_clocks_t**. A record type containing clock speed bin generic elements.
- **mig_clocks_common_t**. A record type containing clock common generic elements.
- **mig_common_t**. A record type containing bank common generic elements.
- **mig_dqs_col0_4_t**. A record type containing 4 DQS groups in column 0 generic elements.
- **mig_dqs_col2_4_t**. A record type containing 4 DQS groups in column 2 generic elements.
- **mig_dqs_col01_2_t**. A record type containing 2 DQS groups in columns 0 and 1 generic elements.

MIG DDR3 SDRAM core constants

- **MIG_CLOCKS_800**. Constant of type **mig_clocks_t** defining clock DDR3-800 speed bin generics.
- **MIG_CLOCKS_COMMON**. Constant of type **mig_clocks_common_t** defining clock common generics.
- **MIG_COMMON**. Constant of type **mig_common_t** defining bank common generics.
- **MIG_DQS_COL0_4**. Constant of type **mig_dqs_col0_4_t** defining DQS groups common generics.
- **MIG_DQS_COL2_4**. Constant of type **mig_dqs_col2_4_t** defining DQS groups common generics.
- **MIG_DQS_COL01_2**. Constant of type **mig_dqs_col01_2_t** defining DQS groups common generics.

Component definitions

- ddr3_if_bank (Deprecated)

6.4.3 Memory Interface Components (Deprecated)

6.4.3.1 DDR3 SDRAM Interface Bank (ddr3_if_bank)(Deprecated)

6.4.3.1.1 Introduction

This is a component in the memory interface group. Its function is as follows:

- Conversion of transactions on a single OCP channel to DDR3 SDRAM MIG bank user interface transactions.
- Instantiation of a single bank Xilinx DDR3 SDRAM MIG core.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)
- ADB3 Target Include Package (adb3_target_inc_pkg)
- Memory Interface Package (mem_if_pkg)(Deprecated)

6.4.3.1.2 Interface

The **ddr3_if_bank** component interface is shown in Figure 81 below and described in Table 178.

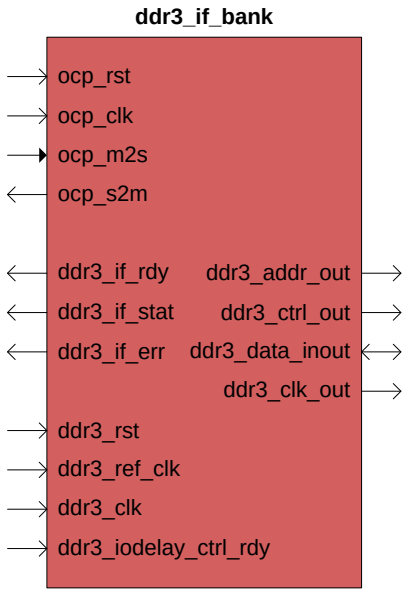

Figure 81 : ddr3_if_bank component interface (deprecated)

Signal	Type	Description
sim	Generic	Simulation select.
bank	Generic	Bank select.

Signal	Type	Description
		OCP Port
ocp_rst	Input	OCP asynchronous reset..
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP port M2S connection.
ocp_s2m	Output	OCP port S2M connection.
		DDR3 SDRAM MIG Core Bank Control/Status
ddr3_rst	Input	MIG core asynchronous reset.
ddr3_clk	Input	MIG core clock.
ddr3_ref_clk	Input	MIG core reference clock.
ddr3_iodelay_ctrl_rdy	Input	MIG core IO delay ready.
ddr3_if_rdy	Output	MIG core ready.
ddr3_if_stat	Output	MIG core status.
ddr3_if_err	Output	MIG core error.
		DDR3 SDRAM Bank Physical Interface
ddr3_addr_out	Output	Bank address.
ddr3_ctrl_out	Output	Bank control.
ddr3_data_inout	Bi-dir	Bank data.
ddr3_clk_out	Output	Bank clocks.

Table 178 : ddr3_if_bank component interface (deprecated)

6.4.3.1.3 Description

This component converts transactions on a single OCP channel to DDR3 SDRAM MIG bank user interface transactions and instantiates a single bank Xilinx DDR3 SDRAM MIG core.

The **ddr3_if_bank** component instantiated depends on the model and MIG version in use. Table 179 lists the available variants:

Model	MIG Version	Filename relative to hdl/vhdl/common/mem_if/ddr3_sdram/
ADM-XRC-6TL	MIG v3.6	admxrc6tl/ddr3_if_bank_6tl.vhd
ADM-XRC-6T1	MIG v3.6	admxrc6t1/ddr3_if_bank_6t1.vhd
ADM-XRC-6TGE	MIG v3.6	admxrc6tge/ddr3_if_bank_6tge.vhd
ADM-XRC-6T-ADV8	MIG v3.6	admxrc6tadv8/ddr3_if_bank_6tadv8.vhd
ADM-XRC-6T-DA1	n/a	n/a
ADPE-XRC-6T	n/a	n/a
ADPE-XRC-6T-L	n/a	n/a
ADM-XRC-7K1	n/a	n/a
ADM-XRC-7V1	n/a	n/a

Table 179 : Available variants of ddr3_if_bank component (deprecated)

It includes the following components:

- adb3_ocp_ocp2ddr3_nb
- DDR3 SDRAM MIG Cores (Deprecated)

6.4.3.1.3.1 adb3_ocp_ocp2ddr3_nb

This component converts ADB3 OCP transactions to DDR3 SDRAM MIG core user interface transactions. It is implemented using the ADB3 OCP component adb3_ocp_ocp2ddr3_nb.

6.4.3.1.3.2 DDR3 SDRAM MIG Core

This component instantiates a single bank Xilinx DDR3 SDRAM MIG core which has been generated using the Xilinx Core Generator MIG tool. Refer to DDR3 SDRAM MIG Cores (Deprecated) for details of the generation procedure.

The component instantiated depends on the bank selected by the **bank** generic. For example, on the ADM-XRC-6T1, **c0_memc_ui_top.vhd** located in **hdl/vhdl/common/mem_if/ddr3_sdram/admxrc6t1/rtl/ip_top/** is used when bank = 0.

The core uses the **MIG_SIM_BYPASS_INIT_CAL**, **MIG_SIM_INIT_OPTION**, and **MIG_SIM_CAL_OPTION** constants which are defined differently for simulation and synthesis. The differing definitions are controlled using synthesis directives. This enables a much shorter initialisation procedure to be used during simulation.

6.5 DDR3 SDRAM Interface

The DDR3 SDRAM interface group is located in the **hdl/vhdl/common/ddr3_sdram_if/** directory and contains the following elements:

- DDR3 SDRAM MIG Cores
- DDR3 SDRAM Interface Package (ddr3_if_pkg)
- DDR3 SDRAM Interface Components

6.5.1 DDR3 SDRAM MIG Cores

These cores are generated by the Xilinx Core Generator software and are instantiated by the DDR3 SDRAM Interface Components. The user is required to generate the appropriate DDR3 SDRAM MIG core files, using a TCL script, prior to simulation or synthesis. The core versions supported by the DDR3 SDRAM Interface group and their availability in Xilinx ISE is as follows:

MIG Version	ISE Versions
MIG6 v3.9	ISE 13.3 - 14.2.
MIG7 v1.4	ISE 13.4 - 14.2.

Table 180 : DDR3 SDRAM MIG Core And ISE Version Compatibility

The core version selected for simulation is defined by the **VAR_MIG** variable in the appropriate simulation script for the model in use.

The core version selected for synthesis is defined in the appropriate synthesis .prj file for the model in use.

Note
This version of the SDK uses MIG6 v3.9 DDR3 SDRAM MIG cores for Virtex-6 model simulation and synthesis, and MIG7 v1.4 DDR3 SDRAM MIG cores for 7 Series model simulation and synthesis.

The files for each supported core version are generated using a different TCL generation script. Table 181 lists the scripts available:

MIG Version	Generation Script
MIG6 v3.9	%ADMXRC3_SDK%/hdl/vhdl/common/ddr3_sdram_if/gen_ddr3_if_mig6_v3_9.tcl
MIG7 v1.4	%ADMXRC3_SDK%/hdl/vhdl/common/ddr3_sdram_if/gen_ddr3_if_mig7_v1_4.tcl

Table 181 : DDR3 SDRAM MIG core generation scripts

Examples are as follows:

To generate MIG6 v3.9 HDL files for an ADM-XRC-6T1 using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\common°dr3_sdram_if
xtclsh gen_ddr3_if_mig6_v3_9.tcl admxrc6t1
```

To generate MIG6 v3.9 HDL files for an ADM-XRC-6T1 using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/common/ddr3_sdram_if
xtclsh ./gen_ddr3_if_mig6_v3_9.tcl admxrc6t1
```

Xilinx documentation is included with the generated Xilinx DDR3 SDRAM MIG core. For example, after generation of the MIG6 v3.9 core for ADM-XRC-6T1 models, its documentation can be found in **hdl/vhdl/common/ddr3_sdram_if/admxrc6t1/mig6_v3_9/mig_temp/mig_v3_9/docs/**.

Similarly its VHDL source files can be found in **hdl/vhdl/common/ddr3_sdram_if/admxrc6t1/mig6_v3_9/rtl/mig_v3_9/**.

Note
The TCL script is run using the Xilinx customized TCL distribution TCL shell **xtclsh**. The path to this shell must be defined for successful script execution.

6.5.2 DDR3 SDRAM Interface Package (ddr3_if_pkg)

The package **ddr3_if_pkg** contains definitions which are used by the DDR3 SDRAM Interface Bank (ddr3_if_bank) components. Different models use different versions of the package. Table 182 lists the versions currently used:

Models	Package Version
Virtex-6	%ADMXRC3_SDK%/hdl/vhdl/common/ddr3_sdram_if/ddr3_if_pkg_v6.vhd
Kintex-7	%ADMXRC3_SDK%/hdl/vhdl/common/ddr3_sdram_if/ddr3_if_pkg_k7.vhd
Virtex-7	%ADMXRC3_SDK%/hdl/vhdl/common/ddr3_sdram_if/ddr3_if_pkg_v7.vhd

Table 182 : DDR3 SDRAM interface packages

6.5.2.1 Virtex-6 DDR3 SDRAM Interface Package (ddr3_if_pkg)

The Virtex-6 version of the package **ddr3_if_pkg** defines types, constants, functions and components as follows:

Memory interface functions

- **conv_t_rfc_option**. Returns the value of **t_rfc** in ps that is appropriate for the **DDR3_BANK_ROW_WIDTH** value in the variant of the adb3_target_inc_pkg that has been selected.
- **conv_sim_bypass_init_cal**. Selects simulation init option "FAST" or "OFF".
- **conv_sim_init_option**. Selects simulation init option "SKIP_PU_DLY" or "NONE".
- **conv_sim_cal_option**. Selects simulation init option "FAST_CAL" or "NONE".

MIG DDR3 SDRAM core types

- **mig_clocks_t**. A record type containing clock speed bin generic elements.
- **mig_clocks_common_t**. A record type containing clock common generic elements.
- **mig_common_t**. A record type containing bank common generic elements.
- **mig_dqs_col0_4_t**. A record type containing 4 DQS groups in column 0 generic elements.
- **mig_dqs_col1_4_t**. A record type containing 4 DQS groups in column 1 generic elements.
- **mig_dqs_col2_4_t**. A record type containing 4 DQS groups in column 2 generic elements.
- **mig_dqs_col01_2_t**. A record type containing 2 DQS groups in columns 0 and 1 generic elements.

MIG DDR3 SDRAM core constants

- **MIG_CLOCKS_800**. Constant of type **mig_clocks_t** defining clock DDR3-800 speed bin generics.
- **MIG_CLOCKS_COMMON**. Constant of type **mig_clocks_common_t** defining clock common generics.
- **MIG_COMMON**. Constant of type **mig_common_t** defining bank common generics.

- **MIG_DQS_COL0_4**. Constant of type **mig_dqs_col0_4_t** defining DQS groups common generics.
- **MIG_DQS_COL1_4**. Constant of type **mig_dqs_col1_4_t** defining DQS groups common generics.
- **MIG_DQS_COL2_4**. Constant of type **mig_dqs_col2_4_t** defining DQS groups common generics.
- **MIG_DQS_COL01_2_1032**. Constant of type **mig_dqs_col01_2_t** defining DQS groups common generics.
- **MIG_DQS_COL01_2_2031**. Constant of type **mig_dqs_col01_2_t** defining DQS groups common generics.

Component definitions

- ddr3_if_bank

6.5.2.2 Kintex-7 DDR3 SDRAM Interface Package (ddr3_if_pkg)

The Kintex-7 version of the package **ddr3_if_pkg** defines types, constants, functions and components as follows:

Memory interface functions

- **conv_clock_option**. Returns the value of the speed bin generics in use.
- **conv_t_rfc_option**. Returns the value of **t_rfc** in ps that is appropriate for the **DDR3_BANK_ROW_WIDTH** value in the variant of the adb3_target_inc_pkg that has been selected.
- **conv_t_rrd_option**. Returns the value of **t_rrd** in ps that is appropriate for the speed bin generics in use.
- **conv_t_faw_option**. Returns the value of **t_faw** in ps that is appropriate for the speed bin generics in use.
- **conv_cl_option**. Returns the value of **CL** that is appropriate for the speed bin generics in use.
- **conv_cwl_option**. Returns the value of **CWL** that is appropriate for the speed bin generics in use.
- **conv_sim_bypass_init_cal**. Selects simulation init option "FAST" or "OFF".

MIG DDR3 SDRAM core types

- **mig_clocks_t**. A record type containing clock speed bin generic elements.
- **mig_clocks_common_t**. A record type containing clock common generic elements.
- **mig_common_t**. A record type containing bank common generic elements.
- **mig_byte_lanes_t**. A record type containing bank common byte lane generic elements.
- **mig_data_ctl_t**. A record type containing bank common data control generic elements.
- **phy_bitlanes_t**. A record type containing bank common phy bitlanes generic elements.
- **mig_map_t**. A record type containing signal mapping generic elements.
- **mig_map_array_t**. A bank array of **mig_map_t** elements.

MIG DDR3 SDRAM core constants

- **MIG_CLOCKS_800**. Constant of type **mig_clocks_t** defining clock DDR3-800 speed bin generics.
- **MIG_CLOCKS_1600**. Constant of type **mig_clocks_t** defining clock DDR3-1600 speed bin generics.
- **MIG_CLOCKS_COMMON**. Constant of type **mig_clocks_common_t** defining clock common generics.
- **MIG_COMMON**. Constant of type **mig_common_t** defining bank common generics.
- **MIG_BYTE_LANES**. Constant of type **mig_byte_lanes_t** defining bank common byte lane generic elements.
- **MIG_DATA_CTL**. Constant of type **mig_data_ctl_t** defining bank common data control generic elements.
- **MIG_PHY_BITLANES**. Constant of type **phy_bitlanes_t** bank common phy bitlanes generic elements.

Component definitions

- ddr3_if_bank

6.5.2.3 Virtex-7 DDR3 SDRAM Interface Package (ddr3_if_pkg)

The Virtex-7 version of the package **ddr3_if_pkg** defines types, constants, functions and components as follows:

Memory interface functions

- **conv_t_rfc_option**. Returns the value of **t_rfc** in ps that is appropriate for the **DDR3_BANK_ROW_WIDTH** value in the variant of the adb3_target_inc_pkg that has been selected.
- **conv_t_rrd_option**. Returns the value of **t_rrd** in ps that is appropriate for the speed bin generics in use.
- **conv_t_faw_option**. Returns the value of **t_faw** in ps that is appropriate for the speed bin generics in use.
- **conv_sim_bypass_init_cal**. Selects simulation init option "FAST" or "OFF".

MIG DDR3 SDRAM core types

- **mig_clocks_t**. A record type containing clock speed bin generic elements.
- **mig_clocks_common_t**. A record type containing clock common generic elements.
- **mig_common_t**. A record type containing bank common generic elements.
- **mig_byte_lanes_t**. A record type containing bank common byte lane generic elements.
- **mig_data_ctl_t**. A record type containing bank common data control generic elements.
- **phy_bitlanes_t**. A record type containing bank common phy bitlanes generic elements.
- **mig_map_t**. A record type containing signal mapping generic elements.
- **mig_map_array_t**. A bank array of **mig_map_t** elements.

MIG DDR3 SDRAM core constants

- **MIG_CLOCKS_800**. Constant of type **mig_clocks_t** defining clock DDR3-800 speed bin generics.
- **MIG_CLOCKS_1600**. Constant of type **mig_clocks_t** defining clock DDR3-1600 speed bin generics.
- **MIG_CLOCKS_COMMON**. Constant of type **mig_clocks_common_t** defining clock common generics.
- **MIG_COMMON**. Constant of type **mig_common_t** defining bank common generics.
- **MIG_BYTE_LANES**. Constant of type **mig_byte_lanes_t** defining bank common byte lane generic elements.
- **MIG_DATA_CTL**. Constant of type **mig_data_ctl_t** defining bank common data control generic elements.

Component definitions

- ddr3_if_bank

6.5.3 DDR3 SDRAM Interface Components

6.5.3.1 DDR3 SDRAM Interface Bank (ddr3_if_bank)

6.5.3.1.1 Introduction

This is a component in the DDR3 SDRAM interface group. Its function is as follows:

- Conversion of transactions on a single OCP channel to DDR3 SDRAM MIG bank user interface transactions.
- Instantiation of a single bank DDR3 SDRAM MIG core or 'light weight' behavioural model.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- ADB3 OCP Component Declaration Package (adb3_ocp_comp)
- ADB3 Target Include Package (adb3_target_inc_pkg)
- DDR3 SDRAM Interface Package (ddr3_if_pkg)

6.5.3.1.2 Interface

The **ddr3_if_bank** component interface is shown in Figure 82 below and described in Table 183.

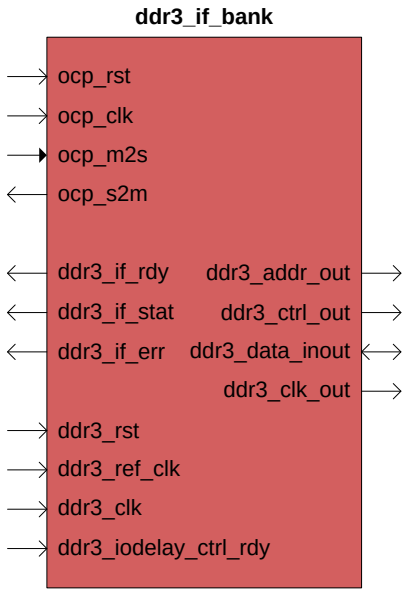

Figure 82 : ddr3_if_bank component interface

Signal	Type	Description
sim	Generic	Simulation select.
bank	Generic	Bank select.

Signal	Type	Description
		OCP Port
ocp_rst	Input	OCP asynchronous reset..
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP port M2S connection.
ocp_s2m	Output	OCP port S2M connection.
		DDR3 SDRAM MIG Core Bank Control/Status
ddr3_rst	Input	MIG core asynchronous reset.
ddr3_clk	Input	MIG core clock.
ddr3_ref_clk	Input	MIG core reference clock.
ddr3_iodelay_ctrl_rdy	Input	MIG core IO delay ready.
ddr3_if_rdy	Output	MIG core ready.
ddr3_if_stat	Output	MIG core status.
ddr3_if_err	Output	MIG core error.
		DDR3 SDRAM Bank Physical Interface
ddr3_addr_out	Output	Bank address.
ddr3_ctrl_out	Output	Bank control.
ddr3_data_inout	Bi-dir	Bank data.
ddr3_clk_out	Output	Bank clocks.

Table 183 : ddr3_if_bank component interface

6.5.3.1.3 Description

The **ddr3_if_bank** component instantiated depends on the model and MIG version in use. Table 184 lists the available variants:

Model	Filename relative to hdl/vhdl/common/ddr3_sdram_if/
	Light Weight Behavioural
All models	lwb/ddr3_if_bank_lwb.vhd
	Xilinx MIG6 v3.9
ADM-XRC-6TL	admxrc6tl/mig6_v3_9/ddr3_if_bank_admxrc6tl_mig6_v3_9.vhd
ADM-XRC-6T1	admxrc6t1/mig6_v3_9/ddr3_if_bank_admxrc6t1_mig6_v3_9.vhd
ADM-XRC-6TGE	admxrc6tge/mig6_v3_9/ddr3_if_bank_admxrc6tge_mig6_v3_9.vhd
ADM-XRC-6T-ADV8	admxrc6tadv8/mig6_v3_9/ddr3_if_bank_admxrc6tadv8_mig6_v3_9.vhd
ADM-XRC-6T-DA1	admxrc6tda1/mig6_v3_9/ddr3_if_bank_admxrc6tda1_mig6_v3_9.vhd
ADPE-XRC-6T	adpexrc6t/mig6_v3_9/ddr3_if_bank_adpexrc6t_mig6_v3_9.vhd
ADPE-XRC-6T-L	adpexrc6tl/mig6_v3_9/ddr3_if_bank_adpexrc6tl_mig6_v3_9.vhd
	Xilinx MIG7 v1.4
ADM-XRC-7K1	admxrc7k1/mig7_v1_4/ddr3_if_bank_admxrc7k1_mig7_v1_4.vhd
ADM-XRC-7V1	admxrc7v1/mig7_v1_4/ddr3_if_bank_admxrc7v1_mig7_v1_4.vhd

Table 184 : Available variants of ddr3_if_bank component

6.5.3.1.3.1 Light Weight Behavioural Variant

This variant can be used to increase simulation speed and is not suitable for synthesis. It includes the following components:

- adb3_ocp_ocp2ddr3_nb
- DDR3 SDRAM MIG Core Light Weight Behavioural Model (mig_ddr3_lwb)

OCP transactions on a single OCP channel are converted to DDR3 SDRAM MIG bank user interface transactions by the adb3_ocp_ocp2ddr3_nb component. These are passed to a behavioural model of the DDR3 SDRAM MIG core and DDR3 SDRAM on-board memory parts.

The **ddr3_data_inout.dqs_p(0)** and **ddr3_data_inout.dqs_n(0)** signals interface to the DDR3 SDRAM Model (ddr3_sdram) 'light weight' behavioural variant. They are used to drive the **init_start** and **log_start** inputs to the behavioural model.

The **INIT_FILENAME** and **LOG_FILENAME** constants define the filenames of the DDR3 SDRAM initialisation and log files respectively .

6.5.3.1.3.2 Xilinx MIG Variants

These variants can be used during simulation or synthesis. They include the following components:

- adb3_ocp_ocp2ddr3_nb
- DDR3 SDRAM MIG Cores

OCP transactions on a single OCP channel are converted to DDR3 SDRAM MIG bank user interface transactions by the adb3_ocp_ocp2ddr3_nb component. These are passed to a DDR3 SDRAM MIG Core which interfaces to the DDR3 SDRAM on-board memory parts.

Bank specific actions are performed in the component using the **bank** generic.

The DDR3 SDRAM MIG Core uses the **MIG_SIM_BYPASS_INIT_CAL**, **MIG_SIM_INIT_OPTION**, and **MIG_SIM_CAL_OPTION** constants which are defined differently for simulation and synthesis. The differing definitions are controlled using the **sim** generic and synthesis directives during instantiation. This enables a much shorter initialisation procedure to be used during simulation.

6.5.3.2 DDR3 SDRAM MIG Core Light Weight Behavioural Model (mig_ddr3_lwb)

6.5.3.2.1 Introduction

This is a component in the DDR3 SDRAM interface group. Its function is as follows:

- Light weight behavioural model of DDR3 SDRAM MIG core and DDR3 SDRAM on-board memory parts

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)
- DDR3 SDRAM Interface Package (ddr3_if_pkg)
- DDR3 SDRAM Model Internal Package (ddr3_sdram_int_pkg)

6.5.3.2.2 Interface

The **mig_ddr3_lwb** component interface is shown in Figure 83 below and described in Table 185.

Figure 83 : mig_ddr3_lwb component interface

Signal	Type	Description
message_level	Generic	Select message reporting level (default = 0).
addr_width	Generic	Width of **app_addr** input.

Signal	Type	Description
rst	Input	Asynchronous reset..
clk	Input	Clock.
		MIG user interface inputs
app_addr	Input	User interface command address (aligned to bank data width).
app_cmd	Input	User interface command.
app_en	Input	User interface command enable.
app_sz	Input	User interface command 1/2 cycle select.
app_wdf_data	Input	User interface write command data.
app_wdf_end	Input	User interface write command data end.
app_wdf_mask	Input	User interface write command data mask (active low).
app_wdf_wren	Input	User interface write command data enable.
		MIG user interface outputs
dfi_init_complete	Output	User interface phy calibration complete.
app_rd_data	Output	User interface read command data.
app_rd_data_valid	Output	User interface read command data valid.
app_rdy	Output	User interface command ready.
app_wdf_rdy	Output	User interface write data ready.
		File control
init_start	Input	Load data initialisation file (default = false).
init_filename	Input	Initialisation file name (default = "init.txt").
log_start	Input	Save data log file (default = false).
log_filename	Input	Log file name (default = "log.txt").

Table 185 : mig_ddr3_lwb component interface

6.5.3.2.3 Description

6.5.3.2.3.1 MIG User Interface FIFO

A synchronous FIFO buffers MIG user interface commands and write data. Illegal commands will generate an error. The FIFO full output is used to generate the user interface **app_rdy** and **app_wdf_rdy** outputs.

6.5.3.2.3.2 MIG User Interface State Machine

A state machine is used to check the sequencing of buffered user interface commands. It generates a read enable **read_en**, write enable **write_en** and FIFO read advance **mig_cmd_adv** depending on the command sequence. Illegal commands or sequencing will generate an error.

6.5.3.2.3.3 Memory log

Memory contents are written to **log_filename** on **log_start**. Refer to Memory Contents Initalisation for log file format.

6.5.3.2.3.4 Memory init

Memory contents are read from **init_filename** on **init_start**. Refer to Memory Contents Logging for init file format.

6.5.3.2.3.5 Memory write

MIG user interface write data is written to the **memory** array. The memory bank, row, and column written to is indicated by the **memory_wr_bank**, **memory_wr_row**, and **memory_wr_col** signals. The write data and its mask are indicated by the **memory_wr_data**, and **memory_wr_mask** signals.

6.5.3.2.3.6 Memory read

MIG user interface read data is read from the **memory** array. The memory bank, row, and column read from is indicated by the **memory_rd_bank**, **memory_rd_row**, and **memory_rd_col** signals. The read data is indicated by the **memory_rd_data** signal.

6.6 Memory Model

The Memory model group is located in the **hdl/vhdl/common/mem_tb/** directory and contains the following elements:

- DDR3 SDRAM Memory Model

6.6.1 DDR3 SDRAM Memory Model

The DDR3 SDRAM Memory model is located in the **hdl/vhdl/common/mem_tb/ddr3_sdram/** directory and contains the following elements:

- DDR3 SDRAM Model Package (ddr3_sdram_pkg)
- DDR3 SDRAM Model Internal Package (ddr3_sdram_int_pkg)
- DDR3 SDRAM Model Components

6.6.1.1 DDR3 SDRAM Model Package (ddr3_sdram_pkg)

The package **ddr3_sdram_pkg** defines types, constants, and components relating to parts supported by the DDR3 SDRAM model.

Definitions are as follows:

DDR3 SDRAM part types

- **part_size_t**. Record type for different part sizes.
- **speed_grade_cl_cwl_t**. Array type for timing parameters which vary with speed grade, CL, and CWL.
- **speed_grade_t**. Record type for timing parameters which vary with speed grade.
- **part_t**. Record type for overall part used by generic model.

Supported part_size_t constants

- **M8_X_B8_X_D16**. 8Mb Array x 8 banks x 16 data bits = 1Gib part.
- **M16_X_B8_X_D16**. 16Mb Array x 8 banks x 16 data bits = 2Gib part.
- **M32_X_B8_X_D16**. 32Mb Array x 8 banks x 16 data bits = 4Gib part.

Supported speed_grade_cl_cwl_t constants

- **MT41J_125E_CL_CWL_MIN**. Micron MT41JxMx_125E (minimum values).
- **MT41J_125E_CL_CWL_MAX**. Micron MT41JxMx_125E (maximum values).
- **MT41J_15E_CL_CWL_MIN**. Micron MT41JxMx_15E (minimum values).
- **MT41J_15E_CL_CWL_MAX**. Micron MT41JxMx_15E (maximum values).
- **MT41J_187E_CL_CWL_MIN**. Micron MT41JxMx_187E (minimum values).
- **MT41J_187E_CL_CWL_MAX**. Micron MT41JxMx_187E (maximum values).

Supported speed_grade_t constants

- **MT41J_125E**. Micron MT41JxMx_125E.
- **MT41J_15E**. Micron MT41JxMx_15E.
- **MT41J_187E**. Micron MT41JxMx_187E.

Supported part_t constants

- **MT41J64M16_125E**. Micron MT41J64M16_125E (1Gib part).
- **MT41J64M16_15E**. Micron MT41J64M16_15E (1Gib part).

- **MT41J64M16_187E**. Micron MT41J64M16_187E (1Gib part).
- **MT41J128M16_125E**. Micron MT41J128M16_125E (2Gib part).
- **MT41J128M16_15E**. Micron MT41J128M16_15E (2Gib part).
- **MT41J128M16_187E**. Micron MT41J128M16_187E (2Gib part).
- **MT41J256M16_125E**. Micron MT41J256M16_125E (4Gib part).
- **MT41J256M16_15E**. Micron MT41J256M16_15E (4Gib part).
- **MT41J256M16_187E**. Micron MT41J256M16_187E (4Gib part).

Component definitions

- ddr3_sdram

6.6.1.2 DDR3 SDRAM Model Internal Package (ddr3_sdram_int_pkg)

The package **ddr3_sdram_int_pkg** defines types, constants, functions, procedures and components which are used internally by the DDR3 SDRAM model.

6.6.1.3 DDR3 SDRAM Model Components

6.6.1.3.1 DDR3 SDRAM Model (ddr3_sdram)

6.6.1.3.1.1 Introduction

This is a component in the memory model group. Its function is to provide a testbench generic simulation model which may be customised to represent specific DDR3 SDRAM parts.

Dependencies

- DDR3 SDRAM Model Package (ddr3_sdram_pkg)
- DDR3 SDRAM Model Internal Package (ddr3_sdram_int_pkg)

6.6.1.3.1.2 Interface

The **ddr3_sdram** component interface is shown in Figure 84 below and described in Table 186.

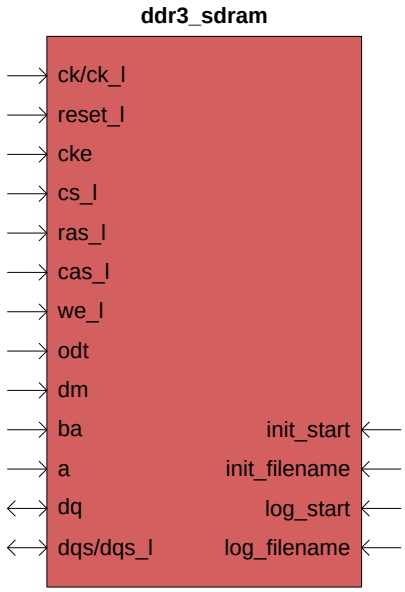

Figure 84 : ddr3_sdram component interface

Signal	Type	Description
message_level	Generic	Select message reporting level (default = 0).
part	Generic	Select component part.
short_init_dly	Generic	Select shortened initialisation sequence.

Signal	Type	Description
		Control/Data
ck+ck_l	Input	Clock (differential).
reset_l	Input	Reset (active low).
cke	Input	Clock enable.
cs_l	Input	Chip select (active low).
ras_l	Input	Row access strobe (active low).
cas_l	Input	Column active strobe (active low).
we_l	Input	Write enable (active low).
odt	Input	On-die termination.
dm	Input	Input data mask.
ba	Input	Bank address.
a	Input	Address.
dq	Bi-dir	Data.
dqs+dqs_l	Bi-dir	Data strobe (differential).
		Init/Log files
init_start	Input	Load data initialisation file (default = false).
init_filename	Input	Initialisation file name (default "init.txt").
log_start	Input	Save data log file (default = false).
log_filename	Input	Log file name (default "log.txt").

Table 186 : ddr3_sdram component interface

6.6.1.3.1.3 Description

The **ddr3_sdram** component instantiated depends on whether the 'light weight' behavioural or Xilinx MIG core DDR3 SDRAM interface is in use. Table 187 lists the available variants:

Variant	Filename
'Light Weight' Behavioural	hdl/vhdl/common/ddr3_sdram_if/lwb/ddr3_sdram_lwb.vhd
Xilinx MIG core	hdl/vhdl/common/mem_tb/ddr3_sdram.vhd

Table 187 : Available variants of ddr3_sdram component

6.6.1.3.1.3.1 Light Weight Behavioural Variant

This is used with the 'light weight' behavioural variant of the DDR3 SDRAM Interface Bank (ddr3_if_bank).

It is a place holder component whose only function is to connect the **init_start** and **log_start** inputs from the testbench to the **dqs(0)** and **dqs_l(0)** signals to the DDR3 SDRAM Interface Bank (ddr3_if_bank).

The **init_filename** and **log_filename** inputs are ignored. Initialisation and logging filenames are defined in the 'light weight' behavioural variant of the DDR3 SDRAM Interface Bank (ddr3_if_bank).

6.6.1.3.1.3.2 Xilinx MIG Variants

This is used with the Xilinx MIG Variants of the DDR3 SDRAM Interface Bank (ddr3_if_bank).

It is instantiated in a testbench to model a specific DDR3 SDRAM part.

6.6.1.3.1.3.2.1 Message Reporting

The generic **message_level** controls the type of 'note' level messages reported by the model. 'warning', 'error', and 'failure' level messages are always reported. Options are as follows:

- 0 - No additional messages.
- 1 - Write additional messages only.
- 2 - Read additional messages only.
- 3 - Info additional messages only.
- 4 - Write and read additional messages.
- 5 - Write and info additional messages.
- 6 - Read and info additional messages.
- 7 - Write and read and info additional messages.

6.6.1.3.1.3.2.2 Initialisation State Machine

This controls the model initialisation procedure. Deviations from the recommended initialisation sequence will generate warning messages of the form **DDR3 SDRAM Init FSM (xx) : message**. The **init_complete** signal becomes true on completion.

The generic **short_init_dly** controls the length of the 200us reset delay and 500us clock enable delay. The length of these delays may be reduced during simulation by setting this generic to 'true'

6.6.1.3.1.3.2.3 Main State Machine

This controls the model command state sequencing. Violations of model command sequencing or timing parameters will generate warning/error messages as appropriate of the form **DDR3 SDRAM Main FSM (xx) : message**.

6.6.1.3.1.3.2.4 Mode Registers

This implements the **MR0**, **MR1**, **MR2**, and **MR3** mode registers. Register bits are decoded and provided to the rest of the model. Out of range values will generate error messages.

6.6.1.3.1.3.2.5 Memory Write/Read

This section controls writing to/reading from the model memory array. When the main state machine decodes a write or read command, mode register and model timing parameters are used to generate writes/reads to/from the appropriate bank, row and column of the memory array.

6.6.1.3.1.3.2.6 Memory Contents Logging

Saving of data to a log file from the model is initiated by an event occuring on the **log_start** input signal which results in it being 'true'. Only memory data that has been initialised or written to is output to the log file.

The log file name is specified by the **log_filename** input signal, and will be located in the same directory as the modelsim macro file in use.

The format of each line in the log file is as follows:

- Start BANK (decimal 0..7).
- Start ROW (decimal 0..8191 1 Gib, 0..16383 2 Gib, 0..32767 4 Gib).
- Start COL (decimal 0..1023).
- Start data BYTE (decimal 0..1 16-bit chip).
- Data Bytes from starting byte (up to 16 bytes per ROW).

An example log file is shown below:

```
0    5  512 0 0x04 0x00 0x05 0x00 0x06 0x00 0x03 0x00 0x08 0x00 0x09 0x00 0x0A 0x00
0x07 0x00
0    5  520 0 0x0C 0x00 0x0D 0x00 0x0E 0x00 0x0B 0x00 0x10 0x00 0x11 0x00 0x12 0x00
0x0F 0x00
0 4104  512 0 0x08 0x00 0x09 0x00 0x0A 0x00 0x07 0x00 0x0C 0x00 0x0D 0x00 0x0E 0x00
0x0B 0x00
0 4104  520 0 0x10 0x00 0x11 0x00 0x12 0x00 0x0F 0x00 0x14 0x00 0x15 0x00 0x16 0x00
0x13 0x00
2    1  511 0 0xU0 0x00 0x00 0x0U 0x55
2    1  514 1 0x55 0x04 0x00 0x05 0x00 0x06 0x00 0x03 0x00 0x08 0x00 0x09 0x00 0x0A
0x00 0x07
2    1  522 1 0x00 0x0C 0x00 0x0D 0x00 0x0E 0x00 0x0B 0x00 0x10 0x00 0x11 0x00 0x12
0x00 0x0F
2    1  546 1 0x00
2    1 1023 0 0x77 0x77
2    2    0 0 0x99 0x99
2    2  511 0 0xU0 0x66 0x66 0x0U 0xAA
2    2  514 1 0xAA 0x06 0x00 0x07 0x00 0x08 0x00 0x05 0x00 0x0A 0x00 0x0B 0x00 0x0C
0x00 0x09
2    2  522 1 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00 0x0D 0x00 0x12 0x00 0x13 0x00 0x14
0x00 0x11
2    2  546 1 0x00
2    2 1023 0 0x88 0x88
5    5    0 0 0x02 0x00 0x03 0x00 0x04 0x00 0x01 0x00 0x06 0x00 0x07 0x00 0x08 0x00
0x05 0x00
5    5    8 0 0x0A 0x00 0x0B 0x00 0x0C 0x00 0x09 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00
0x0D 0x00
6    5  768 0 0x02 0x00 0x03 0x00 0x04 0x00 0x01 0x00 0x06 0x00 0x07 0x00 0x08 0x00
0x05 0x00
6    5  776 0 0x0A 0x00 0x0B 0x00 0x0C 0x00 0x09 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00
0x0D 0x00
7    5  512 0 0x02 0x00 0x03 0x00 0x04 0x00 0x01 0x00 0x06 0x00 0x07 0x00 0x08 0x00
0x05 0x00
7    5  520 0 0x0A 0x00 0x0B 0x00 0x0C 0x00 0x09 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00
0x0D 0x00
```

6.6.1.3.1.3.2.7 Memory Contents Initalisation

Loading of data from an init file to the model is initiated by an event occuring on the **init_start** input signal which results in it being 'true'.

The init file name is specified by the **init_filename** input signal, and should be located in the same directory as the modelsim macro file in use.

The format of each line in the init file should be as follows:

- Start BANK (decimal 0..7).
- Start ROW (decimal 0..8191 1 Gib, 0..16383 2 Gib, 0..32767 4 Gib).
- Start COL (decimal 0..1023).

- Start data BYTE (decimal 0..1 16-bit part).
- Data Bytes from starting byte (up to 16 bytes per ROW).

An example init file is shown below:

```
2    1  511 0 0xU0 0x00 0x00 0x0U 0x55
2    1  514 1 0x55 0x04 0x00 0x05 0x00 0x06 0x00 0x03 0x00 0x08 0x00 0x09 0x00 0x0A
0x00 0x07
2    1  522 1 0x00 0x0C 0x00 0x0D 0x00 0x0E 0x00 0x0B 0x00 0x10 0x00 0x11 0x00 0x12
0x00 0x0F
2    1  530 1 0x00 0x14 0x00 0x15 0x00 0x16 0x00 0x13 0x00 0x18 0x00 0x19 0x00 0x1A
0x00 0x17
2    1  538 1 0x00 0x1C 0x00 0x1D 0x00 0x1E 0x00 0x1B 0x00 0x20 0x00 0x21 0x00 0x22
0x00 0x1F
2    1  546 1 0x00
2    1 1023 0 0x77 0x77
2    2    0 0 0x99 0x99
2    2  511 0 0xU0 0x66 0x66 0x0U 0xAA
2    2  514 1 0xAA 0x06 0x00 0x07 0x00 0x08 0x00 0x05 0x00 0x0A 0x00 0x0B 0x00 0x0C
0x00 0x09
2    2  522 1 0x00 0x0E 0x00 0x0F 0x00 0x10 0x00 0x0D 0x00 0x12 0x00 0x13 0x00 0x14
0x00 0x11
2    2  530 1 0x00 0x16 0x00 0x17 0x00 0x18 0x00 0x15 0x00 0x1A 0x00 0x1B 0x00 0x1C
0x00 0x19
2    2  538 1 0x00 0x1E 0x00 0x1F 0x00 0x20 0x00 0x1D 0x00 0x22 0x00 0x23 0x00 0x24
0x00 0x21
2    2  546 1 0x00
2    2 1023 0 0x88 0x88
```

6.6.1.3.1.3.2.8 Timing Parameter Generation And Checking

The generic **part** selects the DDR3 SDRAM part to be simulated by the model. Supported parts are defined in the DDR3 SDRAM Model Package (ddr3_sdram_pkg).

Values for the timing parameters required by the model are calculated using the constants specified by the **part** generic.

Timing parameter checks are implemented by an instantiation of the **ddr3_sdram_timing_chk** block. Timing checks are enabled using the **timing_enabled** signal.

6.7 Memory Application

The memory application group is located in the **hdl/vhdl/common/mem_apps/** directory and contains the following elements:

- Memory Application Components

6.7.1 Memory Application Components

6.7.1.1 Memory Test Block (blk_mem_test)

6.7.1.1.1 Introduction

This is a component in the memory application group. Its function is to generate test stimulus, and analyse test responses on a single ADB3 OCP channel.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)

6.7.1.1.2 Interface

The **blk_mem_test** component interface is shown in Figure 85 below and described in Table 188.

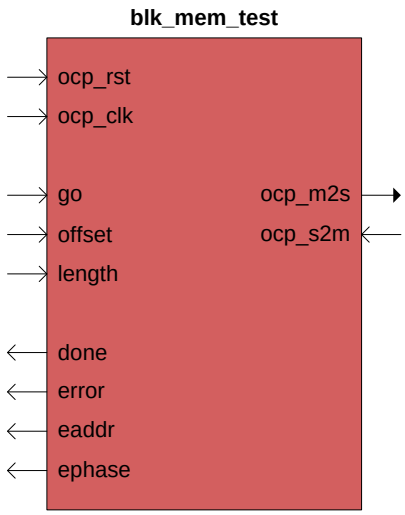

Figure 85 : blk_mem_test component interface

Signal	Type	Description
a_width	Generic	Number of logical bits in OCP port address.
d_width	Generic	Number of logical bits in OCP port data word.
rd_width	Generic	Number of physical data pins on memory bank.
tag_base	Generic	Tag base value.
tag_incr	Generic	Tag value increment.
tag_mask	Generic	Tag check mask bits.

Signal	Type	Description
		OCP Port
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP port M2S connection.
ocp_s2m	Input	OCP port S2M connection.
		Memory Test Control/Status
go	Input	Initiate test.
offset	Input	Starting logical address of test (16-byte address).
length	Input	Number of logical words to test - 1 (16-byte words).
done	Input	Test finished/idle.
error	Input	Error has occurred (qualified by **done**).
eaddr	Input	First error address (16-byte words)(qualified by **done** and **error**).
ephase	Input	First error phase (qualified by **done** and **error**).

Table 188 : blk_mem_test component interface

6.7.1.1.3 Description

6.7.1.1.3.1 Executive State Machine

This state machine controls the initiation of test, and the progression through the different test phases. Test is initiated by the **go** input. There are 6 test phases as follows:

Phase	Test Data
0	"55555555"
1	"AAAAAAAA"
2	"5555AAAA"
3	Address
4	Reversed address
5	Random

6.7.1.1.3.2 OCP Command Issue State Machine

This state machine controls the issue of OCP write and read commands within a test phase. The **offset** and **length** inputs are used in generation of the OCP address and burst length.

Command tags are generated using the **tag_base**, **tag_incr**, and **tag_mask** generics.

6.7.1.1.3.3 OCP Data Transfer State Machine

This state machine controls the issue of OCP write data, and the acceptance of OCP response data.

6.7.1.1.3.4 Data Generation And Verification

OCP write data is generated appropriate for the current test phase.

OCP response data is accepted and compared with expected data for the current test phase.

6.8 Clock Frequency Measurement

The clock frequency measurement group is located in the **hdl/vhdl/examples/uber/common/** directory and contains the following elements:

- Clock Frequency Measurement Components

6.8.1 Clock Frequency Measurement Components

6.8.1.1 Clock Frequency Measurement Block (blk_clock_freq)

6.8.1.1.1 Introduction

This is a component in the clock frequency measurement group. Its function is to count the number of edges present on a sample clock in a measurement period.

Dependencies

- None

6.8.1.1.2 Interface

The **blk_clock_freq** component interface is shown in Figure 86 below and described in Table 189.

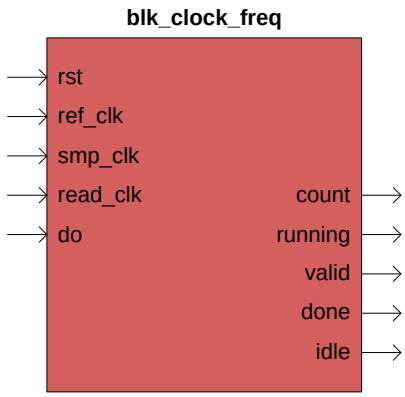

Figure 86 : blk_clock_freq component interface

Signal	Type	Description
ref_clk_tcval	Generic	Measurement period in **ref_clk** cycles.
smp_clk_div_stages	Generic	Number of ripple-divide stages for **smp_clk**.

Signal	Type	Description
		Reset/Clocks
rst	Input	Asynchronous reset.
ref_clk	Input	Reference clock.
smp_clk	Input	Sample clock (to be measured).
		Read Clock Domain

Table 189 : blk_clock_freq component interface (continued on next page)

Signal	Type	Description
read_clk	Input	Read clock.
do	Input	Start a measurement.
count	Output	Number of **smp_clk** cycles counted (qualified by **valid**).
running	Output	**smp_clk** is running (qualified by **valid**).
valid	Output	**count** and **running** are valid.
done	Output	Measurement completed (Active for 1 cycle).
idle	Output	Measurement not in progress.

Table 189 : blk_clock_freq component interface

6.8.1.1.3 Description

This block works by prescaling the clock whose frequency is being measured (input via the **smp_clk** port) by a power of 2, sampling it, and counting rising edges during a certain number of **ref_clk** cycles. Thus, in order to prevent incorrect measurements resulting from aliasing of the sampled clock, the following relationship must hold between the frequencies of **ref_clk** and **smp_clk**, and the number of divider stages (the **smp_clk_div_stages** generic) used in each **blk_clock_freq** instance:

- ref_clk frequency > smp_clk frequency * 2 / (2**smp_clk_div_stages)

For small values of **smp_clk_div_stages**, the accuracy of a measured clock frequency is approximately equal to the accuracy of **ref_clk**.

6.9 ChipScope

The Chipscope group is located in the **hdl/vhdl/common/chipscope/** directory and contains the following elements:

- ChipScope Components

6.9.1 ChipScope Components

6.9.1.1 ChipScope Block (blk_chipscope)

6.9.1.1.1 Introduction

This is a component in the ChipScope group. Its function is to instantiate up to 3 Xilinx ChipScope interfaces, each connected to an ADB3 OCP channel.

Dependencies

- ADB3 OCP Profile Definition Package (adb3_ocp)

6.9.1.1.2 Interface

The **blk_chipscope** component interface is shown in Figure 87 below and described in Table 190.

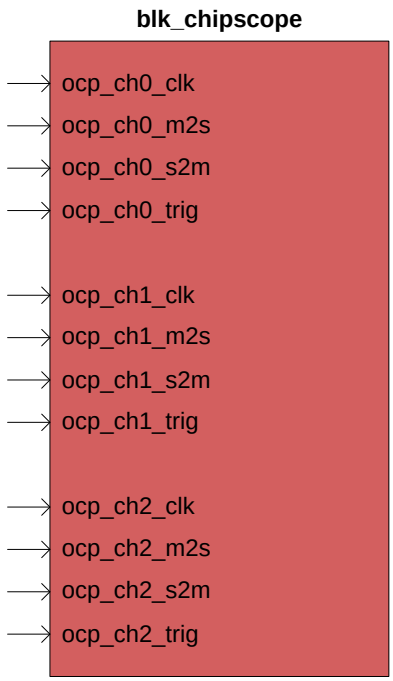

Figure 87 : blk_chipscope component interface

Signal	Type	Description
instantiate	Generic	Enables generation of this component.

Signal	Type	Description
		ChipScope 0
ocp_ch0_clk	Input	OCP port clock.
ocp_ch0_m2s	Input	OCP port M2S.
ocp_ch0_s2m	Input	OCP port S2M.
ocp_ch0_trig	Input	Trigger.
		ChipScope 1
ocp_ch1_clk	Input	OCP port clock.
ocp_ch1_m2s	Input	OCP port M2S.
ocp_ch1_s2m	Input	OCP port S2M.
ocp_ch1_trig	Input	Trigger.
		ChipScope 2
ocp_ch2_clk	Input	OCP port clock.
ocp_ch2_m2s	Input	OCP port M2S.
ocp_ch2_s2m	Input	OCP port S2M.
ocp_ch2_trig	Input	Trigger.

Table 190 : blk_chipscope component interface

6.9.1.1.3 Description

Instantiation of this component is controlled by the value of the **instantiate** generic. A true value is required for instantiation.

6.9.1.1.3.1 Synthesis

During synthesis, **blk_chipscope.vhd** in **$ADMXRC3_SDK/hdl/vhdl/common/chipscope/** contains the **blk_chipscope** component.

For each chipscope channel, a Xilinx **chipscope_ila** core is instantiated with connections as follows:

ILA clk input

- OCP port clock.

ILA data input

- OCP port M2S: Addr(39:0), Data, BurstLength, DataByteEn, Tag.
- OCP port S2M: Data, Tag.
- ILA trig0.
- ILA trig1.

ILA trig0 input

- OCP port M2S: RespAccept, DataValid, Cmd(1:0).
- OCP port S2M: Resp, DataAccept, CmdAccept.

ILA trig1 input

- Trigger input

ILA trig_out output

- Unconnected

A Xilinx **chipscope_icon** core is also instantiated.

Refer to Xilinx ChipScope Core Generation (ICON/ILA) for details of the ILA and ICON core generation procedure.

6.9.1.1.3.2 OCP-Only/Full MPTL Simulation

During simulation, **blk_chipscope_sim.vhd** in **$ADMXRC3_SDK/hdl/vhdl/common/chipscope/** contains the **blk_chipscope** component.

In this simulation version, signals are generated as in the full version, but no **chipscope_ila** and **chipscope_icon** components are instantiated.

6.9.1.1.4 Xilinx ChipScope Core Generation (ICON/ILA)

Prior to the initial bitstream build of a design using a Xilinx ChipScope interface, its ICON and ILA .ngc files will need to be generated using the **gen_chipscope.tcl** script. Examples are as follows:

To generate .ngc files for an ADM-XRC-6T1 using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\common\chipscope
xtclsh gen_chipscope.tcl admxrc6t1
```

To generate .ngc files for an ADM-XRC-6T1 using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/common/chipscope
xtclsh ./gen_chipscope.tcl admxrc6t1
```

Once generated, the Xilinx ChipScope interface .ngc files are located in **hdl/vhdl/common/chipscope/admxrc6t1/ngc/**.

> **Note**
> The TCL script is run using the Xilinx customized TCL distribution TCL shell **xtclsh**. The path to this shell must be defined for sucessful script execution.

7 FPGA Design Guide

This section provides guidelines for FPGA designs targeting third generation Alpha Data hardware.

7.1 ADB3 OCP Protocol Reference

7.1.1 Introduction

Open Core Protocols (OCP-IP) in general allow interfacing between two modules, with one module the master (in control of the transactions) and one module the slave. Each OCP-IP Protocol must have at least a command signal (**Cmd**), however the definition of other sideband signals is fairly flexible.

- Master Port - Initiates all transactions. Multiple transactions may be active at any one time if the slave can also handle multiple transactions.

- Slave Port - Responds to master transactions only, does not initiate any transactions.

The main groupings of signals used in the **ADB3 OCP** protocol are an address channel, synchronous with the **Cmd** signal, and data transfer channels both from master to slave (write data channel) and slave to master (read data channel). Each of these channels is acknowledged independently allowing the flow to be controlled.

The target MPTL interface provides the user with a bank of OCP ports through which data is passed as Read or Write transactions. The ports are as follows:

- A Master port for direct read and write transactions from the host via PCIe Bars 2/3 and 4/5 (64 bit bars).
- A Master port for each DMA engine in the bridge FPGA.
- For advanced systems where the target FPGA design has a requirement for DMA to the host, an MPTL interface can be provided that has an additional Slave Port.

All OCP ports operate independently. With multiple DMA engines in the bridge FPGA, the user can initiate multiple data streams into and out of the target FPGA design.

7.1.2 Port Signal Definitions

The master port outputs to the slave port are shown in Table 191 below. The AMBA AXI4 (Advanced eXtensible Interface) signal equivalents are also shown.

Signal	Channel	Type	Width	Description	AMBA AXI4
Cmd	Address	ocp_CmdT	3	Transaction type/valid	AWVALID/ARVALID
Addr	Address	std_logic_vector	64	Transaction address	AWADDR/ARADDR
BurstLength	Address	std_logic_vector	12	Transaction burst length	AWLEN/ARLEN
Tag	Address	std_logic_vector	8	Transaction identifier	AWID/ARID
DataValid	Write data	std_logic	1	Transfer valid	WVALID
Data	Write data	std_logic_vector	128	Transfer data	WDATA
DataByteEn	Write data	std_logic_vector	16	Transfer data byte enables	WSTRB
RespAccept	Read data	std_logic	1	Transfer ready	RREADY

Table 191 : ADB3 OCP Master Port To Slave Port Signals

Notes:

- All port signals are active high.
- **Cmd** can be **Idle**, **Write**, or **Read**.
- **Addr** is a byte address which is 16-byte aligned. The 4 LSBs are unused.
- **Data** consists of 16 bytes which corresponds to write command addresses **Addr**-**Addr**+15.
- Writing occurs for the bytes of **Data** enabled by **DataByteEn**.

The slave port outputs to the master port are shown in Table 192 below. The AMBA AXI4 (Advanced eXtensible Interface) signal equivalents are also shown.

Signal	Channel	Type	Width	Description	AMBA AXI4
CmdAccept	Address	std_logic	1	Transaction ready	AWREADY/ARREADY
DataAccept	Write data	std_logic	1	Transfer ready	WREADY
Resp	Read data	ocp_RespT	2	Transfer type/valid	RRESP/RVALID
Data	Read data	std_logic_vector	128	Transfer data	RDATA
Tag	Read data	std_logic_vector	8	Transaction identifier	RID

Table 192 : ADB3 OCP Slave Port To Master Port Signals

Notes:

- All port signals are active high.
- **Resp** can be **None**, or **Valid**.
- **Data** consists of 16 bytes which corresponds to read command addresses **Addr**-**Addr**+15.
- Reading occurs for the full 16-bytes of **Data**.

7.1.3 Port Operation

Each OCP Link operates as follows:

Address Channel

1 When required, the master port initiates a transaction by asserting **Cmd**, together with its associated **Addr**, **Tag**, and **BurstLength** signals.

2 The slave port indicates its readiness to accept master port address channel signals using **CmdAccept**. For each cycle in which there is a valid **Cmd** present together with an active **CmdAccept**, the address channel signals will be accepted by the slave port. The slave port asserts **CmdAccept** as and when it is able to.

3 The next valid **Cmd** may be asserted in the cycle following the acceptance of the previous address channel signals.

Write Data Channel

1 When required, the master port initiates a write data transfer by asserting **DataValid**, together with its associated **Data** and **DataByteEn** signals.

2 The slave port indicates its readiness to accept master port write data channel signals using **DataAccept**. For each cycle in which there is a active **DataValid** present together with an active **DataAccept**, the write data channel signals will be accepted by the slave port. The slave port asserts **DataAccept** as and when it is able to.

3 The next valid **DataValid** may be asserted in the cycle following the acceptance of the previous write data channel signals.

Read Data Channel

1 When required, the slave port initiates a read data transfer by asserting **Resp**, together with its associated **Data** and **Tag** signals.

2 The master port indicates its readiness to accept slave port read data channel signals using **RespAccept**. For each cycle in which there is a active **Resp** present together with an active **RespAccept**, the read data channel signals will be accepted by the master port. The master port asserts **RespAccept** as and when it is able to.

3 The next active **Resp** may be asserted in the cycle following the acceptance of the previous read data channel signals.

In summary:

- Initiation of transactions and write data transfer is controlled by the master port.
- Acceptance of transactions and write data transfer is controlled by the slave port.
- Initiation of read data transfer is controlled by the slave port.
- Acceptance of read data transfer is controlled by the master port.

Operations between channels are independent. Write commands may be provided before, during, or after write data. Write commands may be accepted before, during, or after write data. It is the responsibility of the master port to ensure that the number of write data transfers corresponds with the write command burst length. It is the responsibility of the slave port to ensure that the number of read data transfers corresponds with the read command burst length.

Operations within channels are dependent. Write data transfers should be initiated in the same order as write commands are issued. Read data transfers will be returned in the same order as read commands are issued.

Write data transfers are enabled on a byte by byte basis using **DataByteEn**. Read data transfers consist of the full 16-bytes of read data.

7.1.4 Example ADB3 OCP Transaction Waveforms

This section contains timing diagrams for write and read transactions which illustrate operation of the protocol.

The diagrams show different transfer sequences, all of them valid OCP transactions. This is to show the different timing sequences of commands and data transfers that are possible.

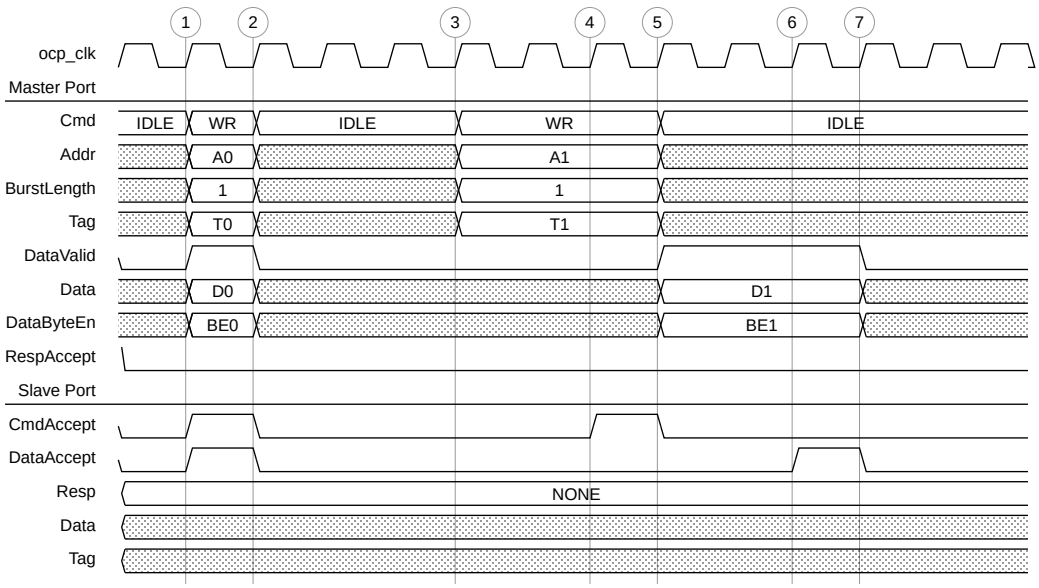

Figure 88 : ADB3 OCP Single Beat Write Transactions

Figure 88 shows two single beat write transactions. The **Addr**, **BurstLength** and **Tag** must be valid while the **Cmd** is set to **Write**.

In the first case, the **Cmd** is accepted in the same cycle as it is asserted, and so returns to **Idle** in the next cycle. The **Data** and **DataByteEn** are also asserted and accepted in the same clock cycle.

In the second case, the **Cmd** is not accepted until the third cycle after it is asserted (possibly due to the slave being busy). The master in this case does not assert the **DataValid** signal until the **Cmd** has been accepted. The **Data** is not accepted immediately, and therefore **DataValid** must remain high until it is. Both examples are legal

within the protocol.

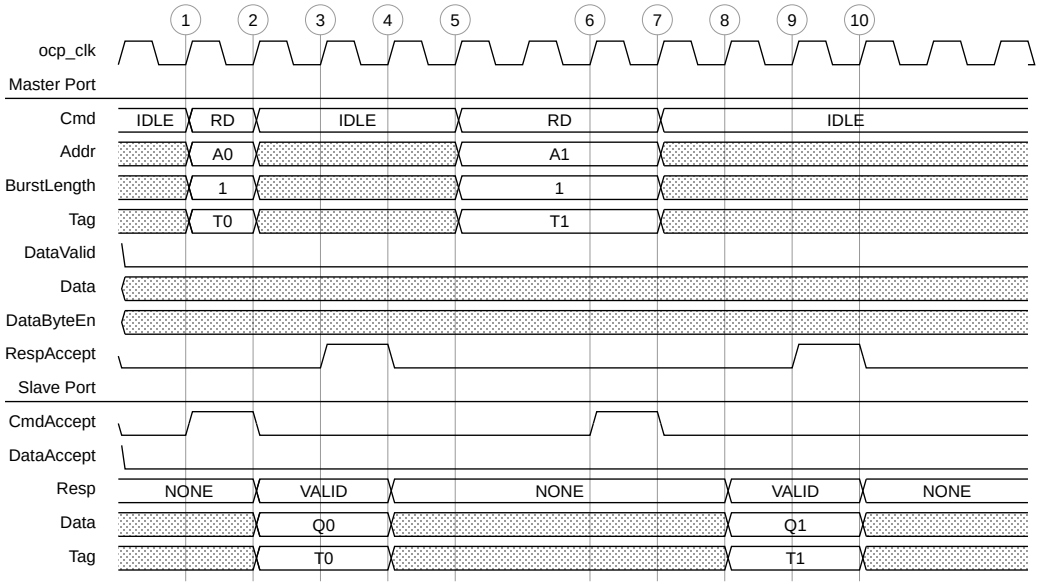

Figure 89 : ADB3 OCP Single Beat Read Transactions

Figure 89 shows two single beat read transactions. The **Addr**, **BurstLength** and **Tag** must be valid while the **Cmd** is set to **Read**.

In the first case the **Cmd** is accepted in the same cycle as it is asserted, and so returns to **Idle** in the next cycle. The slave port returns **Data** on the following clock cycle. The **Tag** sent with the read **Cmd** is returned with the response **Data**. **Resp** indicates when the response **Data** and **Tag** are valid. The valid response is accepted on the following clock cycle.

The second example shows a delayed command accept, delayed response data and a delayed response accept. Both examples are legal within the protocol.

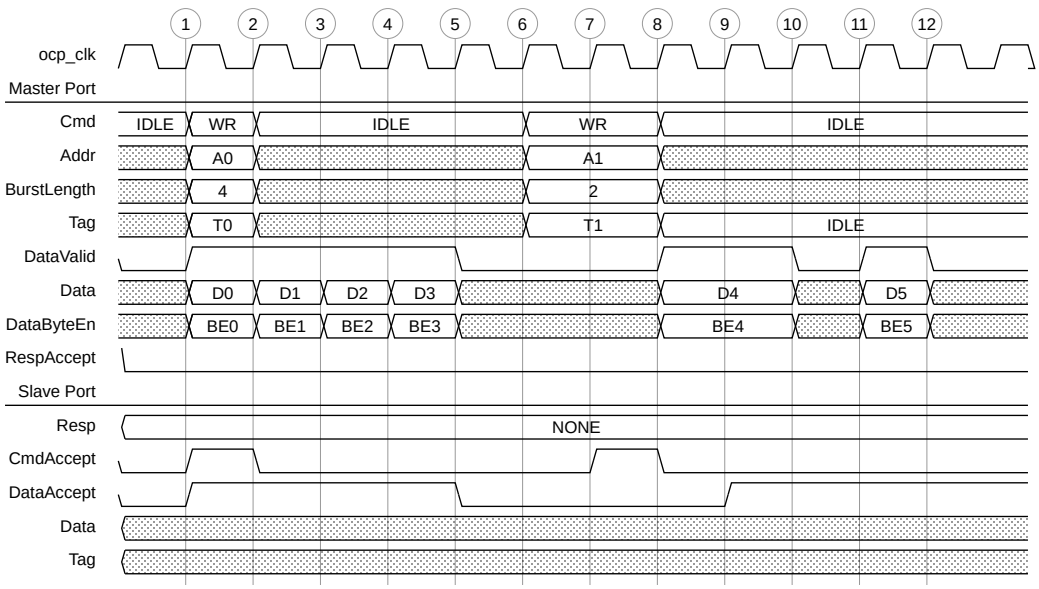

Figure 90 : ADB3 OCP Burst Write Transactions

Figure 90 shows two burst write transactions. A single **Cmd** is issued for multiple write **Data** transfers. The command protocol operates in exactly the same manner as for single beat transfers. Write **Data** transfers only occur when both **DataValid** and **DataAccept** are asserted. The master port must wait on **DataAccept** being asserted before providing the next write **Data** transfer. The slave port must check that **DataValid** is asserted before accepting the write **Data** transfer.

Note
The **DataAccept** signal indicates that the slave port is ready to accept master port write data present on **Data**. The slave port may assert **DataAccept** even if **DataValid** is not asserted.

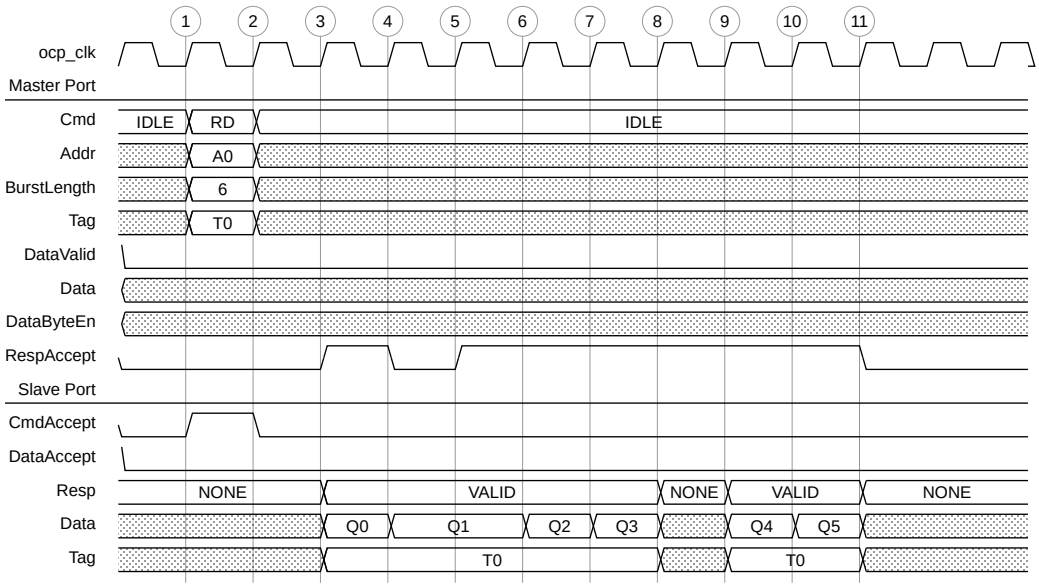

Figure 91 : ADB3 OCP Burst Read Transactions

Figure 91 shows a single burst read transaction. A single **Cmd** is issued for multiple read **Data** transfers. The command protocol operates in exactly the same manner as for single beat transfers. Read **Data** transfers only occur when both **Resp** is **Valid** and **RespAccept** is asserted. The slave port must wait on **RespAccept** being asserted before providing the next read **Data** transfer. The master port must check that **Resp** is **Valid** before accepting the read **Data** transfer.

Note
The **RespAccept** signal indicates that the master port is ready to accept slave port read response data present on **Data**. The master port may assert **RespAccept** even if **Resp** is not **Valid**.

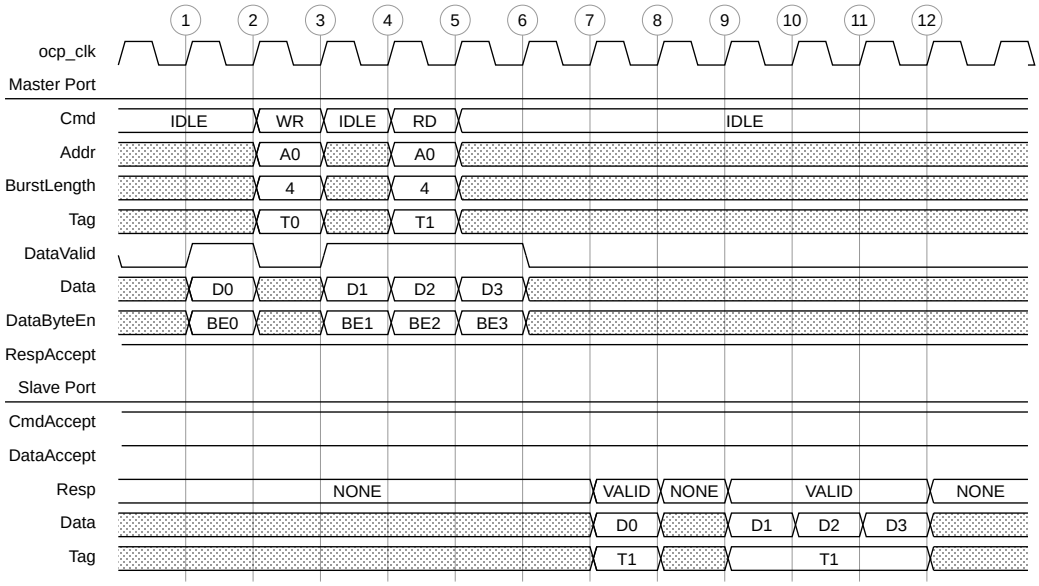

Figure 92 : ADB3 OCP 'Valid' Controlled Transactions

Figure 92 shows a single burst write transaction followed by a single burst read transaction. The write **Data** transfers are provided and accepted in the same cycle when **DataValid** is active, as **DataAccept** is always active. The write and read commands are accepted in the same cycle, as **CmdAccept** is always active. Address and write data channel signals are operating independently of each other.

The read **Data** transfers are provided and accepted in the same cycle when **Resp** is **Valid**, as **RespAccept** is always active.

Note
The **CmdAccept** signal indicates that the slave port is ready to accept master port commands present on **Cmd**. The slave port may assert **CmdAccept** even if **Cmd** is **Idle**.

7.2 ADB3 OCP Lite Protocol Reference

7.2.1 Introduction

Open Core Protocols (OCP-IP) in general allow interfacing between two modules, with one module the master (in control of the transactions) and one module the slave. Each OCP-IP Protocol must have at least a command signal (**Cmd**), however the definition of other sideband signals is fairly flexible.

- Master Port - Initiates all transactions. Multiple transactions may be active at any one time if the slave can also handle multiple transactions.
- Slave Port - Responds to master transactions only, does not initiate any transactions.

The main groupings of signals used in the **ADB3 OCP Lite** protocol are an address channel, synchronous with the **Cmd** signal, and data transfer channels both from master to slave (write data channel) and slave to master (read data channel). The address channel is acknowledged allowing the flow to be controlled.

The **ADB3 OCP Lite** protocol is designed to be used for low speed and throughput communication, for example in register access. Conversion between full and lite protocols is achieved using the adb3_ocp_full2lite_b component.

7.2.2 Port Signal Definitions

The master port outputs to the slave port are shown in Table 193 below. The AMBA AXI4 (Advanced eXtensible Interface) signal equivalents are also shown.

Signal	Channel	Type	Width	Description	AMBA AXI4
Cmd	Address	ocp_CmdT	3	Transaction type/valid	AWVALID/ARVALID
Addr	Address	std_logic_vector	32	Transaction address	AWADDR/ARADDR
Data	Write data	std_logic_vector	32	Transfer data	WDATA
DataByteEn	Write data	std_logic_vector	4	Transfer data byte enables	WSTRB

Table 193 : ADB3 OCP Lite Master Port To Slave Port Signals

Notes:

- All port signals are active high.
- **Cmd** can be **Idle**, **Write**, or **Read**.
- **Addr** is a byte address which is 4-byte aligned. The 2 LSBs are unused.
- There is only one data burst transfer per transaction.
- **Data** consists of 4 bytes which corresponds to write command addresses **Addr**-**Addr**+3.
- Writing occurs for the bytes of **Data** enabled by **DataByteEn**.

The slave port outputs to the master port are shown in Table 194 below. The AMBA AXI4 (Advanced eXtensible Interface) signal equivalents are also shown.

Signal	Channel	Type	Width	Description	AMBA AXI4
CmdAccept	Address	std_logic	1	Transaction ready	AWREADY/ARREADY
Resp	Read data	ocp_RespT	2	Transfer type/valid	RRESP/RVALID
Data	Read data	std_logic_vector	32	Transfer data	RDATA

Table 194 : ADB3 OCP Lite Slave Port To Master Port Signals

Notes:

- All port signals are active high.
- **Resp** can be **None**, or **Valid**.
- **Data** consists of 4 bytes which corresponds to read command addresses **Addr**-**Addr**+4.
- Reading occurs for the full 4-bytes of **Data**.

7.2.3 Port Operation

Each OCP Link operates as follows:

Address and Write Data Channels

1. When required, the master port initiates a transaction by asserting **Cmd**, together with its associated **Addr** signal. If the transaction is a write, the master port initiates a write data transfer by asserting **Data**, together with its associated **DataByteEn** signal.
2. The slave port indicates its readiness to accept master port address channel signals using **CmdAccept**. For each cycle in which there is a valid **Cmd** present together with an active **CmdAccept**, the address channel signals will be accepted by the slave port. The slave port indicates its readiness to accept master port write data channel signals using **CmdAccept**. For each cycle in which there is a valid write **Cmd** present together with an active **CmdAccept**, the write data channel signals will be accepted by the slave port. The slave port asserts **CmdAccept** as and when it is able to.
3. The next valid **Cmd** may be asserted in the cycle following the acceptance of the previous address channel signals.

Read Data Channel

1. When required, the slave port initiates a read data transfer by asserting **Resp**, together with its associated **Data** signal.
2. For each cycle in which there is a valid **Resp** present together with an active **RespAccept**, the read data channel signals will be accepted by the master port.
3. The next valid **Resp** may be asserted in the cycle following the previous read data channel signals.

In summary:

- Initiation of transactions and write data transfer is controlled by the master port.
- Acceptance of transactions and write data transfer is controlled by the slave port.
- Initiation of read data transfer is controlled by the slave port.

Read data transfers will be returned in the same order as read commands are issued.

Write data transfers are enabled on a byte by byte basis using **DataByteEn**. Read data transfers consist of the full 4-bytes of read data.

7.2.4 Example ADB3 OCP Lite Transaction Waveforms

This section contains timing diagrams for write and read transactions which illustrate operation of the protocol.

The diagrams show different transfer sequences, all of them valid OCP transactions. This is to show the different

timing sequences of commands and data transfers that are possible.

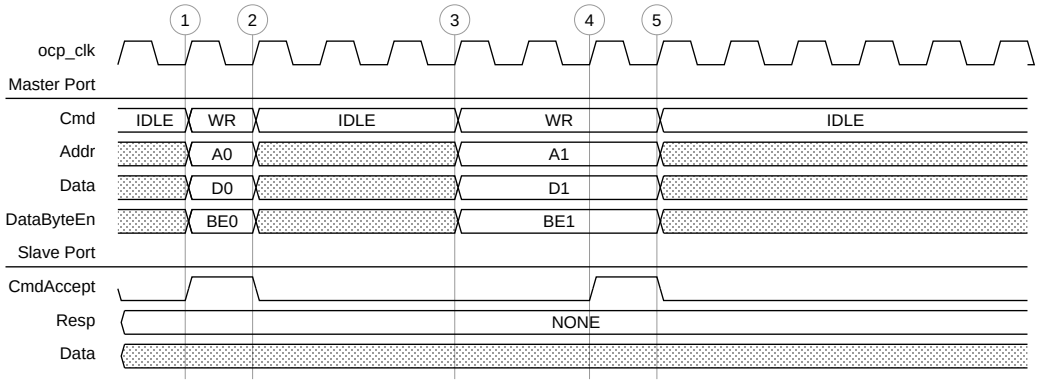

Figure 93 : ADB3 OCP Lite Write Transactions

Figure 93 shows two write transactions. The **Addr**, **Data** and **DataByteEn** must be valid while the **Cmd** is set to **Write**.

In the first case, the **Cmd** is accepted in the same cycle as it is asserted, and so returns to **Idle** in the next cycle. In the second case, the **Cmd** is not accepted until the third cycle after it is asserted (possibly due to the slave being busy).

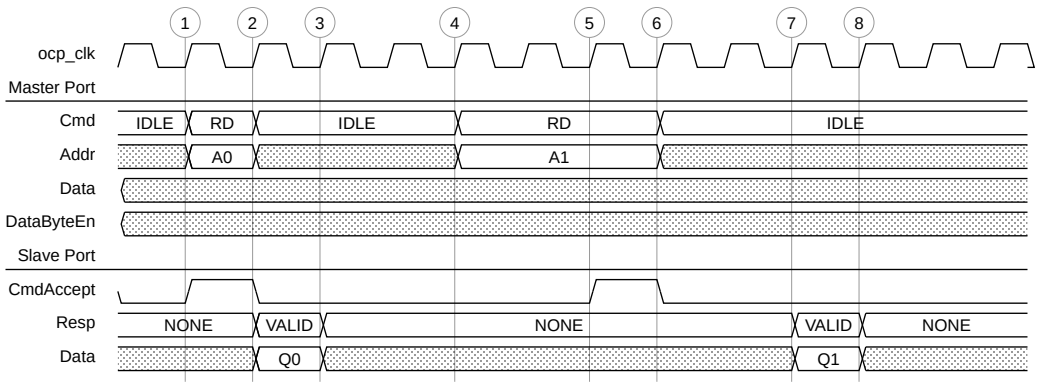

Figure 94 : ADB3 OCP Lite Read Transactions

Figure 94 shows two read transactions. The **Addr** must be valid while the **Cmd** is set to **Read**.

In the first case the **Cmd** is accepted in the same cycle as it is asserted, and so returns to **Idle** in the next cycle. The slave port returns **Data** on the following clock cycle. **Resp** indicates when the response **Data** is valid.

The second example shows a delayed command accept and delayed response data. Both examples are legal within the protocol.

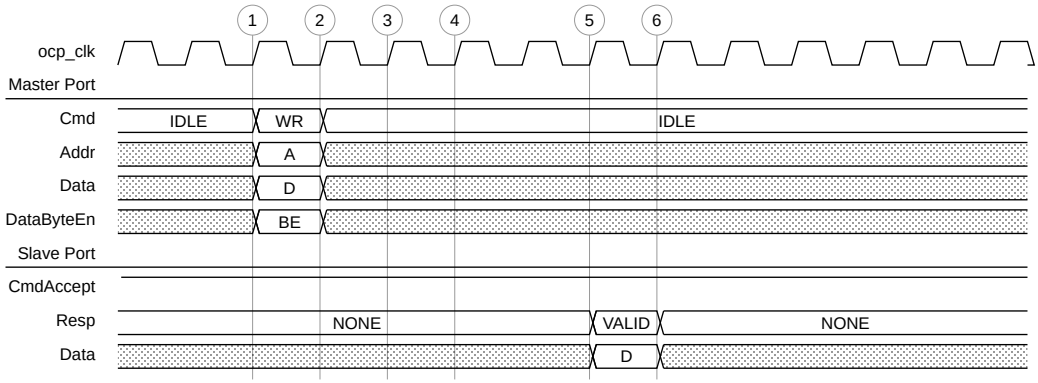

Figure 95 : ADB3 OCP Lite 'Valid' Controlled Transactions

Figure 95 shows a write transaction followed by a read transaction. The write and read commands are accepted in the same cycle, as **CmdAccept** is always active.

Note
The **CmdAccept** signal indicates that the slave port is ready to accept master port commands present on **Cmd**. The slave port may assert **CmdAccept** even if **Cmd** is **Idle**.

8 The ADMXRC3 API

The ADMXRC3 API is the application programming interface that applications, including the ones in this SDK, use to communicate with third generation Alpha Data hardware. This API is documented in the ADMXRC3 API Specification.

Revision History

Date	Revision	Nature of Change
20 May 2010	1.0	Initial version
26 Jul 2010	1.1	Updated for release 1.1.0 Added SDK structure diagram. Added information about example applications.
21 Sep 2010	1.2	Updated for release 1.2.0 Added section for getting started in VxWorks. Documented VxWorks example applications.
04 Mar 2011	1.3	Updated for release 1.3.0 Documented new MEMTESTH example application. Documented new options in existing example applications and utilities. Documented DDR3 memory interface additions to UBER design. Added outlines of common HDL components provided by SDK. Corrected error in DEBUG column of table showing naming conventions for VxWorks example binaries.
11 May 2011	1.4	Updated for release 1.3.1
05 Sep 2011	1.5	Updated for release 1.4.0 Documented improvements to SYSMON utility. Documented new test_board_clks and adb3_ocp_simple_bus_if_nb components in Common HDL Components section. Added timing diagrams to adb3_ocp_simple_bus_if and adb3_ocp_simple_bus_if_nb components in Common HDL Components section. Expanded ADB3 OCP Protocol Reference section. Documented new adb3_target_tb_inc_pkg package in Common HDL Components section. Updated Example HDL FPGA Designs section to reflect changes.
10 Sep 2012	1.6	Updated for release 1.5.0: Documented building Windows example applications with Microsoft Visual Studio 2010. Completed documentation of Common HDL Components section. Added new common group DDR3 SDRAM Interface. Added new ADB3 OCP components. Added ADB3 OCP lite protocol reference to design guide. Documented new ITest and SimpleDMA example FPGA designs. Added PlanAhead information for example FPGA designs.

Page Intentionally left blank

Address: 4 West Silvermills Lane,
Edinburgh, EH3 5BD, UK
Telephone: +44 131 558 2600
Fax: +44 131 558 2700
email: sales@alpha-data.com
website: http://www.alpha-data.com

Address: 3507 Ringsby Court Suite 105,
Denver, CO 80216
Telephone: (303) 954 8768
Fax: (866) 820 9956 toll free
email: sales@alpha-data.com
website: http://www.alpha-data.com