ALPHA DATA

ADM-XRC Gen 3 SDK 1.4.0 User Guide

Revision: 1.5
Date: 24th August 2011

	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

Tables

Figures

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 the forward slash (***/***). A pathname or directory name in a Windows environment has forward slashes replaced by backslashes. For example, the path **hdl/vhdl** is also **hdl/vhdl** in a Linux environment, but is **hdl\vhdl** in a Windows environment.

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

Unless stated otherwise or preceded by a forward 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-1.4.0/apps/src/itest/itest.c** in a Linux environment, assuming that the root of the SDK is **/opt/admxrcg3sdk-1.4.0/**.

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 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

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-1.4.0/** in 32-bit editions of Windows.
- **%ProgramFiles(x86)%/ADMXRCG3SDK-1.4.0/** 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-1.4.0/**.

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-1.4.0
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

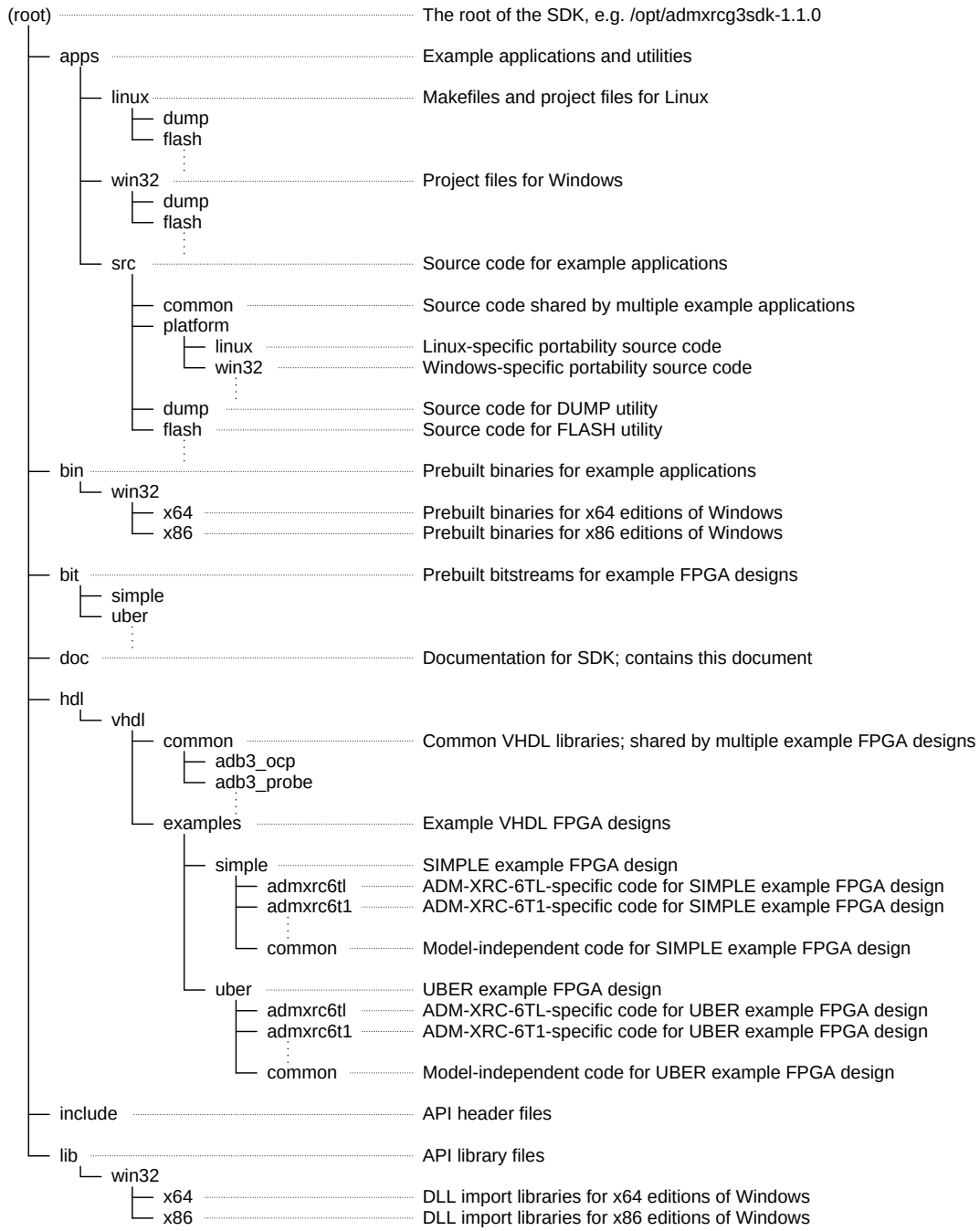

Figure 1: Structure of the ADM-XRC Gen 3 SDK

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: **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: **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: **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
$ ./configure
```

This detects certain features of the operating system environment so that the example applications can be built. Next, change directory to the Linux application directory:

```
$ cd apps/linux
$ 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.

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
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
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 in Windows

A Microsoft Visual Studio 2008 solution **apps/win32/apps.sln** is provided, containing all of the Windows examples. To build all of the examples, use the "Batch Build" command in Visual Studio.

3.2 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 3.10.2**.

3.3 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.

3.4 FLASH utility

WARNING: Incorrect use of the **+failsafe** option may impact long-term reliability of a reconfigurable computing card. Please refer to **Section 3.4.1** 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 program 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
```

3.4.1 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.

Xilinx answer record 35055 elaborates on protecting Virtex-6 GTX transceivers from performance degradation over time.

3.5 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
```

3.6 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.

Summary

Demonstrates consumption of FPGA interrupt notifications.

Description

This example demonstrates how to consume FPGA interrupt notifications in an application. It uses the interrupt test register block of the **Uber example FPGA design**, described in **Section 5.5.4.4.2** 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.

3.7 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
```

3.8 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
```

3.9 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 SIMPLE FPGA design (default).
+uber	Uses UBER FPGA design.

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
```

3.10 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:

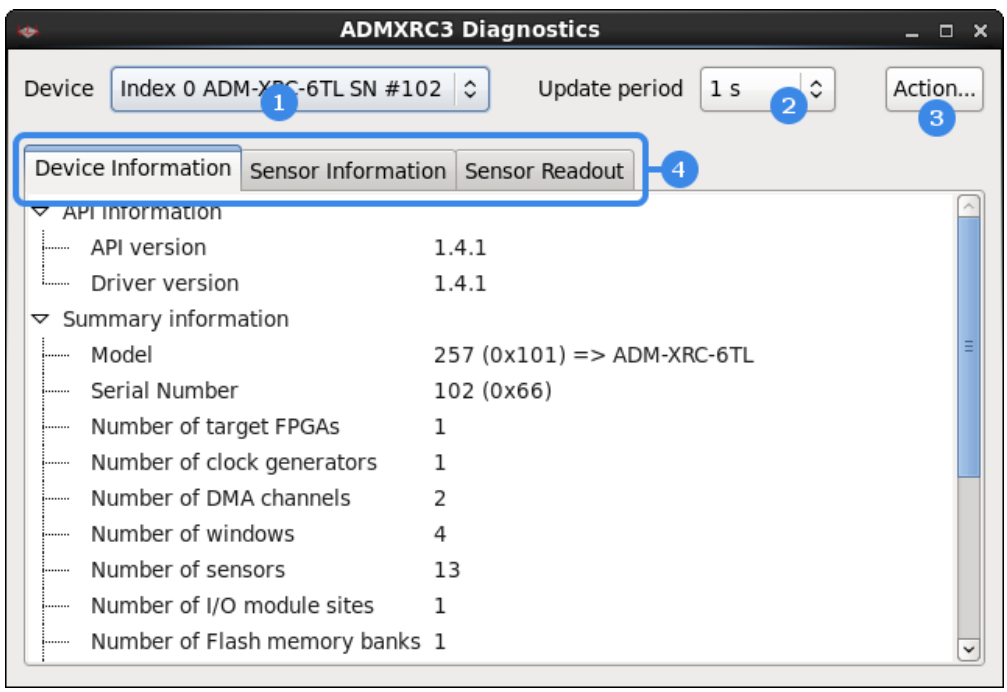

Figure 2: SYSMON user interface

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 3.10.1** 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 **'scope tab** displays sensor data graphically in up to four 'scopes.

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

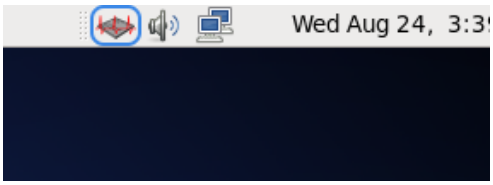

Figure 3: SYSMON notification area icon

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 3.10.1** 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:

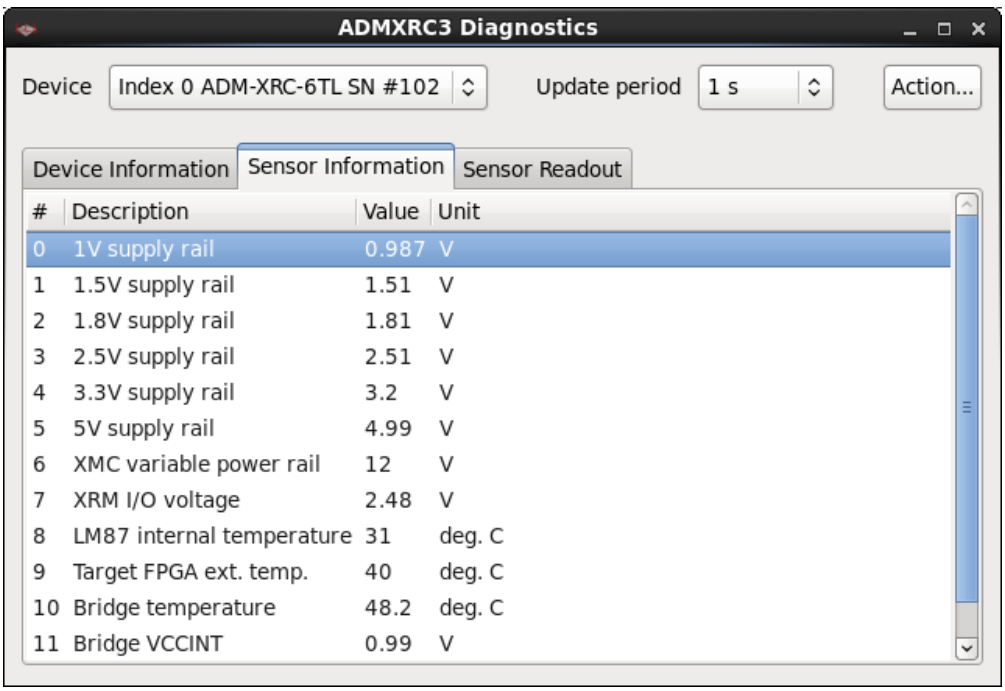

Figure 4: SYSMON sensor information tab

SYSMON 'scope tab

The 'scope tab displays sensor readings in graphical form:

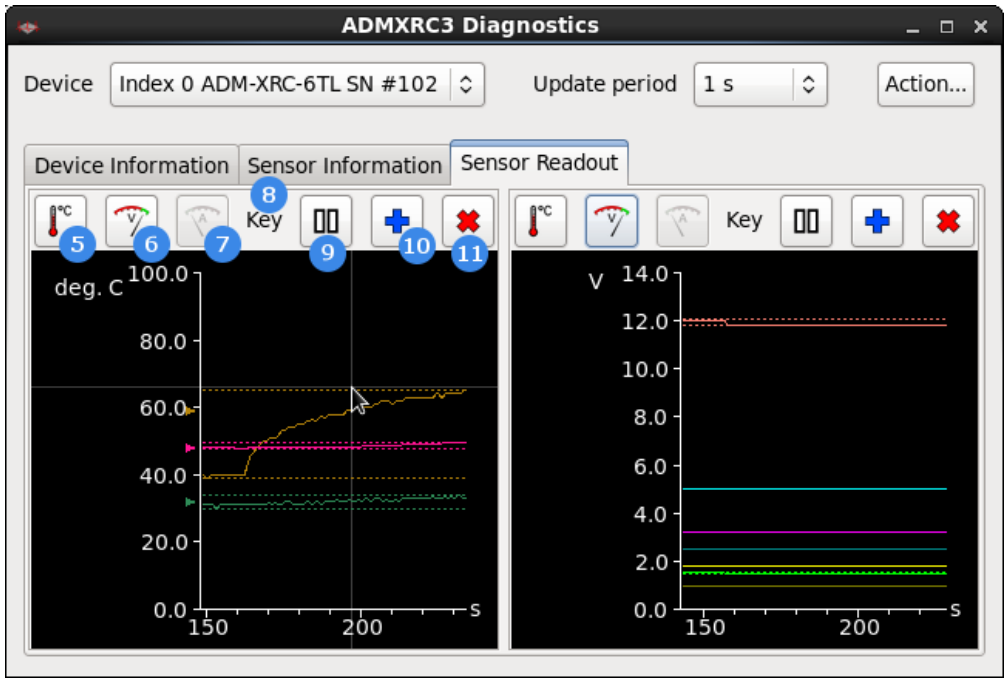

Figure 5: SYSMON 'scope tab

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.

3.10.1 SYSMON sensor data logging

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

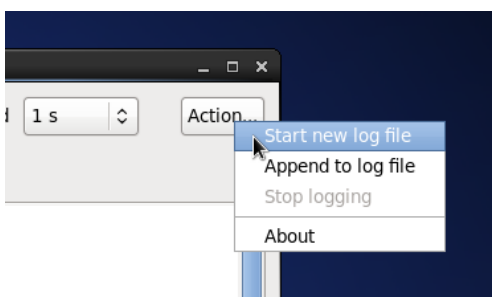

Figure 6: SYSMON Action menu in Linux

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

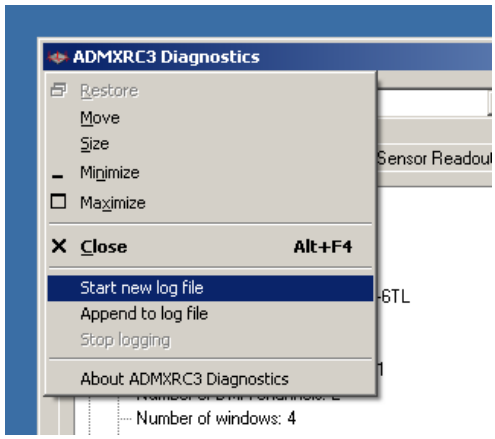

Figure 7: SYSMON Action menu in Windows

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.

3.10.2 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**.

3.11 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:

(a) *address* - the byte address within VPD memory at which to begin filling

(b) *n* - byte count; the number of bytes of VPD memory to fill

(c) *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
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
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 reconfigurable computing device (default 0). However, this may be interpreted as a serial number instead of an index if *flags* contains 0x1.
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.

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	specifies the index of the card to open (default 0).
flags	is the bitwise OR of zero or more of the following flags (default 0): **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>
```

where

index specifies the index of the card to open (default 0).

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 5.5.4.4.2** 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 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
```

4.8 MONITOR utility (VxWorks)

Invocation in VxWorks shell

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

where

index	specifies the index of the card to open (default 0).
flags	is a bitwise OR of flags that modify the behavior of this utility (default 0); must be 0 as there are currently no flags defined.
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
```

4.9 SIMPLE example (VxWorks)

Invocation in VxWorks shell

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

where

index	specifies the index of the card to open (default 0).
flags	is the bitwise OR of zero or more of the following flags (default 0): **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
```

4.10 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	specifies the index of the card to open (default 0).
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:

(a) *address* - the byte address within VPD memory at which to begin filling

(b) *n* - byte count; the number of bytes of VPD memory to fill

(c) *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 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 ................
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 the Virtex-6 based ADM-XRC series of cards 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:

Simple	Minimal design that demonstrates implementation of host-accessible registers. The **SIMPLE example application** (**Windows and Linux** / **VxWorks**) uses this design.
Uber	Demonstrates implementation of host-accessible registers. The **SIMPLE example application** (**Windows and Linux** / **VxWorks**) uses this design 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 design.
	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 Design Simulation Using Modelsim

Testbench code and macro files compatible with Modelsim are provided for simulation of each example FPGA design. For details specific to each example design, refer to its **Design Simulation** section.

> **Note:** VHDL source code is compiled for simulation using the 1993 standard.

Two types of simulation are currently available, termed "Full MPTL" and "OCP-only". They are selected by the **TARGET_USE** constant in the package **adb3_target_inc_pkg**. There are several variants of this package, They are listed in **Table 97**.

5.2.1 Full MPTL Simulation (TARGET_USE = SIM_MPTL)

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.

5.2.2 OCP-Only Simulation (TARGET_USE = SIM_OCP)

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.

5.3 Bitstream Build Using Xilinx ISE

> **Note:** Xilinx ISE versions 12.3 onwards is required by this version of the SDK. ISE version 13.2 onwards is recommended.

Bitstreams for all supported combinations of example FPGA design, board, 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. Bitstream build in the Windows environment uses the Microsoft Visual Studio **NMAKE** utility. Bitstream build in the Linux environment uses **GNU Make**.

5.3.1 Building All Example Bitstreams for Windows

An Makefile compatible with **NMAKE** is provided for building all bitstreams for all example FPGA designs in Windows. It is located in the **hdl/vhdl/examples/** directory. As many bitstream files are generated, it may take from minutes to hours to run to completion. To perform the build, start a shell and issue the following commands:

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

To completely rebuild all example bitstreams, issue the commands:

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

To install the resulting bitstream files in the **bit/** directory, start a shell and issue the following commands:

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

Note: The above commands build the bitstream files, if necessary, before installing them.

5.3.2 Building All Example Bitstreams for Linux

A Makefile compatible with **GNU Make** is provided for building all bitstreams for all example FPGA designs in Linux. It is located in the **hdl/vhdl/examples** directory. As many bitstream files are generated, it may take from minutes to hours to run to completion. To perform the build, start a shell and issue the following commands:

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

To completely rebuild all example bitstreams, issue the commands:

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

To install the resulting bitstream files in the **bit/** directory, start a shell and issue the following commands:

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

Note: The above commands build the bitstream files, if necessary, before installing them.

5.3.3 Building Specific Example/Board/Device Bitstreams

For each example FPGA design, a Makefile is provided for building all its bitstreams, or a specific board/device bitstream. For details specific to each example design, refer to its **Design Synthesis and Bitstream Build** section.

5.4 Simple Example FPGA Design

5.4.1 Board Support

The **Simple** FPGA design is compatible with all Virtex-6 based boards.

5.4.2 Source Location

The **Simple** FPGA design is located in **hdl/vhdl/examples/simple/**. Source files common to all boards are located in the **hdl/vhdl/examples/simple/common/** directory. These include the design and testbench top levels.

5.4.2.1 VHDL Source Files for Simulation

For a complete list of the source files used during simulation, refer to the appropriate Modelsim macro file located in the board design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.do** for OCP-only simulation of an ADM-XRC-6T1.

5.4.2.2 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 board design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.prj** for an ADM-XRC-6T1.

5.4.2.3 XST Files

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

5.4.2.4 Implementation Constraint Files

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

5.4.3 Design Synthesis and Bitstream Build

A Makefile is provided for building the **Simple** 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 board-device combination, or bitstreams can be built for all supported board-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, 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.

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

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

Table 4: Simple Design Makefile Targets

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

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

Filenames of any bitstreams built are thus of the form:

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

When a target of class "clean" is executed, the **build/<*board*>-<*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 board fitted with an 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 board fitted with an 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 board fitted with an 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
```

5.4.4 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 5** 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-6TADV8	PCIe	simple_l_pcie.vhd

Table 5: 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 8 below shows the main elements of the **Simple** design using MPTL interface IP.

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

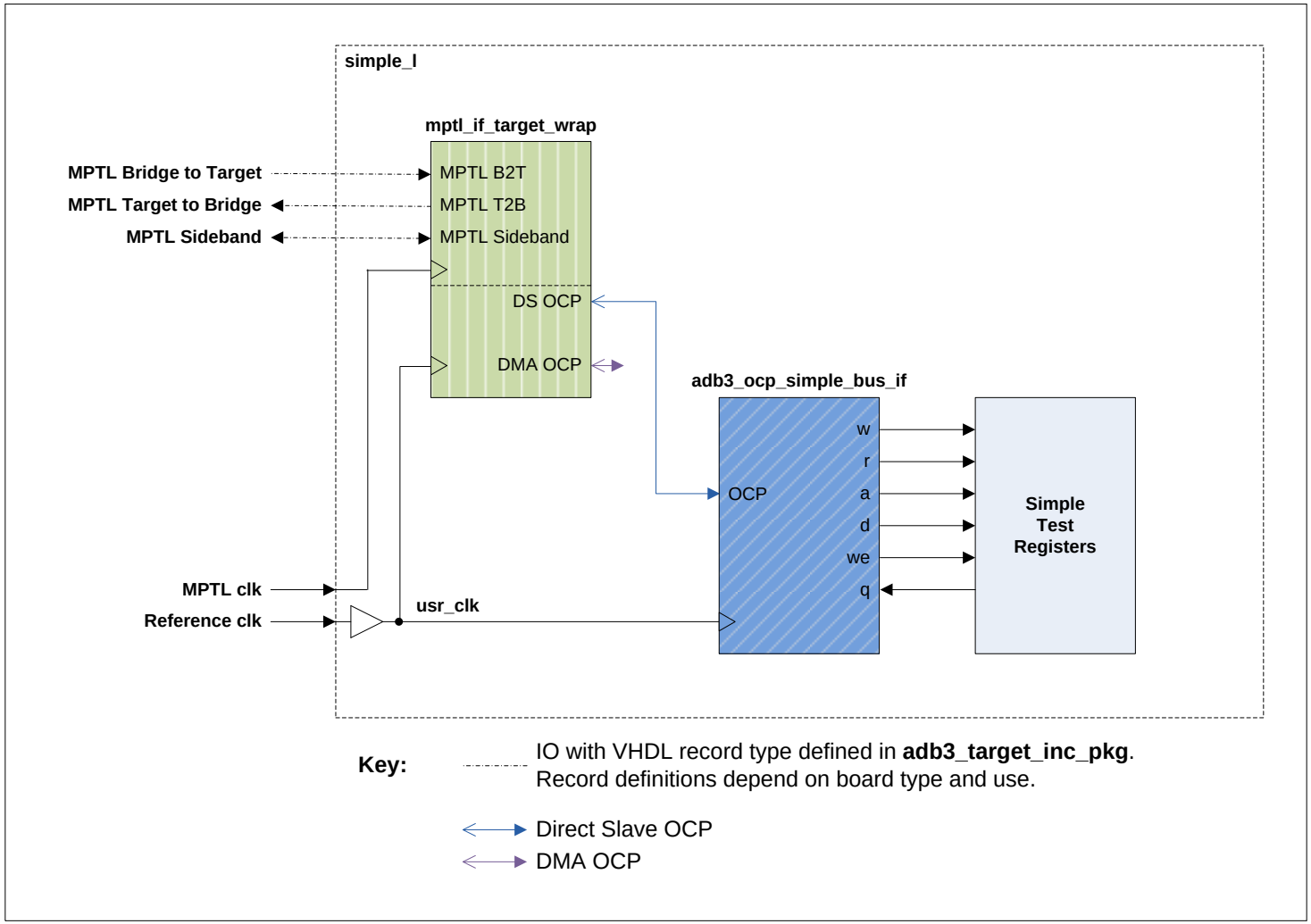

Figure 8: Simple Design Block Diagram (MPTL)

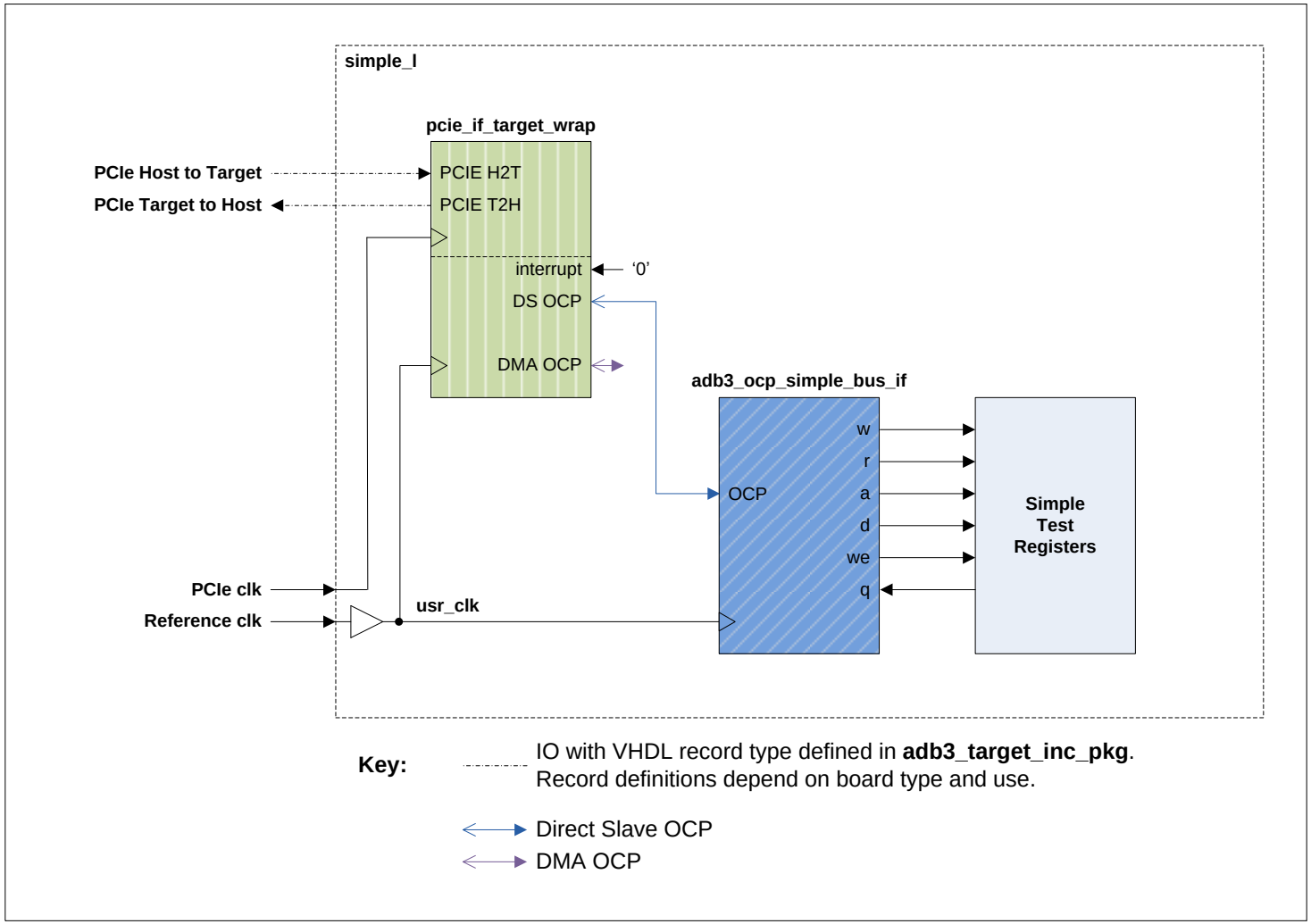

Figure 9: Simple Design Block Diagram (PCIe)

5.4.4.1 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 board selected, and is defined in the constraints file located in the board-specific design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-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 board selected, and is defined in the constraints file located in the board-specific design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-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 board selected, and is defined in the constraints file located in the board-specific design directory; for example, **hdl/vhdl/examples/simple/admxrc6t1/simple-admxrc6t1.ucf** for an ADM-XRC-6T1.

OCP Reset

- The **Simple** example design is reset by an OCP reset named **usr_rst**.
- The reset is active on power-on and is held high for 32 cycles of **usr_clk**.

5.4.4.2 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 board 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.

5.4.4.3 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 board 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.

5.4.4.4 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.

5.4.4.5 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.

5.4.4.5.1 Register Description

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

Name	Type	Address
DATA	RW	0x000000

Table 6: 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 7: Simple Design, DATA Register (0x000000)

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

5.4.5 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 8** 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-6TADV8	PCIe	test_simple_pcie.vhd

Table 8: 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 10 shows the testbench and embedded **simple** FPGA design (MPTL).

Figure 11 shows the testbench and embedded **simple** FPGA design (PCIe).

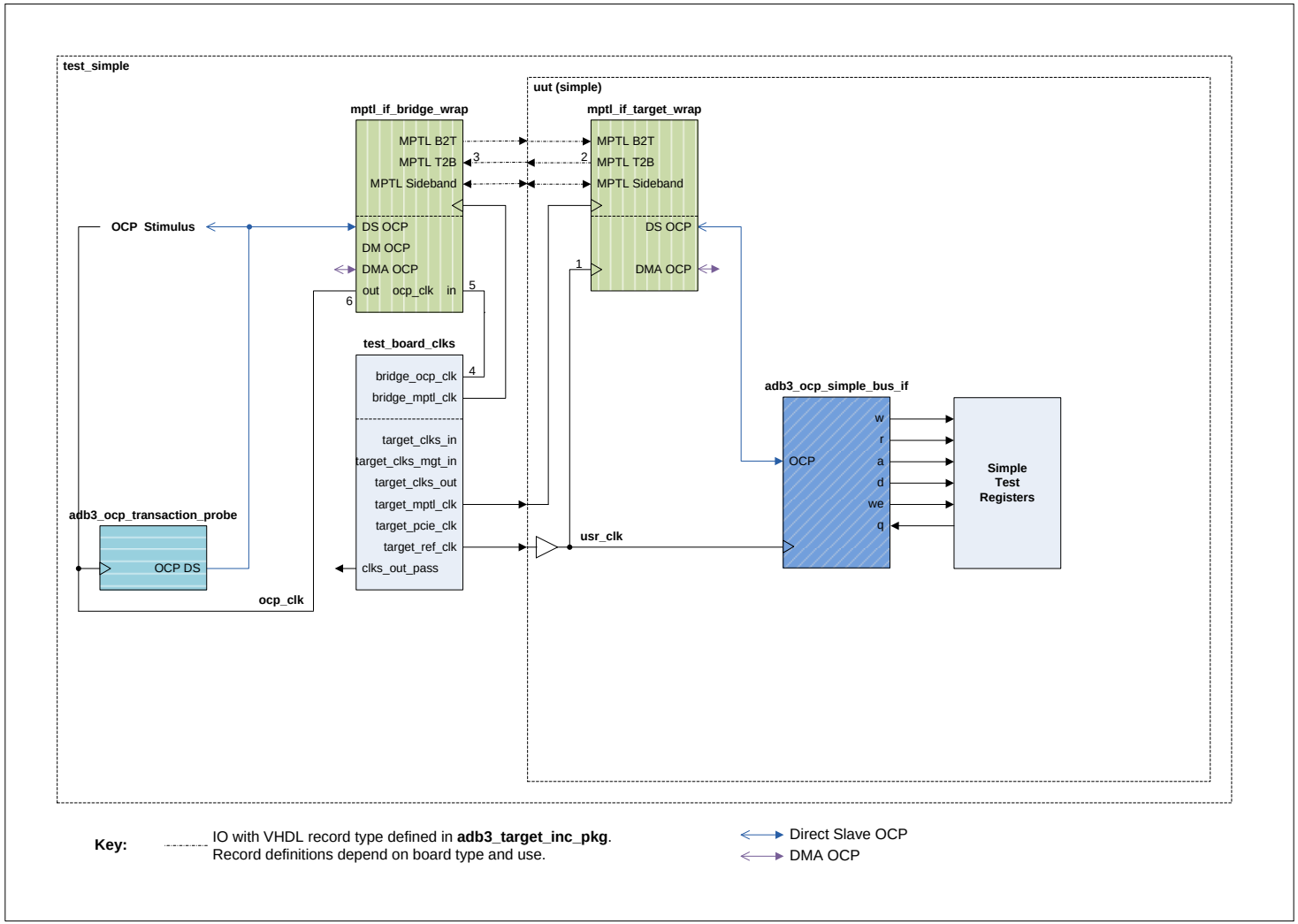

Figure 10: Simple Design Testbench and Top Level Block Diagram (MPTL)

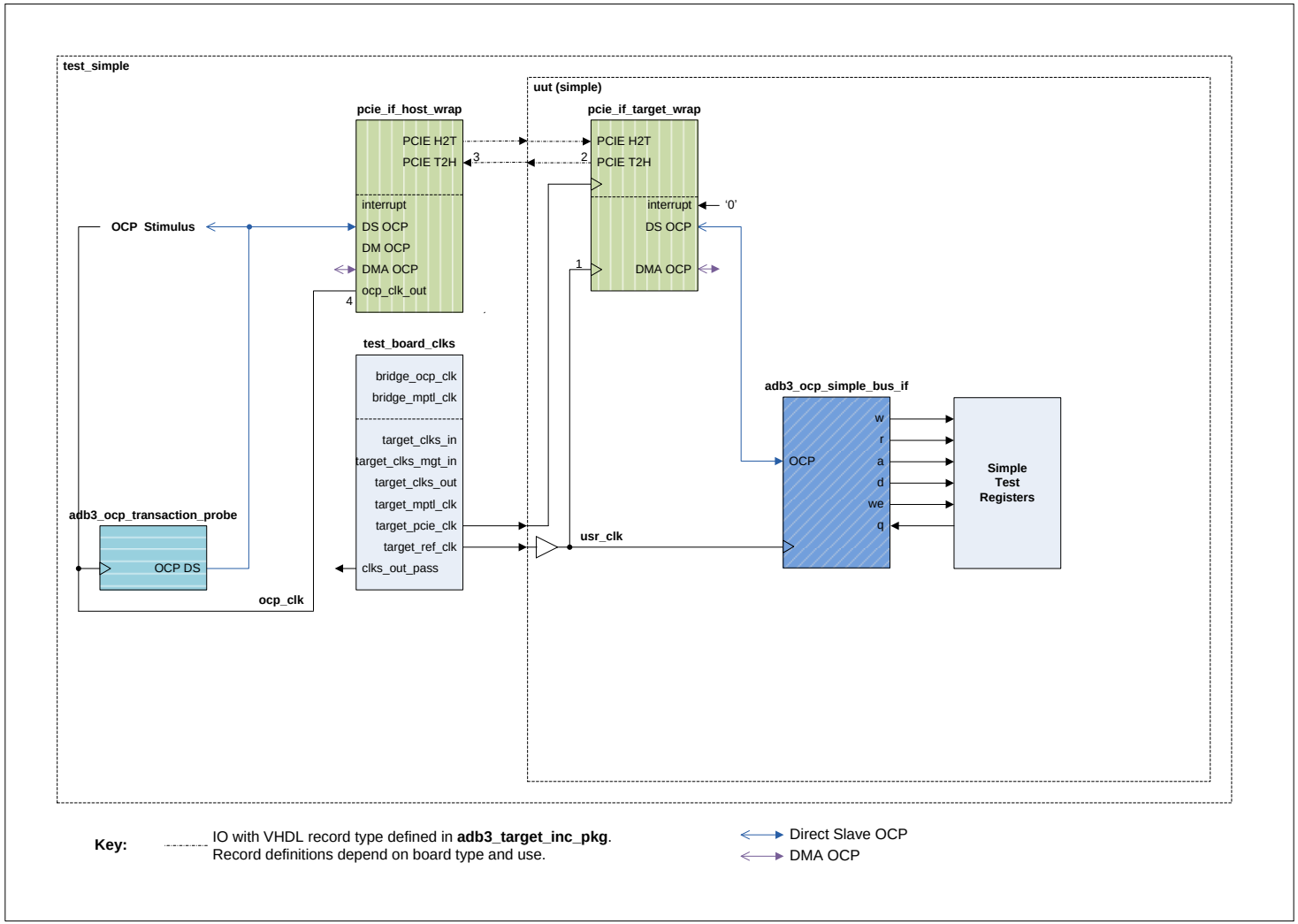

Figure 11: Simple Design Testbench and Top Level Block Diagram (PCIe)

5.4.5.1 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 board is selected. These clocks drive the unit under test (**simple**).

Bridge MPTL Interface Clock

- It generates the **bridge_mptl_clk** clock according to which board 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 board 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.

5.4.5.2 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 board 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 10** 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 10** 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 during a full MPTL simulation.

5.4.5.3 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 board 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 **mptl_if_bridge_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 11** as the route consisting of points 1, 2, 3 and 4.

5.4.5.4 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**.

5.4.5.5 Stimulus Generation and Verification

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

5.4.5.5.1 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.

5.4.5.5.1.1 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**.

5.4.6 Design Simulation

Modelsim macro files are located in each of the board-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 board 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 Modelsim 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.

Expected simulation results are shown below.

5.4.6.1 Initialisation Results (MPTL)

Modelsim output during initialisation of simulation is of the form:

```
# ** Note: Board Type : adm_xrc_6t1
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Target Use : sim_ocp
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Waiting for MPTL online....
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
```

5.4.6.2 Direct Slave OCP Channel Results

Modelsim 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
```

5.4.6.3 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
```

5.5 Uber Example FPGA Design

5.5.1 Board Support

The **Uber** FPGA design is compatible with all Virtex-6 based boards.

5.5.2 Source Location

The **Uber** FPGA design is located in **hdl/vhdl/examples/uber/**. Source files common to all boards are located in the **hdl/vhdl/examples/uber/common/** directory. These include the design and testbench top levels.

5.5.2.1 VHDL Source Files for Simulation

For a complete list of the source files used during simulation, refer to the appropriate Modelsim macro file located in the board design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1.do** for OCP-only simulation of the ADM-XRC-6T1.

5.5.2.2 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 board design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1-6vlx240t.prj** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

5.5.2.3 XST Files

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

5.5.2.4 Implementation Constraint Files

Implementation constraint files (**.ucf**) are located in the board design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/uber-admxrc6t1-6vlx240t.ucf** for the ADM-XRC-6T1 with a 6VLX240T device.

5.5.3 Design Synthesis and Bitstream Build

A Makefile is 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 board-device combination, or bitstreams can be built for all supported board-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 the script **gen_mem_if.tcl**. Refer to **Xilinx DDR3 SDRAM MIG Core Generation** 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 board and device combinations.
bit_<model>_<device>		Builds the **.bit** file for the board specified by **<model>** with a device specified by **<device>**.
install	install	Builds and installs all **.bit** files for all supported board and device combinations in the directory **bit/uber/**.
inst_<model>_<device>		Builds the **.bit** file for the board specified by **<model>** with a device specified by **<device>** and copies it to the directory **bit/uber/**.
clean	clean	Deletes all **.bit** files and intermediate build files for all supported board 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 board specified by **<model>** with a device specified by **<device>** (but does not delete any files from **bit/uber/**).

Table 9: Uber Design Makefile Targets

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

hdl/vhdl/examples/uber/build/*<board>-<device>*/

Filenames of any bitstreams built are thus of the form:

hdl/vhdl/examples/uber/build/*<board>-<device>*/uber-*<board>-<device>*.bit.

When a target of class "clean" is executed, the **build/*<board>-<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 board fitted with an 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 board fitted with an 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 board fitted with an 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
```

5.5.3.1 Date/Time Package Generation

If XST is required to be run during bitstream build, the makefile 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 board selected and is located in the board design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/today_pkg_admxrc6t1_6vlx240t.vhd** for the ADM-XRC-6T1 with a 6VLX240T device. Script output is of the form:

```
--
-- today_pkg_admxrc6t1_sim.vhd
-- 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"00010400";
  constant TODAYS_DATE : std_logic_vector(31 downto 0) := X"08102010";
  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 simulation.

5.5.4 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 10** 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-6TADV8	PCIe	uber_pcie.vhd

Table 10: 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 12 shows the main elements of the **Uber** design using MPTL interface IP.

Figure 13 shows the main elements of the **Uber** design using PCIe interface IP.

Figure 14 shows the hierarchy of the **Uber** design using MPTL interface IP.

Figure 15 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)**
- **ADB3 target package (adb3_target_pkg)**
- **Design package (uber_pkg)**

Figure 16 shows the design package dependencies.

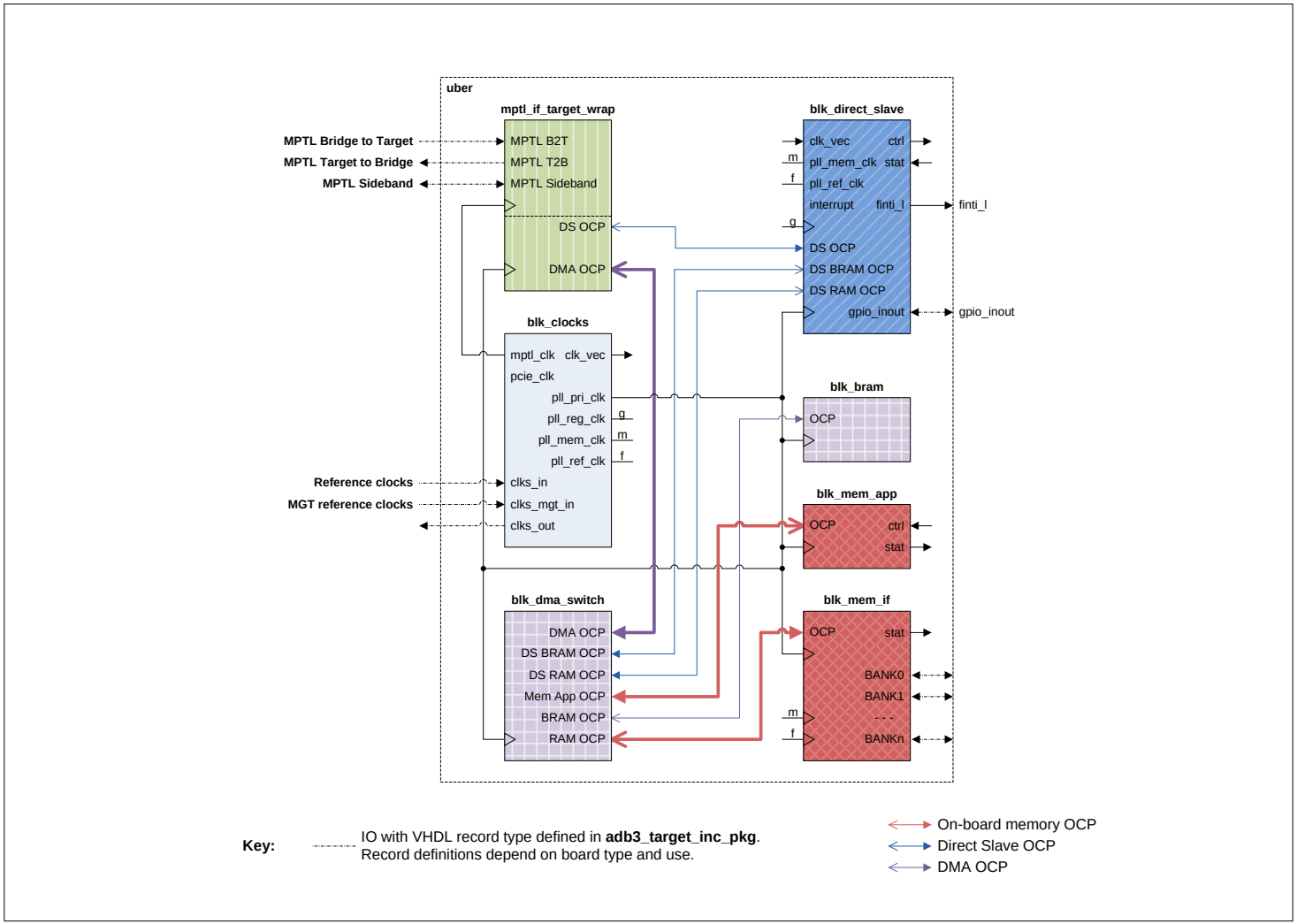

Figure 12: Uber Design Top Level Block Diagram (MPTL)

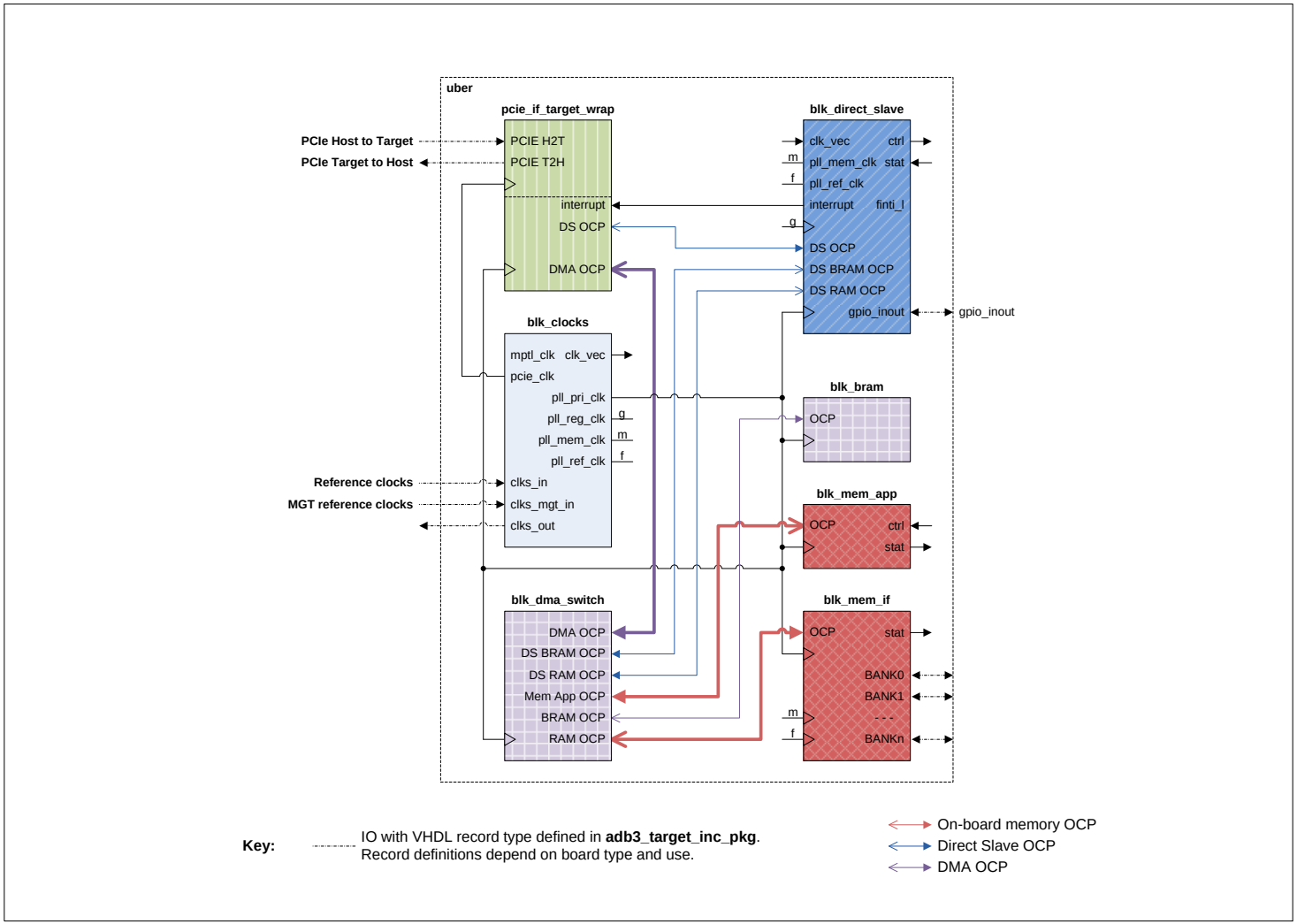

Figure 13: Uber Design Top Level Block Diagram (PCIe)

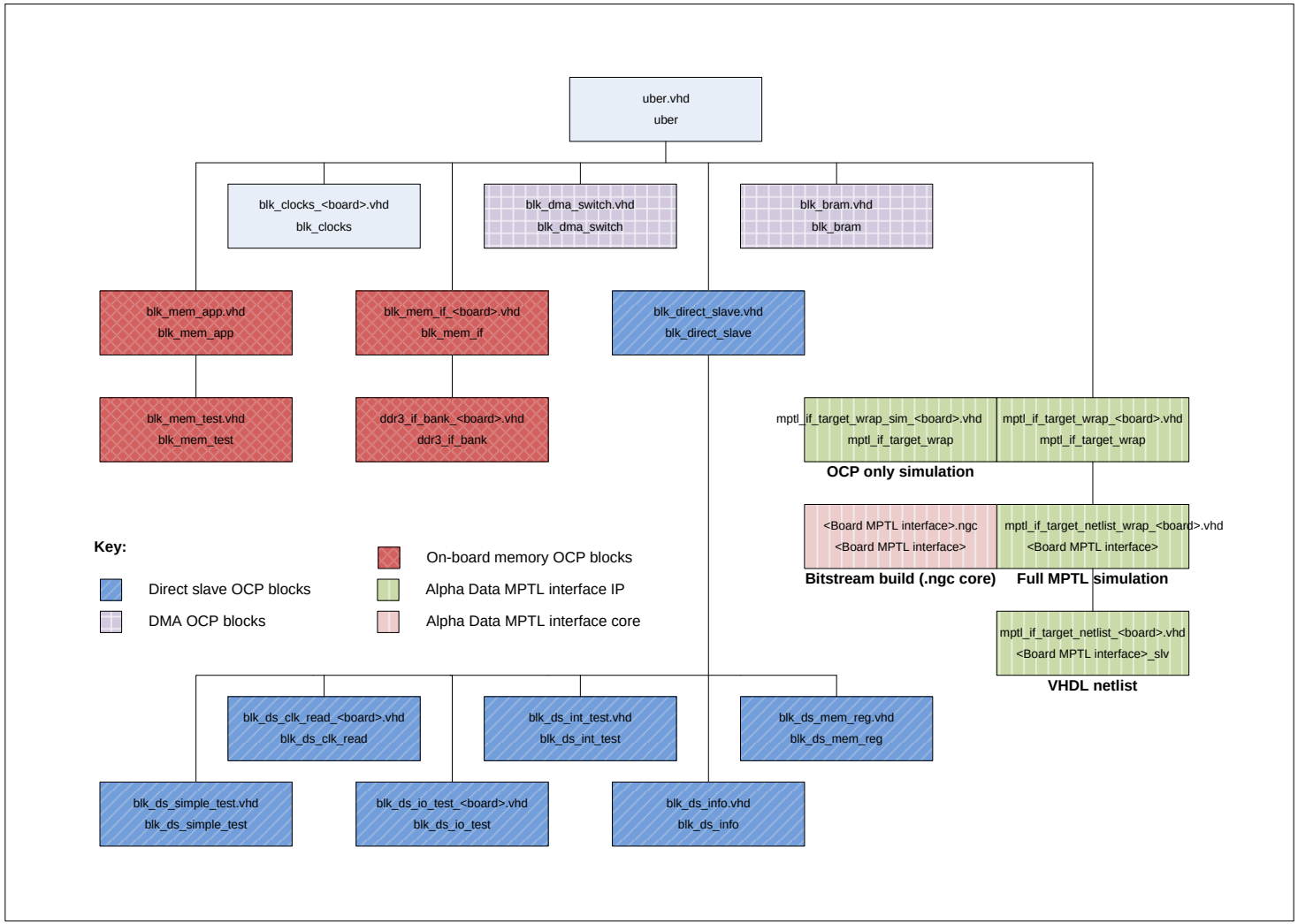

Figure 14: Uber Design Top Level Hierarchy (MPTL)

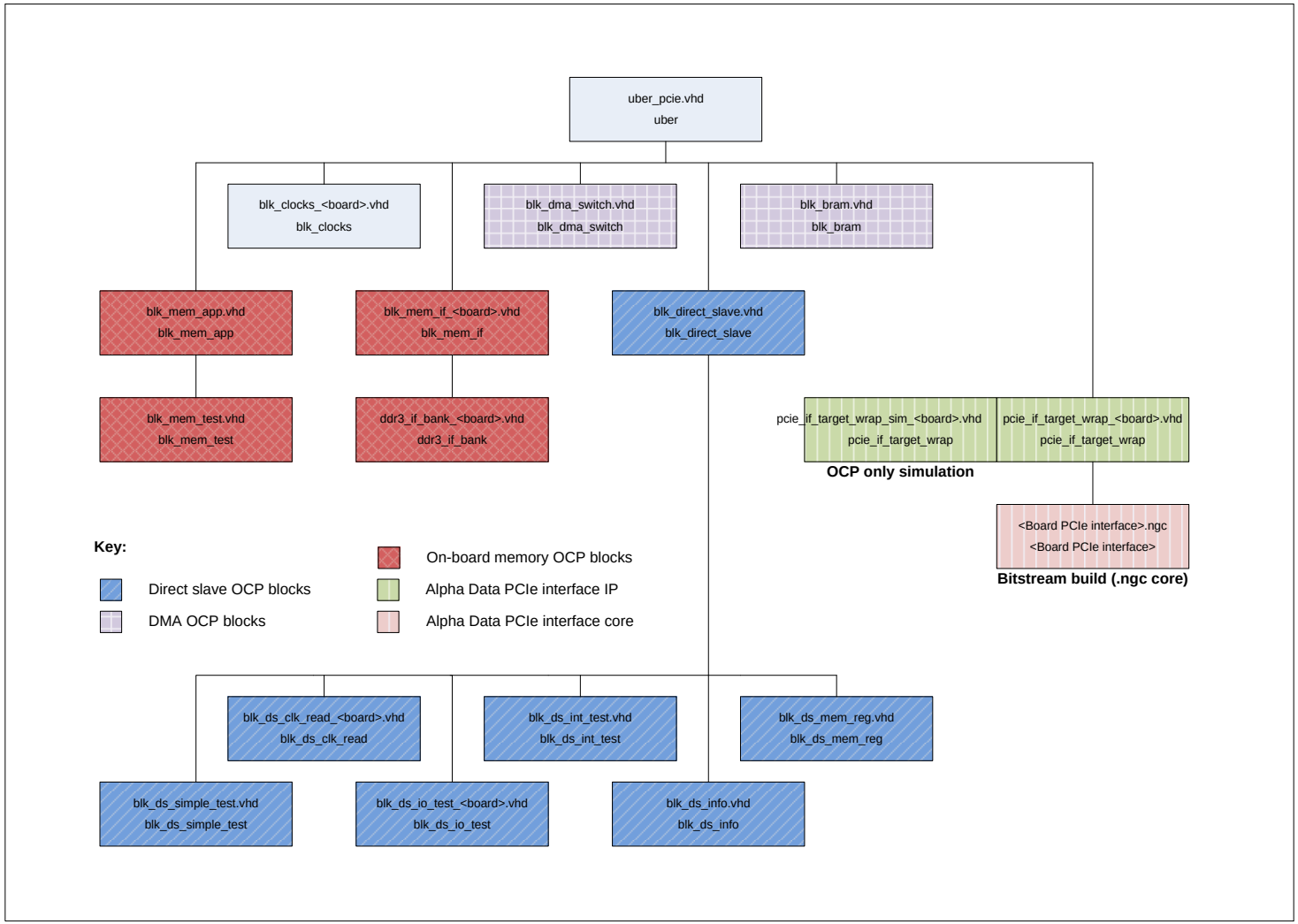

Figure 15: Uber Design Top Level Hierarchy (PCIe)

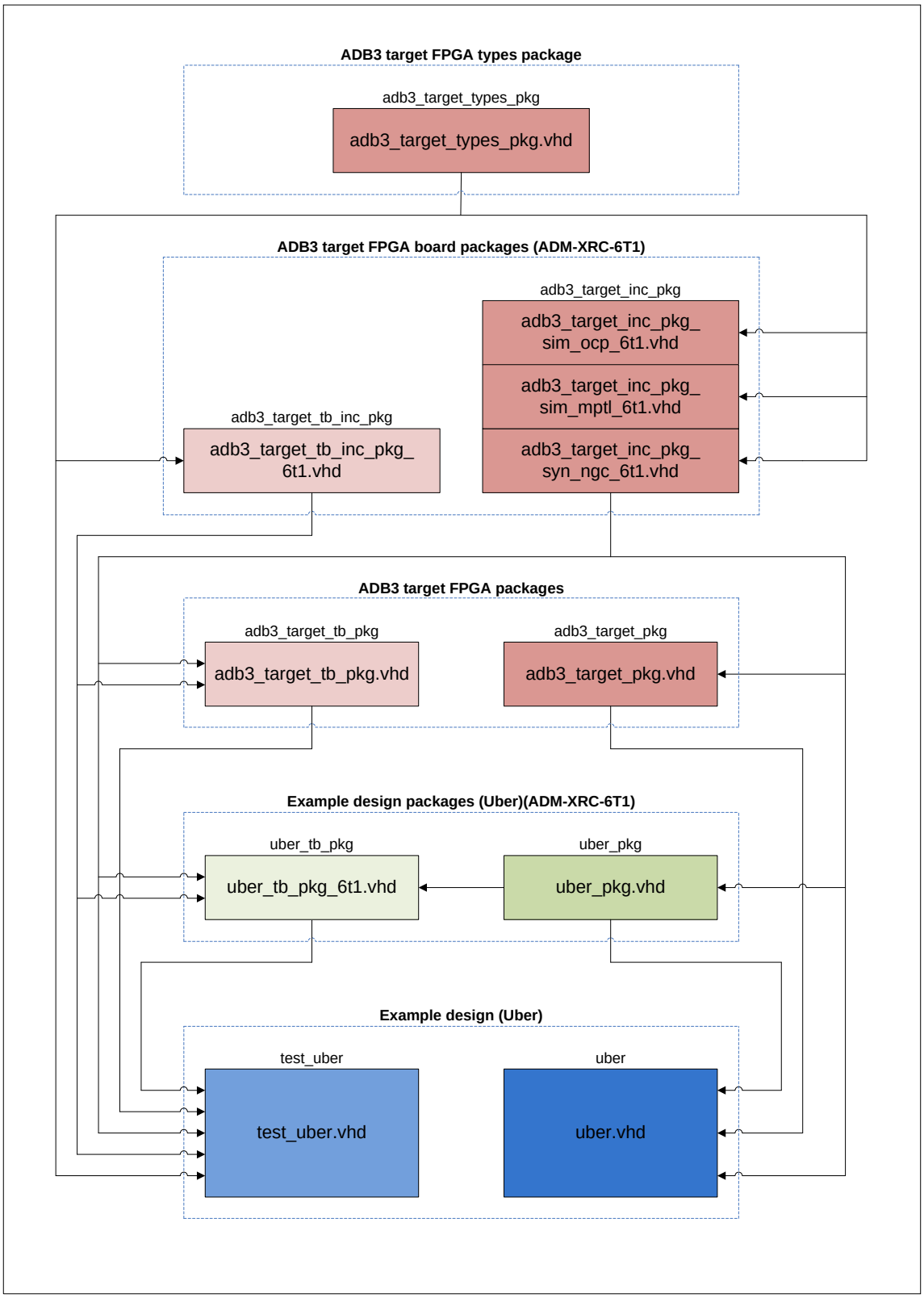

Figure 16: Uber Design Package Dependencies

5.5.4.1 Clock and Reset Generation

The clock and reset generation block is implemented by the **blk_clks** block which is board dependent.

Table 11 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-6TADV8	admxrc6tadv8/blk_clks_6tadv8.vhd

Table 11: 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**

5.5.4.1.1 Internal Clock Generation (MMCM)

This consists of an Xilinx MMCM block driven by the **clks_in.ref_clk** global clock input. It generates three output clocks: **pll_pri_clk**, **pll_reg_clk**, and **pll_mem_clk**. Refer to **Figure 17**.

pll_ref_clk

- This is used as a reference clock by the design.
- It is fixed at 200 MHz and used to measure the frequencies of the other clocks in the clock frequency measurement section, as well as being the reference clock for the IODELAYCTRL instances used in the DDR3 SDRAM interfaces. The three clocks immediately below are derived from this clock.
- The source of this clock is the **clks_in.ref_clk** global clock input.

pll_pri_clk

- This clock is used as the primary OCP clock by the design.
- It is derived from **pll_ref_clk** and set to 200 MHz. It drives much of the OCP logic in the **Uber** design, including the DMA OCP section.

pll_reg_clk

- This is used as a low frequency clock by the design.
- It is derived from **pll_ref_clk** and set to 80 MHz. It drives the low-frequency OCP Direct Slave register section.
- Its frequency need not be related to any of the other clocks.

pll_mem_clk

- This is used as the clock for the DDR3 SDRAM memory interfaces in the design.
- It is derived from **pll_ref_clk** and set to 400 MHz. It drives the on-board memory interface section.

5.5.4.1.2 Internal Reset Generation (MMCM)

An active high asynchronous user reset **pll_rst** is generated from the MMCM locked signal. Refer to **Figure 17**.

5.5.4.1.3 MPTL Interface Clock Generation

The **target MPTL interface** requires an unbuffered differential **mptl_clk** clock input. Its source is dependent on the board selected. Refer to **Figure 18**.

5.5.4.1.4 PCIe Interface Clock Generation

The **target PCIe interface** requires an unbuffered differential **pcie_clk** clock input. Its source is dependent on the board selected. Refer to **Figure 18**.

5.5.4.1.5 Input Clock Buffering

Clocks are input on the **clks_in** signal of type **clks_in_t** and are buffered. Clock support is dependent on the board selected. Type **clks_in_t** is defined in the **ADB3 target include package (adb3_target_inc_pkg)**.

Refer to **Figure 18**.

5.5.4.1.6 Input Clock Extraction (MGT Sourced)

MGT sourced clocks are input on the **clks_mgt_in** signal of type **clks_mgt_in_t**. MGT sourced clock support is dependent on the board 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 18**.

5.5.4.1.7 Output Clock Generation

Clocks are generated and output on the **clks_out** signal of type **clks_out_t**. Clock support is dependent on the board selected. Type **clks_out_t** is defined in the **ADB3 target include package (adb3_target_inc_pkg)**.

Refer to **Figure 18**.

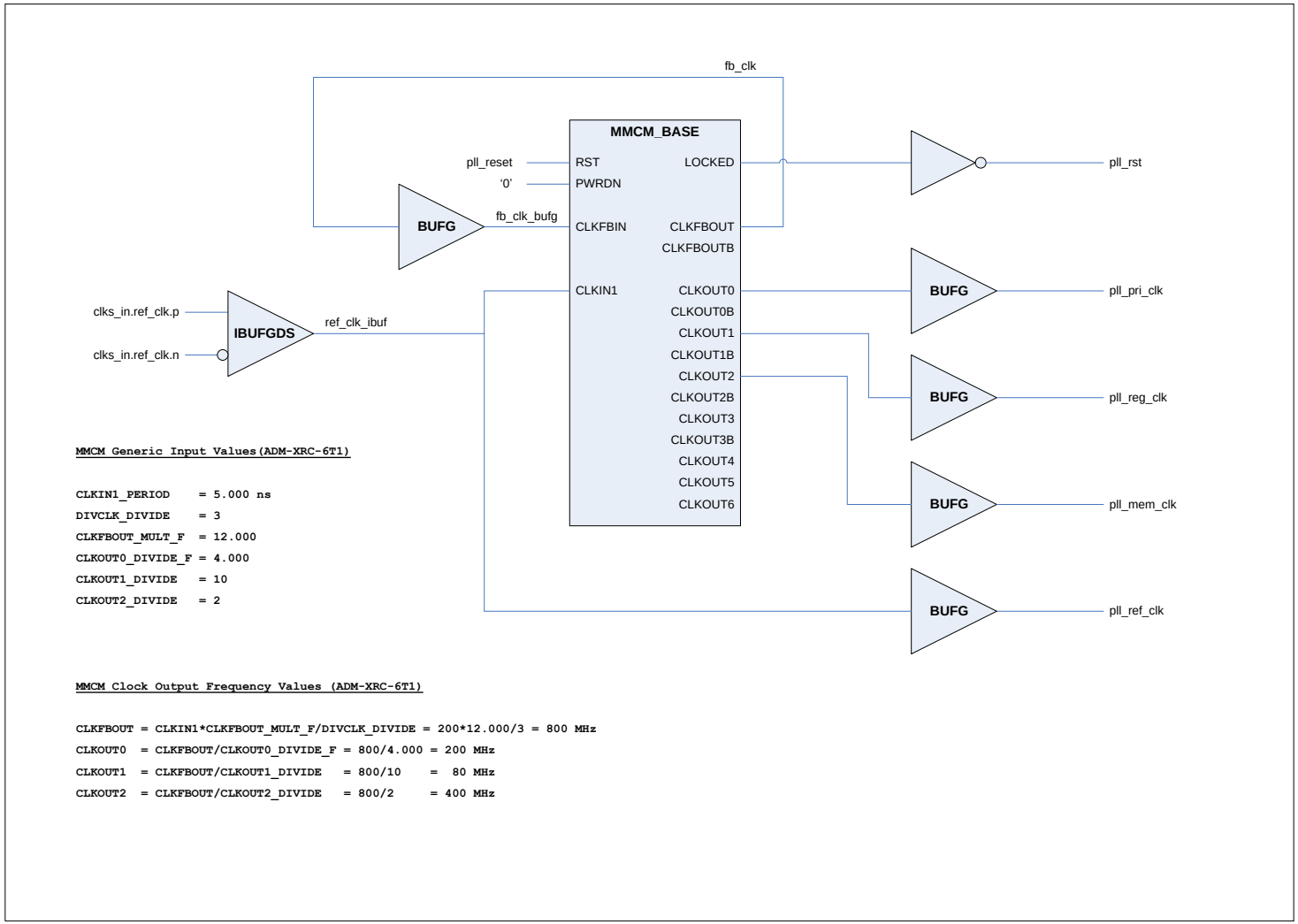

Figure 17: Uber Design Internal Clock Generation (MMCM)

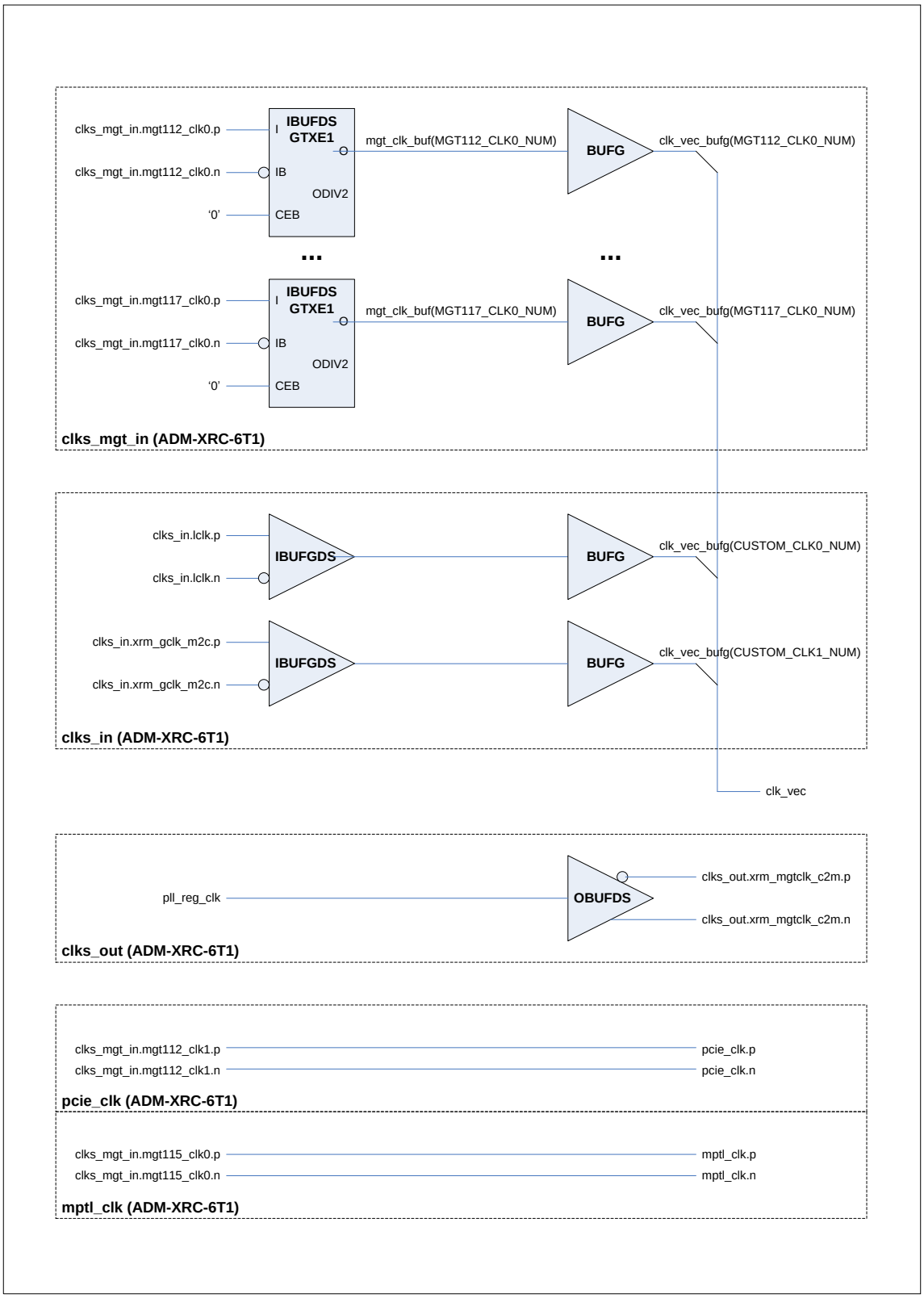

Figure 18: Uber Design Clock Buffering/Extraction

5.5.4.2 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 board 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 a call to api macro **ADMXRC3_ConfigureFromFile** to wait until MPTL communication has been completed, thus guaranteeing that the Target FPGA is in the proper state for software on the host to communicate with it.

> **Note:** The Direct Slave and DMA address spaces supported by the Bridge FPGA are smaller than the full ADB3 OCP address space. For the board 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.

5.5.4.3 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 board 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 board 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.

5.5.4.4 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 19**.

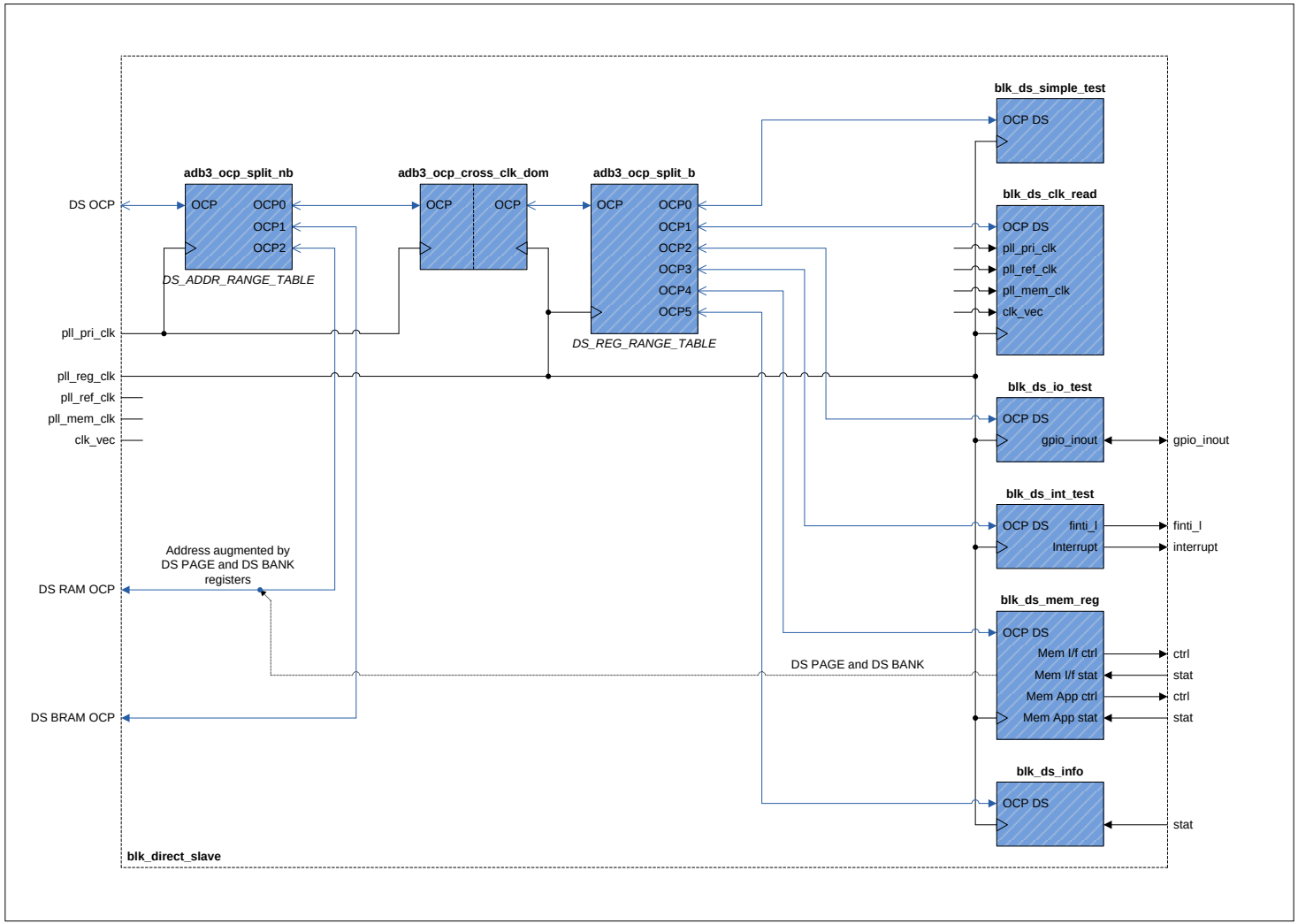

Figure 19: Uber Direct Slave Block Diagram

5.5.4.4.1 Direct Slave Address Space Splitter

An instance of the ADB3 OCP component **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 12** 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 12: 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.

5.5.4.4.2 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.

5.5.4.4.2.1 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**.

5.5.4.4.2.2 Direct Slave Register Address Space Splitter

An instance of the ADB3 OCP component **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

So, the address bits used in comparison = 0x00000000_003FFFC0.

When an incoming OCP address must be decoded, decoding is performed as follows for address range 1:

Incoming OCP address (for example) = 0x00000000_000000D0
Masked incoming OCP address = 0x00000000_000000C0

So, the address hits address range 1, as masked incoming OCP address = address range 1 base.

Table **Table 13** 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 13: 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.

5.5.4.4.2.3 Simple Test Register Block

5.5.4.4.2.3.1 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.

5.5.4.4.2.3.2 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 14: 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 15: Simple Test Register Block, DATA Register (0x000000)

5.5.4.4.2.4 Clock Frequency Measurement Register Block

5.5.4.4.2.4.1 Description

The clock frequency measurement register block is implemented by the **blk_ds_clk_read** block which is board dependent.

Table 16 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-6TADV8	admxrc6tadv8/blk_ds_clk_read_6tadv8.vhd

Table 16: 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 (board 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)

Table 17: Internally Generated Clock Frequency Measurement

Clock frequency measurement components **blk_clock_freq** are also instantiated for each board-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 18: Externally Sourced Clock Frequency Measurement (ADM-XRC-6T1)

Within this block, a function **conv_ref_clk_tcval** returns the clock frequency measurement period, and hence the measurement resolution, as a function of the **TARGET_USE** constant from the package **adb3_target_inc_pkg**. The **REF_CLK_TCVAL** constant defines the measurement period in **pll_ref_clk** cycles as follows:

OCP-only simlation (TARGET_USE = SIM_OCP)

- Period = (REF_CLK_FREQ_HZ/1000000) ref_clk cycles = 1µs.
- Resolution = 1MHz.

Full MPTL simlation (TARGET_USE = SIM_MPTL)

- Period = (REF_CLK_FREQ_HZ/1000000) ref_clk cycles = 1µs.
- Resolution = 1MHz.

Synthesis (TARGET_USE = SYN_NGC)

- Period = (REF_CLK_FREQ_HZ) ref_clk cycles = 1s.
- Resolution = 1Hz.

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 **Clock Frequency Measurement Block Constraints** still holds for every **blk_clock_freq** instance.

5.5.4.4.2.4.2 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 19: Clock Frequency Measurement Register Block Address Map

Bits	Mnemonic	Type	Function
31:5			(Reserved)
4:0	SEL_CLK	M	Selects which clock's measured frequency and flags are available in the FREQ and STAT registers, respectively. 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 => Unused (4) 00101 => Unused (5) 00110 => Custom Clock 0 (External clock)(06) 00111 => Custom Clock 1 (External clock)(07) 01000 => Custom Clock 2 (External clock)(08) 01001 => Custom Clock 3 (External clock)(09) 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) 11100 => mgt118_clk0 (External MGT clock)(30) 11100 => mgt118_clk1 (External MGT clock)(31)

Table 20: Clock Frequency Measurement Register Block, SEL Register (0x000040)

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.
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 21: 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 22: Clock Frequency Measurement Register Block, FREQ Register (0x000048)

5.5.4.4.2.5 Interrupt Test Register Block

5.5.4.4.2.5.1 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.

5.5.4.4.2.5.2 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 23: 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 24: 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 25: 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 26: 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 27: 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 28: 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.

5.5.4.4.2.6 Informational Register Block

5.5.4.4.2.6.1 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.

5.5.4.4.2.6.2 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 29: 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 30: 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 31: 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 32: 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 33: 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 34: 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 35: 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 36: 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 37: 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 38: Informational Register Block, SDK_VER Register (0x000160)

5.5.4.4.2.7 GPIO Test Register Block

5.5.4.4.2.7.1 Description

The GPIO test register block is implemented by the **blk_ds_io_test** block which is board dependent.

Table 39 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-6TADV8	admxrc6tadv8/blk_ds_io_test_6tadv8.vhd

Table 39: 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)

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.

5.5.4.4.2.7.2 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

Table 40: 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 41: 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 42: 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 43: 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 44: 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 45: 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 46: 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 47: 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 48: 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 49: 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 50: 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 51: 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 52: 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 53: 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 54: 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).
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 55: 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 56: 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 57: 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 58: 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 59: 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 60: 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 61: 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 62: 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 63: 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 64: 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 65: 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 66: 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 67: GPIO Test Register Block, PN6_GPIO_LS_TRI Register (0x000268)

5.5.4.4.2.8 On-Board Memory Register Block

5.5.4.4.2.8.1 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.

5.5.4.4.2.8.2 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 ...

Table 68: On-Board Memory Register Block Address Map (continued on next page)

Name	Address
BANK0_INFO BANK1_INFO ...	0x00032C 0x00034C ...
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 68: 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 69: 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 70: 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.
7:0			(Reserved)

Table 71: 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 72: 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 73: 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 74: 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 75: 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 76: 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. Refer to **ADB3 OCP** for a description.

Table 77: 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. Refer to **ADB3 OCP** for a description.

Table 78: On-Board Memory Register Block, BANKx_IF_ERR Register (0x00033C, 0x00035C, ...)

5.5.4.4.3 Direct Slave BRAM Address Space

5.5.4.4.3.1 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**.

5.5.4.4.3.2 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 79: Direct Slave BRAM Access Window

5.5.4.4.4 Direct Slave On-Board Memory Address Space

5.5.4.4.4.1 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**.

5.5.4.4.4.2 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 80: 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 81**.

5.5.4.5 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 20**.

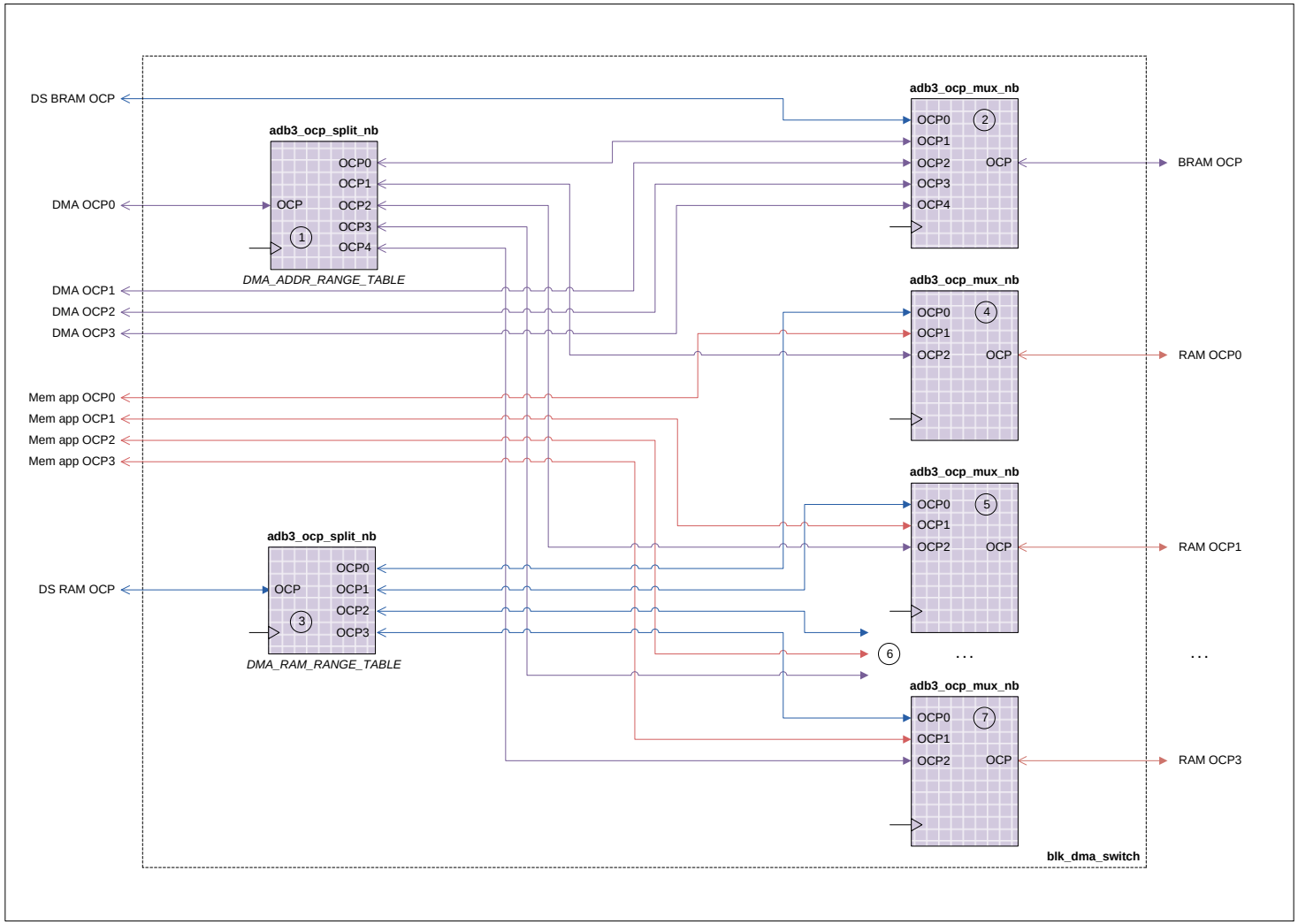

Figure 20: Uber OCP Switching Block

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

5.5.4.5.1 Direct Slave On-Board Memory OCP Address Space Splitter Block

Referring to item 1 in **Figure 20**, 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 81** 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 81: 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.

5.5.4.5.2 BRAM OCP Multiplexor Block

Referring to item 2 in **Figure 20**, 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

5.5.4.5.3 DMA Channel 0 OCP Address Space Splitter Block

Referring to item 3 in **Figure 20**, this instance of **adb3_ocp_split_nb** splits DMA OCP channel 0 into multiple secondary OCP channels according to the address map in **Table 82**.

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 82: 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.

5.5.4.5.4 On-Board Memory Bank OCP Multiplexors

Items 4, 5, 6 and 7 in **Figure 20** 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)

5.5.4.6 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 82**.
- Any other DMA channel, where the BRAM block is aliased throughout the entire OCP address space.

Figure 21 shows the BRAM block and its associated part of the **OCP switching block**.

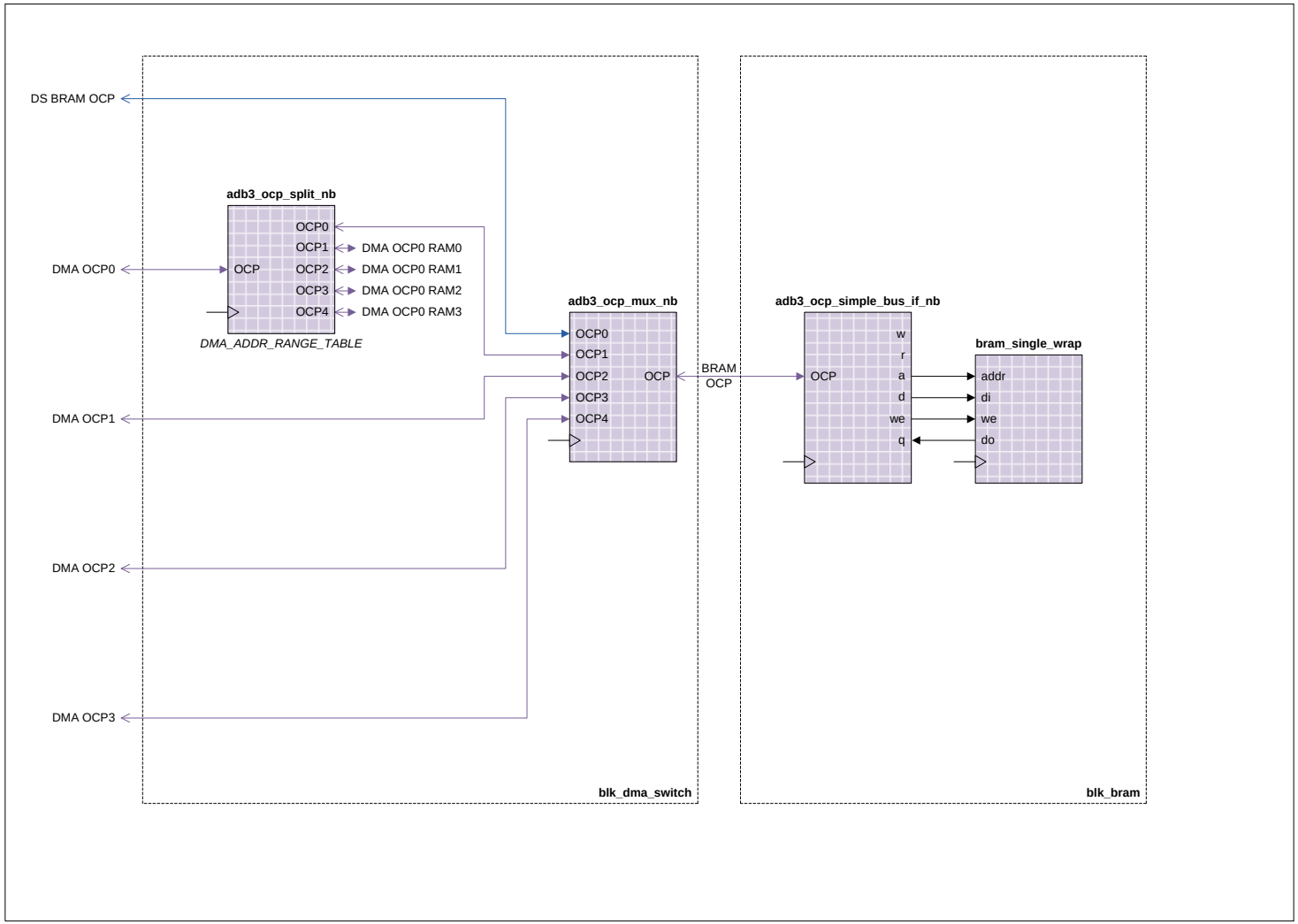

Figure 21: Uber BRAM Block Diagram

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.

5.5.4.7 On-Board Memory Interface Block

The on-board memory interface block is implemented by the **blk_mem_if** block which is board dependent.

Table 83 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-6TADV8	admxrc6tadv8/blk_mem_if_6tadv8.vhd

Table 83: 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 82**.
- **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 boards 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 22 shows the on-board memory interface block and its associated part of the **OCP switching block**.

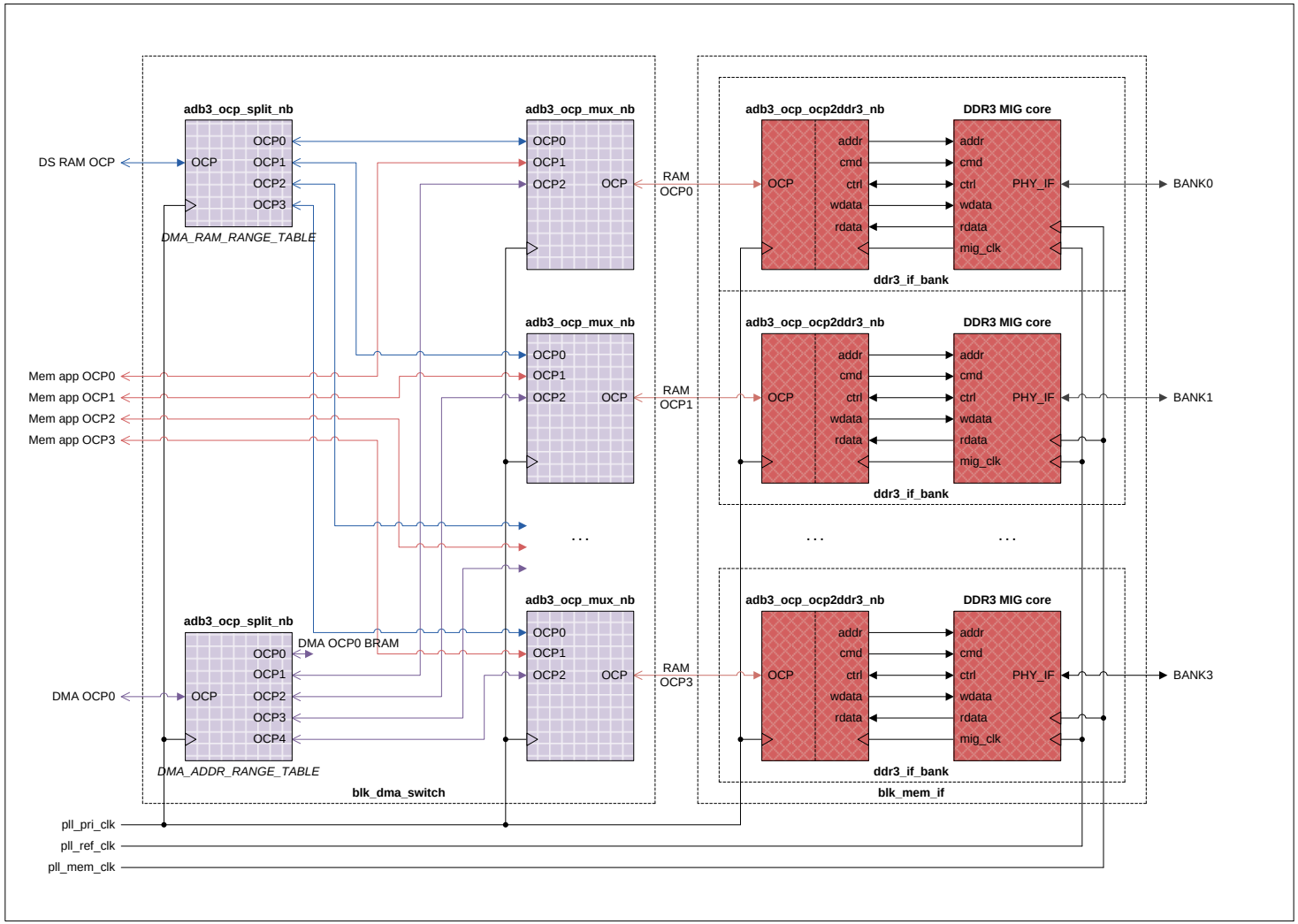

Figure 22: Uber Memory Interface Block Diagram

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. 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**.

5.5.4.8 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.

5.5.4.9 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.

5.5.4.10 Design Package (uber_pkg)

The package **uber_pkg** defines types, constants, and functions which are used by the **Uber** example FPGA design.

Defininitions 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_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**

5.5.5 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 84** 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-6TADV8	PCIe	test_uber_pcie.vhd

Table 84: 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 23 shows the testbench and main elements of the **uber** FPGA design using MPTL interface IP.

Figure 24 shows the testbench and main elements of the **uber** FPGA design using PCIe interface IP.

Figure 25 shows the hierarchy of the **uber** testbench using MPTL interface IP.

Figure 26 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 types definition package (adb3_target_types_pkg)**
- **ADB3 target include package (adb3_target_inc_pkg)**
- **ADB3 target testbench include package (adb3_target_tb_inc_pkg)**
- **ADB3 target testbench package (adb3_target_tb_pkg)**
- **Testbench package (uber_tb_pkg)**

Figure 16 shows the design package dependencies.

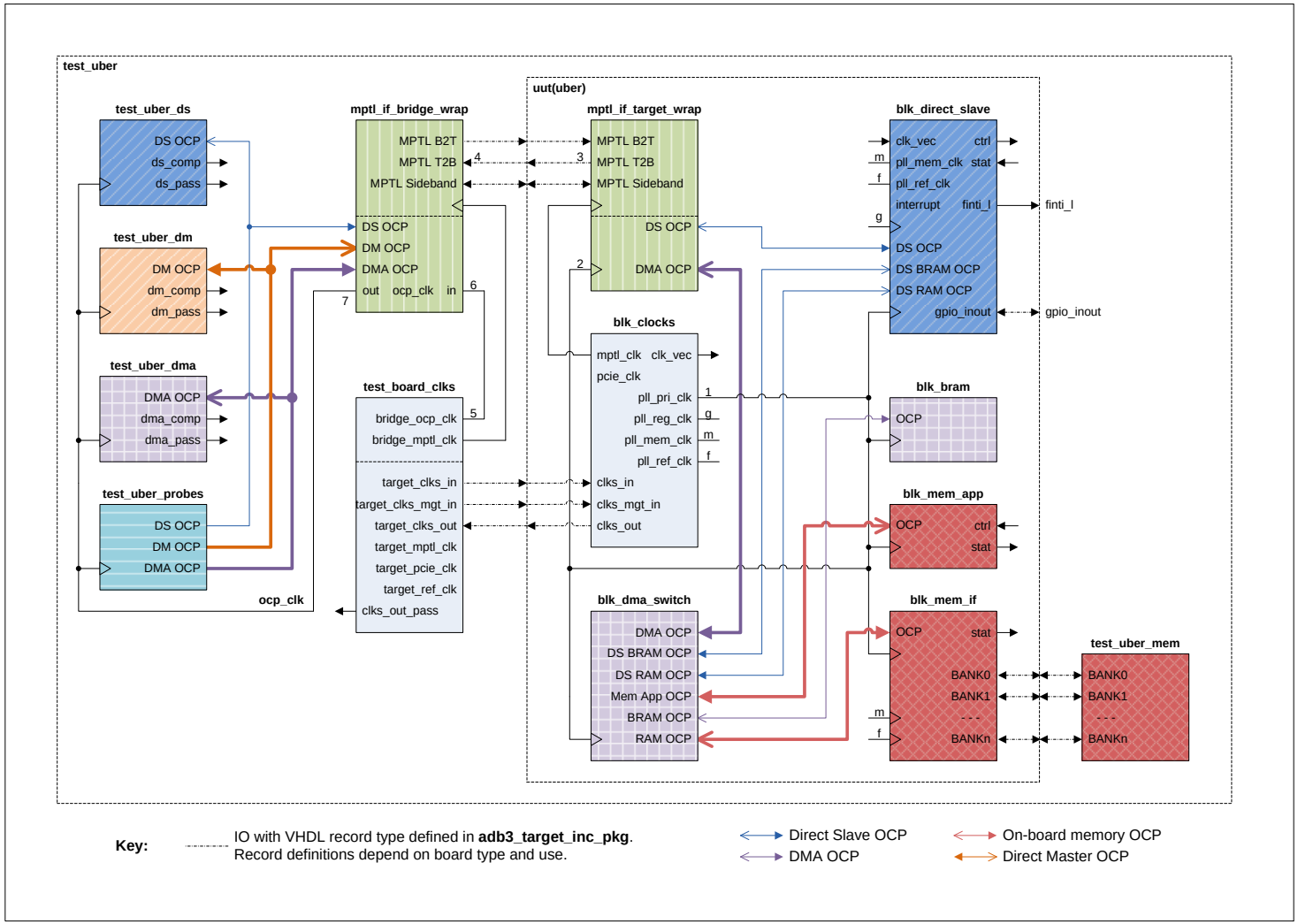

Figure 23: Uber Design Testbench and Top Level Block Diagram (MPTL)

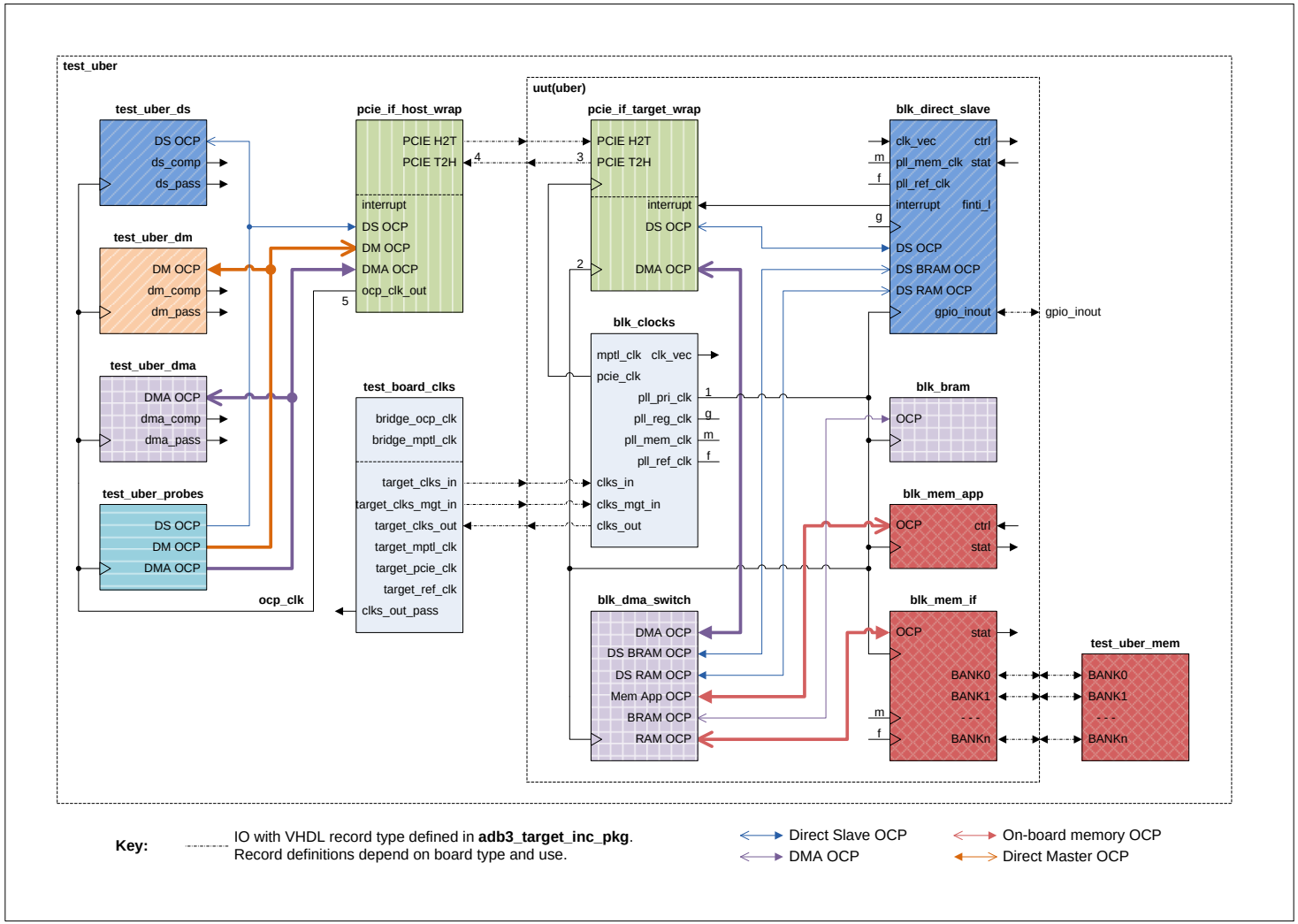

Figure 24: Uber Design Testbench and Top Level Block Diagram (PCIe)

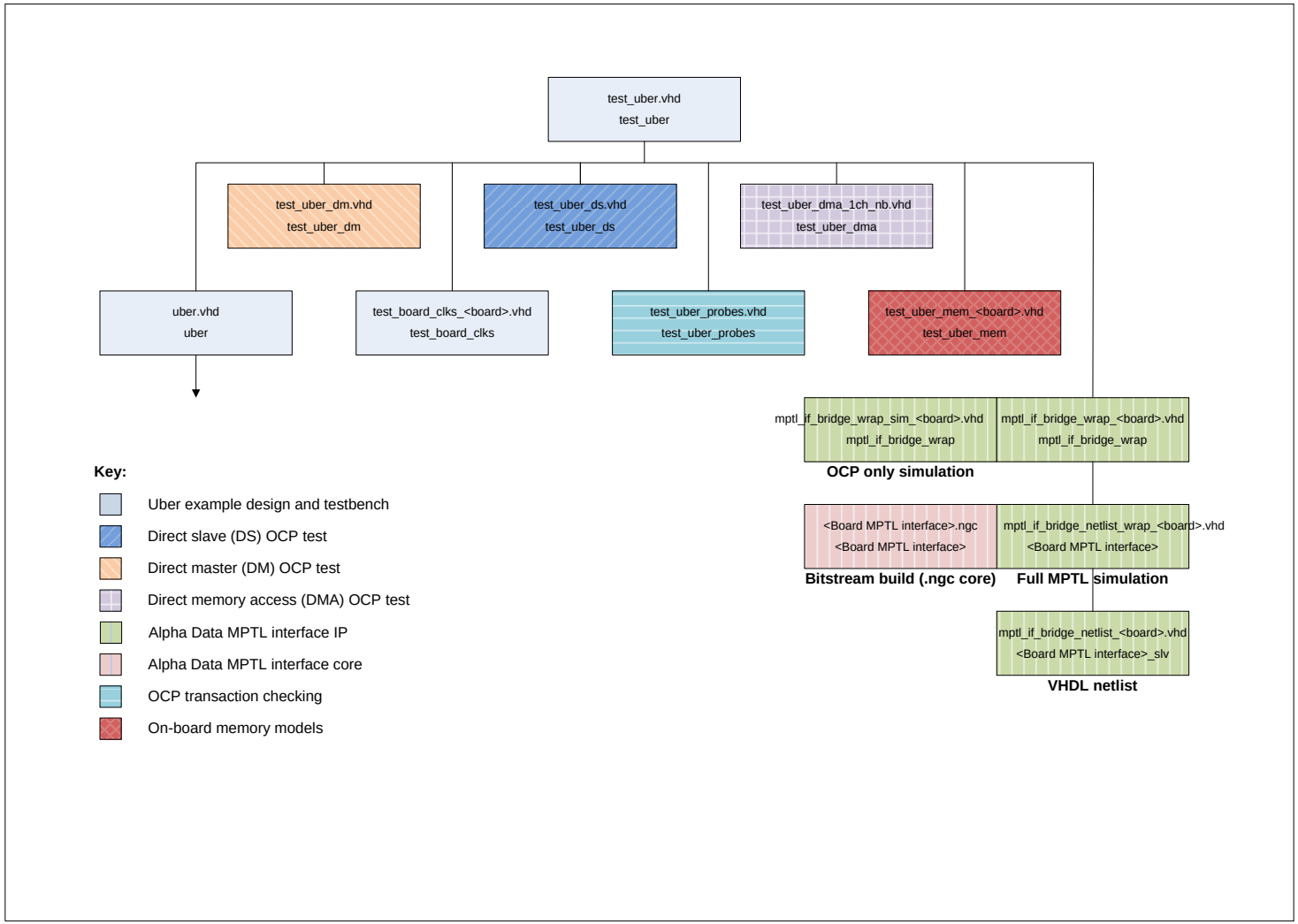

Figure 25: Uber Design Testbench Hierarchy (MPTL)

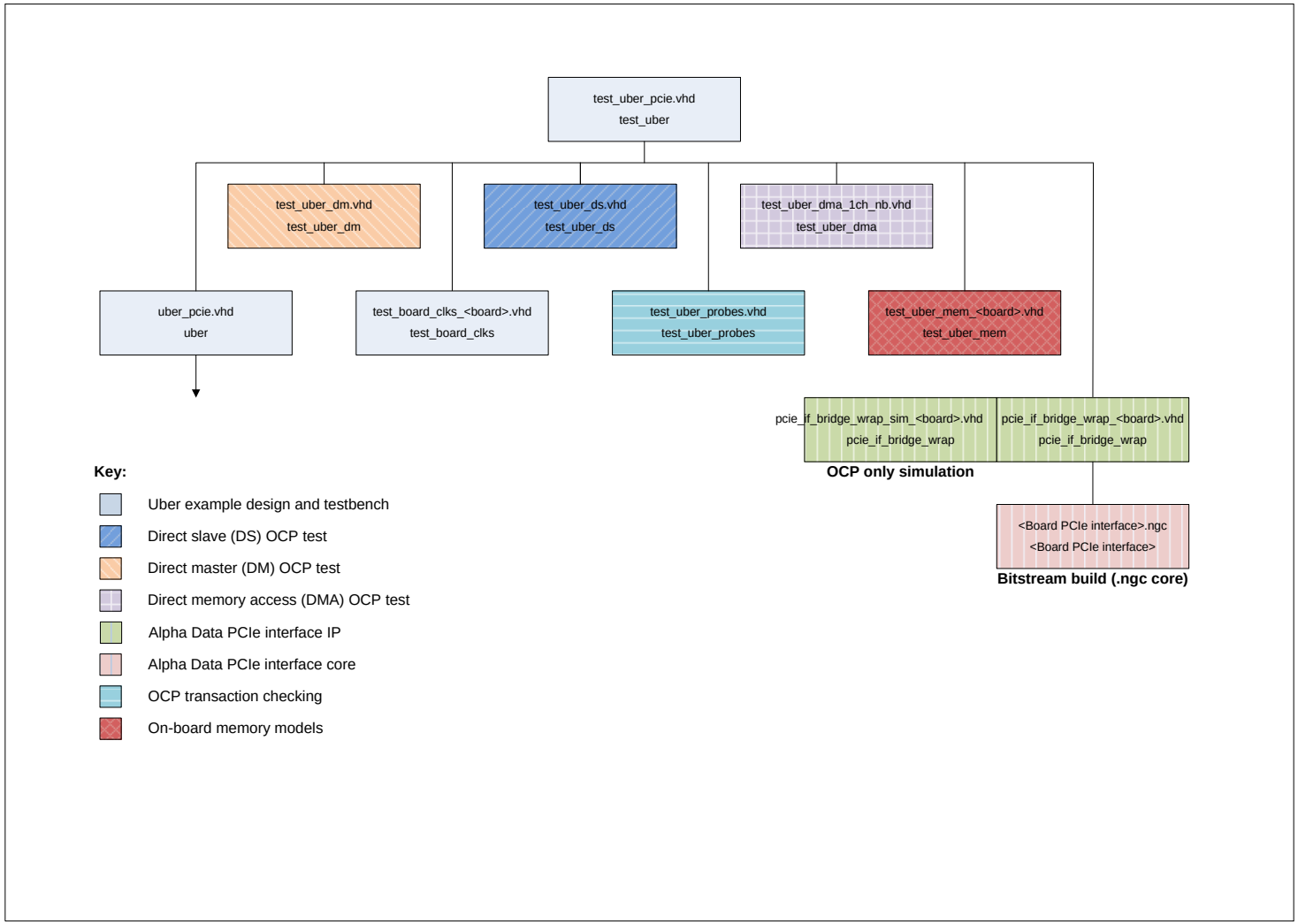

Figure 26: Uber Design Testbench Hierarchy (PCIe)

5.5.5.1 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 board 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 board-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 board-specific way, depending on the package **adb3_target_inc_pkg**.
- It tests the **clks_out** clock frequency according to which board 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 board 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 board 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.

5.5.5.2 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 board 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 23** 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 23** 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 during a full MPTL simulation.

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

5.5.5.3 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 board 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 **mptl_if_bridge_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 24** 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.

5.5.5.4 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**.

5.5.5.5 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.

5.5.5.5.1 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.

5.5.5.5.1.1 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.

5.5.5.5.1.2 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**.

5.5.5.5.1.3 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.

5.5.5.5.2 Direct Slave OCP Channel

Note: all filenames mentioned in this section are relative to the path **hdl/vhdl/examples/uber/common/**.

An instance of the **test_uber_ds** component, implemented in **test_uber_ds.vhd**, provides test stimulus to and verifies test results from the UUT's OCP Direct Slave channel.

test_uber_ds 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.

5.5.5.5.2.1 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**.

5.5.5.5.2.2 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**:

5. Selects **pll_pri_clk** by writing 1 (corresponding to **PLL_PRI_CLK_SEL**) to the **SEL** register.
6. Waits for a measurement to be completed, by polling until bit 31 of the **CTRL/STAT** register is 1.
7. 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**):

8. Selects the **TEST_MGT_CLK** clock by writing **TEST_MGT_CLK_SEL** to the **SEL** register.
9. Waits for a measurement to be completed, by polling until bit 31 of the **CTRL/STAT** register.
10. Reads the **FREQ** register (see **Table 22**) 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**:

11. Selects the **TEST_CUS_CLK** clock by writing **TEST_CUS_CLK_SEL** to the **SEL** register.
12. Waits for a measurement to be completed, by polling until bit 31 of the **CTRL/STAT** register.
13. Reads the **FREQ** register (see **Table 22**) 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**.

5.5.5.5.2.3 XRM GPIO Test

This test 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**.

5.5.5.5.2.4 Pn4/Pn6 GPIO Test

This test 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:

5. 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).
6. 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.
7. 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.
8. 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**.

5.5.5.5.2.5 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**.

5.5.5.5.2.6 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**.

5.5.5.5.2.7 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**.

5.5.5.5.2.8 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**.

5.5.5.5.3 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.

5.5.5.5.3.1 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**.

5.5.5.5.3.2 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**.

5.5.5.6 On-Board Memory Simulation Models

The on-board memory model block is implemented by the **test_uber_mem** block which is board dependent.

Table 85 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-6TADV8	admxrc6tadv8/test_uber_mem_6tadv8.vhd

Table 85: 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 86 lists the available variants:

Model	Filename relative to hdl/vhdl/common/mem_tb/
DDR3 SDRAM	ddr3_sdram/ddr3_sdram.vhd

Table 86: Available Variants of On-Board Memory Models

Refer to **DDR3 SDRAM Memory Model** for its functional description.

The DDR3 SDRAM part to be simulated is defined by selecting either **DDR3_1G_PART**, **DDR3_2G_PART**, or **DDR3_4G_PART** 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 1Gib. The user should verify that this matches the size of the memory parts on the Alpha Data board in use and selected by the package **adb3_target_inc_pkg**.

5.5.5.7 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 87** 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-6TADV8	admxrc6tadv8/uber_tb_pkg_6tadv8.vhd

Table 87: Available Variants of uber_tb_pkg Package

Defininitions are as follows:

General test constants

- **MPTL_GPIO**. The value used for the MPTL GPIO loopback test on this board.

DS clock test constants

- **CLK_TEST_DIFF**. The acceptable difference for clock frequency measurement results on this board.
- **PLL_PRI_CLK_FREQ**. The **pll_pri_clk** expected clock frequency measurement result on this board.
- **PLL_REG_CLK_FREQ**. The **pll_reg_clk** expected clock frequency measurement result on this board.
- **PLL_MEM_CLK_FREQ**. The **pll_mem_clk** expected clock frequency measurement result on this board.
- **PLL_REF_CLK_FREQ**. The **pll_ref_clk** expected clock frequency measurement result on this board.
- **CLKS_OUT_FREQ**. Defines **clks_out** output clocks expected frequencies (MHz).
- **BRIDGE_LCLK_FREQ**. Defines frequency (Hz) of lclk (generated by bridge).
- **TARGET_LCLK_PER**. Defines period of lclk (generated by bridge).
- **TEST_MGT_CLK_SEL**. The **TEST_MGT_CLK** clock frequency measurement clock select index.
- **TEST_CUS_CLK_SEL**. 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 board (rounded).
- **TEST_CUS_CLK_FREQ**. The **TEST_CUS_CLK** expected clock frequency measurement result (MHz) on this board (rounded).

DS GPIO test constants

- **PN4_DATA**. Data used during Pn4 test on this board.
- **PN6_DATA**. Data used during Pn6 test on this board.

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**

5.5.6 Design Simulation

Modelsim macro files are located in each of the target FPGA example 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 board 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 Modelsim 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 **Xilinx DDR3 SDRAM MIG Core Generation** 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 **DDR3_1G_PART**, **DDR3_2G_PART**, or **DDR3_4G_PART** 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 Gib constants are appropriate for the size of the memory parts on the Alpha Data board in use.

Note: The Modelsim macro files always delete any previously compiled data before compiling the **Uber** design.

5.5.6.1 Date/Time Package Generation

Before compiling the **Uber** example design HDL and initiating simulation, the macro file runs a TCL script **gen_today_pkg.tcl** to generate a file containing the **today_pkg** package. This package contains HDL constant definitions containing the SDK version and date/time at which the script was run. The file generated is dependent on the board selected and is located in the board design directory; for example, **hdl/vhdl/examples/uber/admxrc6t1/today_pkg_admxrc6t1_sim.vhd** for the ADM-XRC-6T1. Transcript output is of the form:

```
--
-- today_pkg_admxrc6t1_sim.vhd
-- 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"00010400";
  constant TODAYS_DATE : std_logic_vector(31 downto 0) := X"08102010";
  constant TODAYS_TIME : std_logic_vector(31 downto 0) := X"15264600";

end package today_pkg;
```

Note: The macro file runs the TCL script using the Xilinx customized TCL distribution TCL shell **xtclsh**. The path to this shell must be defined before initiating simulation.

5.5.6.2 Initialisation Results

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

5.5.6.2.1 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
############################################################
```

5.5.6.2.2 Testbench Status (MPTL)

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

```
# ** Note: Board Type : adm_xrc_6t1
#    Time: 0 fs  Iteration: 0  Instance: /test_uber
# ** Note: Target Use : sim_ocp
#    Time: 0 fs  Iteration: 0  Instance: /test_uber
# ** Note: Waiting for MPTL online....
#    Time: 0 fs  Iteration: 0  Instance: /test_uber
```

5.5.6.2.3 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 MIG 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
```

5.5.6.3 Non-OCP Functions Results

5.5.6.3.1 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
```

5.5.6.4 Direct Slave OCP Channel Results

5.5.6.4.1 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
```

5.5.6.4.2 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 0x00000050 from byte address 0x000048
#    Time: 3607500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 80 MHz ±2 MHz
#    Time: 3607500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 80 MHz
#    Time: 3607500 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: 3615 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PLL_PRI_CLK_FREQ 4 bytes 0x000000C8 from byte address 0x000048
#    Time: 4307500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 200 MHz ±2 MHz
#    Time: 4307500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 200 MHz
#    Time: 4307500 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: 4315 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read TEST_MGT_CLK_FREQ 4 bytes 0x000000F8 from byte address 0x000048
#    Time: 5007500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 250 MHz ±2 MHz
#    Time: 5007500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 252 MHz
#    Time: 5007500 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: 5015 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read TEST_CUS_CLK_FREQ 4 bytes 0x000002A0 from byte address 0x000048
#    Time: 5707500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 666 MHz ±2 MHz
#    Time: 5707500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual   freq = 664 MHz
#    Time: 5707500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Clock Read completed: PASSED.
#    Time: 5707500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

Results will be board dependent. Results are shown for ADM-XRC-6T1.

5.5.6.4.3 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 board dependent. Results are shown for ADM-XRC-6T1.

5.5.6.4.4 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 board dependent. Results are shown for ADM-XRC-6T1.

5.5.6.4.5 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
```

5.5.6.4.6 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 0x00010400 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
```

5.5.6.4.7 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
```

5.5.6.4.8 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 0b111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
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 0b11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
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 0b1111111111111111111111111111111111111111111111111111111111111111
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 0b11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
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
```

5.5.6.5 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
```

5.5.6.6 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**
- **Memory Application**
- **Memory Model**
- **Clock Frequency Measurement**
- **ChipScope**

6.1 ADB3 OCP

The ADB3 OCP group is located in **hdl/vhdl/common/adb3_ocp** and contains the following elements:

- **ADB3 OCP Profile Definition Package (adb3_ocp)**
- **ADB3 OCP Component Declaration Package (adb3_ocp_comp)**
- **ADB3 OCP Components**
- **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 profile. This OCP profile is used for many of the reuseable VHDL modules in this SDK, and to connect together the various blocks in the example FPGA designs.

Two main types are defined:

Burst capable data flow from OCP Master to OCP Slave (M2S)

- Command **Cmd** of type **ocp_CmdT** (Idle, Write, Read, Write Non Post).
- Command Start Address **Addr** of type **std_logic_vector** with width **ADB3_OCP_ADDR_WIDTH** = 64.
- Command Burst Length **BurstLength** of type **std_logic_vector** with width **ADB3_OCP_BURST_WIDTH** = 12.
- Command Tag **Tag** of type **std_logic_vector** with width **ADB3_OCP_TAG_WIDTH** = 8.
- Data **Data** of type **std_logic_vector** with width **ADB3_OCP_DATA_WIDTH** = 128.
- Data Byte Enable **DataByteEn** of type **std_logic_vector** with width **ADB3_OCP_BE_WIDTH** = 16.
- Data Valid **DataValid** of type **std_logic**.
- Response Accept **RespAccept** of **type std_logic**.

Burst capable data flow from OCP Slave to OCP Master (S2M)

- Command Accept **CmdAccept** of type **std_logic**.
- Data Accept **DataAccept** of type **std_logic**.
- Response Data **Data** of type **std_logic_vector** with width **ADB3_OCP_DATA_WIDTH** = 128.
- Response Type **Resp** of type **ocp_RespT** (None, Valid, Failed, Error).
- Response Tag **Tag** of type **std_logic_vector** with width **ADB3_OCP_TAG_WIDTH** = 8.

Refer to **ADB3 OCP Protocol Reference** for a description of ADB3 OCP protocol transactions.

6.1.2 ADB3 OCP Component Declaration Package (adb3_ocp_comp)

The package **adb3_ocp_comp** defines functions and components which use the ADB3 OCP profile.

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_simple_bus_if**
- **adb3_ocp_split_b**

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 DMA. 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_ocp2ddr3_nb**
- **adb3_ocp_retime_nb**
- **adb3_ocp_simple_bus_if_nb**
- **adb3_ocp_split_nb**

6.1.3 ADB3 OCP Components

6.1.3.1 adb3_ocp_cross_clk_dom

6.1.3.1.1 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)**

6.1.3.1.2 Interface

The **adb3_ocp_cross_clk_dom** component interface is shown in **Figure 27** below and described in **Table 88**.

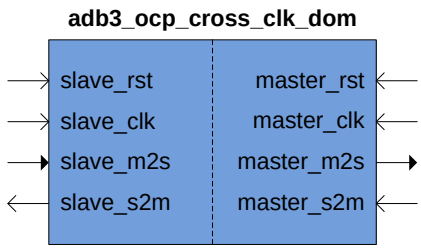

Figure 27: adb3_ocp_cross_clk_dom Component Interface

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 88: adb3_ocp_cross_clk_dom Component Interface

6.1.3.1.3 Description

The **adb3_ocp_cross_clk_dom** component block diagram is shown in **Figure 28** below.

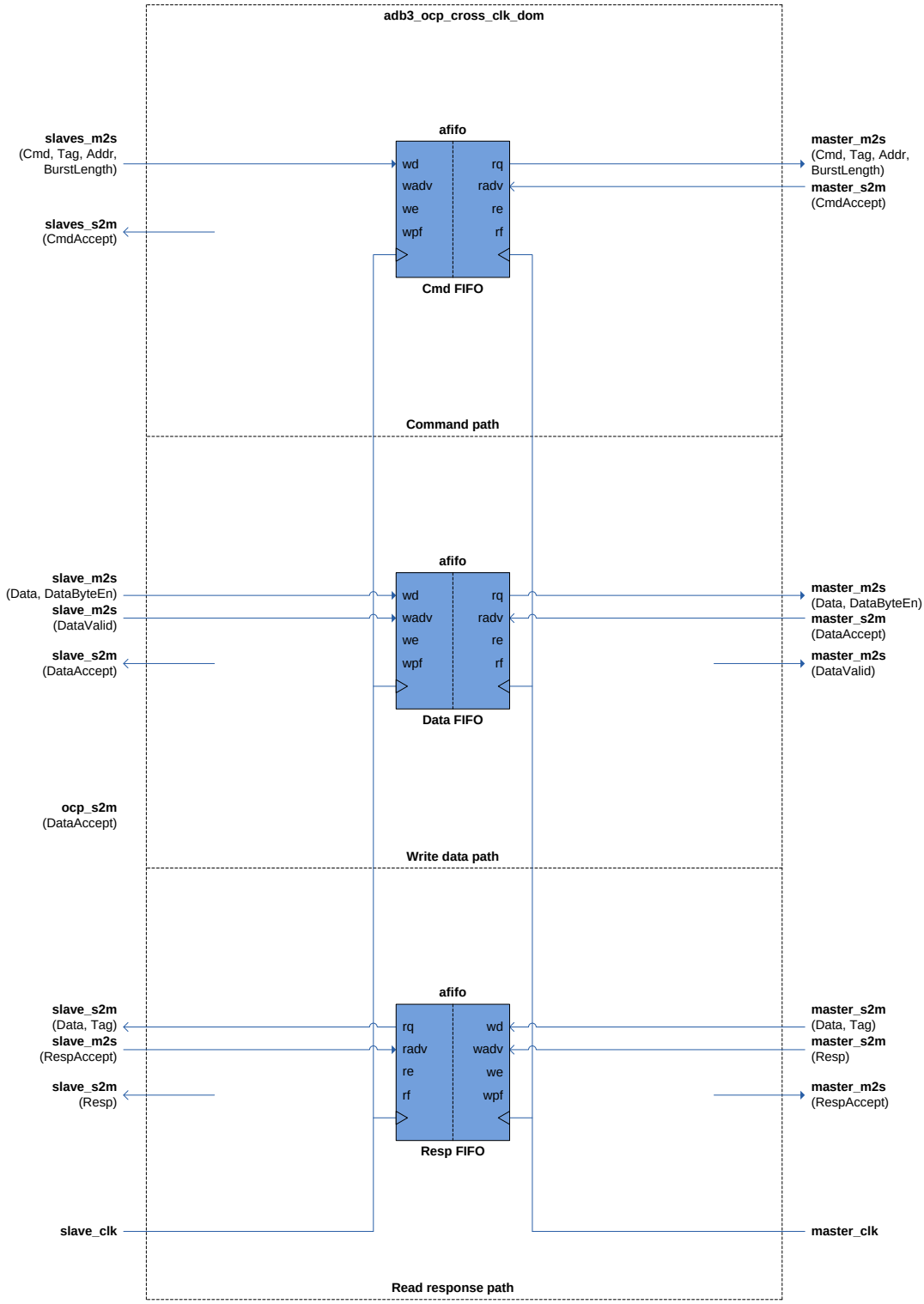

Figure 28: adb3_ocp_cross_clk_dom Block Diagram

The component consists of three instances of the Asychronous FIFO block **afifo**. One for command signals, one for data signals, and the third for response signals as follows:

6.1.3.1.3.1 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 empty flag.

6.1.3.1.3.2 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 empty flag.

6.1.3.1.3.3 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 empty flag.

6.1.3.2 adb3_ocp_mux_b

6.1.3.2.1 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.

6.1.3.2.2 Interface

The **adb3_ocp_mux_b** component interface is shown in **Figure 29** below and described in **Table 89**.

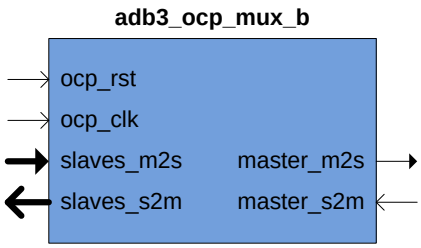

Figure 29: adb3_ocp_mux_b Component Interface

Signal	Type	Description
mux_inputs	Generic	Number of primary OCP channels to be multiplexed.
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 89: adb3_ocp_mux_b Component Interface

6.1.3.2.3 Description

TBD

6.1.3.3 adb3_ocp_mux_nb

6.1.3.3.1 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)**

6.1.3.3.2 Interface

The **adb3_ocp_mux_nb** component interface is shown in **Figure 30** below and described in **Table 90**.

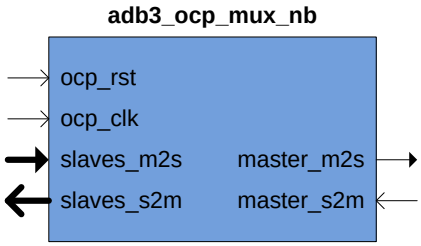

Figure 30: adb3_ocp_mux_nb Component Interface

Signal	Type	Description
mux_inputs	Generic	Number of primary OCP channels to be multiplexed.
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 90: adb3_ocp_mux_nb Component Interface

6.1.3.3.3 Description

The **adb3_ocp_mux_nb** component block diagram is shown in **Figure 31** below.

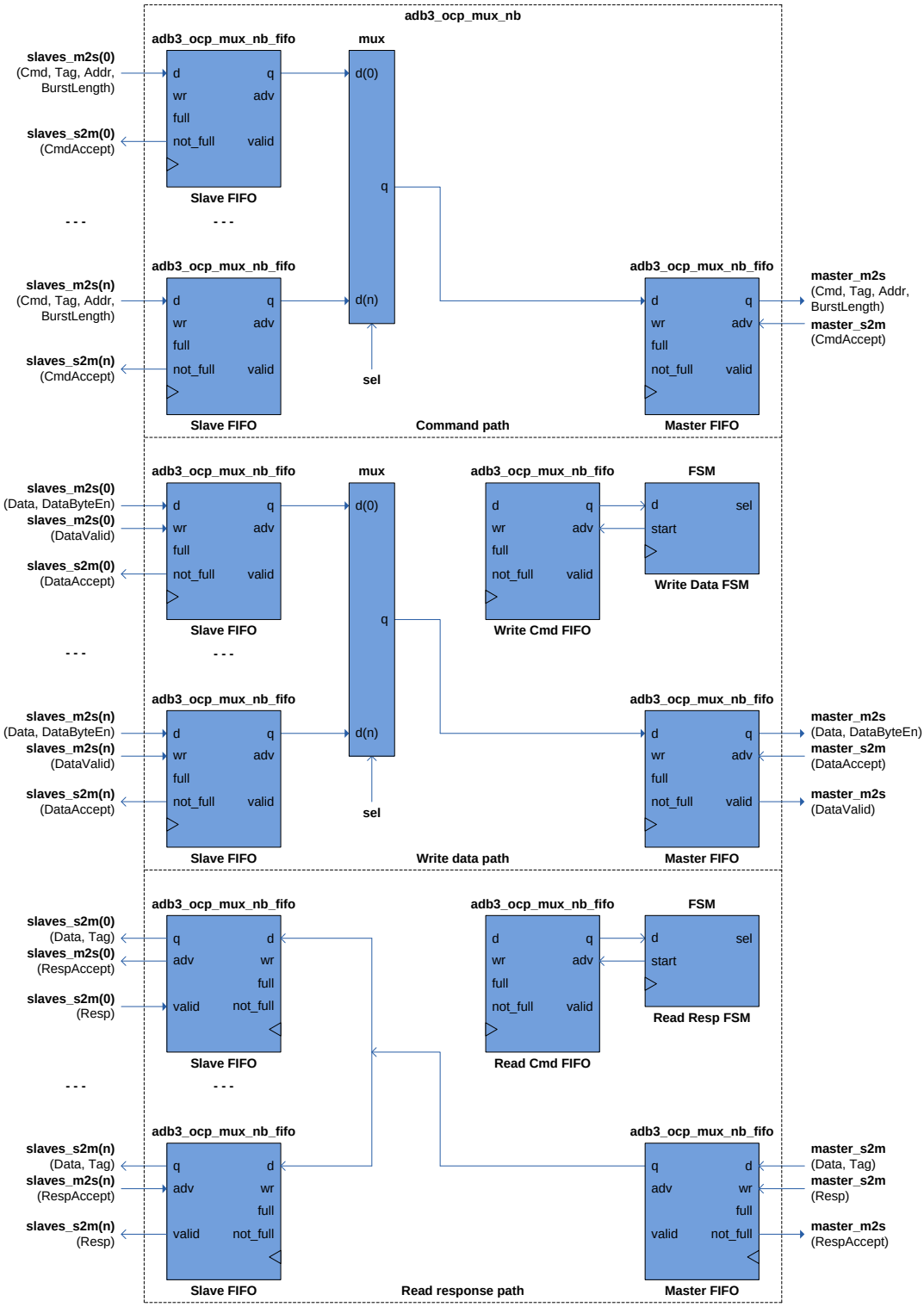

Figure 31: adb3_ocp_mux_nb Block Diagram

6.1.3.3.3.1 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 from the slave command select.
- 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.

6.1.3.3.3.2 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 from the write data select.
- 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.

6.1.3.3.3.3 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.

6.1.3.4 adb3_ocp_ocp2ddr3_nb

6.1.3.4.1 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)**

6.1.3.4.2 Interface

The **adb3_ocp_ocp2ddr3_nb** component interface is shown in **Figure 32** below and described in **Table 91**.

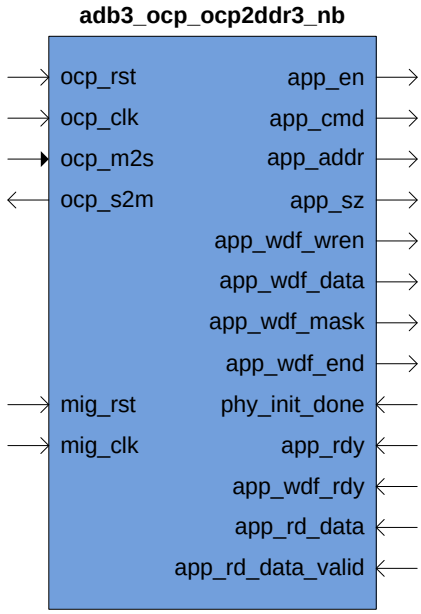

Figure 32: adb3_ocp_ocp2ddr3_nb Component Interface

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 (4-byte address).
		OCP Interface
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.

Table 91: adb3_ocp_ocp2ddr3_nb Component Interface (continued on next page)

Signal	Type	Description
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 (4-byte address).
app_sz	Output	User interface command on the fly BL8/BC4 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 91: adb3_ocp_ocp2ddr3_nb Component Interface

6.1.3.4.3 Description

The **adb3_ocp_ocp2ddr3_nb** component block diagram is shown in **Figure 33** below.

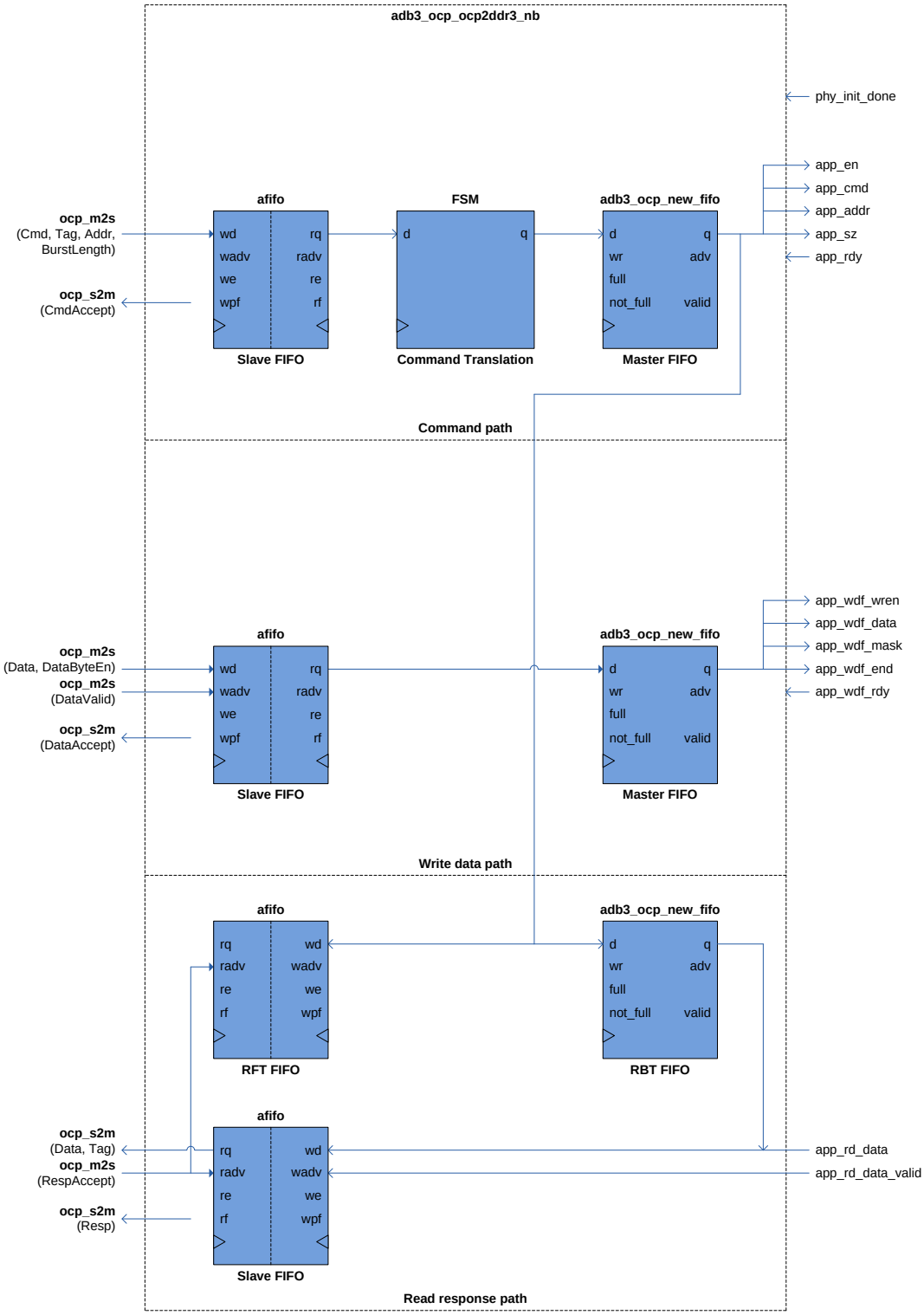

Figure 33: adb3_ocp_ocp2ddr3_nb Block Diagram

6.1.3.4.3.1 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 in the **ocp_clk** domain are interfaced to the command translation FSM in the **mig_clk** domain 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

- This block operates in the **mig_clk** domain.
- Slave command FIFO data is converted into MIG core user interface commands which are then written to the master command FIFO.
- The FSM output **n_mcx_bstart** is used to generate the slave command FIFO read advance.
- The FSM output **mcx_mst_wr** is used to generate the master command FIFO write advance.
- The FSM outputs **n_mcx_mst_wr**, and **n_mcx_cmd_wr** are used to generate the slave write data FIFO read advance and the master write data FIFO write advance.

Master Command FIFO

- The command translation FSM is interfaced to the MIG core user interface command signals via the master command FIFO.
- The master command FIFO write advance is generated from the FSM master write output **mcx_mst_wr**.
- The master command FIFO read advance is generated from the master command FIFO not empty, read burst tracker and read full tracker FIFOs not full, and the MIG core user interface ready signals.
- The MIG **app_addr** input is produced by reordering the address component of the master 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.

Read Burst Tracking FIFO

- The master command FIFO outputs **app_cmd_bl8** and **app_cmd_tag** are written to the read burst tracking FIFO on every MIG core user interface read command.
- The read burst tracking FIFO output **rbt_q_tag** is used as the tag value for OCP read response written into the slave read response FIFO.
- The read burst tracking FIFO output **rbt_q_bl8** is compared with the number of OCP response data words and this is used to generate **rbt_bl_comp** signal.
- The read burst tracking FIFO write advance is generated from the master command FIFO read advance and the master command FIFO output **app_cmd_rd**.
- The read burst tracking FIFO read advance is generated from the read burst tracking FIFO not empty, the **rbt_bl_comp** signal, and the slave read response FIFO write advance.

Read Full Tracking FIFO

- The master command FIFO output **app_cmd_bl8** is written to the read full tracking FIFO on every MIG core user interface read command.
- The read full tracking FIFO output **rft_q_bl8** is compared with the number of OCP response data words and this is used to generate **rft_bl_comp** signal.
- The read full tracking FIFO write advance is generated from the master command FIFO read advance and the master command FIFO output **app_cmd_rd**.
- The read full tracking FIFO read advance is generated from the read full tracking FIFO not empty, the **rft_bl_comp** signal, and the slave read response FIFO read advance.

6.1.3.4.3.2 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 in the **ocp_clk** domain are interfaced to the master write data FIFO in the **mig_clk** domain 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 FSM master write and write command outputs **n_mcx_mst_wr** and **n_mcx_cmd_wr**.

Master Write Data FIFO

- The slave write data FIFO is interfaced to the MIG core user interface write data signals via the master write data FIFO.
- The master write data FIFO write advance is generated from the FSM master write and write command outputs **n_mcx_mst_wr** and **n_mcx_cmd_wr**.
- The master write data FIFO read advance is generated from the master write data FIFO not empty, the **app_cmd_wr** signal, read burst tracker and read full tracker FIFOs not full, and the MIG core user interface ready signals.

6.1.3.4.3.3 Read Response 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 Response FIFO

- The MIG core user interface read data signals in the **mig_clk** domain are interfaced to the **ocp_s2m** port read response elements in the **ocp_clk** domain 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 MIG core user interface read data signal **app_rd_data_valid**. 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.

6.1.3.5 adb3_ocp_retime_nb

6.1.3.5.1 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.

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)**

6.1.3.5.2 Interface

The **adb3_ocp_retime_nb** component interface is shown in **Figure 34** below and described in **Table 92**.

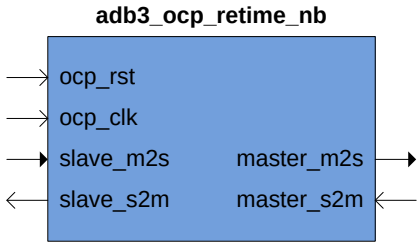

Figure 34: adb3_ocp_retime_nb Component Interface

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 92: adb3_ocp_retime_nb Component Interface

6.1.3.5.3 Description

The **adb3_ocp_retime_nb** component block diagram is shown in **Figure 35** below.

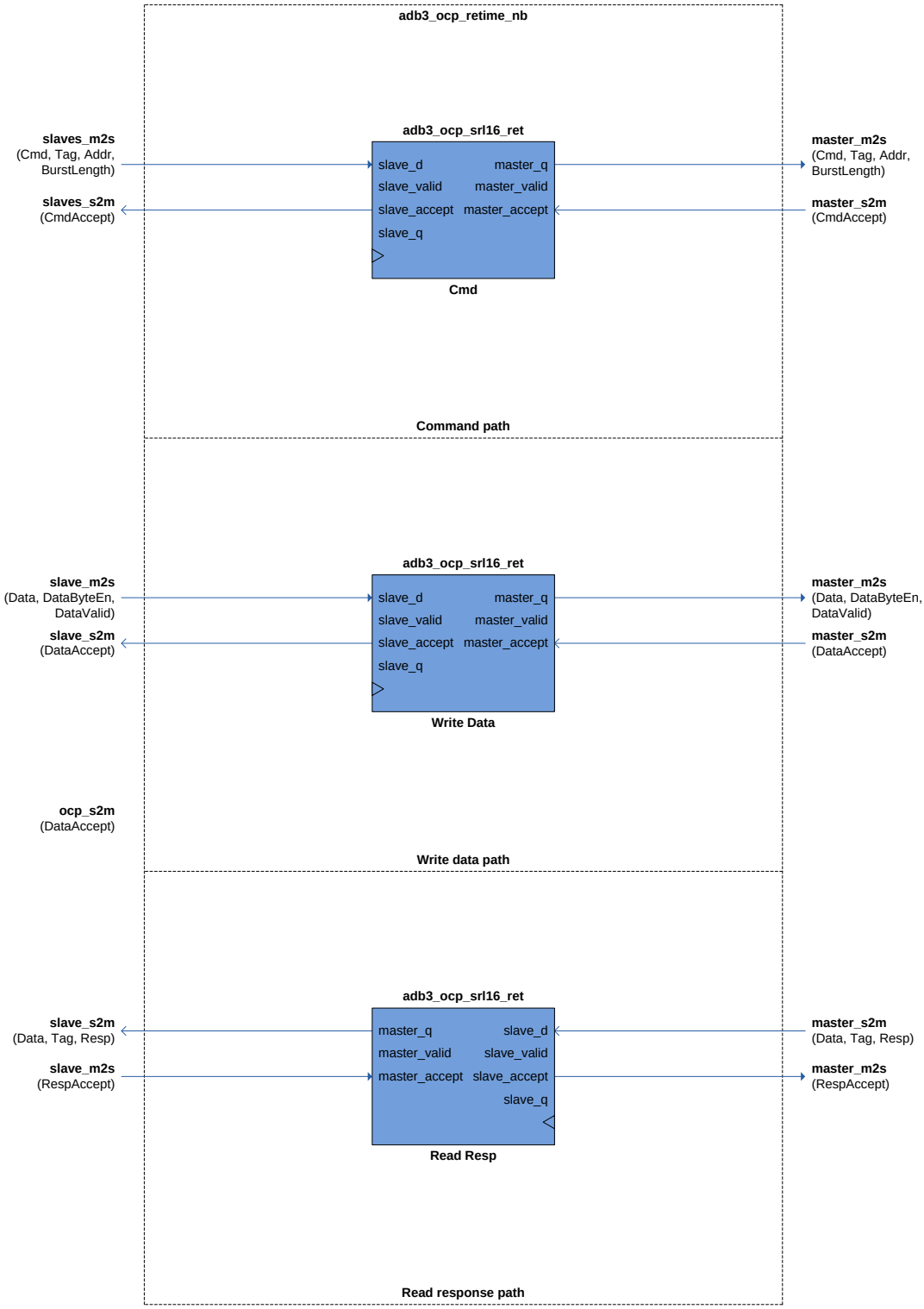

Figure 35: adb3_ocp_retime_nb Block Diagram

6.1.3.5.3.1 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.

6.1.3.5.3.2 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.

6.1.3.5.3.3 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.

6.1.3.5.3.4 SRL16E Retime Block (adb3_ocp_srl16_ret)

The **adb3_ocp_srl16_ret** component block diagram is shown in **Figure 36** below.

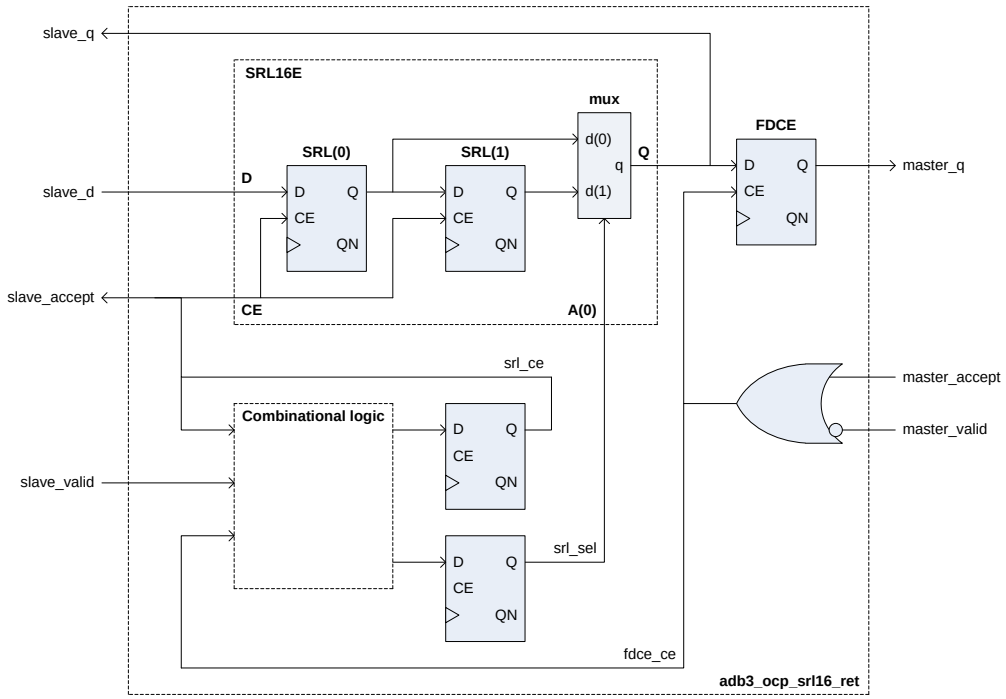

Figure 36: adb3_ocp_srl16_ret Block Diagram

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**.

6.1.3.6 adb3_ocp_simple_bus_if

6.1.3.6.1 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.

6.1.3.6.2 Interface

The **adb3_ocp_simple_bus_if** component interface is shown in **Figure 37** below and described in **Table 93**.

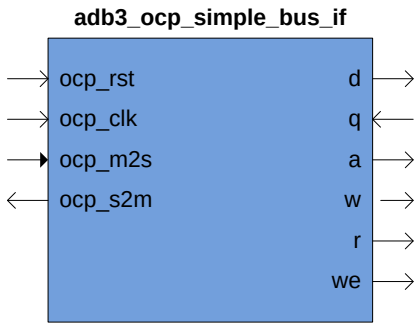

Figure 37: adb3_ocp_simple_bus_if Component Interface

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.
		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.
we	Output	Write data byte enable of width data_width/8.

Table 93: adb3_ocp_simple_bus_if Component Interface

6.1.3.6.3 Description

TBD

6.1.3.6.3.1 Example Waveforms

In the following example, we are performing 2 OCP writes with burst length of 1, to a 32-bit simple bus. 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 32-bit simple bus write address **a** ls nibble will therefore always sequence through values 0x0, 0x4, 0x8, 0xC.

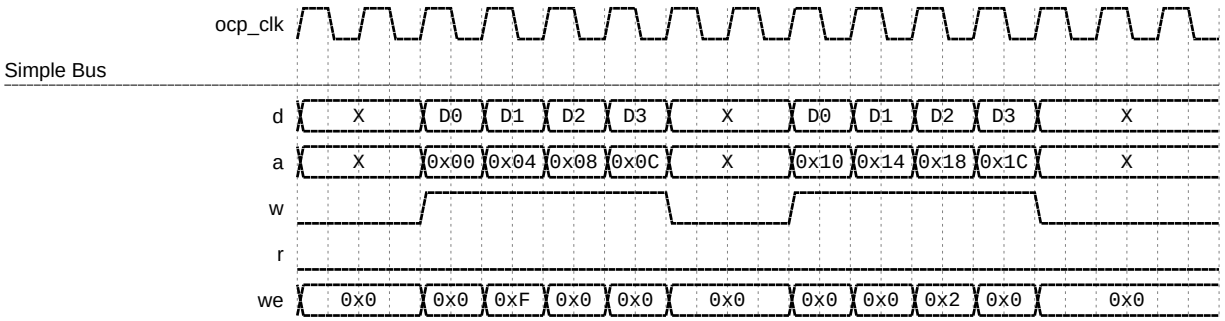

Figure 38: OCP Writes (Burst Length = 1) To 32-bit Simple Bus

The first OCP write is 32-bits to a byte address starting at 16-byte offset = 0x4.

The second OCP write is 8-bits to a byte address starting at 16-byte offset = 0x9.

In the following example, we are performing 1 OCP read with burst length of 1, from a 32-bit simple bus with **read_latency** of 1. 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 32-bit simple bus read address **a** ls nibble will therefore always sequence through values 0x0, 0x4, 0x8, 0xC.

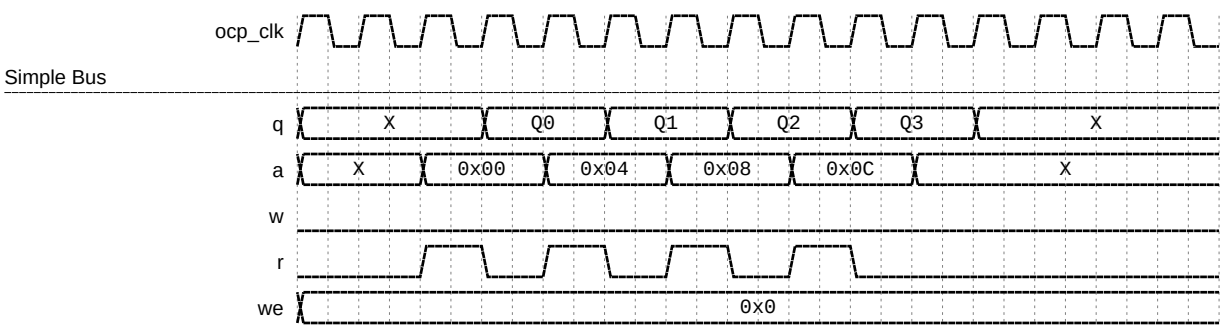

Figure 39: OCP Read From 32-bit Simple Bus (Read Latency = 1)

The OCP read is 128-bits from a byte address starting at 16-byte offset = 0x0.

In the following example, we are performing 2 OCP writes with burst length of 1, to a 128-bit simple bus. 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 128-bit simple bus write address **a** ls nibble will therefore always have value 0x0.

In the following example, we are performing 2 OCP reads with burst length of 1, from a 128-bit simple bus with **read_latency** of 1. 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 128-bit simple bus read address **a** ls nibble will therefore always have value 0x0.

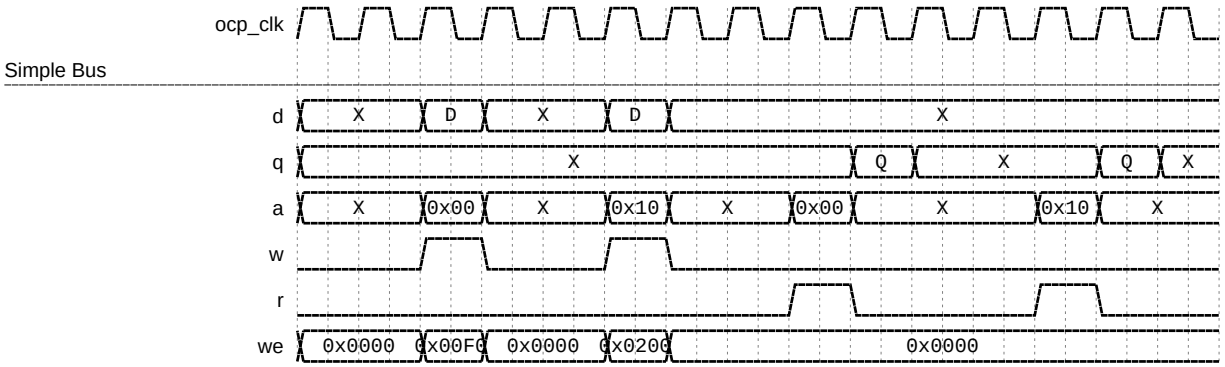

Figure 40: OCP Writes/Reads (Burst Length = 1) To/From 128-bit Simple Bus

The first OCP write is 32-bits to a byte address starting at 16-byte offset = 0x4.

The second OCP write is 8-bits to a byte address starting at 16-byte offset = 0x9.

The first OCP read is 128-bits from a byte address starting at 16-byte offset = 0x0.

The second OCP read is 128-bits from a byte address starting at 16-byte offset = 0x0.

6.1.3.7 adb3_ocp_simple_bus_if_nb

6.1.3.7.1 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)**

6.1.3.7.2 Interface

The **adb3_ocp_simple_bus_if_nb** component interface is shown in **Figure 41** below and described in **Table 94**.

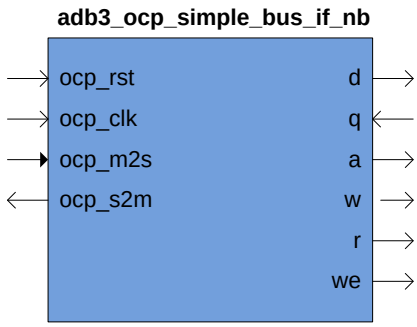

Figure 41: adb3_ocp_simple_bus_if_nb Component Interface

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.
		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.
we	Output	Write data byte enable of width data_width/8.

Table 94: adb3_ocp_simple_bus_if_nb Component Interface

6.1.3.7.3 Description

The **adb3_ocp_simple_bus_if_nb** component block diagram is shown in **Figure 42** below.

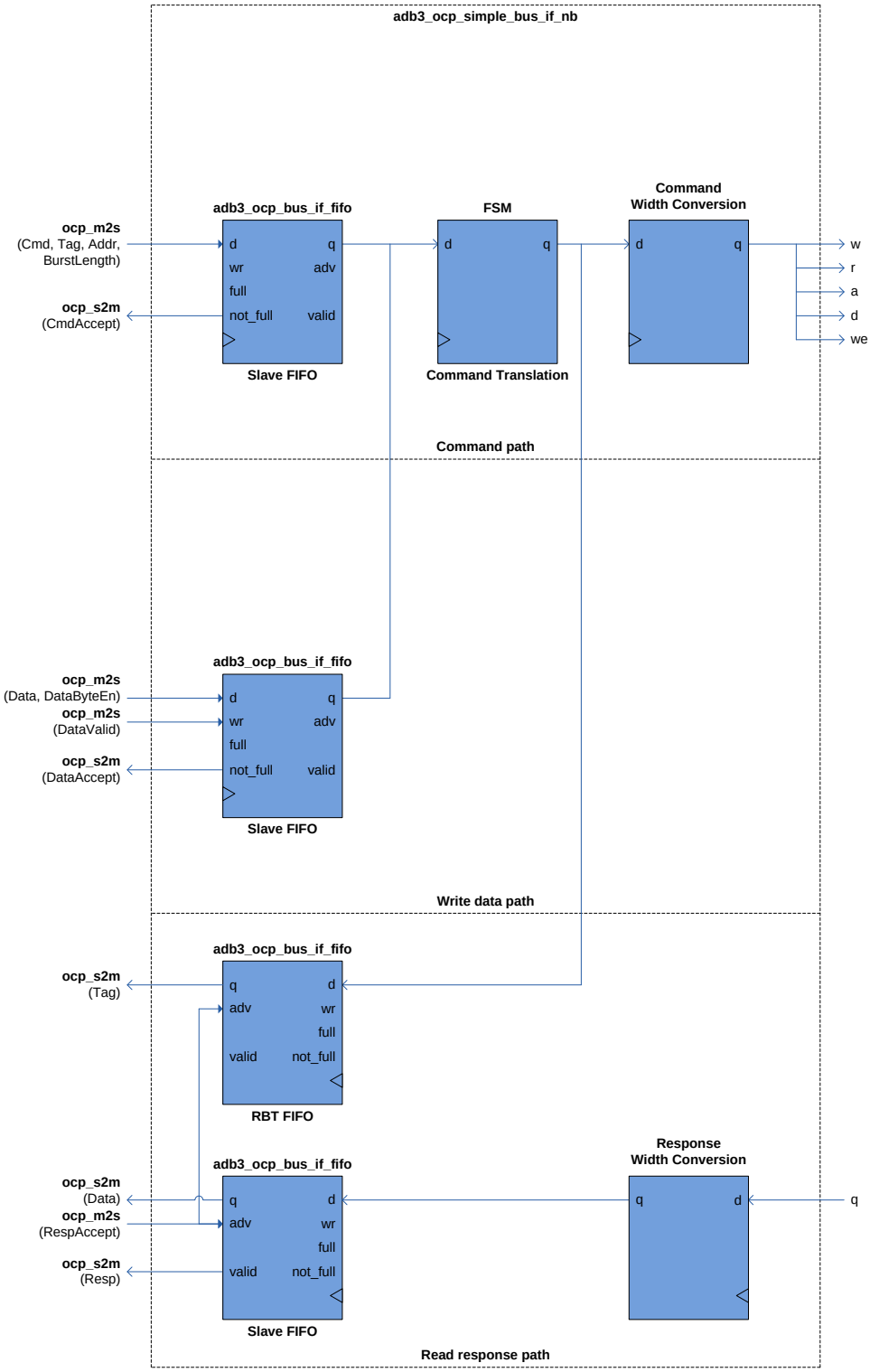

Figure 42: adb3_ocp_simple_bus_if_nb Block Diagram

6.1.3.7.3.1 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.

6.1.3.7.3.2 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**.

6.1.3.7.3.3 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.

6.1.3.7.3.4 Example Waveforms

In the following example, we are performing 2 OCP writes with burst length of 1, to a 32-bit simple bus. 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 32-bit simple bus write address **a** ls nibble will therefore always sequence through values 0x0, 0x4, 0x8, 0xC.

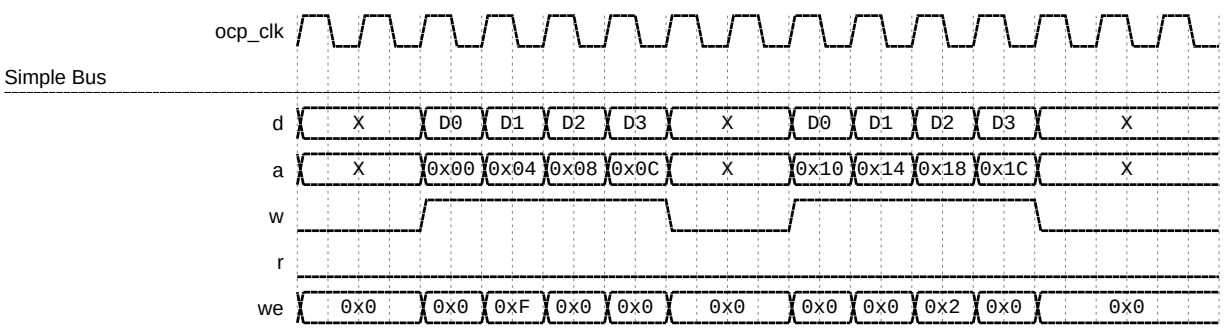

Figure 43: OCP Writes (Burst Length = 1) To 32-bit Simple Bus

The first OCP write is 32-bits to a byte address starting at 16-byte offset = 0x4.

The second OCP write is 8-bits to a byte address starting at 16-byte offset = 0x9.

In the following example, we are performing 2 OCP reads with burst length of 1, from a 32-bit simple bus with **read_latency** of 1. 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 32-bit simple bus read address **a** ls nibble will therefore always sequence through values 0x0, 0x4, 0x8, 0xC.

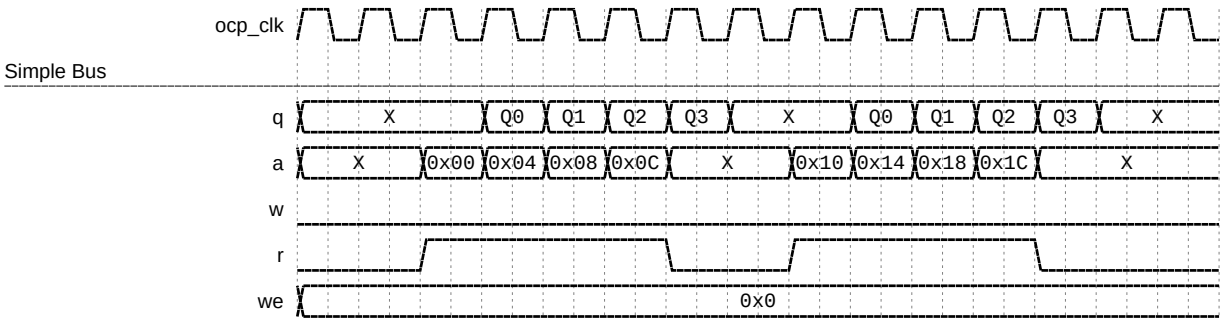

Figure 44: OCP Read From 32-bit Simple Bus (Read Latency = 1)

The OCP read is 128-bits from a byte address starting at 16-byte offset = 0x0.

In the following example, we are performing 2 OCP writes with burst length of 1, to a 128-bit simple bus. 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 128-bit simple bus write address **a** ls nibble will therefore always have value 0x0.

In the following example, we are performing 2 OCP reads with burst length of 1, from a 128-bit simple bus with **read_latency** of 1. 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 128-bit simple bus read address **a** ls nibble will therefore always have value 0x0.

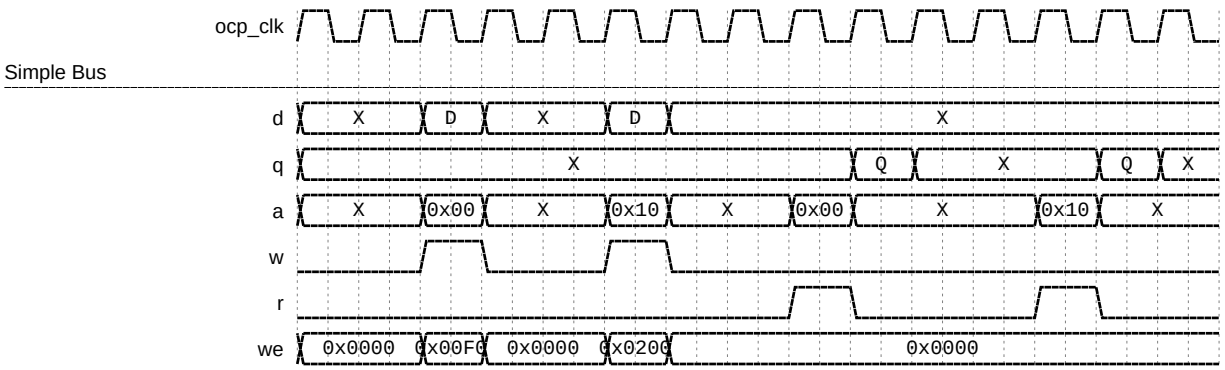

Figure 45: OCP Writes/Reads (Burst Length = 1) To/From 128-bit Simple Bus

The first OCP write is 32-bits to a byte address starting at 16-byte offset = 0x4.

The second OCP write is 8-bits to a byte address starting at 16-byte offset = 0x9.

The first OCP read is 128-bits from a byte address starting at 16-byte offset = 0x0.

The second OCP read is 128-bits from a byte address starting at 16-byte offset = 0x0.

6.1.3.8 adb3_ocp_split_b

6.1.3.8.1 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. The de-multiplex is controlled by the primary channel command address.

6.1.3.8.2 Interface

The **adb3_ocp_split_b** component interface is shown in **Figure 46** below and described in **Table 95**.

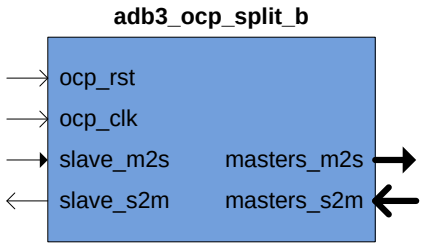

Figure 46: adb3_ocp_split_b Component Interface

Signal	Type	Description
addr_range_table	Generic	Table defining the address ranges to be used to control the split operation.
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 95: adb3_ocp_split_b Component Interface

6.1.3.8.3 Description

TBD

6.1.3.9 adb3_ocp_split_nb

6.1.3.9.1 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. The de-multiplex 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)**

6.1.3.9.2 Interface

The **adb3_ocp_split_nb** component interface is shown in **Figure 47** below and described in **Table 96**.

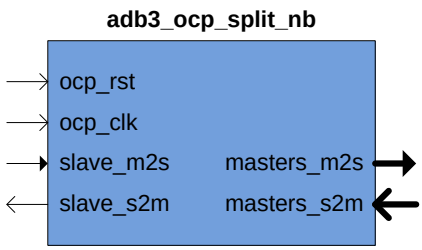

Figure 47: adb3_ocp_split_nb Component Interface

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.
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 96: adb3_ocp_split_nb Component Interface

6.1.3.9.3 Description

The **adb3_ocp_split_nb** component block diagram is shown in **Figure 48** below.

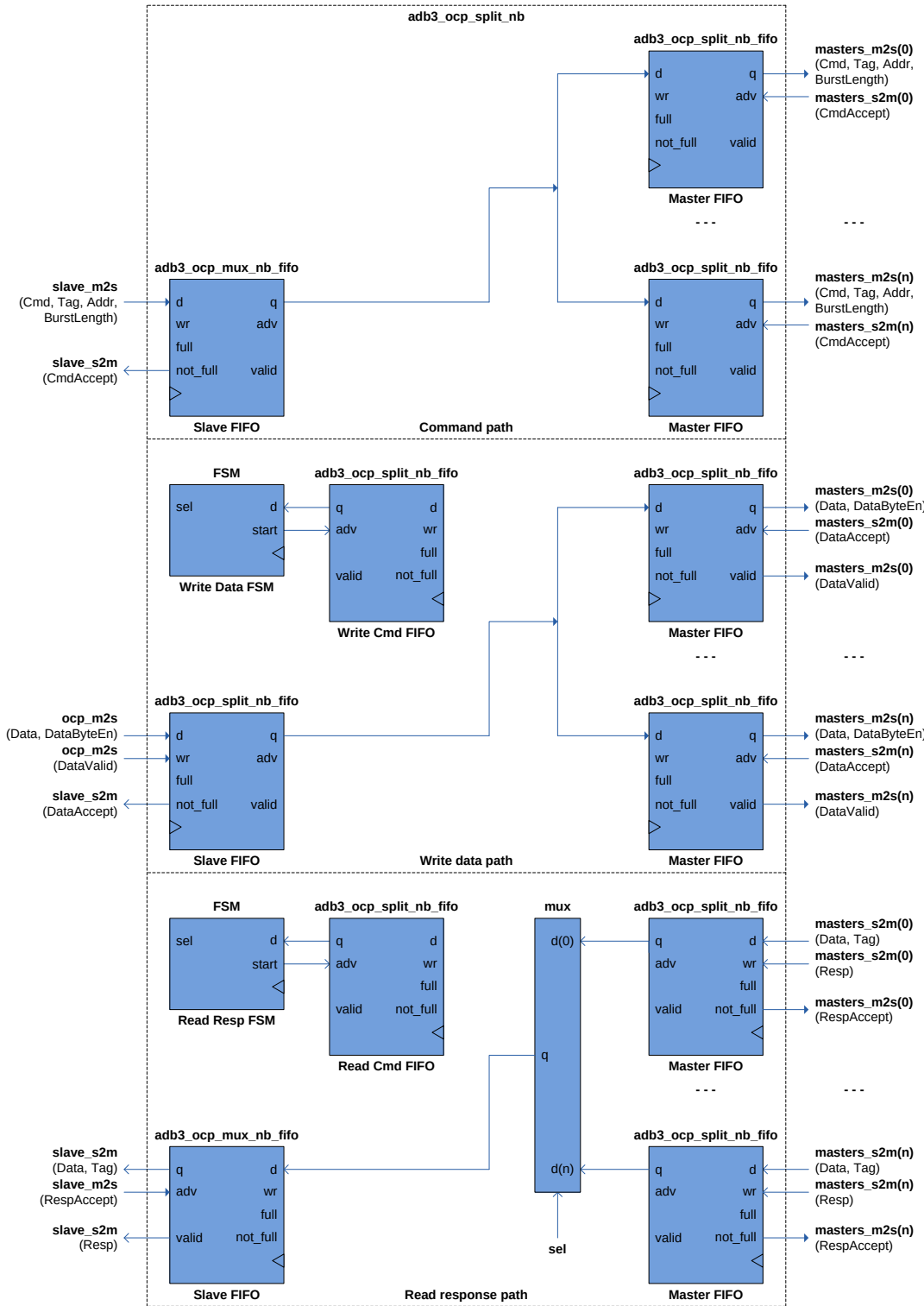

Figure 48: adb3_ocp_split_nb Block Diagram

6.1.3.9.3.1 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 Selector

- The slave command select is generated by comparison of the slave command FIFO **Addr** element with the address ranges in the **addr_range_table** generic.

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.

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.

6.1.3.9.3.2 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.

6.1.3.9.3.3 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.

6.1.4 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_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 Types Definition Package (adb3_target_types_pkg)**
- **ADB3 Target Include Package (adb3_target_inc_pkg)**
- **ADB3 Target Package (adb3_target_pkg)**
- **ADB3 Target Components**
- **ADB3 Target Testbench Include Package (adb3_target_tb_inc_pkg)**
- **ADB3 Target Testbench Package (adb3_target_tb_pkg)**
- **ADB3 Target Testbench Components**

6.2.1 ADB3 Target Types Definition Package (adb3_target_types_pkg)

The **adb3_target_types_pkg** package defines types and constants which are used by the ADB3 target include packages.

Type Definitions

- **target_use_t**. An enumerated type containing an element for each end use supported by the SDK. Valid end uses are currently: **SIM_OCP** for OCP-only simulation; **SIM_MPTL** for Full MPTL simulation; and **SYN_NGC** for synthesis.

Maximum Value Constant Definitions

- **MAX_DS_CHANNELS**. Direct slave OCP channels.
- **MAX_DMA_CHANNELS**. DMA OCP channels.
- **MAX_DM_CHANNELS**. Direct master OCP channels.
- **MAX_MEM_BANKS**. On-board memory interfaces.
- **MAX_MPTL_SER_WIDTH**. Width of MPTL serial data interfaces.
- **MAX_PCIE_SER_WIDTH**. Width of PCIe serial data interfaces.
- **MAX_GPIO_FR_WIDTH**. Width of the XMC front GPIO interface.
- **MAX_GPIO_SE_WIDTH**. Width of single ended GPIO interfaces.
- **MAX_GPIO_DE_WIDTH**. Width of double ended GPIO interfaces.

6.2.2 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 board selected. This includes whether synthesis or simulation is being performed. 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.

The **adb3_target_inc_pkg** package exists in several variants, one for each supported combination of board and usage. **Table 97** lists the available variants:

Model	TARGET_USE	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	SIM_MPTL	admxrc6tl/adb3_target_inc_pkg_sim_mptl_6tl.vhd
	SIM_OCP	admxrc6tl/adb3_target_inc_pkg_sim_ocp_6tl.vhd
	SYN_NGC	admxrc6tl/adb3_target_inc_pkg_syn_ngc_6tl.vhd
ADM-XRC-6T1	SIM_MPTL	admxrc6t1/adb3_target_inc_pkg_sim_mptl_6t1.vhd
	SIM_OCP	admxrc6t1/adb3_target_inc_pkg_sim_ocp_6t1.vhd
	SYN_NGC	admxrc6t1/adb3_target_inc_pkg_syn_ngc_6t1.vhd
ADM-XRC-6TGE	SIM_MPTL	admxrc6tge/adb3_target_inc_pkg_sim_mptl_6tge.vhd
	SIM_OCP	admxrc6tge/adb3_target_inc_pkg_sim_ocp_6tge.vhd
	SYN_NGC	admxrc6tge/adb3_target_inc_pkg_syn_ngc_6tge.vhd
ADM-XRC-6TADV8	SIM_OCP	admxrc6tadv8/adb3_target_inc_pkg_sim_ocp_6tadv8_pcie.vhd
	SYN_NGC	admxrc6tadv8/adb3_target_inc_pkg_syn_ngc_6tadv8_pcie.vhd

Table 97: Available Variants of the adb3_target_inc_pkg Package

The following definitions are available in this package:

General Definitions

- **TARGET_USE**. Defines the end use according to the **target_use_t** enumerated type; for example, **SIM_OCP** for OCP-only simulation.
- **std_logic_dbl_t**. Type defining a general-purpose differential std_logic signal.

Clock Definitions

- **REF_CLK_FREQ_HZ**. The frequency in Hz of the reference clock input used by the target FPGA design on this board.
- **clks_in_t**. Record defining target FPGA clock inputs on this board.
- **MGT_CLKS_VALID** Vector defining target FPGA MGT clock inputs which require to be buffered. Clock order is MGT118 Clk(1:0), ..., MGT110 Clk(1:0).
- **clks_mgt_in_t**. Record defining target FPGA MGT clock inputs on this board.
- **clks_out_t**. Record defining target FPGA clock outputs on this board.

GPIO Definitions

- **XRM_GPIO_WIDTH**. Indicates width of XRM GPIO interface on this board.
- **PN4_GPIO_WIDTH**. Indicates width of Pn4 GPIO interface on this board.
- **PN6_GPIO_WIDTH**. Indicates width of Pn6 GPIO interface on this board.
- **xrm_gpio_t**. Record defining target FPGA XRM GPIO interface on this board.

- **pn4_gpio_t**. Record defining target FPGA Pn4 GPIO interface on this board.
- **pn6_gpio_t**. Record defining target FPGA Pn6 GPIO interface on this board.
- **gpio_inout_t**. Record defining target FPGA GPIO interface on this board.

On-Board Memory Interface Definitions

- **DDR3_BANKS**. Indicates the number of target FPGA DDR3 SDRAM bank interfaces on this board.
- **DDR3_BANK_ROW_WIDTH***. Indicates the maximum width of the target FPGA DDR3 SDRAM row address interface on this board.
- **DDR3_BANK_DATA_WIDTH**. Indicates the width of the target FPGA DDR3 SDRAM data interface on this board
- **DDR3_BYTE_ADDR_WIDTH**. Indicates the maximum width of the target FPGA DDR3 SDRAM byte address interface on this board.
- **ddr3_addr_out_t**. Record defining target FPGA DDR3 SDRAM bank address interface on this board.
- **ddr3_ctrl_out_t**. Record defining target FPGA DDR3 SDRAM bank control interface on this board.
- **ddr3_data_inout_t**. Record defining target FPGA DDR3 SDRAM bank data interface on this board.
- **ddr3_clk_out_t**. Record defining target FPGA DDR3 SDRAM bank clock interface on this board.
- **ddr3_addr_out_array_t**. Array defining target FPGA DDR3 SDRAM address interface on this board.
- **ddr3_ctrl_out_array_t**. Array defining target FPGA DDR3 SDRAM control interface on this board.
- **ddr3_data_inout_array_t**. Array defining target FPGA DDR3 SDRAM data interface on this board.
- **ddr3_clk_out_array_t**. Array defining target FPGA DDR3 SDRAM clock interface on this board.
- **MEM_BANKS**. Indicates the number of target FPGA on-board memory interfaces on this board.
- **DDR3_BANK0**. Indicates the bank number of the first target FPGA DDR3 SDRAM bank interface on this board.
- **mem_byte_addr_width_array_t**. Array defining target FPGA on-board memory bank address widths on this board.
- **MEM_BYTE_ADDR_WIDTH_ARRAY**. Indicates the address width of each bank of on-board memory on this board.
- **mem_addr_out_t**. Record defining target FPGA address interface on this board.
- **mem_ctrl_out_t**. Record defining target FPGA control interface on this board.
- **mem_data_inout_t**. Record defining target FPGA data interface on this board.
- **mem_clk_out_t**. Record defining target FPGA clock interface on this board.

* 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 **DDR3_1G_PART**, **DDR3_2G_PART**, or **DDR3_4G_PART** 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 Gib constants are appropriate for the size of the memory parts on the Alpha Data board in use.

OCP Interface Definitions

- **DS_CHANNELS**. Indicates the number of direct slave OCP channels on this board.
- **DMA_CHANNELS**. Indicates the number of dma OCP channels on this board.
- **DM_CHANNELS**. Indicates the number of direct master OCP channels on this board.

- **DS_ADDR_WIDTH**. Indicates the address space size for a direct slave OCP channel on this board.
- **DMA_ADDR_WIDTH**. Indicates the address space size for a dma OCP channel on this board.
- **DM_ADDR_WIDTH**. Indicates the address space size for a direct master OCP channel on this board.

MPTL Interface Definitions

- **MPTL_SER_WIDTH**. Indicates the width of the MPTL serial data interface that exists on this board.
- **mptl_t2b_t**. Type defining the MPTL interface signals between the target and bridge FPGAs. Definition depends on board and end use.
- **mptl_b2t_t**. Type defining the MPTL interface signals between the bridge and target FPGAs. Definition depends on board and end use.
- **mptl_sb_b2t_t**. Type defining the MPTL sideband interface signals from the bridge to the target. Definition depends on board and end use.
- **mptl_sb_t2b_t**. Type defining the MPTL sideband interface signals from the target to the bridge. Definition depends on board and end use.

PCIe Interface Definitions

- **PCIE_SER_WIDTH**. Indicates the width of the PCIe serial data interface that exists on this board.
- **pcie_t2h_t**. Type defining the PCIe interface signals between the target FPGA and host. Definition depends on board and end use.
- **pcie_h2t_t**. Type defining the PCIe interface signals between the host and target FPGA. Definition depends on board and end use.

Custom Interface Definitions

- **adb3_model_in_t**. Type defining the adb3 model interface input signals. Definition depends on board and end use.
- **adb3_model_out_t**. Type defining the adb3 model interface output signals. Definition depends on board and end use.
- **adb3_model_inout_t**. Type defining the adb3 model interface bi-directional signals. Definition depends on board and end use.
- **custom_in_t**. Type defining the custom interface input signals. Definition depends on board and end use.
- **custom_out_t**. Type defining the custom interface output signals. Definition depends on board and end use.
- **custom_inout_t**. Type defining the custom interface bi-directional signals. Definition depends on board and end use.

6.2.3 ADB3 Target Package (adb3_target_pkg)

The package **adb3_target_pkg** defines functions and components which relate to target example FPGAs.

The **adb3_target_pkg** package exists in two variants, one for MPTL interface IP, the other for PCIe interface IP. **Table 98** lists the available variants:

Interface	Filename relative to hdl/vhdl/common/adb3_target/
MPTL	adb3_target_pkg.vhd
PCIe	adb3_target_pkg_pcie.vhd

Table 98: Available Variants of the adb3_target_pkg Package

Function Definitions (MPTL)

- **make_defined**. Return the input vector with any bits that are not '0' or '1' set to '0'.
- **make_defined_s2m**. Return record of type **adb3_ocp_s2mT** with **Data** and **Tag** values modified using make_defined function.

Component Definitions (MPTL)

- **mptl_if_target_wrap**

Component Definitions (PCIe)

- **pcie_if_target_wrap**

6.2.4 ADB3 Target Components

6.2.4.1 Target MPTL Interface Wrapper (mptl_if_target_wrap)

6.2.4.1.1 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 Types Definition Package (adb3_target_types_pkg)**
- **ADB3 Target Include Package (adb3_target_inc_pkg)**
- **ADB3 Target Package (adb3_target_pkg)**

6.2.4.1.2 Interface

The **mptl_if_target_wrap** component interface is shown in **Figure 49** below and described in **Table 99**.

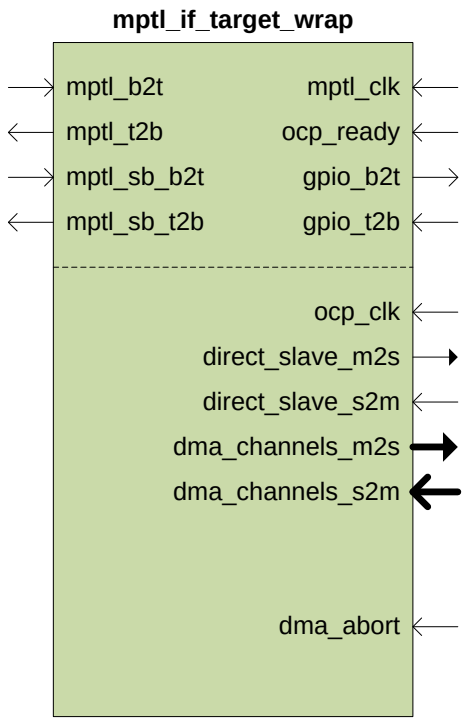

Figure 49: mptl_if_target_wrap Component Interface

Signal	Type	Description
		OCP Interface
ocp_clk	Input	OCP clock (from target FPGA).
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).
		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_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 99: mptl_if_target_wrap Component Interface

6.2.4.1.3 Description

The type of Target MPTL interface that is instantiated depends upon which variant of the **adb3_target_inc_pkg** is in use, through the **TARGET_USE** constant.

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 interface blocks. 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.

6.2.4.1.3.1 OCP-Only Simulation

During OCP-only simulation (selected by **TARGET_USE** = SIM_OCP), a simulation only version of the **mptl_if_target_wrap** component is instantiated. **Table 100** 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

Table 100: 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 to the bridge clock via the **mptl_t2b.target_ocp_clk** signal.
- The **ocp_clk** input drives the **mptl_t2b.target_ocp_clk** signal.

Initialisation

- At power-up, an online delay counter produces the **mptl_sb_t2b.mptl_target_gtp_online_l** output using the **mptl_sb_b2t.mptl_bridge_gtp_online_l** input.
- The **mptl_sb_t2b.mptl_target_configured_l** output is generated using the OCP channels ready **ocp_ready** 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 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 board, 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 boards, the **dma_abort** input from the target FPGA drives the DMA abort request output **mptl_t2b.dma_abort**.

6.2.4.1.3.2 Full MPTL Simulation and Synthesis

During full MPTL simulation (selected by **TARGET_USE** = SIM_MPTL) and synthesis (selected by **TARGET_USE** = SYN_NGC), the **mptl_if_target_wrap** component is instantiated. **Table 101** 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

Table 101: Available Variants of mptl_if_target_wrap Component

6.2.4.1.3.2.1 Full MPTL simulation

During full MPTL simulation, the **mptl_if_target_wrap** component instantiates the MPTL interface HDL netlist appropriate to the board in use. **Table 102** 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

Table 102: Available Variants of Target MPTL Interface Netlist

The **mptl_if_target_wrap** component direct slave OCP channel input (**direct_slave_s2m**) and DMA OCP channels inputs (**dma_channels_s2m**) are processed by the **make_defined_s2m** function to ensure that they only contain '0' or '1' data. Other data values may cause the simulation of the MPTL interface to fail.

The remainder of the **mptl_if_target_wrap** component signals are connected to their equivalents on the MPTL interface HDL netlist.

6.2.4.1.3.2.2 Synthesis

During synthesis, the **mptl_if_target_wrap** component instantiates the MPTL interface core (**.ngc**) appropriate to the board in use. **Table 103** 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

Table 103: Available Variants of MPTL Interface Core

The remainder of the **mptl_if_target_wrap** component signals are connected to their equivalents on the MPTL interface core.

6.2.4.2 Target PCIe Interface Wrapper (pcie_if_target_wrap)

6.2.4.2.1 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)**

6.2.4.2.2 Interface

The **pcie_if_target_wrap** component interface is shown in **Figure 50** below and described in **Table 104**.

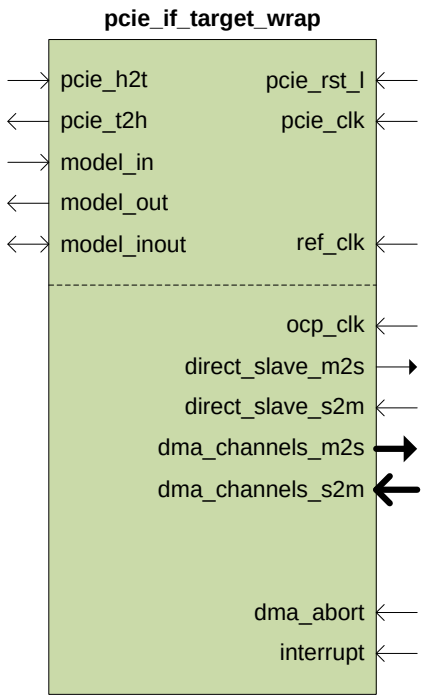

Figure 50: pcie_if_target_wrap Component Interface

Signal	Type	Description
		OCP Interface
ocp_clk	Input	OCP clock (from target FPGA).
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).
interrupt	Input	Interrupt request (from target).

Table 104: pcie_if_target_wrap Component Interface (continued on next page)

Signal	Type	Description
		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 104: pcie_if_target_wrap Component Interface

6.2.4.2.3 Description

The type of Target PCIe interface that is instantiated depends upon which variant of the **adb3_target_inc_pkg** is in use, through the **TARGET_USE** constant.

The PCIe interface signals **pcie_t2h** and **pcie_h2t** connect the host and target PCIe interface. 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 interface. 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 board specific interfaces on boards without a bridge FPGA.

6.2.4.2.3.1 OCP-Only Simulation

During OCP-only simulation (selected by **TARGET_USE** = SIM_OCP), a simulation only version of the **pcie_if_target_wrap** component is instantiated. **Table 105** lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TADV8	admxrc6tadv8/pcie_target/pcie_if_target_wrap_sim_6tadv8.vhd

Table 105: Available Variants of Simulation Only Version of pcie_if_target_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** 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**.

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**.

6.2.4.2.3.2 Synthesis

During synthesis (selected by **TARGET_USE** = SYN_NGC), the **pcie_if_target_wrap** component is instantiated. **Table 106** lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TADV8	admxrc6tadv8/pcie_target/pcie_if_target_wrap_6tadv8.vhd

Table 106: 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 board in use. **Table 107** lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TADV8	admxrc6tadv8/pcie_target/admxrc6tadv8_pcie_x4.ngc

Table 107: Available Variants of PCIe Interface Core

The remainder of the **pcie_if_target_wrap** component signals are connected to their equivalents on the PCIe interface core.

6.2.5 ADB3 Target Testbench Include Package (adb3_target_tb_inc_pkg)

The **adb3_target_tb_inc_pkg** package defines constants and types which characterise the board selected. The package exists in several variants, one for each supported board. **Table 108** 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-6TADV8	admxrc6tadv8/adb3_target_tb_inc_pkg_6tadv8.vhd

Table 108: Available Variants of the adb3_target_tb_inc_pkg Package

The following definitions are available in this package:

General Definitions

- **BOARD_TYPE**. Defines a string containing the board name.
- **clks_out_exp_t**. Record defining expected frequencies of target FPGA clock outputs on this board.
- **DDR3_1G_PART**. Defines DDR3 SDRAM 1Gib part if relevant on this board.
- **DDR3_2G_PART**. Defines DDR3 SDRAM 2Gib part if relevant on this board.
- **DDR3_4G_PART**. Defines DDR3 SDRAM 4Gib part if relevant on this board.

Build Options

- **OPTION_M**. Defines DDR3 SDRAM part option if relevant on this board.
- **OPTION_P**. Defines Pn4 connector fit option if relevant on this board.
- **OPTION_E**. Defines ethernet link fit option if relevant on this board.
- **OPTION_G**. Defines Pn6 clock option if relevant on this board.

Clock Generation

- Defines board clock period/frequency constants for this board. There are used by the **test_board_clks** block to generate board clocks.

On-Board Memory

- **BOARD_DDR3_PART**. Part number of DDR3 SDRAM component selected for simulation (**OPTION_M**).
- **BOARD_DDR3_BANK_ROW_WIDTH**. Row address width of DDR3 SDRAM component selected for simulation.
- **BOARD_DDR3_BANK_COL_WIDTH**. Column address width of DDR3 SDRAM component selected for simulation.
- **BOARD_DDR3_BANK_BNK_WIDTH**. Bank address width of DDR3 SDRAM component selected for simulation.
- **BOARD_DDR3_BYTE_ADDR_WIDTH**. Byte address width of DDR3 SDRAM component selected for simulation.
- **BOARD_MEM_BYTE_ADDR_WIDTH_ARRAY**. Byte address width of banks of on-board memory present on this board.

6.2.6 ADB3 Target Testbench Package (adb3_target_tb_pkg)

The package **adb3_target_tb_pkg** defines components which relate to target example FPGA testbenches.

The **adb3_target_tb_pkg** package exists in two variants, one for MPTL interface IP, the other for PCIe interface IP. **Table 109** lists the available variants:

Interface	Filename relative to hdl/vhdl/common/adb3_target/
MPTL	adb3_target_tb_pkg.vhd
PCIe	adb3_target_tb_pkg_pcie.vhd

Table 109: Available Variants of the adb3_target_tb_pkg Package

Component Definitions (MPTL)

- **mptl_if_bridge_wrap**
- **test_board_clks**

Component Definitions (PCIe)

- **pcie_if_host_wrap**
- **test_board_clks**

6.2.7 ADB3 Target Testbench Components

6.2.7.1 Bridge MPTL Interface Wrapper (mptl_if_bridge_wrap)

6.2.7.1.1 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 Types Definition Package (adb3_target_types_pkg)**
- **ADB3 Target Include Package (adb3_target_inc_pkg)**
- **ADB3 Target Testbench Package (adb3_target_tb_pkg)**

6.2.7.1.2 Interface

The **mptl_if_bridge_wrap** component interface is shown in **Figure 51** below and described in **Table 110**.

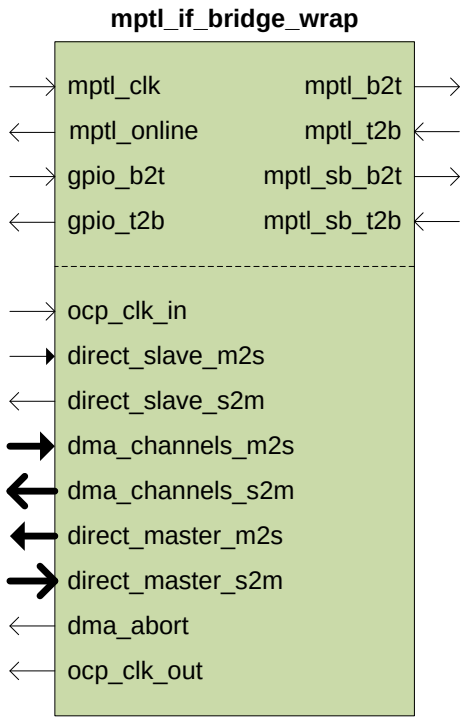

Figure 51: mptl_if_bridge_wrap Component Interface

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).
direct_masters_m2s	Output	Direct mater 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 110: mptl_if_bridge_wrap Component Interface

6.2.7.1.3 Description

The type of Bridge MPTL interface that is instantiated depends upon which variant of the **adb3_target_inc_pkg** is in use, through the **TARGET_USE** constant.

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interface blocks. 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 interface blocks. During full MPTL simulation and synthesis, these signals transfer MPTL data between the bridge and target MPTL interface blocks.

The MPTL interface sideband signals **mptl_sb_t2b** and **mptl_sb_b2t** connect the bridge and target MPTL interface blocks. 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 interface blocks.

6.2.7.1.3.1 OCP-Only Simulation

During OCP-only simulation (selected by **TARGET_USE** = SIM_OCP), a simulation only version of the **mptl_if_bridge_wrap** component is instantiated. **Table 111** 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

Table 111: 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

- At power-up, an online delay counter produces the **mptl_sb_b2t.mptl_bridge_gtp_online_l** output.
- 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 board, 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 boards, the **mptl_t2b.dma_abort** input drives the DMA abort request output **dma_abort**.

6.2.7.1.3.2 Full MPTL Simulation

During full MPTL simulation (selected by **TARGET_USE** = SIM_MPTL), the **mptl_if_bridge_wrap** component is instantiated. **Table 112** 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

Table 112: 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 board in use. **Table 113** 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

Table 113: 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.

OCP Interface

The **mptl_if_bridge_wrap** component direct master OCP channel inputs (**direct_master_s2m**) are processed by the **make_defined_s2m** function to ensure that they only contain '0' or '1' data. Other data values may cause the simulation of the MPTL interface to fail.

The remainder of the **mptl_if_bridge_wrap** component signals are connected to their equivalents on the MPTL interface HDL netlist.

6.2.7.2 Host PCIe Interface Wrapper (pcie_if_host_wrap)

6.2.7.2.1 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)**

6.2.7.2.2 Interface

The **pcie_if_host_wrap** component interface is shown in **Figure 52** below and described in **Table 114**.

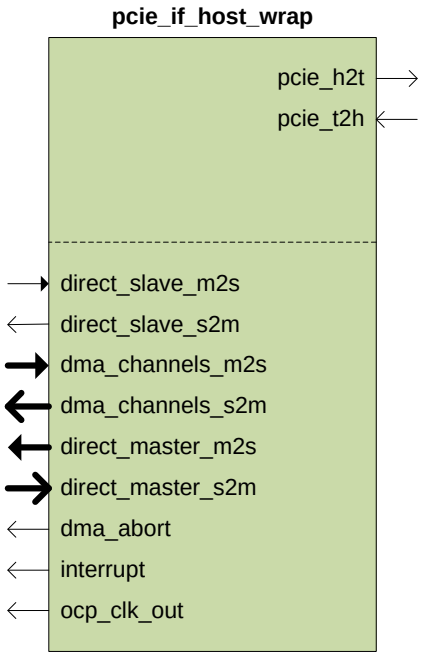

Figure 52: pcie_if_host_wrap Component Interface

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 mater 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).
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 114: pcie_if_host_wrap Component Interface

6.2.7.2.3 Description

The type of host PCIe interface that is instantiated depends upon which variant of the **adb3_target_inc_pkg** is in use, through the **TARGET_USE** constant.

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.

6.2.7.2.3.1 OCP-Only Simulation

During OCP-only simulation (selected by **TARGET_USE** = SIM_OCP), a simulation only version of the **pcie_if_host_wrap** component is instantiated. **Table 115** lists the available variants:

Model	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TADV8	admxrc6tadv8/pcie_host/pcie_if_host_wrap_sim_6tadv8.vhd

Table 115: 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**.

6.2.7.3 Board Clock Generation and Test (test_board_clks)

6.2.7.3.1 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)**

6.2.7.3.2 Interface

The **test_board_clks** component interface is shown in **Figure 53** below and described in **Table 116**.

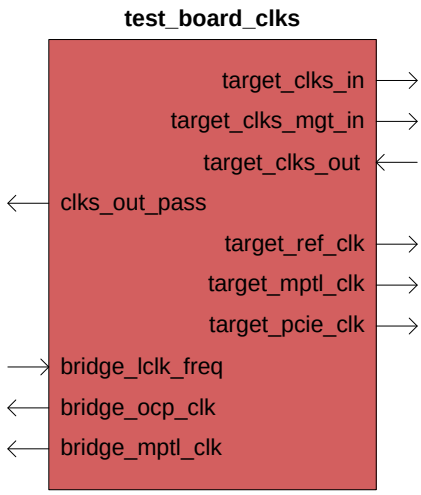

Figure 53: test_board_clks Component Interface

Signal	Type	Description
		Generics
clks_out_freq	Generic	Target clks_out expected frequencies.
		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
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 116: test_board_clks Component Interface

6.2.7.3.3 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 board in use.
- Test of target FPGA clock outputs **clks_out** using generic input **clks_out_freq** for the board in use.
- Generation of target FPGA reference, MPTL and PCIe clock inputs for the board 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 board in use.
- Generation of Bridge (testbench) MPTL clock for the board in use.

Table 117 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-6TADV8	admxrc6tadv8/pcie_host/test_board_clks_6tadv8.vhd

Table 117: 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 54** below and described in **Table 118**.

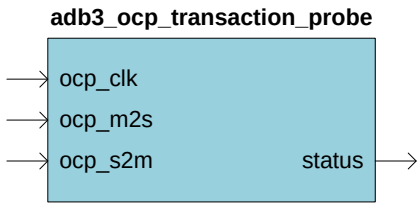

Figure 54: adb3_ocp_transaction_probe Component Interface

Signal	Type	Description
		Generics
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.
		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 118: 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

The memory interface group is located in the **hdl/vhdl/common/mem_if/** directory and contains the following elements:

- **Memory Interface Package (mem_if_pkg)**
- **Memory Interface Components**
- **Xilinx DDR3 SDRAM MIG Cores**

6.4.1 Memory Interface Package (mem_if_pkg)

The package **mem_if_pkg** defines types, constants, and functions which are used by the memory interface components.

Definitions are as follows:

DDR3 SDRAM bank physical interface types

- **ddr3_addr_out_t**. A record type containing address elements (outputs).
- **ddr3_ctrl_out_t**. A record type containing control elements (outputs).
- **ddr3_data_inout_t**. A record type containing data elements (bi-dir).
- **ddr3_clk_out_t**. A record type containing clock elements (outputs).

Memory interface functions

- **conv_sim_bypass_init_cal**. Returns the value of **sim_bypass_init_cal** that is appropriate for the **TARGET_USE** value in the variant of the **adb3_target_inc_pkg** that has been selected.
- **conv_sim_init_option**. Returns the value of **sim_init_option** that is appropriate for the **TARGET_USE** value in the variant of the **adb3_target_inc_pkg** that has been selected.
- **conv_sim_cal_option**. Returns the value of **sim_cal_option** that is appropriate for the **TARGET_USE** value in the variant of the **adb3_target_inc_pkg** that has been selected.
- **conv_t_rfc_option**. Returns the value of **t_rfc** that is appropriate for the **DDR3_BANK_ROW_WIDTH** value in the variant of the **adb3_target_inc_pkg** that has been selected.

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**

6.4.2 Xilinx DDR3 SDRAM MIG Cores

These cores are used by DDR3 SDRAM on-board memory interface components. Different core versions are supported by different versions of the Xilinx ISE tools as follows:

MIG Version	ISE Versions
MIG 3.6	ISE 12.3 onwards.

Table 119: MIG vs ISE Version Compatibility

Each supported board may use a different version of Xilinx DDR3 SDRAM MIG core. **Table 120** lists the versions currently used:

Model	MIG DDR3 SDRAM Core Version
ADM-XRC-6TL	3.6
ADM-XRC-6T1	3.6
ADM-XRC-6TGE	3.6
ADM-XRC-6TADV8	3.6

Table 120: Versions of DDR3 SDRAM MIG Core in Use

6.4.2.1 Xilinx DDR3 SDRAM MIG Core Generation

Prior to the initial simulation or bitstream build of a design using a Xilinx DDR3 SDRAM MIG core, its HDL files will need to be generated using the **gen_mem_if.tcl** script. Examples are as follows:

To generate MIG HDL files for ADM-XRC-6T1 boards using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\common\mem_if\ddr3_sdram
xtclsh gen_mem_if.tcl admxrc6t1
```

To generate MIG HDL files for ADM-XRC-6T1 boards using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/common/mem_if/ddr3_sdram
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 core for ADM-XRC-6T1 boards, 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 sucessful script execution.

6.4.3 Memory Interface Components

6.4.3.1 DDR3 SDRAM Memory Interface Bank (ddr3_if_bank)

6.4.3.1.1 Introduction

This is a component in the memory interface group. Its function is as follows:

- Conversion of single bank OCP channel transactions to DDR3 SDRAM MIG 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)**

6.4.3.1.2 Interface

The **ddr3_if_bank** component interface is shown in **Figure 55** below and described in **Table 121**.

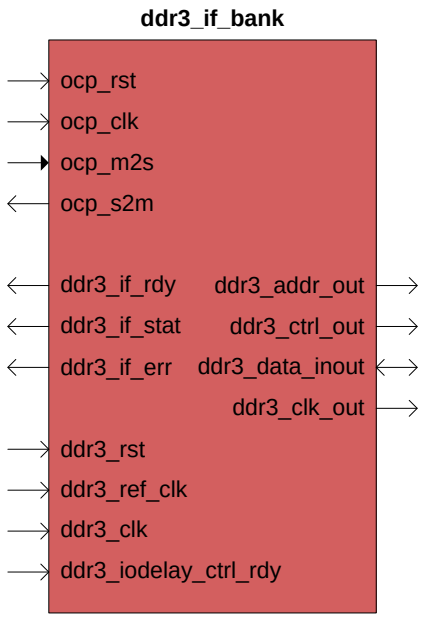

Figure 55: ddr3_if_bank Component Interface

Signal	Type	Description
bank	Generic	Bank select.
		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 121: ddr3_if_bank Component Interface

6.4.3.1.3 Description

This component converts single bank OCP channel transactions to DDR3 SDRAM MIG core user interface transactions and instantiates a single bank Xilinx DDR3 SDRAM MIG core.

The **ddr3_if_bank** component instantiated is board dependent. **Table 122** lists the available variants:

Model	Filename relative to hdl/vhdl/common/mem_if/ddr3_sdram/
ADM-XRC-6TL	admxrc6tl/ddr3_if_bank_6tl.vhd
ADM-XRC-6T1	admxrc6t1/ddr3_if_bank_6t1.vhd
ADM-XRC-6TGE	admxrc6tge/ddr3_if_bank_6tge.vhd
ADM-XRC-6TADV8	admxrc6tadv8/ddr3_if_bank_6tadv8.vhd

Table 122: Available Variants of ddr3_if_bank Component

It includes the following components:

- **adb3_ocp_ocp2ddr3_nb**
- Xilinx DDR3 SDRAM MIG core

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 Xilinx 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 **Xilinx DDR3 SDRAM MIG Core Generation** 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 board, **c0_memc_ui_top.vhd** located in **hdl/vhdl/common/mem_if/ddr3_sdram/admxrc6t1/rtl/mig_v3_6/ip_top/** is used when bank = 0.

6.5 Memory Application

The memory application group is located in the **hdl/vhdl/common/mem_apps/** directory and contains the following elements:

- **Memory Application Components**

6.5.1 Memory Application Components

6.5.1.1 Memory Test Block (blk_mem_test)

6.5.1.1.1 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)**

6.5.1.1.2 Interface

The **blk_mem_test** component interface is shown in **Figure 56** below and described in **Table 123**.

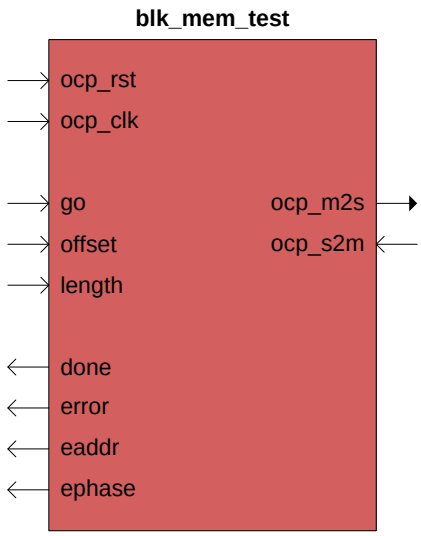

Figure 56: blk_mem_test Component Interface

Signal	Type	Description
a_width	Generic	Number of logical address bits in memory port.
d_width	Generic	Number of logical bits in a memory port 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.

Table 123: blk_mem_test Component Interface (continued on next page)

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	Test start (16-byte words).
length	Input	Test length-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 123: blk_mem_test Component Interface

6.5.1.1.3 Description

TBD

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 Components**

6.6.1.1 DDR3 SDRAM Model Package (ddr3_sdram_pkg)

The package **ddr3_sdram_pkg** defines types, constants, and components which are used 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_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).
- **MT41J_25E_CL_CWL_MIN**. Micron MT41JxMx_25E (minimum values).
- **MT41J_25E_CL_CWL_MAX**. Micron MT41JxMx_25E (maximum values).

Supported speed_grade_t constants

- **MT41J_15E**. Micron MT41JxMx_15E.
- **MT41J_187E**. Micron MT41JxMx_187E.
- **MT41J_25E**. Micron MT41JxMx_25E.

Supported part_t constants

- **MT41J64M16_15E**. Micron MT41J64M16_15E (1Gib part).

- **MT41J64M16_187E**. Micron MT41J64M16_187E (1Gib part).
- **MT41J128M16_15E**. Micron MT41J128M16_15E (2Gib part).
- **MT41J128M16_187E**. Micron MT41J128M16_187E (2Gib 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 Components

6.6.1.2.1 DDR3 SDRAM Model (ddr3_sdram)

6.6.1.2.1.1 Introduction

This is a component in the memory model group. Its function is to provide a generic simulation model which may be customised to represent specific DDR3 SDRAM parts.

Dependencies

- **DDR3 SDRAM Model Package (ddr3_sdram_pkg)**

6.6.1.2.1.2 Interface

The **ddr3_sdram** component interface is shown in **Figure 57** below and described in **Table 124**.

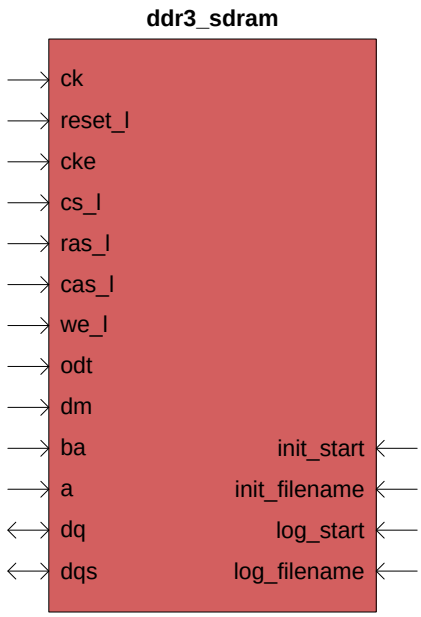

Figure 57: ddr3_sdram Component Interface

Signal	Type	Description
message_level	Generic	Select message reporting level.
part	Generic	Select component part.
short_init_dly	Generic	Select shortened initialisation sequence.
		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).

Table 124: ddr3_sdram Component Interface (continued on next page)

Signal	Type	Description
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.
init_filename	Input	Initialisation file name (default "init.txt").
log_start	Input	Save data log file.
log_filename	Input	Log file name (default "log.txt").

Table 124: ddr3_sdram Component Interface

6.6.1.2.1.3 Description

TBD

6.6.1.2.1.3.1 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.

6.6.1.2.1.3.2 Part Selection

The generic **part** selects the DDR3 SDRAM part to be simulated by the model.

6.6.1.2.1.3.3 Initialisation Delay Selection

The generic **short_init_dly** controls the DDR3 SDRAM initialisation sequence. The length of this sequence may be reduced during simulation by setting this generic to 'true'

6.6.1.2.1.3.4 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 1Gib, 0..16383 2Gib, 0..32767 4Gib).
- Start COL (decimal 0-1023).
- Start data BYTE (decimal 0-1).
- Data Bytes from starting byte.

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 16 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
```

6.6.1.2.1.3.5 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 modified 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 1Gib, 0..16383 2Gib, 0..32767 4Gib).
- Start COL (decimal 0-1023).
- Start data BYTE (decimal 0-1).
- Data Bytes from starting byte.

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
```

6.7 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**

6.7.1 Clock Frequency Measurement Components

6.7.1.1 Clock Frequency Measurement Block (blk_clock_freq)

6.7.1.1.1 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

6.7.1.1.2 Interface

The **blk_clock_freq** component interface is shown in **Figure 58** below and described in **Table 125**.

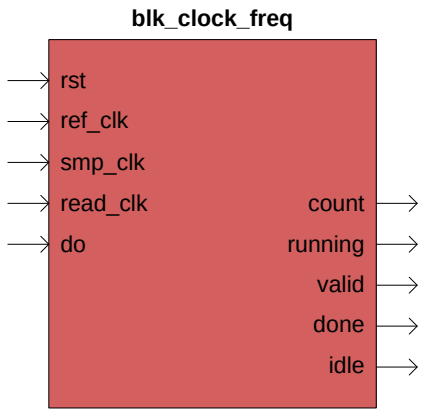

Figure 58: blk_clock_freq Component Interface

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**.
		Reset/Clocks
rst	Input	Asynchronous reset.
ref_clk	Input	Reference clock.
smp_clk	Input	Sample clock (to be measured).
		Read Clock Domain
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 125: blk_clock_freq Component Interface

6.7.1.1.3 Description

TBD

6.7.1.1.3.1 Clock Frequency Measurement Block Constraints

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**.

6.8 ChipScope

The Chipscope group is located in the **hdl/vhdl/common/chipscope/** directory and contains the following elements:

- **ChipScope Components**

6.8.1 ChipScope Components

6.8.1.1 ChipScope Block (blk_chipscope)

6.8.1.1.1 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)**

6.8.1.1.2 Interface

The **blk_chipscope** component interface is shown in **Figure 59** below and described in **Table 126**.

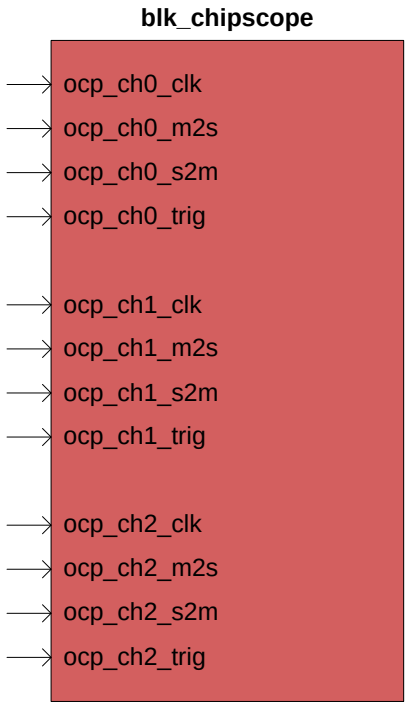

Figure 59: blk_chipscope Component Interface

Signal	Type	Description
instantiate	Generic	Enables generation of this component.
		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 126: blk_chipscope Component Interface

6.8.1.1.3 Description

Instantiation of this component is controlled by the value of the **instantiate** generic. A true value is required for instantiation.

6.8.1.1.3.1 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.

6.8.1.1.3.2 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.

6.8.1.1.4 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 ADM-XRC-6T1 boards 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 ADM-XRC-6T1 boards 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

OCP-IP Protocols 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 a command group, synchronous with the **Cmd** signal, and data transfer groups both from master to slave (write) and slave to master (read). Each of these groupings 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 the following signals to the slave port:

Signal	Group	Type	Width	Description
Cmd	Command	ocp_CmdT	3	Command (Idle, Write, Read)
Addr	Command	std_logic_vector	64	Command address (16-byte aligned)
BurstLength	Command	std_logic_vector	12	Command burst length
Tag	Command	std_logic_vector	8	Command tag
Data	Data	std_logic_vector	128	Write data beat
DataByteEn	Data	std_logic_vector	16	Write data beat byte enables
DataValid	Data	std_logic	1	Write data group valid
RespAccept	Response	std_logic	1	Read response group accept

Table 127: Master Port To Slave Port Signals

Notes:

- Port signals are active high.
- **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.
- **DataByteEn** enables writing of **Data** on a byte by byte basis.

The slave port outputs the following signals to the master port:

Signal	Group	Type	Width	Description
CmdAccept	Command	std_logic	1	Command group accept
DataAccept	Data	std_logic	1	Write data group accept
Data	Response	std_logic_vector	128	Read response data beat
Resp	Response	ocp_RespT	2	Read response group valid (None, Valid)
Tag	Response	std_logic_vector	8	Read response tag

Table 128: Slave Port To Master Port Signals

Notes:

- Port signals are active high.
- **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 OCP Port Operation

Each OCP Link operates as follows:

Command Group

1. When required, the Master Port outputs a write/read command, together with its associated address, tag, and burst length.
2. The Slave Port indicates its readiness to accept Master Port command group signals using its **CmdAccept** signal. For each cycle in which there is a write/read command present together with an active **CmdAccept**, the command group signals will be accepted by the Slave Port. The Slave Port asserts **CmdAccept** as and when it is able to.
3. The next write/read command may be output in the cycle following the acceptance of the command group.

Data Group (Associated with write transactions only)

1. When required, the Master Port outputs a data beat, together with its associated byte enables and data valid flag.
2. The Slave Port indicates its readiness to accept Master Port data group signals using its **DataAccept** signal. For each cycle in which there is a valid data beat present together with an active **DataAccept**, the data group signals will be accepted by the Slave Port. The Slave port asserts **DataAccept** as and when it is able to.
3. The next valid data beat may be output in the cycle following the acceptance of the data group.

Response Group (Associated with read transactions only)

1. The Slave Port outputs a response data beat, together with its associated tag and response valid flag, as and when it is able to.

2. The Master Port indicates its readiness to accept Slave Port response group signals using its **RespAccept** signal. For each cycle in which there is a valid response data beat present together with an active **RespAccept**, the response group signals will be accepted by the Master Port. The Master port asserts **RespAccept** as and when it is able to.
3. The next response data beat may be output in the cycle following the acceptance of the response group.

In summary:

- Provision of transaction commands and write data is controlled by the Master port.
- Acceptance of transaction commands and write data is controlled by the Slave port.
- Provision of read response data is controlled by the Slave port.
- Acceptance of read response data is controlled by the Master port.

The operation of each group of signals is independent. Write or read Commands may be provided before, during, or after write data. Write or read 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 beats provided corresponds with burst lengths of write commands. It is the responsibility of the slave port to ensure that the number of read response data beats provided corresponds with burst lengths of read commands.

There is dependence of operation within each group of signals. Write data should be provided in the same order as write commands are issued. Response data will be returned in the same order as read commands are issued.

During write transactions, writing of data is enabled on a byte by byte basis using the data group signal **DataByteEn**. During read transactions, the full 16-bytes of response data is read and returned to the Slave Port.

7.1.4 Example OCP Transaction Waveforms

This section contains timing diagrams for most common transactions and illustrates 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.

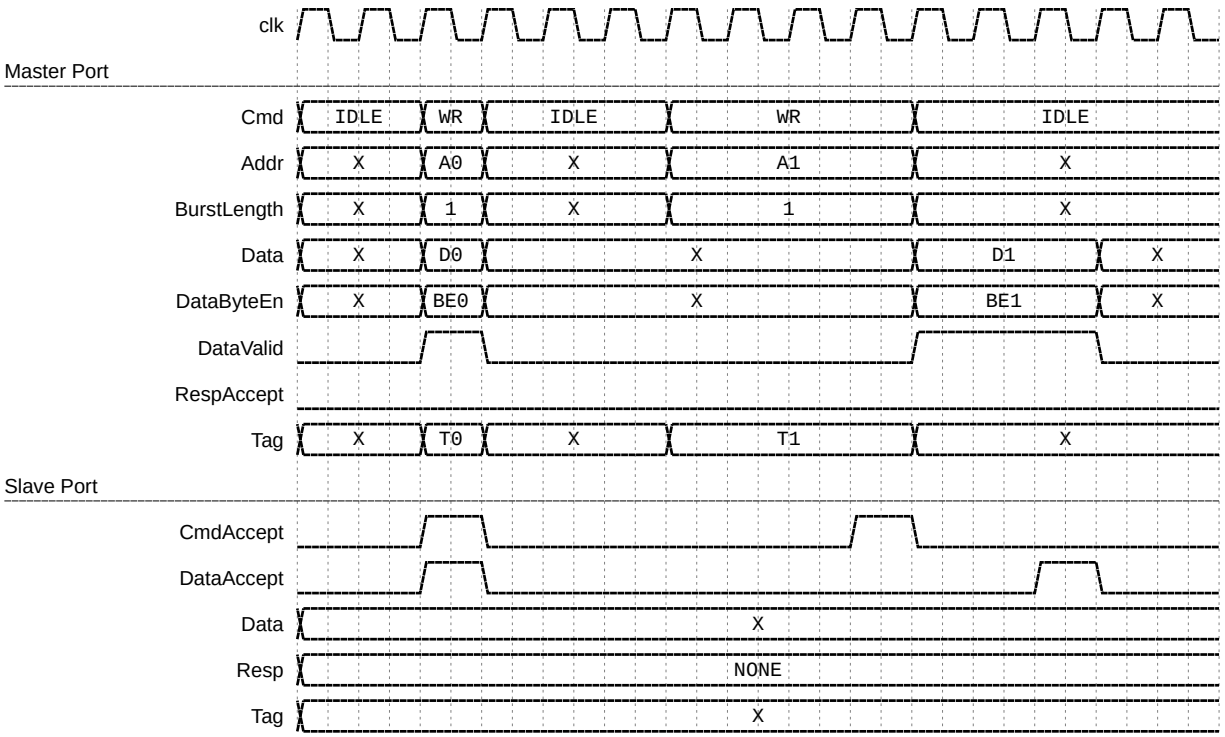

Figure 60: Single Beat Write Transactions

Figure 60 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 4th cycle after it is asserted (possibly due to the slave being busy). The master in this case also does not assert the **DataValid** signal until after the **Cmd**. The **Data** is also not accepted immediately, and therefore the **DataValid** must remain high until the **Data** is accepted. Both examples are legal within the protocol.

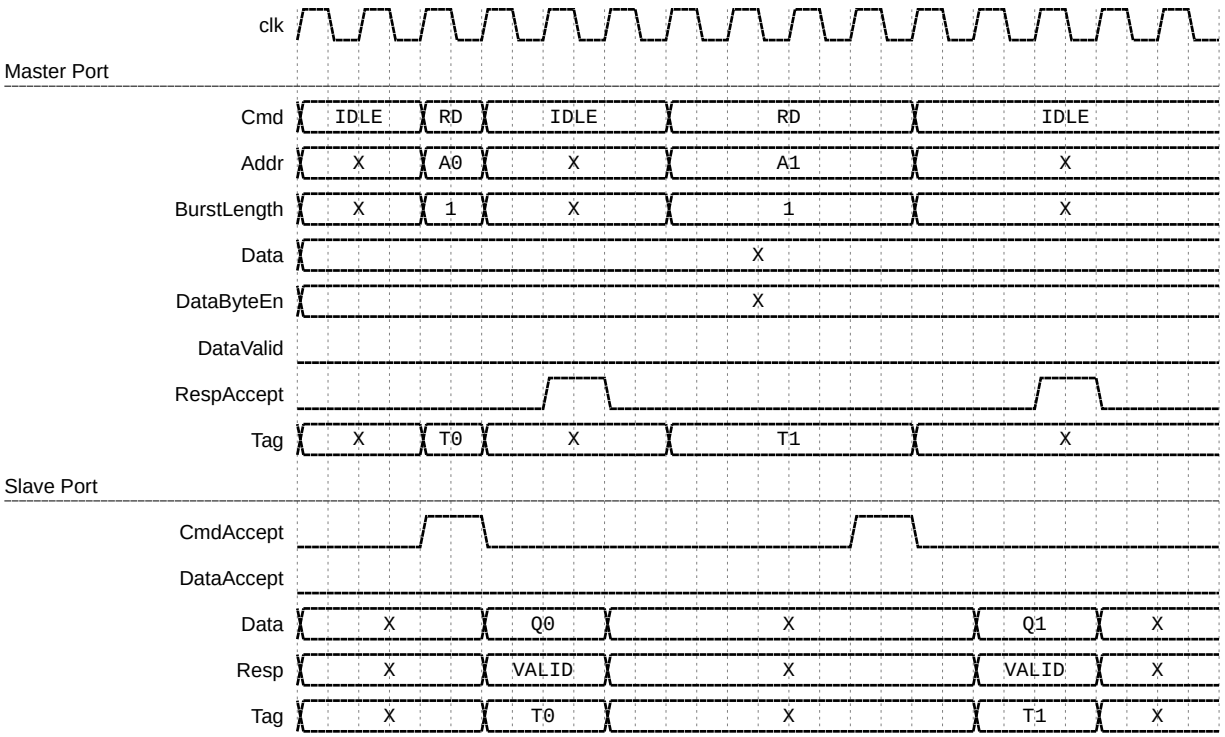

Figure 61: Single Beat Read Transactions

Figure 61 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 immediately accepted. The slave port returns valid response **Data** (Q0) on the following clock cycle. The **Tag** sent with the **Cmd** is returned with the response **Data**. The **Resp** indicates when the response **Data** and **Tag** are valid. The valid response is accepted by **RespAccept** on the following clock cycle.

The second example shows a delayed command accept, a delayed response and a delayed response accept. Both examples are legal within the protocol.

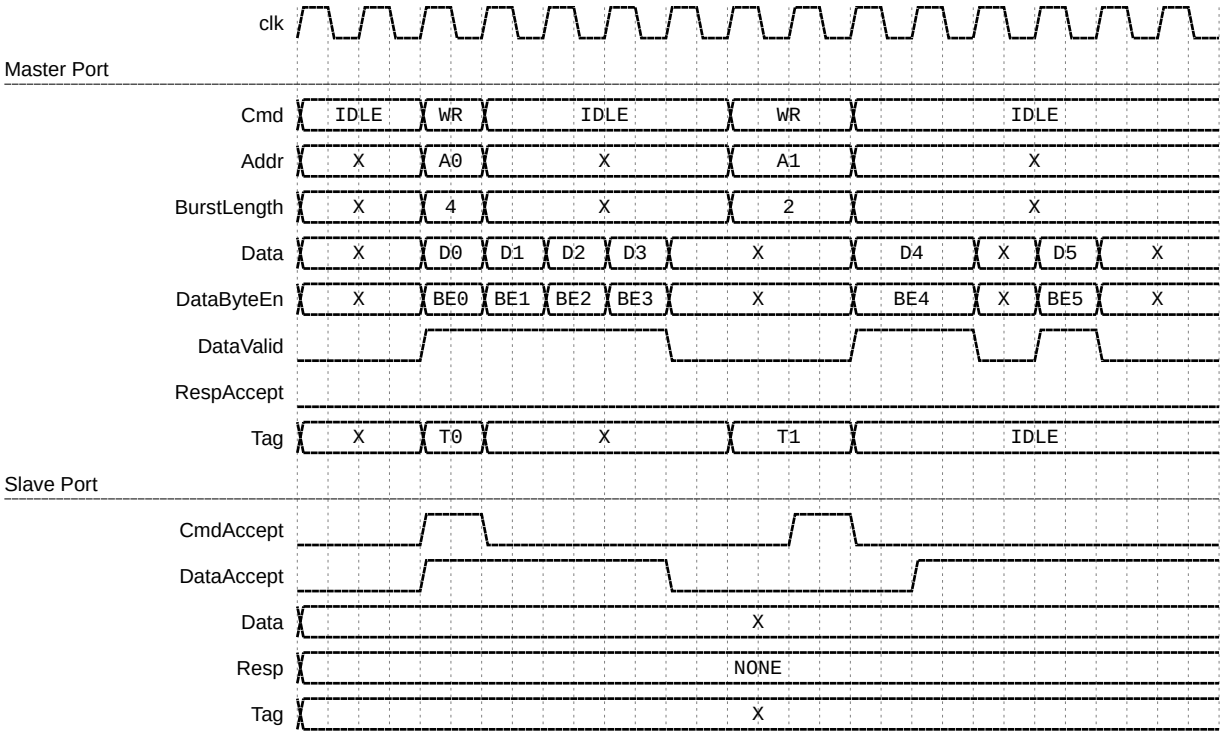

Figure 62: Burst Write Transactions

Figure 62 shows two burst write transactions. A single **Cmd** is issued for multiple write **Data** beat transfers. The command protocol operates in exactly the same manner as for single beat transfers. Multiple write data beat transfers occur for each write **Cmd**. 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** beat. The slave port must check that **DataValid** is asserted before accepting the write **Data** beat.

> **Note:** The **DataAccept** signal indicates that the slave port is able to accept master port write data present on **Data**. The slave port may assert **DataAccept** even if **DataValid** is not asserted.

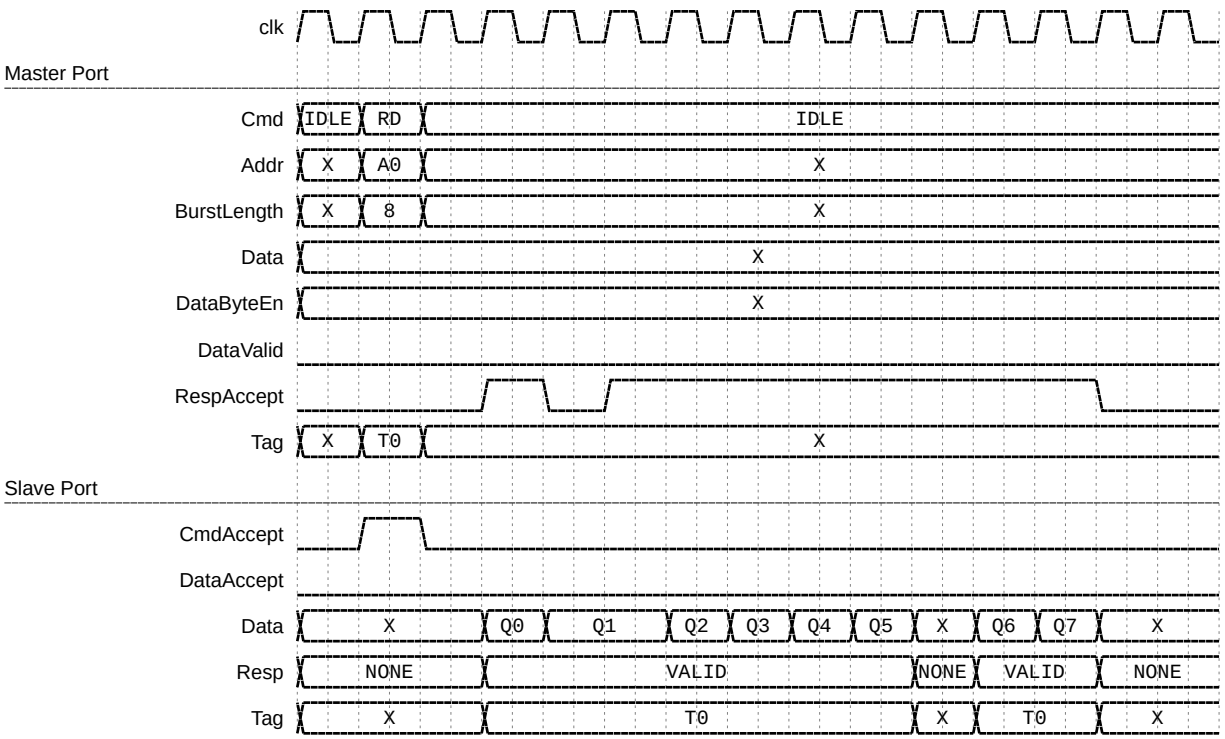

Figure 63: Burst Read Transactions

Figure 63 shows a single burst read transaction. A single **Cmd** is issued for multiple read response **Data** beat transfers. The command protocol operates in exactly the same manner as for single beat transfers. Multiple response data beat transfers occur for each read **Cmd**. Read response **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 response **Data** beat. The master port must check that **Resp** is **Valid** before accepting the read response **Data** beat.

> **Note:** The **RespAccept** signal indicates that the master port is able to accept slave port read response data present on **Data**. The master port may assert **RespAccept** even if **Resp** is not **Valid**.

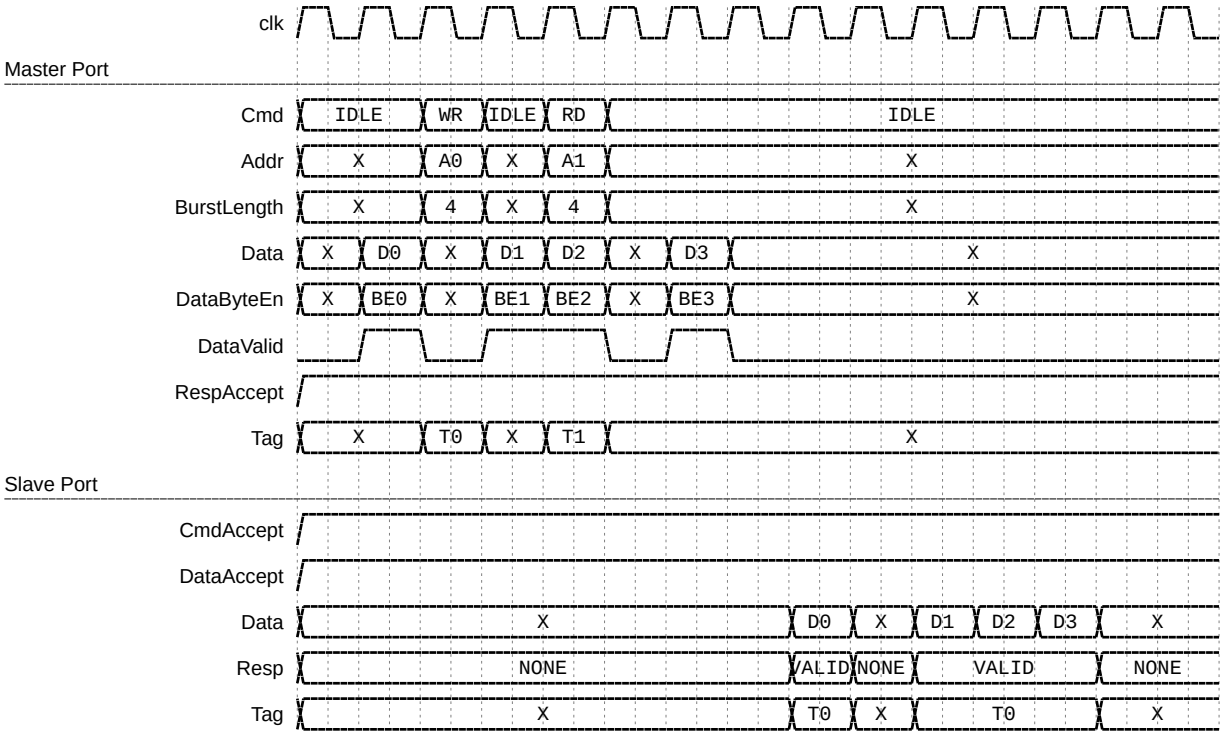

Figure 64: 'Valid' Controlled Transactions

Figure 64 shows a single burst write transaction followed by a single burst read transaction. The write **Data** beats 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. Commands and data group signals are operating independently to each other.

The read response **Data** beats 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 able 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
20/05/2010	1.0	Initial version
26/07/2010	1.1	Updated for release 1.1.0 Added SDK structure diagram. Added information about example applications.
21/09/2010	1.2	Updated for release 1.2.0 Added section for getting started in VxWorks. Documented VxWorks example applications.
04/03/2011	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.
11/05/2011	1.4	Updated for release 1.3.1
05/09/2011	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.

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