ALPHA DATA

ADM-XRC Gen 3 SDK 1.3.0 User Guide

Revision: 1.3
Date: 04th March 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

Page Intentionally left blank.

1 Introduction

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

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

1.1 Document conventions

In order to avoid unnecessary repetition of information pertaining to both Windows and Linux environments, the directory separator character for pathnames in this document is 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-1.3.0/apps/src/itest/itest.c** in a Linux environment, assuming that the root of the SDK is **/opt/admxrcg3sdk-1.3.0/**.

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-1.3.0/** in 32-bit editions of Windows.
- **%ProgramFiles(x86)%/ADMXRCG3SDK-1.3.0/** 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-1.3.0/**.

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-1.3.0
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

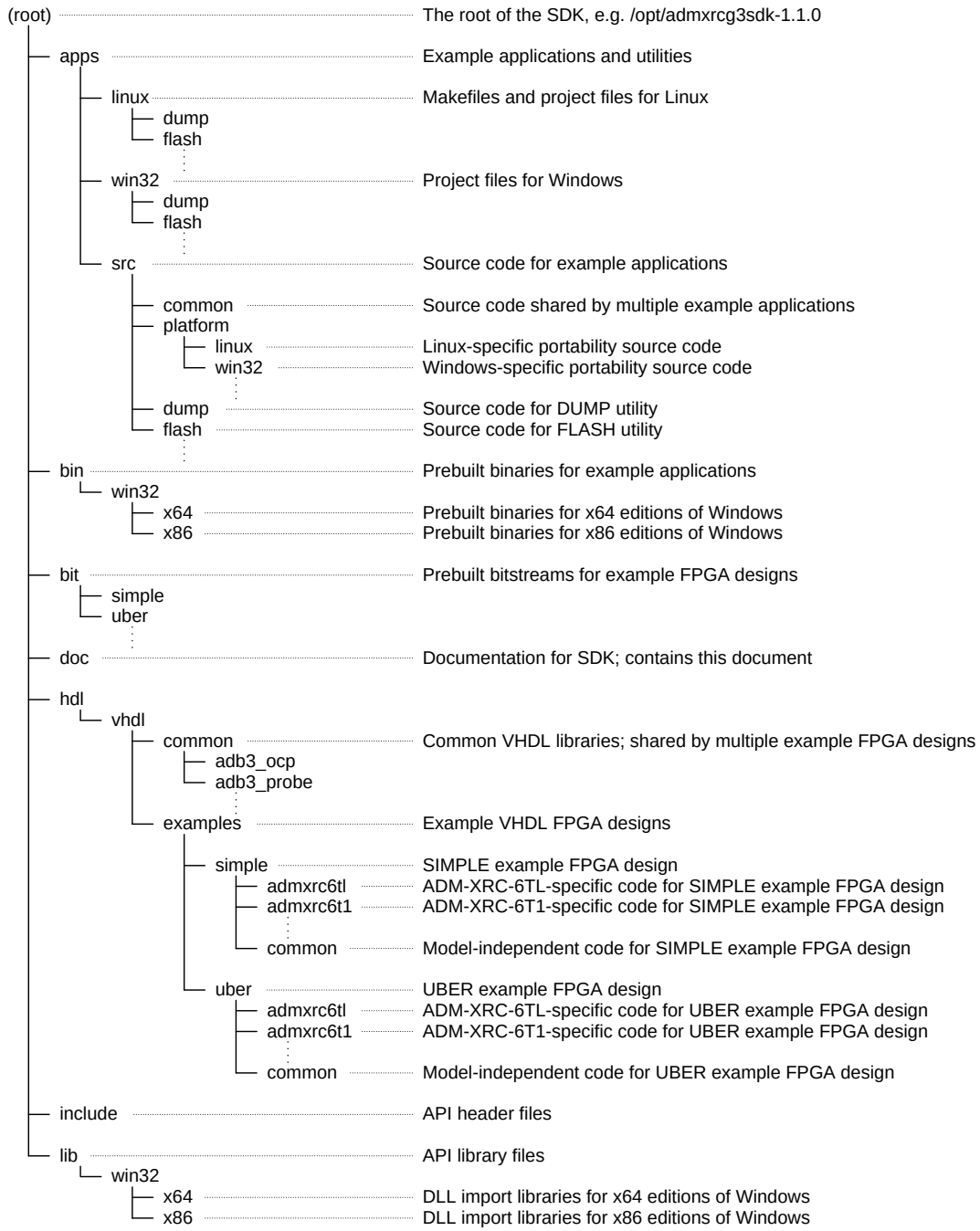

Figure 1: Structure of the ADM-XRC Gen 3 SDK

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 3.10.1**.

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
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**
  Displays information about the Flash memory.
- **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.

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

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 5.5.4.3.5** as a means of generating FPGA interrupt notifications, and starts a thread whose purpose is to wait for and acknowledge interrupts from the target FPGA.

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

A sample session looks like this:

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

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

Remarks

As noted in the ADMXRC3 API Specification (see functions **ADMXRC3_RegisterWin32Event**, **ADMXRC3_RegisterVxwSem** and **ADMXRC3_StartNotificationWait**), the ADMXRC3 API does not queue each type of notification. Therefore, this example works as expected as long as the frequency of target FPGA interrupt notifications is not too fast for the interrupt thread. Since the rate of generation of notifications in this example is limited the user's keyboard input rate, the interrupt thread should be able to keep up (as long as the machine is not heavily loaded with other processes). Nevertheless, it is important to note that in this simple example, there is no mechanism for throttling the rate of notifications so that notifications cannot be lost. In a real application, the preferred design approaches are:

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

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

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:

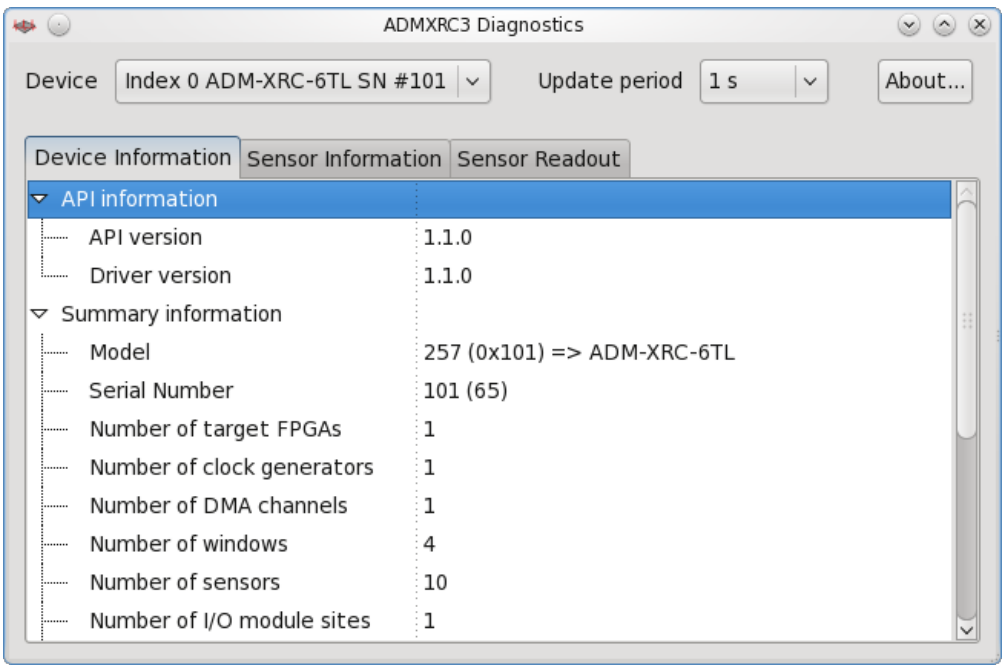

Figure 2: SYSMON user interface - device information

The Windows version of SYSMON offers equivalent functionality, but uses a different GUI technology to that of the Linux version. The second tab shows sensor readings in tabular form:

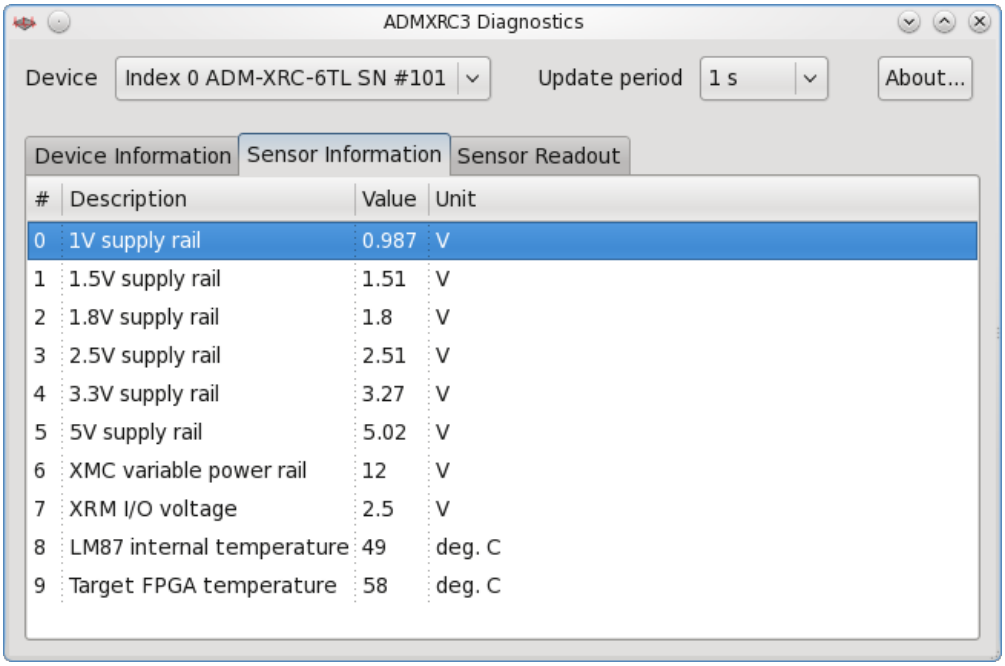

Figure 3: SYSMON user interface - sensor readings

The third tab displays sensor readings in graphical form:

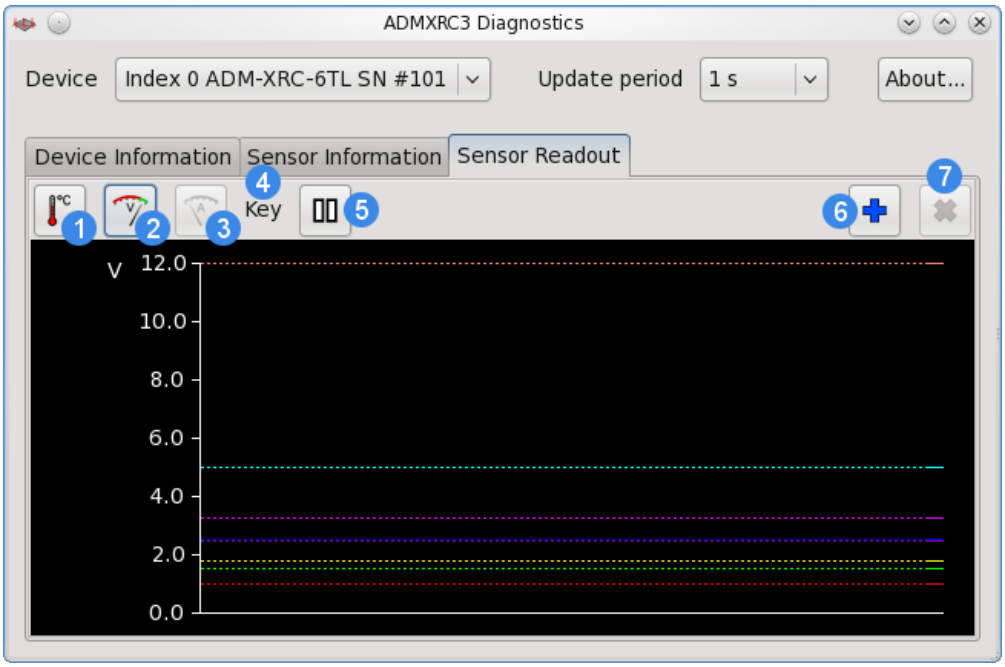

Figure 4: SYSMON user interface - sensor display

Initially, the 'scope is empty and displays no sensors. The above figure shows the effect of clicking the voltage button, labelled 2 in the above figure. The user interface elements of the 'scope toolbar are as follows:

1. The temperature button sets the 'scope to display all temperature sensors in the device. Once some sensors are displayed, updates begin.
2. The voltage button sets the 'scope to display all voltage sensors in the device. Once some sensors are displayed, updates begin.
3. The current button sets the 'scope to display all current sensors in the device. Once some sensors are displayed, updates begin.
4. Mouse over the key to see which sensor corresponds to which colored trace.
5. The pause / resume button can be used to pause and resume update of the 'scope.
6. Item 6 is 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.
7. Item 7 is a button that destroys a 'scope when clicked. If there is only one 'scope, the button is disabled.

3.10.1 Building SYSMON in Linux

The Linux version of the **SYSMON** utility uses **GTKMM-2.4**. This package is present in recent Linux distributions such as Fedora Core 13, but may not be present in all 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"
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.

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 **+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, 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

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 5.5.4.3.5** as a means of generating FPGA interrupt notifications, and starts a thread whose purpose is to wait for and acknowledge interrupts from the target FPGA.

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

A sample session looks like this:

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

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

Remarks

As noted in the ADMXRC3 API Specification (see functions **ADMXRC3_RegisterWin32Event**, **ADMXRC3_RegisterVxwSem** and **ADMXRC3_StartNotificationWait**), the ADMXRC3 API does not queue each type of notification. Therefore, this example works as expected as long as the frequency of target FPGA interrupt notifications is not too fast for the interrupt thread. Since the rate of generation of notifications in this example is limited the user's keyboard input rate, the interrupt thread should be able to keep up (as long as the machine is not heavily loaded with other processes). Nevertheless, it is important to note that in this simple example, there is no mechanism for throttling the rate of notifications so that notifications cannot be lost. In a real application, the preferred design approaches are:

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

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

4.7 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. 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 the **adb3_target_inc_pkg** package. Refer to **Table 83**.

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 version 12.3 or 12.4 is required by this version of the SDK.

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 the 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-admxrc6t1-6vlx240t.prj** for an ADM-XRC-6T1 fitted with a 6VLX240T device.

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-admxrc6t1-6vlx240t.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/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 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 are considered intermediate files of the build process are placed in the directories **hdl/vhdl/examples/simple/build/** and **hdl/vhdl/examples/simple/edif/**. Output files, including **.bit** files, are placed in **hdl/vhdl/examples/simple/output/**. Filenames of any bitstreams built are thus of the form **hdl/vhdl/examples/simple/output/simple-<*board*>-<*device*>.bit**. When a target of class "clean" is executed, output and intermediate files are deleted, but files in **bit/simple/** are unaffected.

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 on the Virtex-6 series of ADM-XRC boards. The design consists of:

- **Clock Generation**
- **Target MPTL interface block**, using an instance of **mptl_if_target_wrap**
- **OCP to simple bus interface block**, using an instance of **adb3_ocp_simple_bus_if**
- **Simple test registers** implemented using VHDL processes.

Figure 5 below shows the main elements of the **Simple** design:

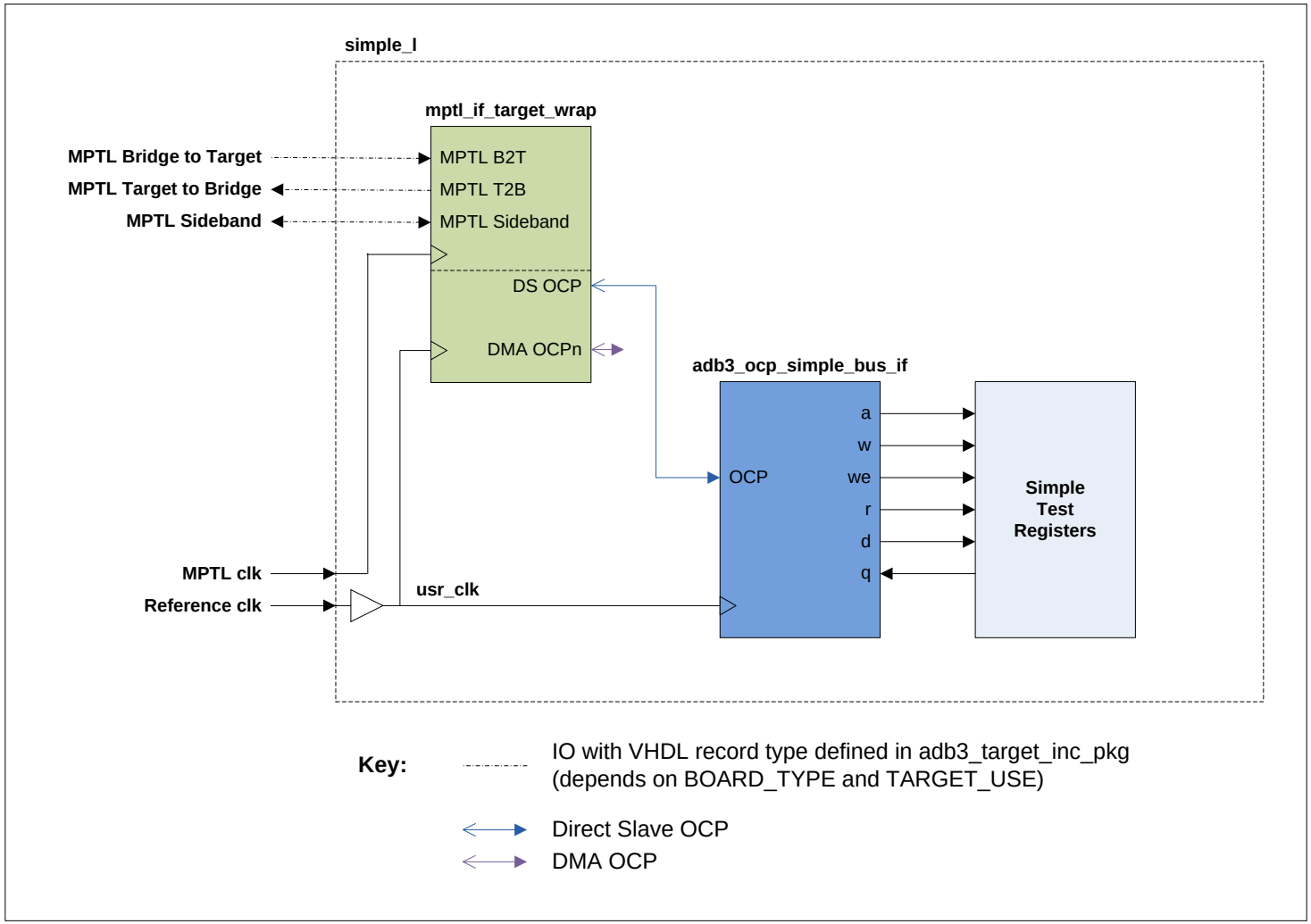

Figure 5: Simple Design Block Diagram

5.4.4.1 Clock Generation

5.4.4.1.1 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 the ADM-XRC-6T1.

5.4.4.1.2 Target MPTL Interface Clock

The target MPTL interface block requires a clock to be input via its **mptl_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 the ADM-XRC-6T1. It is differential and buffered within the MPTL interface block.

5.4.4.2 Target MPTL Interface

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 (the data channel between the Bridge and Target FPGAs) to the Direct Slave and DMA OCP channels within the FPGA design. Refer to the component **mptl_if_target_wrap** for details.

> **Note:** The Direct Slave address space supported by the Bridge is smaller than the full ADB3 OCP address space. For the board in use, it is indicated by the **DS_ADDR_WIDTH** constant in the package **adb3_target_inc_pkg**.

> **Note:** The DMA address space supported by the Bridge is smaller than the full ADB3 OCP address space. For the board in use, it is indicated by the **DMA_ADDR_WIDTH** constant in the package **adb3_target_inc_pkg**.

5.4.4.3 OCP to Simple Bus Interface Block

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. **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 **ds_q** for reads.
2. **ds_w** - Indicates that write data is valid on the signal **ds_d** and write byte enables are valid on the signal **ds_we**.
3. **ds_we** - Byte write enables; qualified by **ds_w**.
4. **ds_d** - Write data; qualified by **ds_w**.
5. **ds_r** - Indicates that valid data must be presented on **ds_q** on the following clock cycle.
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.

5.4.4.4 Simple Test Registers

A set of VHDL processes uses the signals **ds_a**, **ds_w** 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.

5.4.4.4.1 Register Description

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

Name	Type	Address
DATA	RW	0x000000

Table 5: 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 6: 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 testbench for the **Simple** example FPGA design is implemented in **hdl/vhdl/examples/simple/common/test_simple.vhd**. **Figure 6** below shows the testbench, with the **simple_l** FPGA design embedded in it.

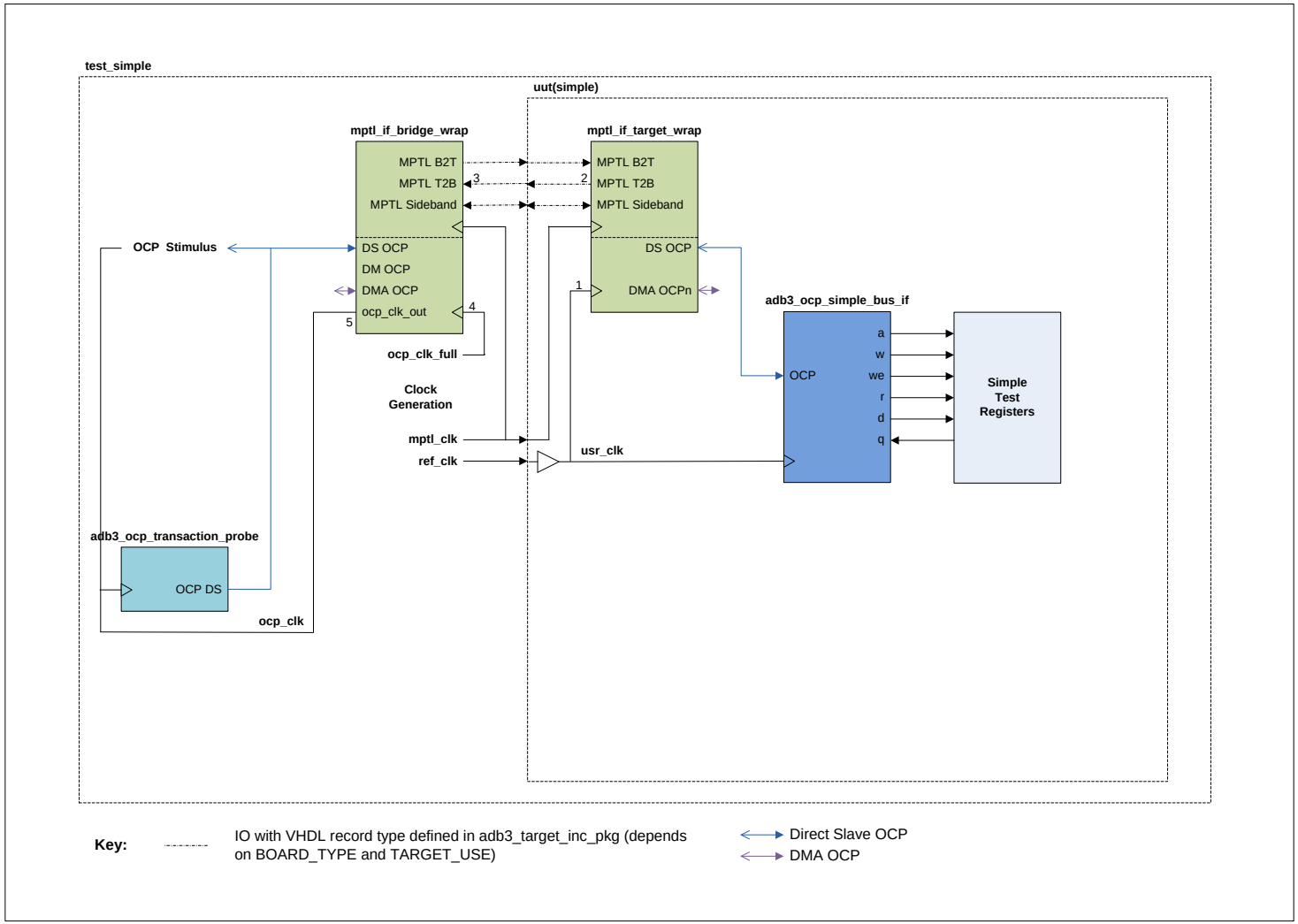

Figure 6: Simple Design Testbench Block Diagram

The **Simple** example FPGA design testbench 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_l** block.
- **The Bridge MPTL interface block**, using an instance of **mptl_if_bridge_wrap**.
- **Direct Slave OCP channel probe**, using an instance of **adb3_ocp_transaction_probe**.
- **Stimulus Generation and Verification**.

5.4.5.1 Clock Generation

The testbench generates the clocks **ref_clk** and **mptl_clk** according to which board is selected, in order to model the hardware, and these drive the unit under test (**simple_l**).

The testbench also feeds **mptl_clk** into the Bridge MPTL Interface (an instance of **mptl_if_bridge_wrap**).

The Bridge MPTL Interface **mptl_if_bridge_wrap** port **ocp_clk_out** drives the OCP clock **ocp_clk** that is used within the testbench for monitoring OCP transactions. This is generated depending on the type of simulation selected by the **TARGET_USE** constant in the package **adb3_target_inc_pkg**:

- In OCP-only simulation (**TARGET_USE = SIM_OCP**), the UUT's main OCP clock (**usr_clk** in this case) is routed out of the UUT (**simple_l**) via the **mptl_if_target_wrap** instance and into the testbench's instance of **mptl_if_bridge_wrap**. The **mptl_if_bridge_wrap** instance outputs this signal as **ocp_clk**. This route is shown in **Figure 6** as the route consisting of points 1, 2, 3 and 5.
- In full MPTL simulation (**TARGET_USE = SIM_MPTL**), **ocp_clk** is entirely independent of any clock within the UUT, and the testbench's **mptl_if_bridge_wrap** instance passes **ocp_clk_full** through to **ocp_clk**. This is shown in **Figure 6** as the route consisting of points 4 and 5.

5.4.5.2 Bridge MPTL Interface

The testbench contains an instance of **mptl_if_bridge_wrap**, which translates Direct Slave and DMA OCP transactions in the testbench to MPTL data. **mptl_if_bridge_wrap** wraps up the Bridge MPTL interface core, instantiating an OCP to MPTL core appropriate for the **BOARD_TYPE** and **TARGET_USE** constants from the package **adb3_target_inc_pkg**.

The **mptl_if_bridge_wrap** output **mptl_sb_b2t.mptl_bridge_gtp_online_l** is combined with the **Simple** example FPGA design output **mptl_sb_t2b.mptl_target_gtp_online_l** to produce the **mptl_online_long** signal. This indicates that the MPTL interface is active and stable.

Note: The testbench monitors **mptl_online_long** and will terminate the simulation 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 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**.

5.4.5.4 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. There is one test section:

5.4.5.4.1 Direct Slave OCP Channel

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

The **test_simple** testbench, implemented in **test_simple.vhd**, provides test stimulus to and verifies test results from the UUT's OCP Direct Slave channel. 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 Direct Slave OCP channels:

- The **Bridge MPTL interface** converts OCP commands and write data originating in **test_simple** to MPTL protocol. Within the target FPGA, the **Target MPTL interface** converts MPTL protocol back into OCP commands and data. Thus, neither **test_simple** nor the UUT (**simple**) is aware that OCP stimulus passes through the MPTL.
- Responses originating in the Target FPGA are correspondingly converted to MPTL protocol by the **Target MPTL interface**, and converted back into OCP responses by the **Bridge MPTL interface**}. Thus, neither **test_simple** nor the UUT (**simple**) is aware that OCP responses pass through the MPTL.

Tests performed are detailed in the following subsections.

5.4.5.4.1.1 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** signals respectively in **test_simple.vhd**.

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

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 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.bat** (Windows) or **gen_mem_if.bash** (Linux) in **hdl/vhdl/common/mem_if/ddr3_sdram/mig_v3_6/**. Refer to **Section 6.5** 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. If **CHIPSCOPE_ON** is **true**, the ChipScope™ ILA core **chipscope_ila.ngc** and ICON core **chipscope_icon.ngc** must be generated using the **gen_ChipScope™.bat** (Windows) or **gen_ChipScope™.bat** (Linux) script in **hdl/vhdl/common/ChipScope™/**. Refer to **Section 6.9** 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 7: Uber Design Makefile Targets

Files that are considered intermediate files of the build process are placed in the directories **hdl/vhdl/examples/uber/build/** and **hdl/vhdl/examples/uber/edif/**. Output files, including **.bit** files, are placed in **hdl/vhdl/examples/uber/output/**. Filenames of any bitstreams built are thus of the form **hdl/vhdl/examples/uber/output/uber-<*board*>-<*device*>.bit**. When a target of class "clean" is executed, output and intermediate files are deleted, but files in **bit/uber/** are unaffected.

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 date and 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_6vlx240t.vhd
-- This file was generated automatically using the file gen_today_pkg.bat
--

library ieee;
use ieee.std_logic_1164.all;

package today_pkg is

  constant TODAYS_DATE : std_logic_vector(31 downto 0) := X"06102010";
  constant TODAYS_TIME : std_logic_vector(31 downto 0) := X"11043305";

end package today_pkg;
```

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.

The design includes the following functional areas:

- Clock generation block (**blk_clocks**)
- MPTL interface block (**mptl_if_target_wrap**)
- OCP Direct Slave block (**blk_direct_slave**), which includes:
  - **Connection between clock domains**, between the **pll_pri_clk** domain and the relatively low frequency **pll_reg_clk** domain.
  - **Direct Slave address space splitter block**
  - 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™**)

The top-level VHDL source file of **Uber** is **hdl/vhdl/examples/uber/common/uber.vhd**. **Figure 7** shows its main elements. **Figure 8** shows the hierarchy of the design.

The design includes the following packages:

- **ADB3 OCP profile definition package (adb3_ocp)**
- **ADB3 OCP library component declaration package (adb3_ocp_comp)**
- **ADB3 target types definition package (adb3_target_types_pkg)**
- **ADB3 target include package (adb3_target_inc_pkg)**
- **ADB3 target package (adb3_target_pkg)**
- **Memory interface library package (mem_if_pkg)**
- **Design package (uber_pkg)**

Figure 9 shows the design package dependencies.

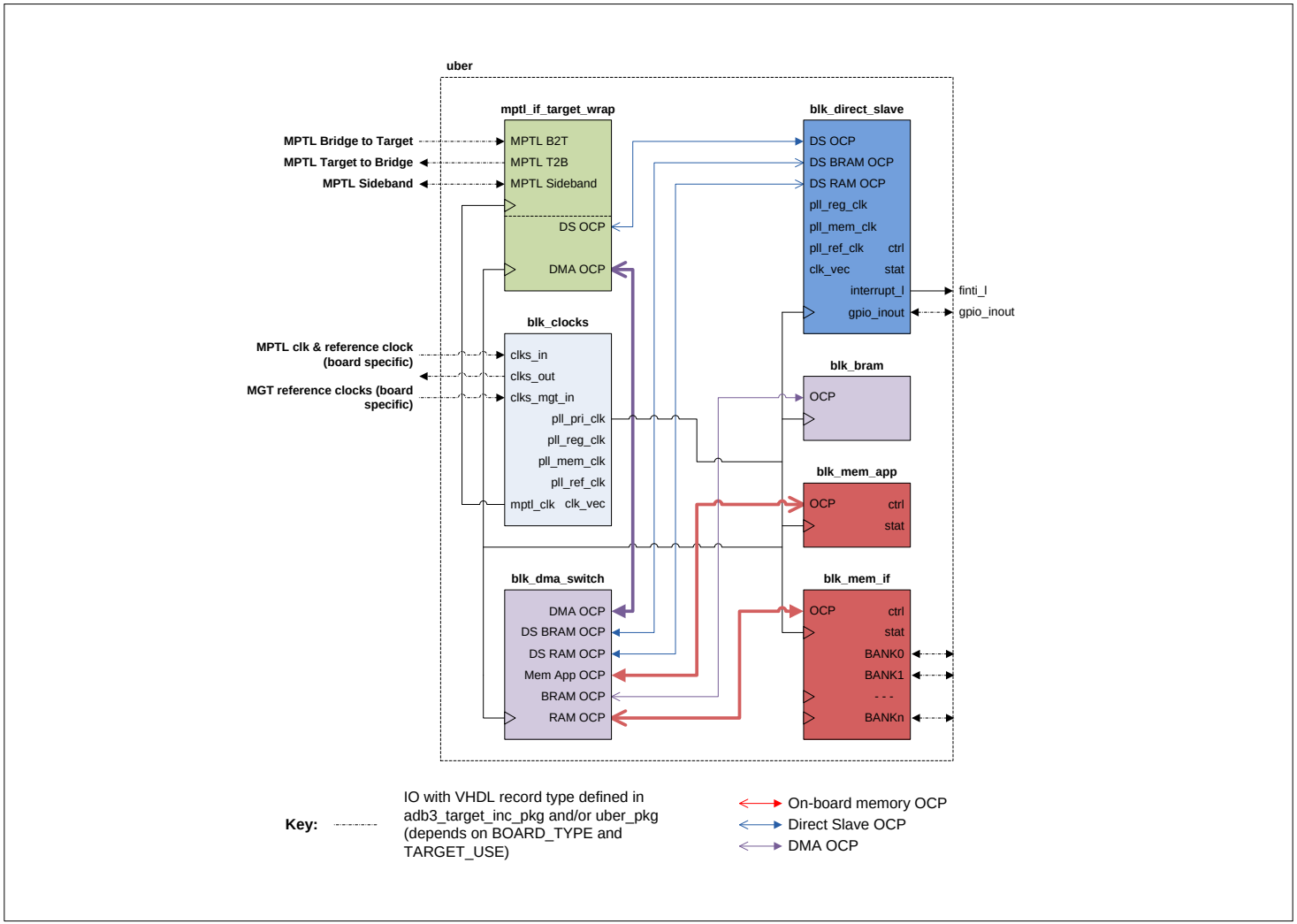

Figure 7: Uber Design Top Level Block Diagram

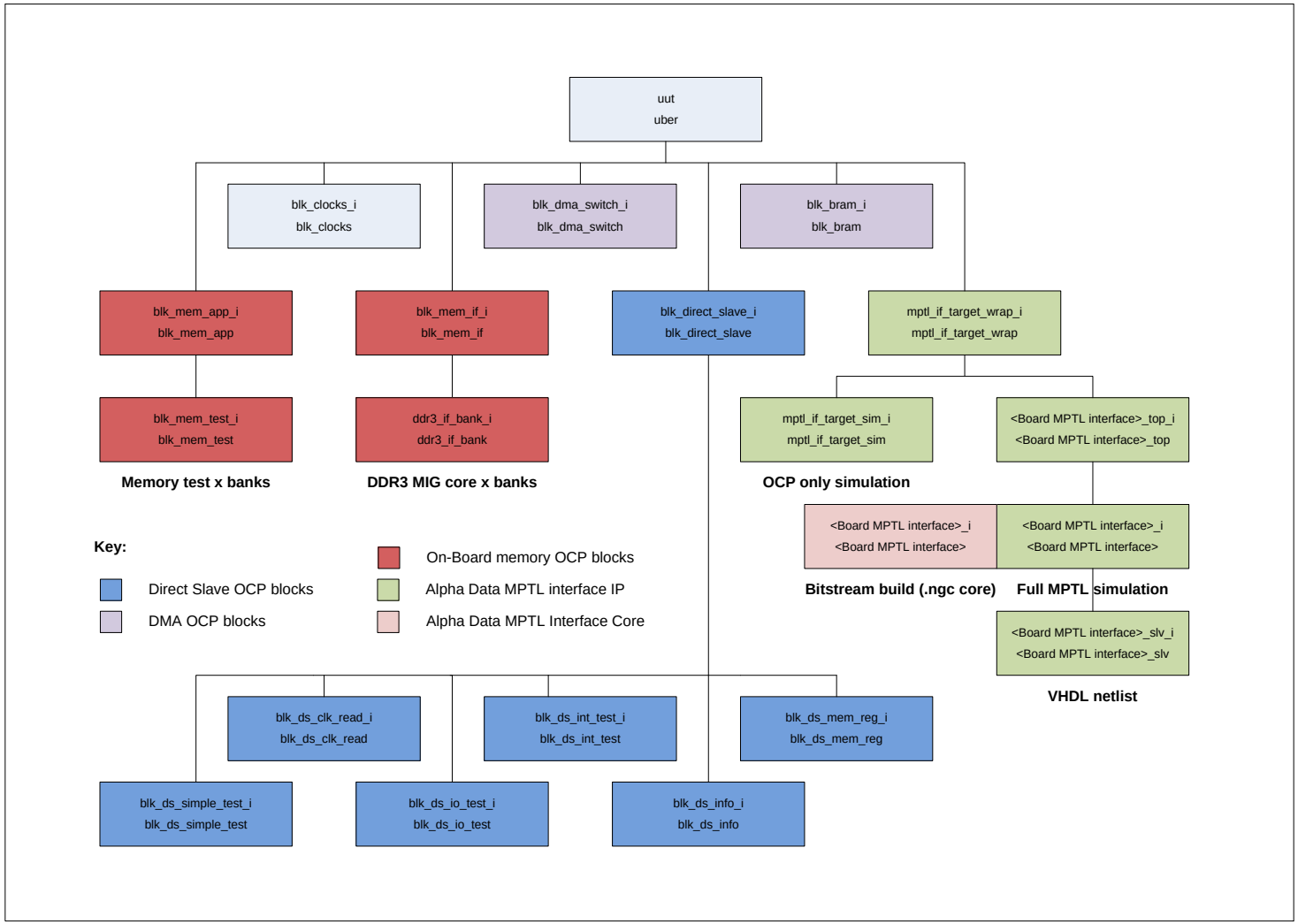

Figure 8: Uber Design Top Level Hierarchy

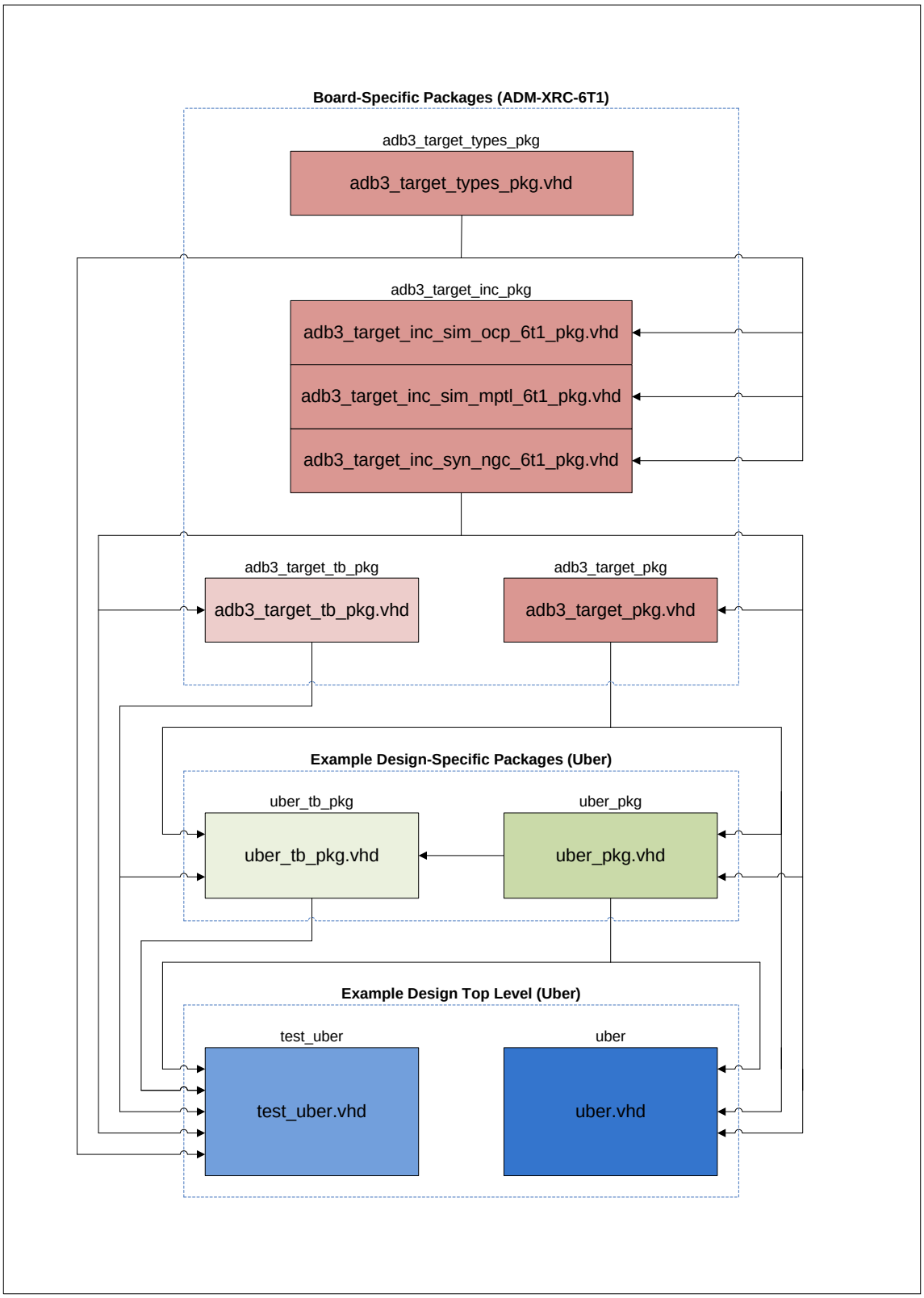

Figure 9: Uber Design Package Dependencies

5.5.4.1 Clock Generation Block

The clock and reset generation block is implemented by **hdl/vhdl/examples/uber/common/blk_clocks.vhd**. It includes the following functional areas:

- **Internal clock generation (MMCM)**
- **Internal reset generation (MMCM)**
- **MPTL 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 10**.

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 10**.

5.5.4.1.3 MPTL Interface Clock Generation

The MPTL interface block requires a differential **mptl_clk** clock input. Its source is dependent on the board selected.

Refer to **Figure 11**.

5.5.4.1.4 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 **uber_pkg** package which is located in **hdl/vhdl/examples/uber/common/**. Refer to **Figure 11**.

5.5.4.1.5 Input Clock Extraction (MGT Sourced)

MGT sourced clocks are input on the **clks_mgt_in** signal of type **clks_mgt_in_t** and are converted from double-ended to single-ended and then buffered. 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. MGT sourced clock support is dependent on the board selected. Type **clks_mgt_in_t** is defined in the **uber_pkg** package which is located in **hdl/vhdl/examples/uber/common/**. Refer to **Figure 11**.

5.5.4.1.6 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 **uber_pkg** package which is located in **hdl/vhdl/examples/uber/common/**. Refer to **Figure 11**.

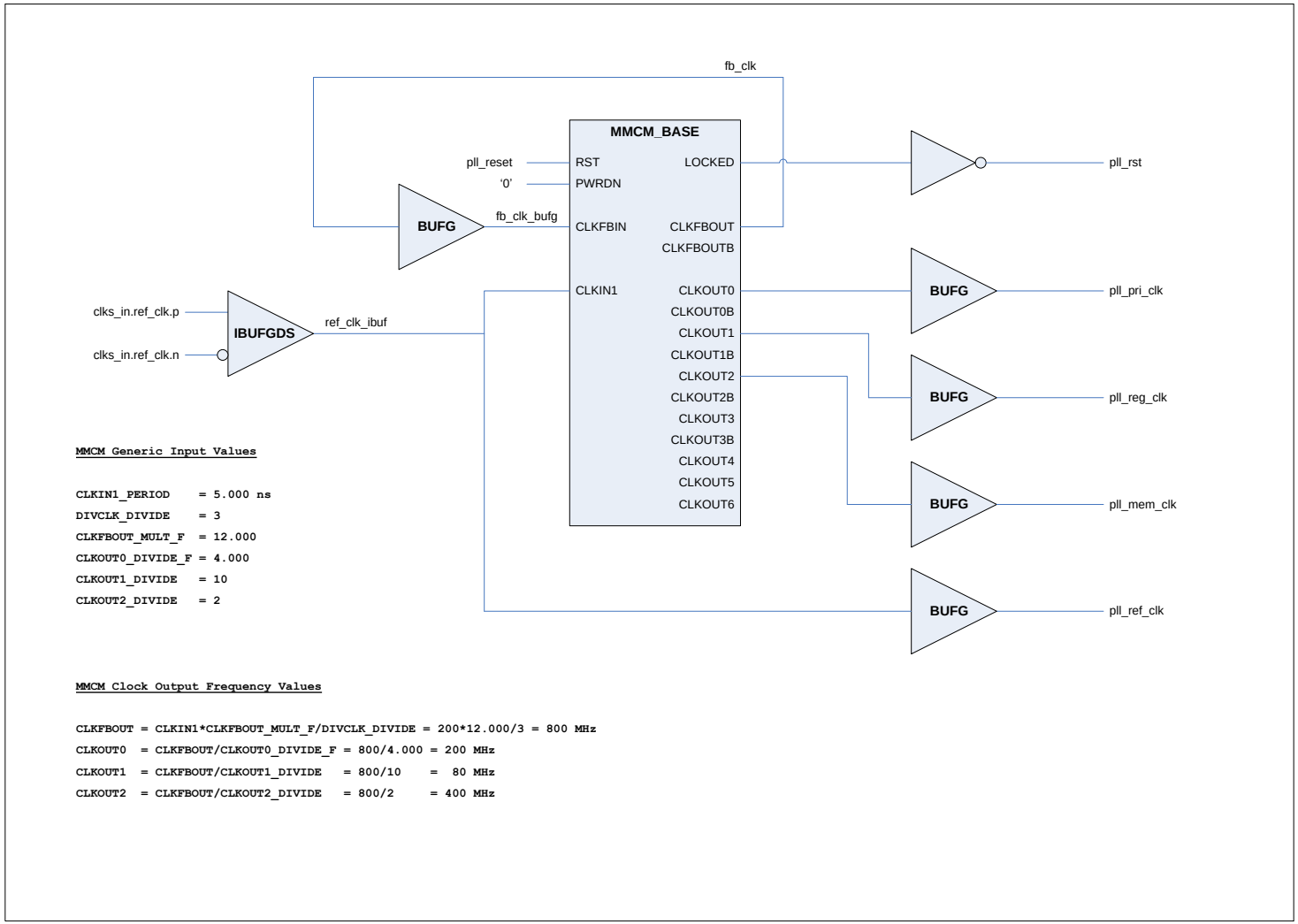

Figure 10: Uber Design Internal Clock Generation (MMCM)

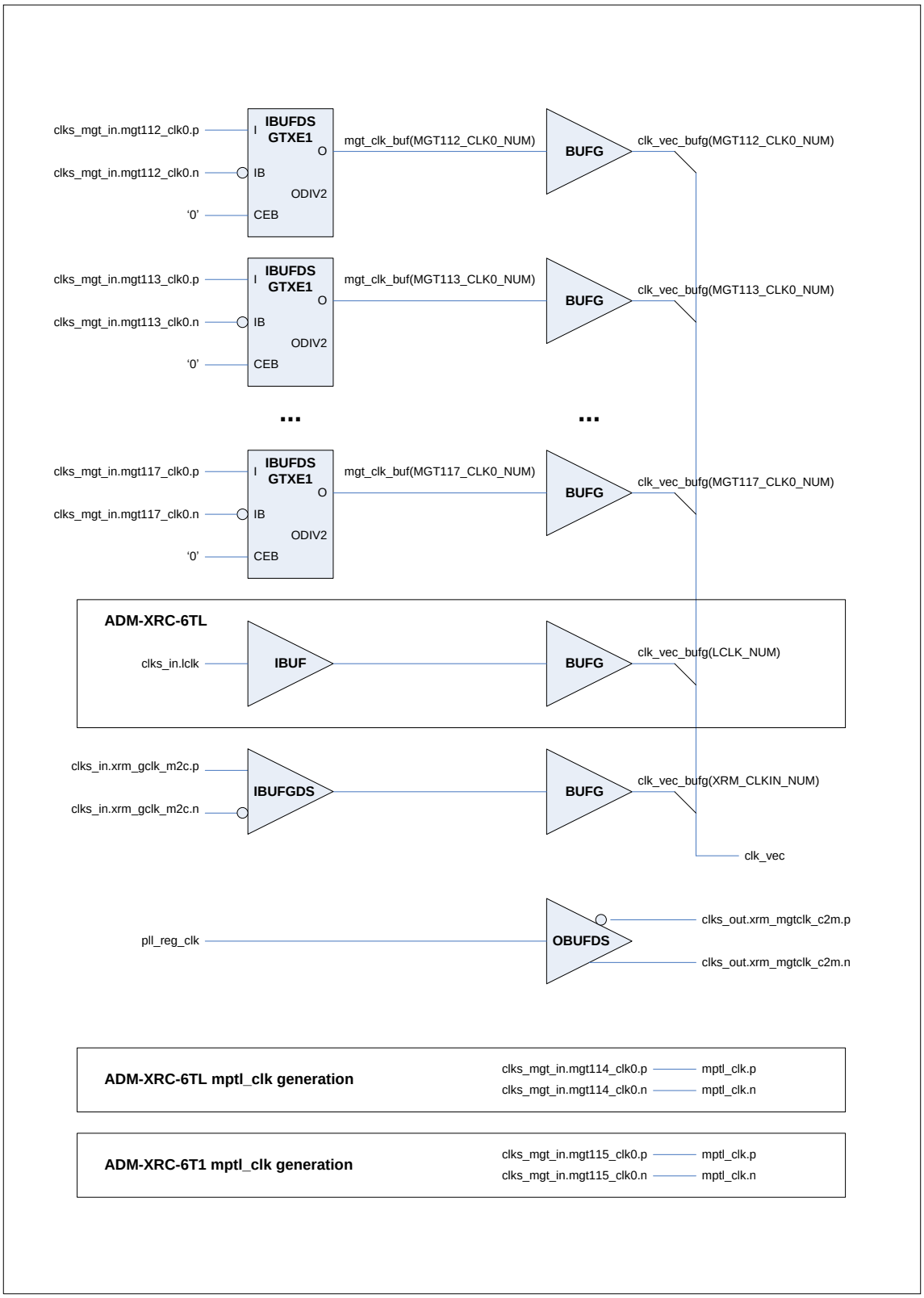

Figure 11: Uber Design Clock Buffering/Extraction

5.5.4.2 Target MPTL Interface

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 (the data channel between the Bridge and Target FPGAs) 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 intitialisation 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 **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 address space supported by the Bridge is smaller than the full ADB3 OCP address space. For the board in use, it is indicated by the **DS_ADDR_WIDTH** constant in the package **adb3_target_inc_pkg**.

> **Note:** The DMA address space supported by the Bridge is smaller than the full ADB3 OCP address space. For the board in use, it is indicated by the **DMA_ADDR_WIDTH** constant in the package **adb3_target_inc_pkg**.

5.5.4.3 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 an OCP address space splitter. 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:

- **Connection between clock domains**, between the **pll_pri_clk** domain and the relatively low frequency **pll_reg_clk** domain.
- **Direct Slave address space splitter block**
- 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 12**.

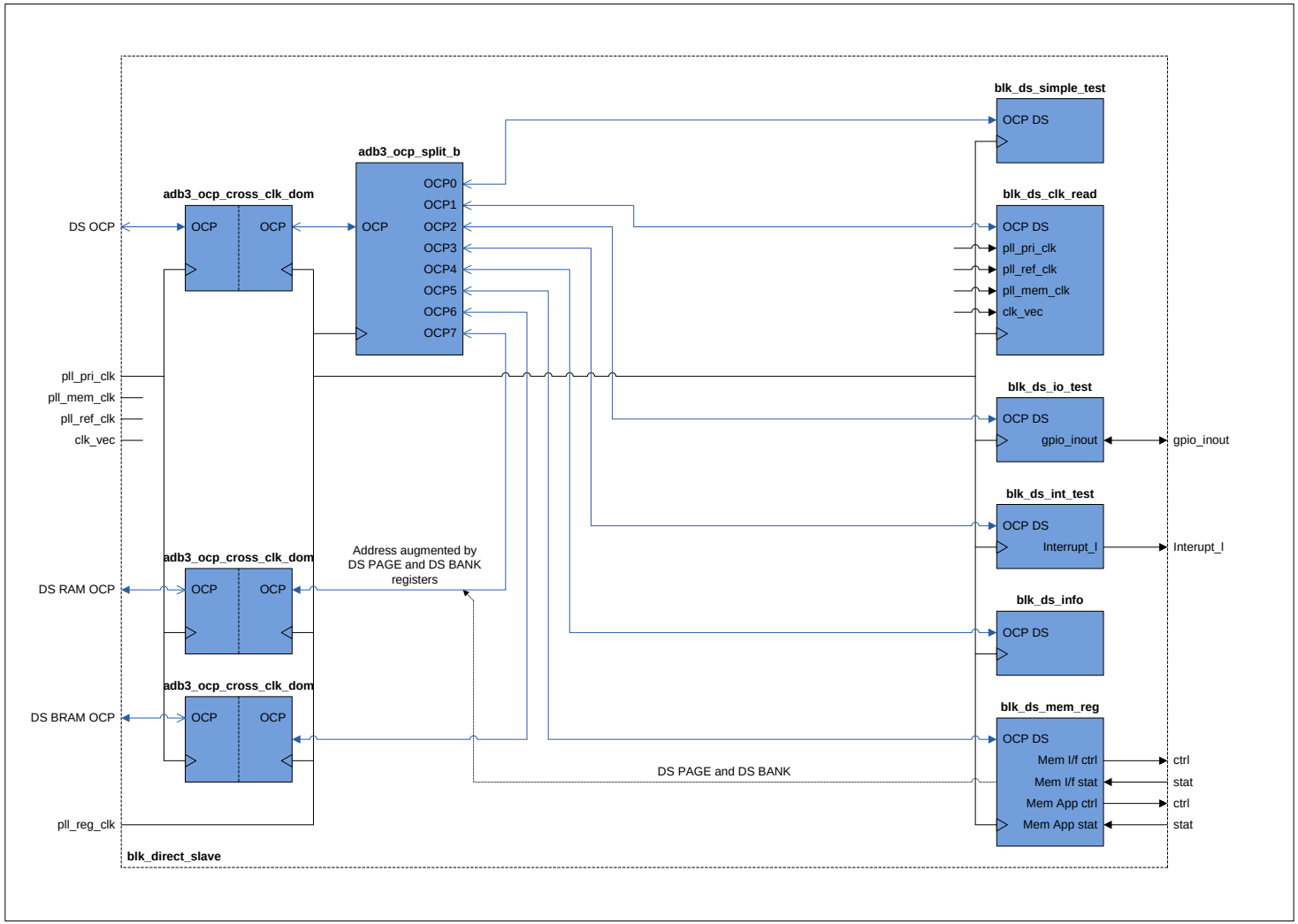

Figure 12: Uber Direct Slave Block Diagram

5.5.4.3.1 OCP Cross-Clock Domain Block

This connects the Direct Slave OCP channel from the higher speed clock domain (**pll_pri_clk**) to the lower speed register clock domain (**pll_reg_clk**), using an instance of the ADB3 OCP library component **adb3_ocp_cross_clk_dom**.

5.5.4.3.2 Direct Slave Address Space Splitter Block

An instance of the ADB3 OCP library component **adb3_ocp_split_b** 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 defined in the **DS_ADDR_RANGE_TABLE** constant in the **uber_pkg** package. The constant **DS_ADDR_RANGE_TABLE** consists of pairs of { base address, mask } for each address range that the splitter recognises. For each range, the lower address is identified by the base address, and the upper address is identified by (base address + mask).

Note: 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. As an optimisation, the **DS_ADDR_RANGE_TABLE** constant in the **uber_pkg** package uses the function **ds_mask_conv** from the package **adb3_target_pkg** to ensure that the topmost DS_ADDR_WIDTH bits of the mask values are all ones, since 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_ADDR_RANGE_TABLE**:

Address range 1 base = ds_base_conv(X"0000C0") = 0x00000000_000000C0
Address range 1 mask = ds_mask_conv(X"00003F") = 0xFFFFFFFF_FFC0003F
=> 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
=> Hits address range 1, since masked incoming OCP address = address range 1 base

Table **Table 8** below shows the information in **DS_ADDR_RANGE_TABLE** and which functional area each index corresponds to:

Address range 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 and status registers**
6	0x080000-0x0FFFFF	**Direct Slave access to BRAM**
7	0x200000-0x3FFFFF	**Direct Slave access to on-board memory**

Table 8: Uber Design Direct Slave Address Map

Note: Reads of undefined areas of the address space return data consisting of 0xDEADC0DE. Writes to undefined areas have no effect.

5.5.4.3.3 Simple Test Register Block

5.5.4.3.3.1 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 library component **adb3_ocp_simple_bus_if** 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 **ds_q** for reads.
2. **ds_w** - Indicates that write data is valid on the signal **ds_d** and write byte enables are valid on the signal **ds_we**.
3. **ds_we** - Byte write enables; qualified by **ds_w**.
4. **ds_d** - Write data; qualified by **ds_w**.
5. **ds_r** - Indicates that valid data must be presented on **ds_q** on the following clock cycle.
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.

5.5.4.3.3.2 Register Description

A set of VHDL processes in uses the signals **ds_a**, **ds_w** 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 9: 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 10: Simple Test Register Block, DATA Register (0x000000)

5.5.4.3.4 Clock Frequency Measurement Register Block

5.5.4.3.4.1 Description

The clock frequency measurement register block is implemented by **hdl/vhdl/examples/uber/common/blk_ds_clk_read.vhd** and performs the following functions:

- Measurement of frequencies of internally generated (MMCM) clocks.
- Measurement of frequencies of externally sourced clocks.

It consists of an instance of **adb3_ocp_simple_bus_if**, multiple instances of the clock frequency measurement block (**blk_clock_freq**), and a set of processes that implement the registers.

The clock frequency measurement component (**blk_clock_freq**) is instantiated for the main OCP clocks of the design, enabling them to be measured:

- pll_ref_clk
- pll_pri_clk
- pll_reg_clk
- pll_mem_clk

blk_clock_freq is also instantiated for each board-dependent clock according to the **CLKS_IN_VALID** constant defined in the **uber_pkg** package.

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_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 **Section 5.5.4.1** 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 **Section 6.8.1.1.3** still holds for every **blk_clock_freq** instance.

5.5.4.3.4.2 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 11: 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) 00001 => pll_pri_clk (Internal) 00010 => pll_ref_clk (Internal) 00011 => pll_mem_clk (Internal) 01100 => lclk (External) 01101 => xrm_clkin (External) 10010 => mgt112_clk0 (External MGT clock) 10100 => mgt113_clk0 (External MGT clock) 10101 => mgt113_clk1 (External MGT clock) 10110 => mgt114_clk0 (External MGT clock) 11000 => mgt115_clk0 (External MGT clock) 11010 => mgt116_clk0 (External MGT clock) 11100 => mgt117_clk0 (External MGT clock)

Table 12: 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 13: 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 14: Clock Frequency Measurement Register Block, FREQ Register (0x000048)

5.5.4.3.5 Interrupt Test Register Block

5.5.4.3.5.1 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 generation.

It consists of an instance of **adb3_ocp_simple_bus_if** and a set of VHDL processes that implement the registers and interrupt generation.

5.5.4.3.5.2 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 15: 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 16: Interrupt Test Register Block, SET Register (0x0000C0)

Bits	Mnemonic	Type	Function
31:0	CLEAR/STAT	R/ W1C	The interrupt output is 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 17: 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 18: 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 output to its inactive state for one cycle of **pll_reg_clk**.

Table 19: 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 20: 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.

5.5.4.3.6 Informational Register Block

5.5.4.3.6.1 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 the design's **.bit** file was built.
- The status of Direct Slave OCP address splitter.
- The base address and size of the BRAM block (**blk_bram**).
- The status of the on-board memory interfaces.

It consists of an instance of **adb3_ocp_simple_bus_if** and a set of VHDL processes that implement the registers.

5.5.4.3.6.2 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

Table 21: 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.

Table 22: 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.

Table 23: 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 24: Informational Register Block, SPLIT Register (0x000148)

Bits	Mnemonic	Type	Function
31:0	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 25: Informational Register Block, BRAM_BASE Register (0x00014C)

Bits	Mnemonic	Type	Function
31:0	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 26: Informational Register Block, BRAM_MASK Register (0x000150)

Bits	Mnemonic	Type	Function
31:0	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 27: Informational Register Block, MEM_BASE Register (0x000154)

Bits	Mnemonic	Type	Function
31:0	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 28: 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 29: Informational Register Block, MEM_BANKS Register (0x00015C)

5.5.4.3.7 GPIO Test Register Block

5.5.4.3.7.1 Description

The GPIO test register block is implemented by **hdl/vhdl/examples/uber/common/blk_ds_io_test.vhd** and 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** 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.

5.5.4.3.7.2 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

Table 30: GPIO Test Register Block Address Map (continued on next page)

Name	Address
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 30: GPIO Test Register Block Address Map

Bits	Mnemonic	Type	Function
31:16	DA_P_OUT	M	Controls/indicates logic levels driven on the **da_p(15:0)** XRM GPIO pins.
15:0	DA_N_OUT	M	Controls/indicates logic levels driven on the **da_n(15:0)** XRM GPIO pins.

Table 31: GPIO Test Register Block, XRM_GPIO_DA_DATAO Register (0x000200)

Bits	Mnemonic	Type	Function
31:16	DA_P_IN	RO	Indicates the actual logic levels on the **da_p(15:0)** XRM GPIO pins.
15:0	DA_N_IN	RO	Indicates the actual logic levels on the **da_n(15:0)** XRM GPIO pins.

Table 32: GPIO Test Register Block, XRM_GPIO_DA_DATAI Register (0x000204)

Bits	Mnemonic	Type	Function
31:16	DA_P_TRI	M	Controls/indicates the 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 the tristate enables for the **da_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 33: 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 the **db_p(15:0)** XRM GPIO pins.
15:0	DB_N_OUT	M	Controls/indicates logic levels driven on the **db_n(15:0)** XRM GPIO pins.

Table 34: GPIO Test Register Block, XRM_GPIO_DB_DATAO Register (0x00020C)

Bits	Mnemonic	Type	Function
31:16	DB_P_IN	RO	Indicates the actual logic levels on the **db_p(15:0)** XRM GPIO pins.
15:0	DB_N_IN	RO	Indicates the actual logic levels on the **db_n(15:0)** XRM GPIO pins.

Table 35: GPIO Test Register Block, XRM_GPIO_DB_DATAI Register (0x000210)

Bits	Mnemonic	Type	Function
31:16	DB_P_TRI	M	Controls/indicates the tristate enables for the **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 the tristate enables for the **db_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 36: 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 the **dc_p(15:0)** XRM GPIO pins.
15:0	DC_N_OUT	M	Controls/indicates logic levels driven on the **dc_n(15:0)** XRM GPIO pins.

Table 37: GPIO Test Register Block, XRM_GPIO_DC_DATAO Register (0x000218)

Bits	Mnemonic	Type	Function
31:16	DC_P_IN	RO	Indicates the actual logic levels on the **dc_p(15:0)** XRM GPIO pins.
15:0	DC_N_IN	RO	Indicates the actual logic levels on the **dc_n(15:0)** XRM GPIO pins.

Table 38: GPIO Test Register Block, XRM_GPIO_DC_DATAI Register (0x00021C)

Bits	Mnemonic	Type	Function
31:16	DC_P_TRI	M	Controls/indicates the tristate enables for the **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 the tristate enables for the **dc_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 39: 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 the **dd_p(15:0)** XRM GPIO pins.
15:0	DD_N_OUT	M	Controls/indicates logic levels driven on the **dd_n(15:0)** XRM GPIO pins.

Table 40: GPIO Test Register Block, XRM_GPIO_DD_DATAO Register (0x000224)

Bits	Mnemonic	Type	Function
31:16	DD_P_IN	RO	Indicates the actual logic levels on the **dd_p(15:0)** XRM GPIO pins.
15:0	DD_N_IN	RO	Indicates the actual logic levels on the **dd_n(15:0)** XRM GPIO pins.

Table 41: GPIO Test Register Block, XRM_GPIO_DD_DATAI Register (0x000228)

Bits	Mnemonic	Type	Function
31:16	DD_P_TRI	M	Controls/indicates the tristate enables for the **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 the tristate enables for the **dd_n(15:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 42: 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 the logic level driven on the **dd_cc_p** XRM GPIO pin.
16	DD_CC_N_OUT	M	Controls/indicates the logic level driven on the **dd_cc_n** XRM GPIO pin.
15	DC_CC_P_OUT	M	Controls/indicates the logic level driven on the **dc_cc_p** XRM GPIO pin.
14	DC_CC_N_OUT	M	Controls/indicates the logic level driven on the **dc_cc_n** XRM GPIO pin.
13	DB_CC_P_OUT	M	Controls/indicates the logic level driven on the **db_cc_p** XRM GPIO pin.
12	DB_CC_N_OUT	M	Controls/indicates the logic level driven on the **db_cc_n** XRM GPIO pin.
11	DA_CC_P_OUT	M	Controls/indicates the logic level driven on the **da_cc_p** XRM GPIO pin.
10	DA_CC_N_OUT	M	Controls/indicates the logic level driven on the **da_cc_n** XRM GPIO pin.
9:6	SD_OUT	M	Controls/indicates the logic levels driven on the **sd(3:0)** XRM GPIO pins.
5:4	SC_OUT	M	Controls/indicates the logic levels driven on the **sc(1:0)** XRM GPIO pins.
3:2	SB_OUT	M	Controls/indicates the logic levels driven on the **sb(1:0)** XRM GPIO pins.
1:0	SA_OUT	M	Controls/indicates the logic levels driven on the **sa(1:0)** XRM GPIO pins.

Table 43: GPIO Test Register Block, XRM_GPIO_CS_DATAO Register (0x000230)

Bits	Mnemonic	Type	Function
31:18			(Reserved)
17	DD_CC_P_IN	RO	Indicates the actual logic level on the **dd_cc_p** XRM GPIO pin.
16	DD_CC_N_IN	RO	Indicates the actual logic level on the **dd_cc_n** XRM GPIO pin.
15	DC_CC_P_IN	RO	Indicates the actual logic level on the **dc_cc_p** XRM GPIO pin.
14	DC_CC_N_IN	RO	Indicates the actual logic level on the **dc_cc_n** XRM GPIO pin.
13	DB_CC_P_IN	RO	Indicates the actual logic level on the **db_cc_p** XRM GPIO pin.
12	DB_CC_N_IN	RO	Indicates the actual logic level on the **db_cc_n** XRM GPIO pin.
11	DA_CC_P_IN	RO	Indicates the actual logic level on the **da_cc_p** XRM GPIO pin.
10	DA_CC_N_IN	RO	Indicates the actual logic level on the **da_cc_n** XRM GPIO pin.
9:6	SD_IN	RO	Indicates the actual logic levels on the **sd(3:0)** XRM GPIO pins.
5:4	SC_IN	RO	Indicates the actual logic levels on the **sc(1:0)** XRM GPIO pins.
3:2	SB_IN	RO	Indicates the actual logic levels on the **sb(1:0)** XRM GPIO pins.
1:0	SA_IN	RO	Indicates the actual logic levels on the **sa(1:0)** XRM GPIO pins.

Table 44: 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 the tristate enable for the **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 the tristate enable for the **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 the tristate enable for the **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 the tristate enable for the **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 the tristate enable for the **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 the tristate enable for the **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 the tristate enable for the **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 the tristate enable for the **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 the tristate enables for the **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 the tristate enables for the **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 the tristate enables for the **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 the tristate enables for the **sa(1:0)** XRM GPIO pins. If a bit is 1, the corresponding pin is tristated (high-impedance).

Table 45: 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 the **gpio_p (PN4_GPIO_WIDTH-1:0)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** is less than 32, the top **(32-PN4_GPIO_WIDTH)** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**.

Table 46: GPIO Test Register Block, PN4_GPIO_P_DATAO Register (0x00023C)

Bits	Mnemonic	Type	Function
31:0	P_DATAI	RO	Indicates the actual logic levels on the **gpio_p(PN4_GPIO_WIDTH-1:0)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** is less than 32, the top **(32-PN4_GPIO_WIDTH)** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**.

Table 47: GPIO Test Register Block, PN4_GPIO_P_DATAI Register (0x000240)

Bits	Mnemonic	Type	Function
31:0	P_TRI	M	Controls/indicates the tristate enables for the **gpio_p (PN4_GPIO_WIDTH-1:0)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** is less than 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 the package **adb3_target_inc_pkg**.

Table 48: 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 the **gpio_n (PN4_GPIO_WIDTH-1:0)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** is less than 32, the top **(32-PN4_GPIO_WIDTH)** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**.

Table 49: GPIO Test Register Block, PN4_GPIO_N_DATAO Register (0x000248)

Bits	Mnemonic	Type	Function
31:0	N_DATAI	RO	Indicates the actual logic levels on the **gpio_n(PN4_GPIO_WIDTH-1:0)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** is less than 32, the top **(32-PN4_GPIO_WIDTH)** bits of this register are unused. The constant **PN4_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**.

Table 50: GPIO Test Register Block, PN4_GPIO_N_DATAI Register (0x00024C)

Bits	Mnemonic	Type	Function
31:0	N_TRI	M	Controls/indicates the tristate enables for the **gpio_n (PN4_GPIO_WIDTH-1:0)** Pn4 GPIO pins. If **PN4_GPIO_WIDTH** is less than 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 the package **adb3_target_inc_pkg**.

Table 51: GPIO Test Register Block, PN4_GPIO_N_TRI Register (0x000250)

Bits	Mnemonic	Type	Function
31:0	MS_DATAO	M	If **PN6_GPIO_WIDTH** is less than or equal to 32, this register is ignored. If **PN6_GPIO_WIDTH** is at least 32, this register controls/indicates logic levels driven on the **gpio(PN6_GPIO_WIDTH:32)** PN6 GPIO pins. If **PN6_GPIO_WIDTH** is less than 64, the top **(64-PN6_GPIO_WIDTH)** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**

Table 52: GPIO Test Register Block, PN6_GPIO_MS_DATAO Register (0x000254)

Bits	Mnemonic	Type	Function
31:0	MS_DATAI	RO	If **PN6_GPIO_WIDTH** is less than or equal to 32, this register is ignored. If **PN6_GPIO_WIDTH** is at least 32, this register indicates the actual logic levels on the **gpio(PN6_GPIO_WIDTH:32)** PN6 GPIO pins. If **PN6_GPIO_WIDTH** is less than 64, the top **(64-PN6_GPIO_WIDTH)** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**

Table 53: GPIO Test Register Block, PN6_GPIO_MS_DATAI Register (0x000258)

Bits	Mnemonic	Type	Function
31:0	MS_TRI	M	If **PN6_GPIO_WIDTH** is less than or equal to 32, this register is ignored. If **PN6_GPIO_WIDTH** is at least 32, this register controls/indicates the tristate enables for the **gpio(PN6_GPIO_WIDTH:32)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** is less than 64, the top **(64-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 the package **adb3_target_inc_pkg**

Table 54: GPIO Test Register Block, PN6_GPIO_MS_TRI Register (0x00025C)

Bits	Mnemonic	Type	Function
31:0	LS_DATAO	M	If **PN6_GPIO_WIDTH** is at least 32, this register controls/indicates logic levels driven on the **gpio(31:0)** Pn6 GPIO pins. If **PN6_GPIO_WIDTH** is less than 32, this register controls/indicates logic levels driven on the **gpio(PN6_GPIO_WIDTH-1:0)** Pn6 GPIO pins, and the top **(32-PN6_GPIO_WIDTH)** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**

Table 55: GPIO Test Register Block, PN6_GPIO_LS_DATAO Register (0x000260)

Bits	Mnemonic	Type	Function
31:0	LS_DATAI	RO	If **PN6_GPIO_WIDTH** is at least 32, this register indicates the actual logic levels on the **gpio(31:0)** Pn6 GPIO pins If **PN6_GPIO_WIDTH** is less than 32, this register indicates the actual logic levels on the **gpio(PN6_GPIO_WIDTH-1:0)** Pn6 GPIO pins, and the top **(32-PN6_GPIO_WIDTH)** bits of this register are unused. The constant **PN6_GPIO_WIDTH** is defined in the package **adb3_target_inc_pkg**

Table 56: GPIO Test Register Block, PN6_GPIO_LS_DATAI Register (0x000264)

Bits	Mnemonic	Type	Function
31:0	LS_TRI	M	If **PN6_GPIO_WIDTH** is at least 32, this register controls/indicates the tristate enables for the **gpio(31:0)** Pn6 GPIO pins If **PN6_GPIO_WIDTH** is less than 32, this register controls/indicates the tristate enables of the **gpio(PN6_GPIO_WIDTH-1:0)** Pn6 GPIO pins, and 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 the package **adb3_target_inc_pkg**

Table 57: GPIO Test Register Block, PN6_GPIO_LS_TRI Register (0x000268)

5.5.4.3.8 On-Board Memory Register Block

5.5.4.3.8.1 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** and a set of VHDL processes that implement the memory control and status registers.

5.5.4.3.8.2 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 ...

Table 58: On-Board Memory Register Block Address Map (continued on next page)

Name	Address
BANK0_LENGTH BANK1_LENGTH ...	0x000328 0x000348 ...
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_DDR3_ERR BANK1_DDR3_ERR ...	0x00033C 0x00035C ...

Table 58: 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 **Table 70** for an explanation of how this register affects access to on-board memory.

Table 59: On-Board Memory Register Block, DS_BANK Register (0x000300)

Bits	Mnemonic	Type	Function
31:0	DS_PAGE	M	Controls which page of 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_DDR3** constant defined in **blk_direct_slave**. Bits 31: PAGE_ADDR_WIDTH_DDR3 are ignored. Refer to **Table 70** for an explanation of how this register affects access to on-board memory.

Table 60: 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 61: 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 (in 16-byte words) for the FPGA-driven on-board memory test for bank x.

Table 62: 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 63: 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_DDR3**. 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	DDR3_16_BYTE_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 by the constant **DDR3_16_BYTE_ADDR_WIDTH**. 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 by the constant **DDR3_BYTE_ADDR_WIDTH**. This is defined in the package **adb3_target_inc_pkg**.

Table 64: 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 65: 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 (in 16-byte words) 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 66: 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 **Section 6.1** for a description.

Table 67: 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 **Section 6.1** for a description.

Table 68: On-Board Memory Register Block, BANKx_DDR3_ERR Register (0x00033C, 0x00035C, ...)

5.5.4.3.9 Direct Slave BRAM Access Block

5.5.4.3.9.1 Description

This block creates a window in the Direct Slave OCP address space through which the **BRAM block** can be read and written. To do accomplish this, secondary port 6 of the **Direct Slave address space splitter** (in the **pll_reg_clk** domain) is connected to the **OCP switching block** (in the **pll_pri_clk** domain) by an instance of the component **adb3_ocp_cross_clk_dom**.

5.5.4.3.9.2 Direct Slave BRAM Access Window

The BRAM access window appears in the Direct Slave OCP address space as follows:

Name	Address
BRAM access window	0x080000-0x0FFFFF

Table 69: Direct Slave BRAM Access Window

5.5.4.3.10 Direct Slave On-Board Memory Access Block

5.5.4.3.10.1 Description

This block creates a window in the Direct Slave OCP address through which the on-board memory interfaces can be read and written. To do accomplish this, secondary port 7 of the **Direct Slave address space splitter** (in the **pll_reg_clk** domain) is connected to the **OCP switching block** (in the **pll_pri_clk** domain) by an instance of the component **adb3_ocp_cross_clk_dom**.

Since the Direct Slave channel has useable OCP address space of 4 MiB, which is not sufficient to access all banks of on-board memory, Direct Slave OCP addresses are augmented by the values of the **DS_BANK** and **DS_PAGE** registers as described in **Table 59** and **Table 60** respectively. The augmented OCP memory address, which is generated in the **pll_reg_clk** domain, is then connected to the **pll_pri_clk** domain by an instance of the component **adb3_ocp_cross_clk_dom**.

5.5.4.3.10.2 Direct Slave On-Board Memory Access Window

In the Direct Slave OCP address space, all on-board memory banks are accessed through a 2 MiB address window. When a Direct Slave OCP address hits this window, it is augmented by the **DS_BANK** and **DS_PAGE** registers in order to be able to access all banks of 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 70: 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 71**.

5.5.4.4 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 13**.

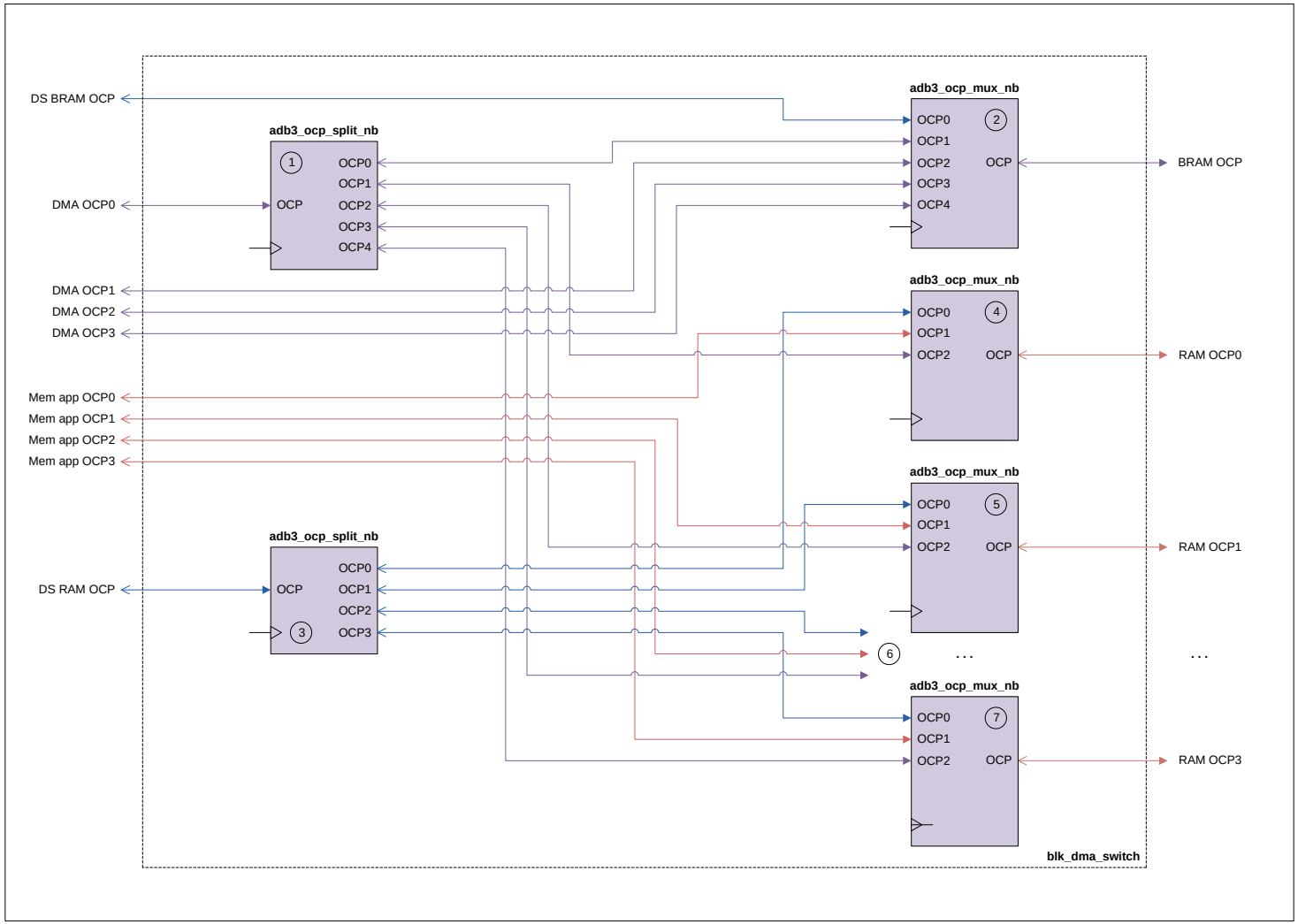

Figure 13: Uber OCP Switching Block

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

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

Referring to item 1 in **Figure 13**, 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 71** below. The address map is defined by the the constant **DS_DDR3_ADDR_RANGE_TABLE** in the **uber_pkg** package.

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

Table 71: Uber Design Direct Slave On-Board Memory Address Map

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 BRAM OCP Multiplexor Block

Referring to item 2 in **Figure 13**, this instance of **adb3_ocp_mux_nb** multiplexes all OCP channels which require to be connected to the **BRAM block**:

- Direct Slave BRAM access OCP channel
- DMA channel 0 splitter secondary OCP channel with index 0
- The other DMA OCP channels

5.5.4.4.3 DMA Channel 0 OCP Address Space Splitter Block

Referring to item 3 in **Figure 13**, this instance of **adb3_ocp_split_nb** splits DMA OCP channel 0 into multiple secondary OCP channels according to the address map in **Table 72**. The address map is defined by the constant **DMA_ADDR_RANGE_TABLE** in the **uber_pkg** package.

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

Table 72: Uber Design DMA Channel 0 Address Map

Note: Reads of undefined areas of the address space return data consisting of 0xDEADC0DE. Writes to undefined areas have no effect.

5.5.4.4.4 On-Board Memory Bank OCP Multiplexors

Items 4, 5, 6 and 7 in **Figure 13** 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 splitter OCP channels with indices 0 to 3
- Memory application OCP channels (FPGA-driven memory test)
- DMA OCP channel 0 splitter indices 1 to 4

5.5.4.5 BRAM Block

This block is implemented by **hdl/vhdl/examples/uber/common/blk_bram.vhd** and contains a RAM composed of BlockRAM primitives that can be read and written via the OCP switching block (see **Section 5.5.4.4**) by:

- The Direct Slave OCP channel, via the **BRAM access window**.
- DMA channel 0, according to the address map in **Table 72**.
- Any other DMA channel, where the BRAM block is aliased throughout the entire OCP address space.

Figure 14 shows the BRAM block connected to the OCP switching block:

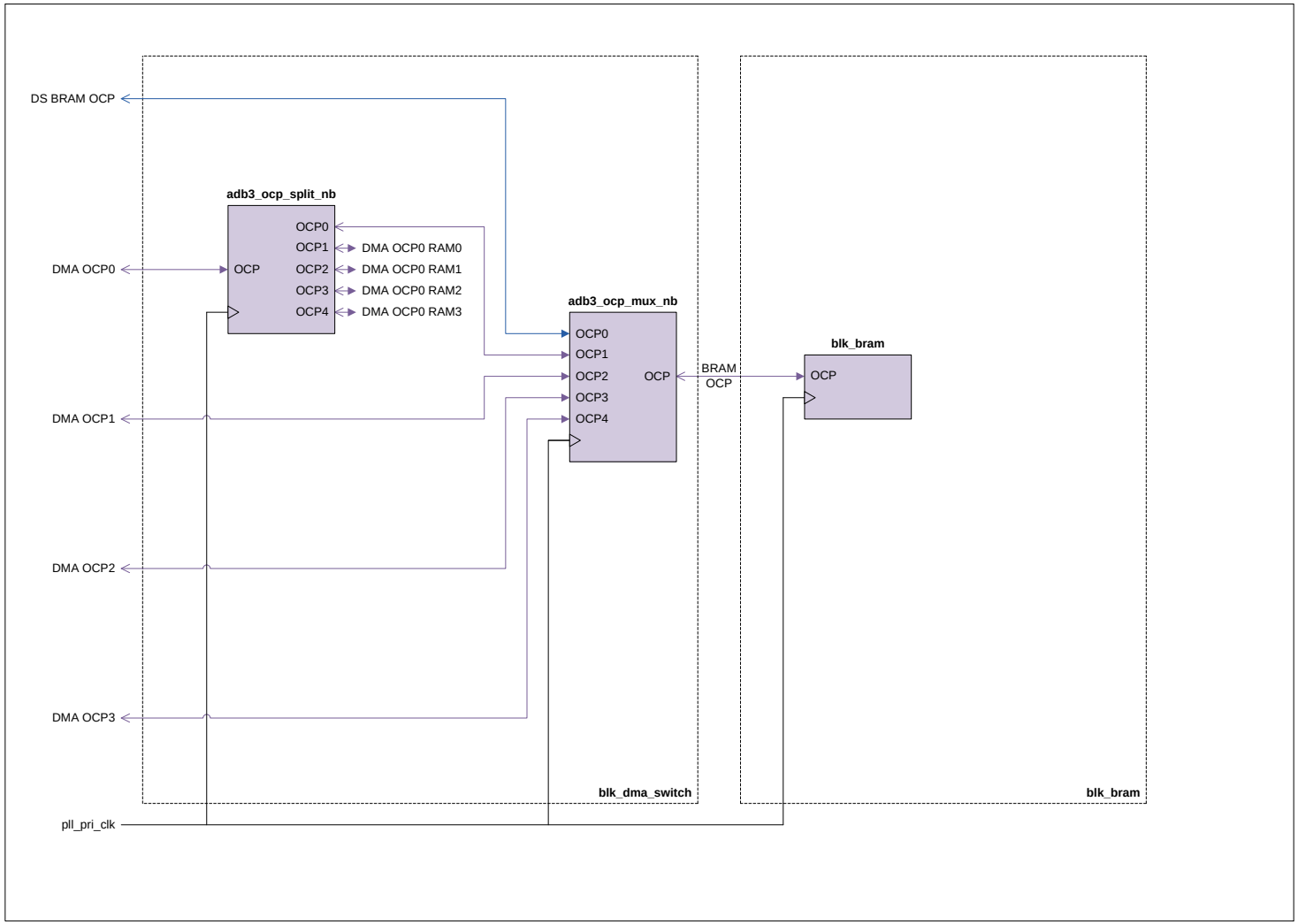

Figure 14: Uber BRAM Block Diagram

An instance of **adb3_ocp_simple_bus_if** could have been used, together with BlockRAM primitives, to implement this block. Although this would work, read and write performance would be unsatisfactory since **adb3_ocp_simple_bus_if** is not designed for high data throughput.

Therefore, instead of the above arrangement, a state machine provides the OCP interface, implementing what is in effect a high-throughput version of **adb3_ocp_simple_bus_if**. A wrapper for a Virtex-6 BlockRAM called **bram_single_wrap** and implemented by **hdl/vhdl/examples/uber/common/bram_single_wrap.vhd** is instantiated multiple times to create a 512 KiB RAM. A shallow FIFO buffers data read from this RAM, partly to mitigate the effect of BlockRAM read latency on throughput and partly to make the implementation of the OCP interface state machine simpler and faster.

5.5.4.6 On-Board Memory Interface Block

This block is implemented by **hdl/vhdl/examples/uber/common/blk_mem_if.vhd** and instantiates a memory interface for each bank of on-board memory. This enables the following agents to read and write on-board memory banks via the OCP switching block (see **Section 5.5.4.4**):

- The Direct Slave OCP channel, via the **on-board memory access window**.
- DMA channel 0, according to the address map in **Table 72**.
- The memory application (see **Section 5.5.4.7**).

The number of memory interfaces and the number of DDR3 SDRAM memory interfaces are defined by the **MEM_BANKS** and **DDR3_BANKS** constants respectively in the **adb3_target_inc_pkg** package. For the ADM-XRC-6TL and ADM-XRC-6T1, which are fitted only with DDR3 SDRAM memory, **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 14 shows the on-board memory interface block connected to the OCP switching block:

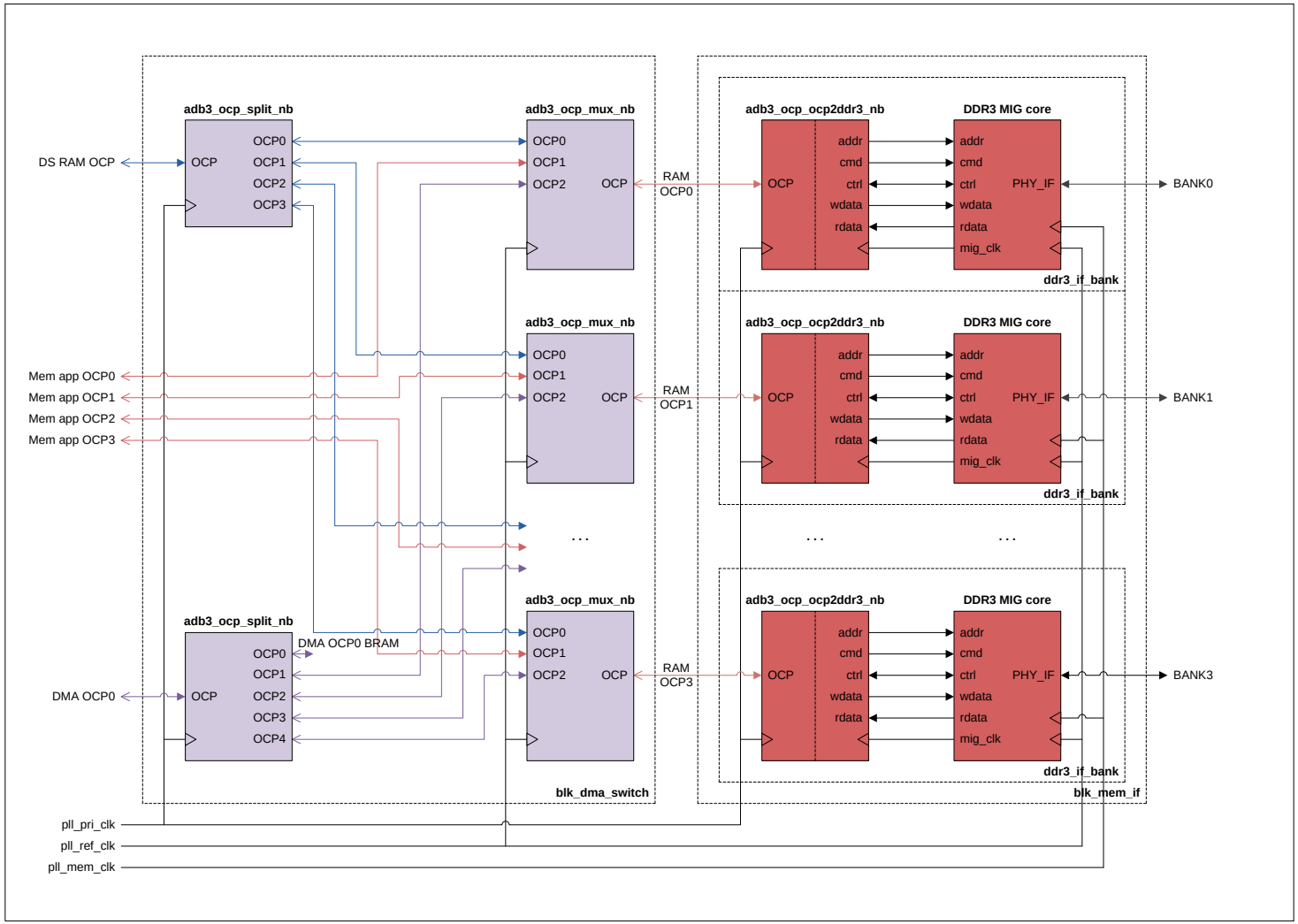

Figure 15: Uber Memory Interface Block Diagram

For each bank of on-board DDR3 SDRAM memory, this block instantiates a **ddr3_mem_if_bank** component (see **Section 6.5**). In addition, this block contains some logic common to all banks of memory such as reset logic and an **IDELAYCTRL** instance.

The status of the memory interfaces, which indicates whether or not training and initialisation was successful for each bank, can be determined via the registers defined in **Section 5.5.4.3.8**.

5.5.4.7 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 module (**blk_mem_test**) 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. Refer to **Section 6.6** for a functional description of **blk_mem_test**.

Since 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.

5.5.4.8 ChipScope™ Connection Block (optional)

This block optionally instantiates logic that enables several ADB3 OCP channels to be monitored using Xilinx™ ChipScope™. It is implemented by **hdl/vhdl/common/ChipScope™/blk_ChipScope™.vhd**.

When the **CHIPSCOPE_ON** constant in **hdl/vhdl/common/ChipScope™/uber.vhd** is **true**, ChipScope™ logic is instantiated.

Note: For simulation, a dummy version of this block is used, implemented by **hdl/vhdl/common/ChipScope™/blk_chipscope_sim.vhd**. Refer to **Section 6.9** for a functional description.

> **Note:** Prior to the initial bitstream build of a design using a Xilinx™ ChipScope™ interface, the ChipScope™ core **.ngc** files must be generated. Refer to **Section 6.9** for a description of the procedure.

5.5.4.9 Design Package

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

Defininitions are as follows:

Top level signal types

- **clks_in_t**. A record type containing non-MGT based input clock elements.
- **clks_mgt_in_t**. A record type containing MGT based input clock elements.
- **clks_out_t**. A record type containing output clock elements.
- **xrm_gpio_t**. A record type containing XRM bi-directional GPIO elements.
- **pn4_gpio_t**. A record type containing PN4 bi-directional GPIO elements.
- **pn6_gpio_t**. A record type containing PN6 bi-directional GPIO elements.
- **gpio_inout_t**. A record type containing all bi-directional GPIO elements.

Direct slave interface memory map constants

- Direct slave memory map sections base address constants (type **adb3_ocp_addr_s**).

- Direct slave memory map sections mask address constants (type **adb3_ocp_addr_s**).
- Direct slave memory map sections range constants (type **addr_range_t**).
- **DS_ADDR_RANGE_TABLE**. Direct slave memory map address range table constant (type **addr_range_table_t**).
- Memory map sections register offsets (type **natural**).
- Memory map sections register offset addresses (type **adb3_ocp_addr_s**).

DMA interface memory map constants

- DMA memory map sections base address constants (type **adb3_ocp_addr_s**).
- DMA memory map sections mask address constants (type **adb3_ocp_addr_s**).
- DMA memory map sections range constants (type **addr_range_t**).
- **DMA_ADDR_RANGE_TABLE**. DMA memory map address range table constant (type **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 clocks/external clock inputs.
- Assignment of an index number (type **clk_vec_range_t**) to all internal clocks/external clock inputs.
- **CLKS_IN_VALID** Clock validity vector (type **clk_vec_t**) for all external clock inputs.

Memory interface 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 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 testbench for the **Uber** example FPGA design is implemented by **hdl/vhdl/examples/uber/common/test_uber.vhd**. Figure **Figure 16** shows the testbench, with the top level of **uber** embedded in it.

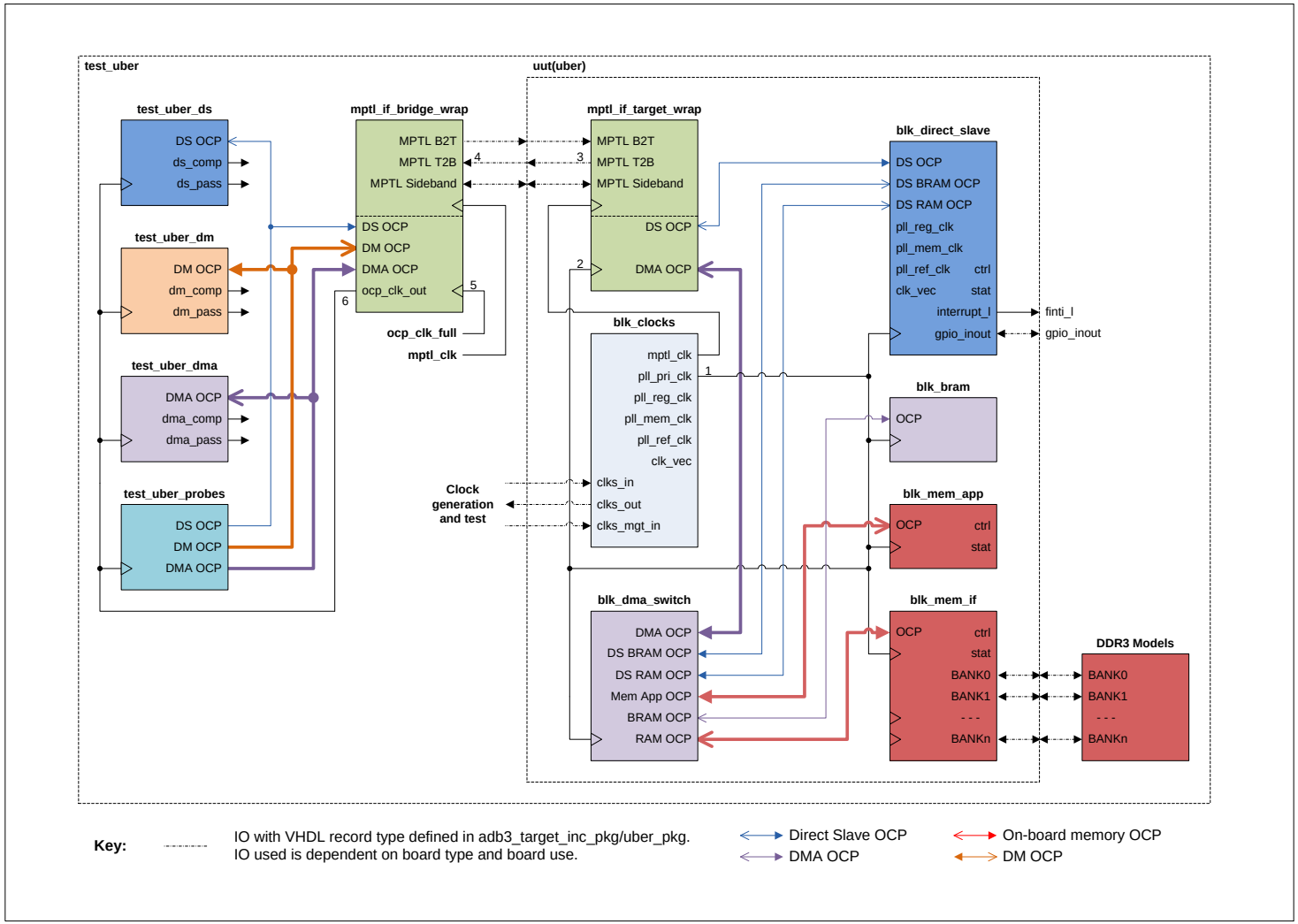

Figure 16: Uber Design Testbench And Top Level Block Diagram

The **Uber** example FPGA design testbench consists of the following functions:

- **Clock generation** for the testbench and the Unit Under Test (UUT).
- The Unit Under Test (UUT), which is the one and only instance of the top-level **uber** block.
- **The Bridge MPTL interface block**, using an instance of **mptl_if_bridge_wrap**.
- **OCP channel probes**, using instances of **adb3_ocp_transaction_probe**.
- **Stimulus Generation and Verification**.
- Instances of the DDR3 SDRAM simulation model (**ddr3_sdram**).

The hierarchical structure of the testbench is shown in **Figure 17**:

The testbench includes the following packages:

- **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)**
- **Memory interface library package (mem_if_pkg)**
- **Design package (uber_pkg)**
- **Testbench package (uber_tb_pkg)**
- **DDR3 SDRAM model package (ddr3_sdram_pkg)**

Figure 9 shows the design package dependencies.

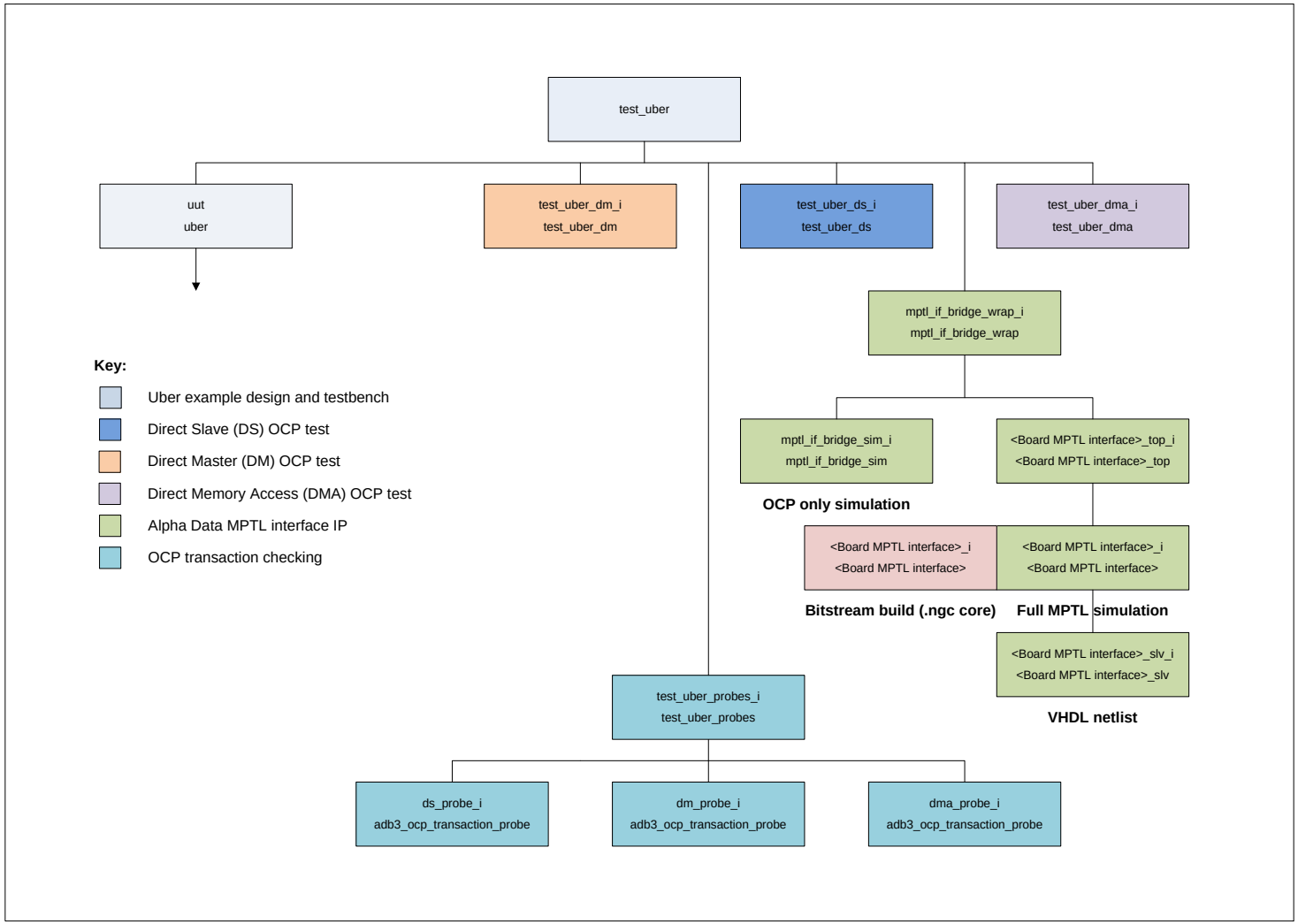

Figure 17: Uber Design Testbench Hierarchy

5.5.5.1 Clock Generation

This function produces the clocks required by the Unit Under Test (**uber**) and the bridge MPTL interface block. Clock generation is performed at the top level of the **Uber** testbench, in **hdl/vhdl/examples/uber/common/test_uber.vhd**. The testbench generates the following clocks:

- **clks_in** is a signal of record type that drives the UUT's top-level **clks_in** port. It is generated in a board-specific way, depending upon the **BOARD_TYPE** constant in the package **adb3_target_inc_pkg**. Among others, it contains the 200 MHz reference clock from which the main clocks in **Uber** are derived (see **Section 5.5.4.1** for details of clock generation in **Uber**).
- **clks_mgt_in** is a signal of record type that drives the UUT's top-level **clks_mgt_in** port, and is a bundle of all of the MGT-related clocks. It is generated in a board-specific way, depending on the **BOARD_TYPE** constant in the package **adb3_target_inc_pkg**.
- The testbench generates the clock **mptl_clk** with a frequency that depends upon the **BOARD_TYPE** constant in the package **adb3_target_inc_pkg**, in order to model the hardware. This clock drives the Bridge MPTL Interface (an instance of **mptl_if_bridge_wrap**) in the testbench.
- **ocp_clk** is a clock driven by the Bridge MPTL Interface so that OCP transactions can be generated and verified by the testbench.

The **ocp_clk** signal requires special explanation. This clock is driven by the Bridge MPTL Interface **mptl_if_bridge_wrap**. It is used within the testbench for monitoring OCP transactions, and how it is generated depends upon the type of simulation selected by the **TARGET_USE** constant in the package **adb3_target_inc_pkg**:

- In OCP-only simulation (**TARGET_USE = SIM_OCP**), the UUT's main OCP clock (**pll_pri_clk** in this case) is routed out of the UUT via the **mptl_if_target_wrap** instance and into the testbench's instance of **mptl_if_bridge_wrap**. The **mptl_if_bridge_wrap** instance outputs this signal as **ocp_clk**. This route is shown in **Figure 16** as the route consisting of points 1, 2, 3, 4 and 6.
- In full MPTL simulation (**TARGET_USE = SIM_MPTL**), **ocp_clk** is entirely independent of any clock within the UUT, and the testbench's **mptl_if_bridge_wrap** instance passes **ocp_clk_full** through to **ocp_clk**. This is shown in **Figure 6** as the route consisting of points 5 and 6.

5.5.5.2 Bridge MPTL Interface

The testbench contains an instance of **mptl_if_bridge_wrap**, which translates Direct Slave and DMA OCP transactions in the testbench to MPTL data. **mptl_if_bridge_wrap** wraps up the Bridge MPTL interface core, instantiating an OCP to MPTL core appropriate for the **BOARD_TYPE** and **TARGET_USE** constants from the package **adb3_target_inc_pkg**.

The **mptl_if_bridge_wrap** output **mptl_sb_b2t.mptl_bridge_gtp_online_l** is combined with the **Simple** example FPGA design output **mptl_sb_t2b.mptl_target_gtp_online_l** to produce the **mptl_online_long** signal. This indicates that the MPTL interface is active and stable.

Note: The testbench monitors **mptl_online_long** and will terminate the simulation 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.

5.5.5.3 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**.

5.5.5.4 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.

5.5.5.4.1 Non-OCP Functions

The top level of the testbench, **hdl/vhdl/examples/uber/common/test_uber.vhd**, 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.

5.5.5.4.1.1 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** in **test_uber.vhd** measures the frequencies of the bundle of clocks **clks_out** driven by the UUT and compares them with expected frequencies.

Test complete and pass/fail indications are returned using the **top_comp.clk_out_complete** and **top_pass.clk_out_passed** signals respectively in **test_uber.vhd**.

Example results from this test are documented in **clock output test results**.

5.5.5.4.1.2 MPTL GPIO Bus Test

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

The process **mptl_gpio_p** in **test_uber.vhd** 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 in **test_uber.vhd**.

Example results from this test are documented in **MPTL GPIO bus test results**.

5.5.5.4.1.3 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** in **test_uber.vhd** 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** 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 in **test_uber.vhd**.

Example results from this test are documented in **DMA abort bus test results**.

5.5.5.4.2 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. 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 Direct Slave OCP channels:

- The **Bridge MPTL interface** converts OCP commands and write data originating in **test_uber_ds** to MPTL protocol. Within the target FPGA, the **Target MPTL interface** converts MPTL protocol back into OCP commands and data. Thus, neither **test_uber_ds** nor the UUT (**uber**) is aware that OCP stimulus passes through the MPTL.
- Responses originating in the Target FPGA are correspondingly converted to MPTL protocol by the **Target MPTL interface**, and converted back into OCP responses by the **Bridge MPTL interface**}. Thus, neither **test_uber_ds** nor the UUT (**uber**) is aware that OCP responses pass through the MPTL.

test_uber_ds performs several tests, which are detailed in the following subsections.

5.5.5.4.2.1 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 in **test_uber.vhd**.

Example results from this test are documented in **simple test results**.

5.5.5.4.2.2 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.

Lastly, the test checks the frequency of the **MGT113_CLK0** clock:

8. Selects the **MGT113_CLK0** clock by writing 20 (corresponding to **MGT113_CLK0_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 14**) and compares it with the expected frequency for **MGT113_CLK0**. The expected frequency of this clock depends upon the **BOARD_TYPE** constant from the package **adb3_target_inc_pkg**.

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**.

5.5.5.4.2.3 XRM GPIO Test

This test exercises with the XRM-related registers of the **GPIO Test Register Block** 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**.

5.5.5.4.2.4 Pn4/Pn6 GPIO Test

This test exercises with the Pn4-related and Pn6-related registers of the **GPIO Test Register Block**. 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**.

5.5.5.4.2.5 Interrupt Test

This test exercises the **Interrupt Test Register Block**. Its 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 31 do
   a. Generate interrupt n, by writing the value 2^n to the **SET** register.
   b. Wait for the interrupt signal **linti_l** 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** 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** high for one cycle ensures that the newly active interrupt source results in another falling edge on **linti_l**.

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**.

5.5.5.4.2.6 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.

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**.

5.5.5.4.2.7 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**.

5.5.5.4.2.8 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.

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_OFF** 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. Set the **memory access window** for accessing the bottom 2 MiB page of memory bank 1, by writing 0 to the **DS_PAGE** register.
10. 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).
11. 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.
12. 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**.
13. Write the 32-bit word 0x47AD0369 to the top of the **memory access window** (**RAM_WIN_ADDR_BASE+ RAM_WIN_ADDR_MASK-3**).
14. 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.
15. Set the **memory access window** for accessing the bottom 2 MiB page, by writing 0 to the **DS_PAGE** register.
16. Write 32 bytes consisting of the 32-bit words { 0xABCDEF12, ... etc. ..., 0xFEDCBA98 } to the bottom of the **memory access window**. This is the case of an even-length burst (two 16-byte words) at an even address (bit 4 of the OCP address is 0).
17. 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.
18. Write 48 bytes consisting of the 32-bit words { 0xBCDEF123, ... etc. ..., 0x3456789A } to the bottom of the **memory access window**. This is the case of an odd-length burst (three 16-byte words) at an even address (bit 4 of the OCP address is 0).
19. 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.
20. Write 32 bytes consisting of the 32-bit words { 0xDEF12345, ... etc. ..., 0xDCBA9876 } 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 (two 16-byte words) at an odd address (bit 4 of the OCP address is 1).

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 48 bytes consisting of the 32-bit words { 0xEF123456, ... 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 (three 16-byte words) at an odd address (bit 4 of the OCP address is 1).
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 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.
25. 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.
26. 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.
27. 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.
28. 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.
29. 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**.

5.5.5.4.3 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.

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 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.

5.5.5.4.3.1 DMA OCP Command and Write Data Process

The process **dma_channel_cmd_p** in **test_uber_dma_1ch_nb.vhd** exercises DMA OCP channel 0 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 channel 0 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.
- 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 channel 0 with 'address' as the address, 'tag' as the tag, and length equal to 'chunk'. Wait until the command has been accepted.
   - Set remaining := remaining - chunk, address := address + chunk, tag := tag + 1
6. end while

The **DMA_SIZE**, **DMA_ADDR_WR** and **DMA_ADDR_RD** constants are defined in the **uber_tb_pkg** package. The values of **DMA_ADDR_WR** and **DMA_ADDR_RD** correspond to byte offset 0x7F00 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**.

5.5.5.4.3.2 DMA OCP Response Process

The process **dma_channel_resp_p** in **test_uber_dma_1ch_nb.vhd** exercises DMA OCP channel 0 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 OCP channel 0
   - 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

The **DMA_SIZE** constant is defined in the **uber_tb_pkg** package.

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**.

5.5.5.5 Memory Device Simulation Models

The testbench instantiates a DDR3 SDRAM simulation model for each memory device in each bank of on-board memory. The DDR3 SDRAM model is located in **hdl/vhdl/common/mem_tb/ddr3_sdram/**. Refer to **Section 6.7** for a functional description.

The DDR3 SDRAM part to be simulated is selected according to the **TB_DDR3_PART** constant defined in the **uber_tb_pkg** package.

5.5.5.6 Testbench Package

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

Defininitions are as follows:

Clock period constants

- Testbench clock period constants (type **time**).
- Testbench clock frequency constsnts (type **natural**).

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.
- **DMA_BL_WRITE**. The OCP burst length used by writes (Host to FPGA) during DMA test.
- **DMA_BL_READ**. The OCP burst length used by reads (FPGA to Host) during DMA test.

On-board RAM part selection constants

- **TB_DDR3_1G_PART**. Part number of 1 Gib DDR3 SDRAM components.
- **TB_DDR3_2G_PART**. Part number of 2 Gib DDR3 SDRAM components.
- **TB_DDR3_PART**. Part number of selected DDR3 SDRAM components.
- **TB_DDR3_ROW**. Row address width of selected DDR3 SDRAM components.
- **TB_DDR3_COL**. Column address width of selected DDR3 SDRAM components.
- **TB_DDR3_BANK**. Bank address width of selected DDR3 SDRAM components.
- **TB_DDR3_BYTE_ADDR_WIDTH**. Byte address width of selected DDR3 SDRAM components.
- **TB_DDR3_16_BYTE_ADDR_WIDTH**. 16-byte address width of selected DDR3 SDRAM components.

Interrupt test constants

- **MASK_EN_ALL**. Interrupt mask register enable all constant.

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.

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

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.bat** (Windows) or **gen_mem_if.bash** (Linux) in **hdl/vhdl/common/mem_if/ddr3_sdram/mig_v3_6/**. Refer to **Section 6.5** for details.

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 date and 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
--
-- 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 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;
```

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

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 Clock Output Test Results

Modelsim transcript output during simulation is of the form:

```
# ** Note: Expected clk_out freq = 80MHz
#    Time: 1483750 ps  Iteration: 10  Instance: /test_uber
# ** Note: Actual   clk_out freq = 80MHz
#    Time: 1483750 ps  Iteration: 10  Instance: /test_uber

# ** Note: Test clk_out completed: PASSED.
#    Time: 1483750 ps  Iteration: 10  Instance: /test_uber
```

5.5.6.3.2 MPTL GPIO Bus Test Results

Modelsim transcript output during simulation is of the form:

```
# ** Note: Test mptl_gpio completed: PASSED.
#    Time: 3028750 ps  Iteration: 13  Instance: /test_uber
```

5.5.6.3.3 DMA Abort Bus Test Results

Modelsim transcript output during simulation is of the form:

```
# ** Note: Test dma_abort completed: PASSED.
#    Time: 2278750 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: 2533750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote PLL_REG_CLK_SEL SEL 4 bytes 0x00000000 with enable 0b1111 to byte address 0x000040
#    Time: 2543750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PLL_REG_CLK FREQ 4 bytes 0x00000050 from byte address 0x000048
#    Time: 4176250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 80 MHz ±2 MHz
#    Time: 4176250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual freq = 80 MHz
#    Time: 4176250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote PLL_PRI_CLK_SEL SEL 4 bytes 0x00000001 with enable 0b1111 to byte address 0x000040
#    Time: 4183750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PLL_PRI_CLK FREQ 4 bytes 0x000000C8 from byte address 0x000048
#    Time: 4751250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 200 MHz ±2 MHz
#    Time: 4751250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual freq = 200 MHz
#    Time: 4751250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote MGT113_CLK0_SEL SEL 4 bytes 0x00000014 with enable 0b1111 to byte address 0x000040
#    Time: 4758750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read MGT113_CLK0 FREQ 4 bytes 0x000000FA from byte address 0x000048
#    Time: 5326250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 250 MHz ±2 MHz
#    Time: 5326250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual freq = 248 MHz
#    Time: 5326250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Clock Read completed: PASSED.
#    Time: 5326250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

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
```

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
```

5.5.6.4.5 Interrupt Test Results

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 0x18022011 from byte address 0x000140
#    Time: 29491250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info TIME 4 bytes 0x10522600 from byte address 0x000144
#    Time: 29741250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info BRAM BASE 4 bytes 0x00080000 from byte address 0x00014C
#    Time: 29991250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info BRAM MASK 4 bytes 0x0007FFFF from byte address 0x000150
#    Time: 30241250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info RAM BASE 4 bytes 0x00200000 from byte address 0x000154
#    Time: 30491250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info RAM MASK 4 bytes 0x001FFFFF from byte address 0x000158
#    Time: 30741250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info RAM INFO 4 bytes 0xXXXXXXX4 from byte address 0x00015C
#    Time: 30991250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test Info completed: PASSED.
#    Time: 30991250 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: 32801250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

```
# ** Note: On-board RAM bank 1 initialised
#    Time: 58551250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Read RAM Bank Info Reg 4 bytes 0x3F10181C from byte address 0x00034C
#    Time: 58801250 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: 58808750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Bank Offset Reg 4 bytes 0x00FFFEFF from byte address 0x000344
#    Time: 59126250 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: 59133750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Bank Length Reg 4 bytes 0x000000FF from byte address 0x000348
#    Time: 59451250 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: 59458750 ps  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: 59468750 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: 59478750 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: 59488750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 4 bytes 0x349AF056 to byte address 0x200000
#    Time: 61401250 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: 61408750 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: 61418750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr top 4 bytes 0x47AD0369 from byte address 0x3FFFFC
#    Time: 62776250 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: 62783750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote RAM Win Addr base 32 bytes 0xFEDCBA987654321FEDCBA987654321FE123456789ABCDEF123456789ABCDEF12
with enable 0b11111111111111111111111111111111 to byte address 0x200000
#    Time: 62798750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 32 bytes 0xFEDCBA987654321FEDCBA987654321FE123456789ABCDEF123456789ABCDEF12
from byte address 0x200000
#    Time: 64021250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 48 bytes
0x3456789ABCDEF123456789ABCDEF1234EDCBA987654321FEDCBA987654321FED23456789ABCDEF123456789ABCDEF123
with enable 0b111111111111111111111111111111111111111111111111 to byte address 0x200000
#    Time: 64038750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 48 bytes
0x3456789ABCDEF123456789ABCDEF1234EDCBA987654321FEDCBA987654321FED23456789ABCDEF123456789ABCDEF123
from byte address 0x200000
#    Time: 65741250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 32 bytes 0xDCBA987654321FEDCBA987654321FEDC456789ABCDEF123456789ABCDEF12345
with enable 0b11111111111111111111111111111111 to byte address 0x200010
#    Time: 65753750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 32 bytes 0xDCBA987654321FEDCBA987654321FEDC456789ABCDEF123456789ABCDEF12345
from byte address 0x200010
#    Time: 67296250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 48 bytes
0x6789ABCDEF123456789ABCDEF1234567DCBA987654321FEDCBA987654321FEDC56789ABCDEF123456789ABCDEF123456
with enable 0b111111111111111111111111111111111111111111111111 to byte address 0x200010
#    Time: 67313750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 48 bytes
0x6789ABCDEF123456789ABCDEF1234567DCBA987654321FEDCBA987654321FEDC56789ABCDEF123456789ABCDEF123456
from byte address 0x200010
#    Time: 68541250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Wrote RAM Win Addr base 4 bytes 0x45000000 with enable 0b1000 to byte address 0x200000
#    Time: 68548750 ps  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: 68558750 ps  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: 68568750 ps  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: 68578750 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read RAM Win Addr base 4 bytes 0x45AB0167 from byte address 0x200000
#    Time: 69791250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Waiting for internal test of on-board RAM bank 1 to complete...
#    Time: 69791250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Read RAM Bank Stat Reg 4 bytes 0x1XXXX0XF from byte address 0x000350
```

```
#    Time: 81541250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Internal test of on-board RAM bank 1 complete
#    Time: 81541250 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i

# ** Note: Test RAM completed: PASSED.
#    Time: 81541250 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 libraries as follows:

- **ADB3 OCP library**
- **MPTL library**
- **ADB3 target library**
- **ADB3 probe library**
- **Memory interface library**
- **Memory application library**
- **Memory model library**
- **Clock frequency measurement library**
- **ChipScope™ library**

6.1 ADB3 OCP Library

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

- **ADB3 OCP profile definition package (adb3_ocp)**
- **ADB3 OCP library component declaration package (adb3_ocp_comp)**
- **ADB3 OCP library components**

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 **Section 7.1** for a description of ADB3 OCP protocol transactions.

6.1.2 ADB3 OCP Library Component Declaration Package (adb3_ocp_comp)

The package **adb3_ocp_comp** defines general-purpose ADB3 OCP library components.

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 in general, but require less FPGA resources. Blocking components in the ADB3 OCP library are as follows:

- **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 in general, but require more FPGA resources. Non-blocking components in the ADB3 OCP library are as follows:

- **adb3_ocp_cross_clk_dom**
- **adb3_ocp_mux_nb**
- **adb3_ocp_ocp2ddr3_nb**
- **adb3_ocp_retime_nb**
- **adb3_ocp_split_nb**

6.1.3 ADB3 OCP Library 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** library. 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.

6.1.3.1.2 Interface

The adb3_ocp_cross_clk_dom component interface is shown in **Figure 18** below and described in **Table 73**.

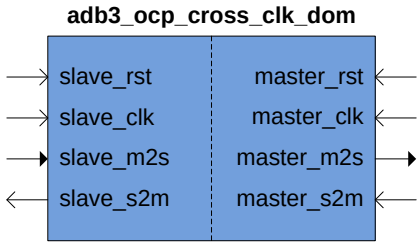

Figure 18: ADB3 OCP Library adb3_ocp_cross_clk_dom Component Interface

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 73: ADB3 OCP Library 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 19** below.

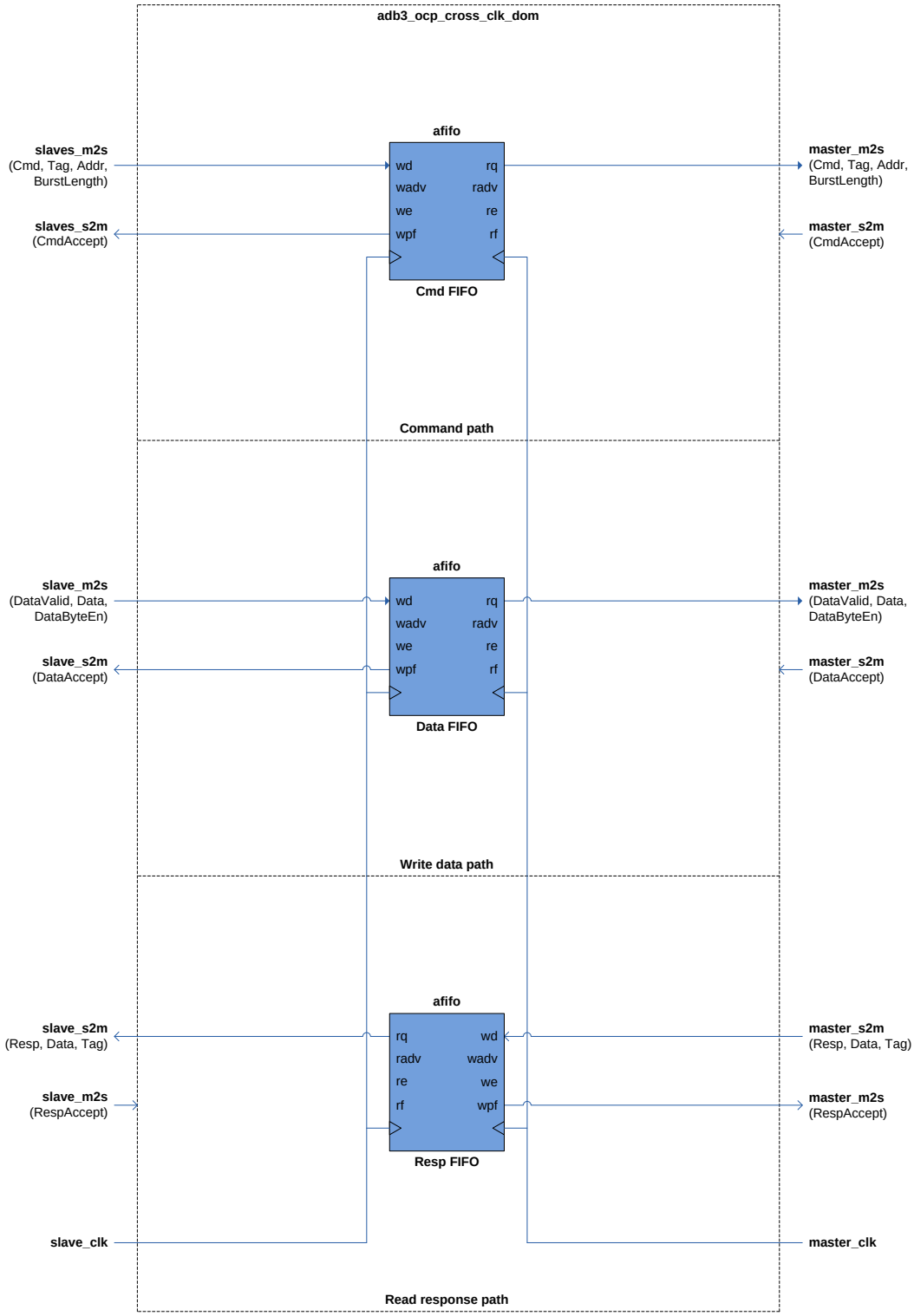

Figure 19: ADB3 OCP Library adb3_ocp_cross_clk_dom Block Diagram

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 **slave_m2s** 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** library. 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 20** below and described in **Table 74**.

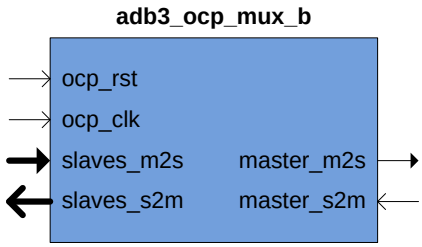

Figure 20: ADB3 OCP Library adb3_ocp_mux_b Component Interface

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 74: ADB3 OCP Library 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** library. 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.

6.1.3.3.2 Interface

The adb3_ocp_mux_nb component interface is shown in **Figure 21** below and described in **Table 75**.

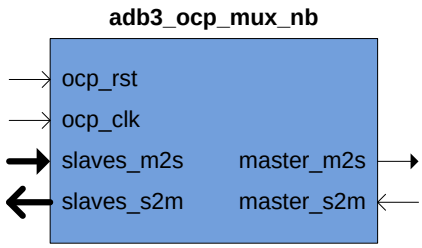

Figure 21: ADB3 OCP Library adb3_ocp_mux_nb Component Interface

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 75: ADB3 OCP Library adb3_ocp_mux_nb Component Interface

6.1.3.3.3 Description

The adb3_ocp_mux_nb component block diagram is shown in **Figure 22** below.

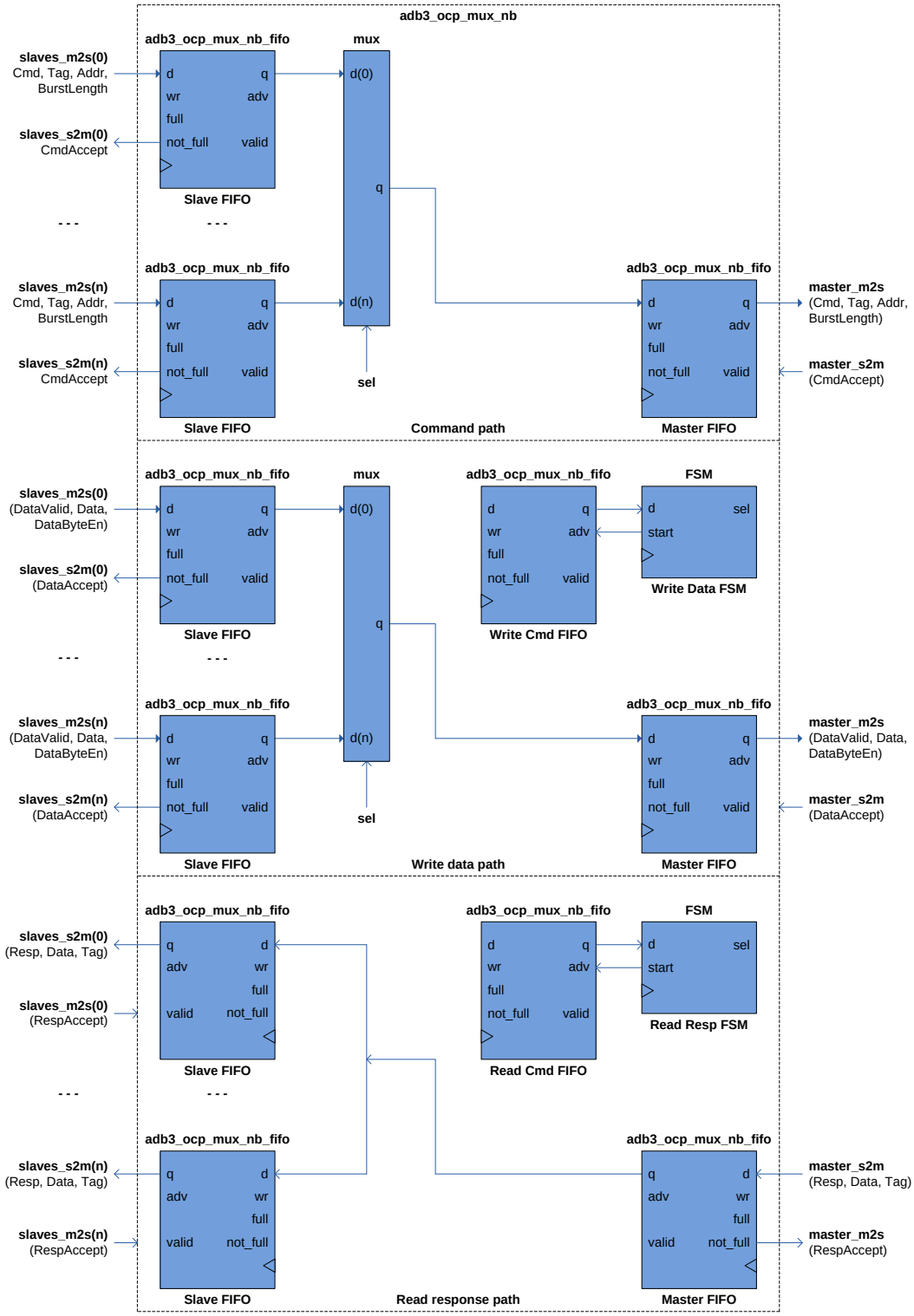

Figure 22: ADB3 OCP Library adb3_ocp_mux_nb Block Diagram

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** library. Its function is to interface a single ADB3 OCP channel to the Xilinx™ DDR3 SDRAM MIG core user interface.

6.1.3.4.2 Interface

The adb3_ocp_ocp2ddr3_nb component interface is shown in **Figure 23** below and described in **Table 76**.

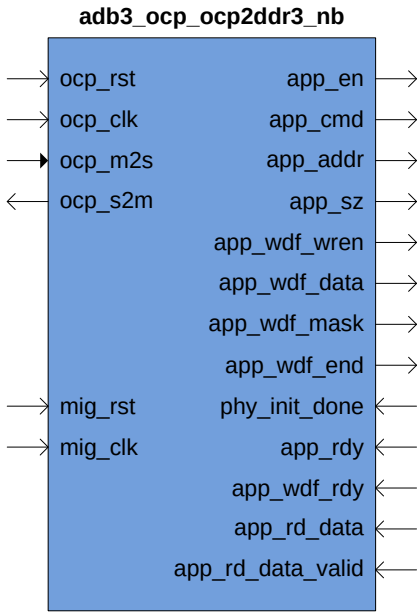

Figure 23: ADB3 OCP Library adb3_ocp_ocp2ddr3_nb Component Interface

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 addressing).
		OCP Interface
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.
ocp_m2s	Input	OCP M2S connection.
ocp_s2m	Output	OCP S2M connection.
		DDR3 SDRAM MIG Core User Interface
mig_rst	Input	User interface reset.
mig_clk	Input	User interface clock.
phy_init_done	Input	User interface phy calibration complete.
app_rdy	Input	User interface command ready.

Table 76: ADB3 OCP Library adb3_ocp_ocp2ddr3_nb Component Interface (continued on next page)

Signal	Type	Description
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.
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 76: ADB3 OCP Library adb3_ocp_ocp2ddr3_nb Component Interface

6.1.3.4.3 Description

The adb3_ocp_ocp2ddr3_nb component block diagram is shown in **Figure 24** below.

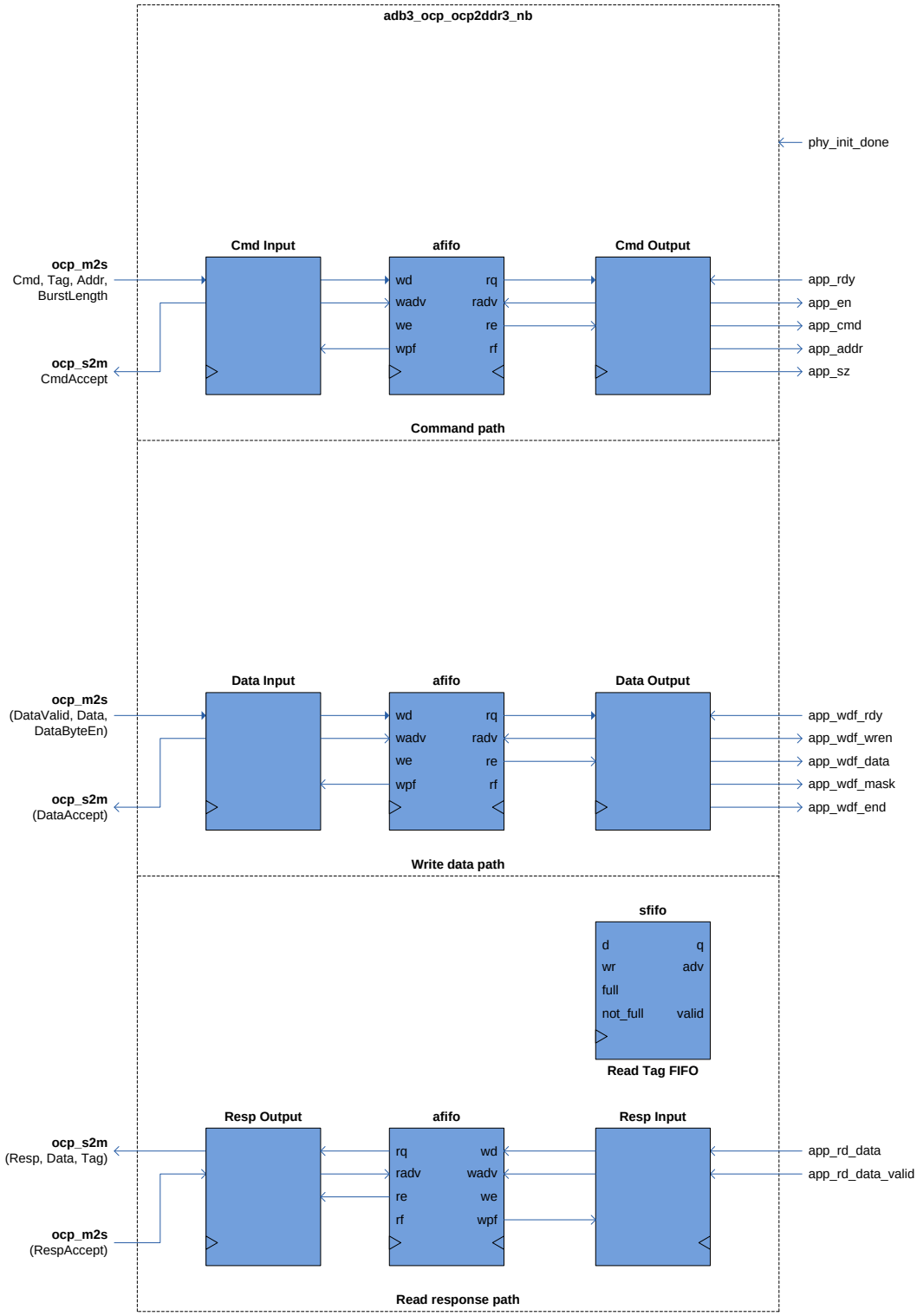

Figure 24: ADB3 OCP Library adb3_ocp_ocp2ddr3_nb Block Diagram

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** signals.

Command Input

- This block operates in the **ocp_clk** domain.
- The **ocp_m2s** port command/data elements and command FIFO full flag are used to produce the command FIFO data and write advance and the **slave_s2m** port **CmdAccept** element.
- **ocp_m2s** port transactions are converted into MIG core user interface transactions which are then written to the command FIFO.

Command FIFO

- MIG core user interface transaction data in the **ocp_clk** domain is interfaced to the **mig_clk** domain.

Command Output

- This block operates in the **mig_clk** domain.
- The command FIFO data output and empty flag are used to produce the MIG core user interface command signals.

Read Tag FIFO

- This block operates in the **mig_clk** domain.
- The command FIFO data outputs **cmd_fifo_bl8_out** and **cmd_fifo_tag_out** are written to the read tag FIFO on every MIG core user interface read command.
- The read tag FIFO output **tag_fifo_tag_out** is used as the tag value for OCP response data written into the response FIFO.
- The read tag FIFO output **tag_fifo_bl8_out** is compared with the number of OCP response data words and this is used to generate the read tag 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** signals.

Write Data Input

- This block operates in the **ocp_clk** domain.
- The **ocp_m2s** port command/data elements and write data FIFO full flag are used to produce the write data FIFO data and write advance and the **slave_s2m** port **DataAccept** element.
- **ocp_m2s** port write data is converted into MIG core user interface write data data which is then written to the write data FIFO.

Write Data FIFO

- MIG core user interface write data in the **ocp_clk** domain is interfaced to the **mig_clk** domain.

Write Data Output

- This block operates in the **mig_clk** domain.

- The write data FIFO data output and empty flag are used to produce the MIG core user interface write data 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** signals.

Read Response Input

- This block operates in the **mig_clk** domain.
- The **app_rd_data**, and **app_rd_data_valid** signals, read tag FIFO output **tag_fifo_tag_out**, and read response FIFO full flag are used to produce the read response FIFO data and write advance.
- MIG core user interface read data signals are converted to OCP response data which is then written to the read response FIFO.

Read Response FIFO

- MIG core user interface read data in the **mig_clk** domain is interfaced to the **ocp_clk** domain.

Read Response Output

- This block operates in the **ocp_clk** domain.
- The read response FIFO data output and empty flag, and **ocp_m2s** port element **RespAccept** are used to produce the **ocp_s2m** port response elements and the response FIFO read advance.

6.1.3.5 adb3_ocp_retime_nb

6.1.3.5.1 Introduction

This is a non-blocking component in the **ADB3 OCP** library. Its function is to re-time a single primary ADB3 OCP channel, producing a single secondary ADB3 OCP channel.

6.1.3.5.2 Interface

The adb3_ocp_retime_nb component interface is shown in **Figure 25** below and described in **Table 77**.

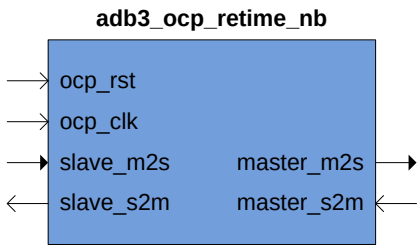

Figure 25: ADB3 OCP Library adb3_ocp_retime_nb Component Interface

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 77: ADB3 OCP Library adb3_ocp_retime_nb Component Interface

6.1.3.5.3 Description

TBD

6.1.3.6 adb3_ocp_simple_bus_if

6.1.3.6.1 Introduction

This is a blocking component in the **ADB3 OCP** library. 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 26** below and described in **Table 78**.

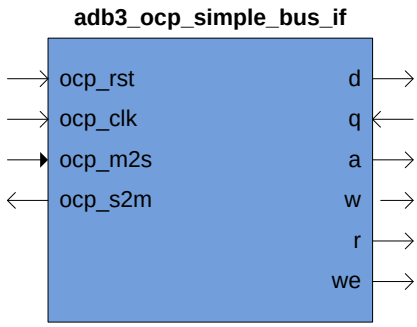

Figure 26: ADB3 OCP Library adb3_ocp_simple_bus_if Component Interface

Signal	Type	Description
addr_width	Generic	Width of the address output a.
data_width	Generic	Width of the data input/output d/q.
read_latency	Generic	Number of cycles delay before read data q 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.
q	Input	Read data.
a	Output	Write/Read address.
w	Output	Write valid.
r	Output	Read valid.
we	Output	Write data byte valid.

Table 78: ADB3 OCP Library adb3_ocp_simple_bus_if Component Interface

6.1.3.6.3 Description

TBD

6.1.3.7 adb3_ocp_split_b

6.1.3.7.1 Introduction

This is a blocking component in the **ADB3 OCP** library. 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.

6.1.3.7.2 Interface

The adb3_ocp_split_b component interface is shown in **Figure 27** below and described in **Table 79**.

Figure 27: ADB3 OCP Library 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 79: ADB3 OCP Library adb3_ocp_split_b Component Interface

6.1.3.7.3 Description

TBD

6.1.3.8 adb3_ocp_split_nb

6.1.3.8.1 Introduction

This is a non-blocking component in the **ADB3 OCP** library. 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_nb component interface is shown in **Figure 28** below and described in **Table 80**.

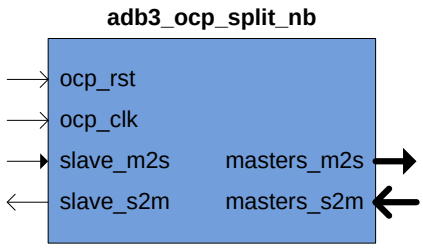

Figure 28: ADB3 OCP Library adb3_ocp_split_nb Component Interface

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 80: ADB3 OCP Library adb3_ocp_split_nb Component Interface

6.1.3.8.3 Description

The adb3_ocp_split_nb component block diagram is shown in **Figure 29** below.

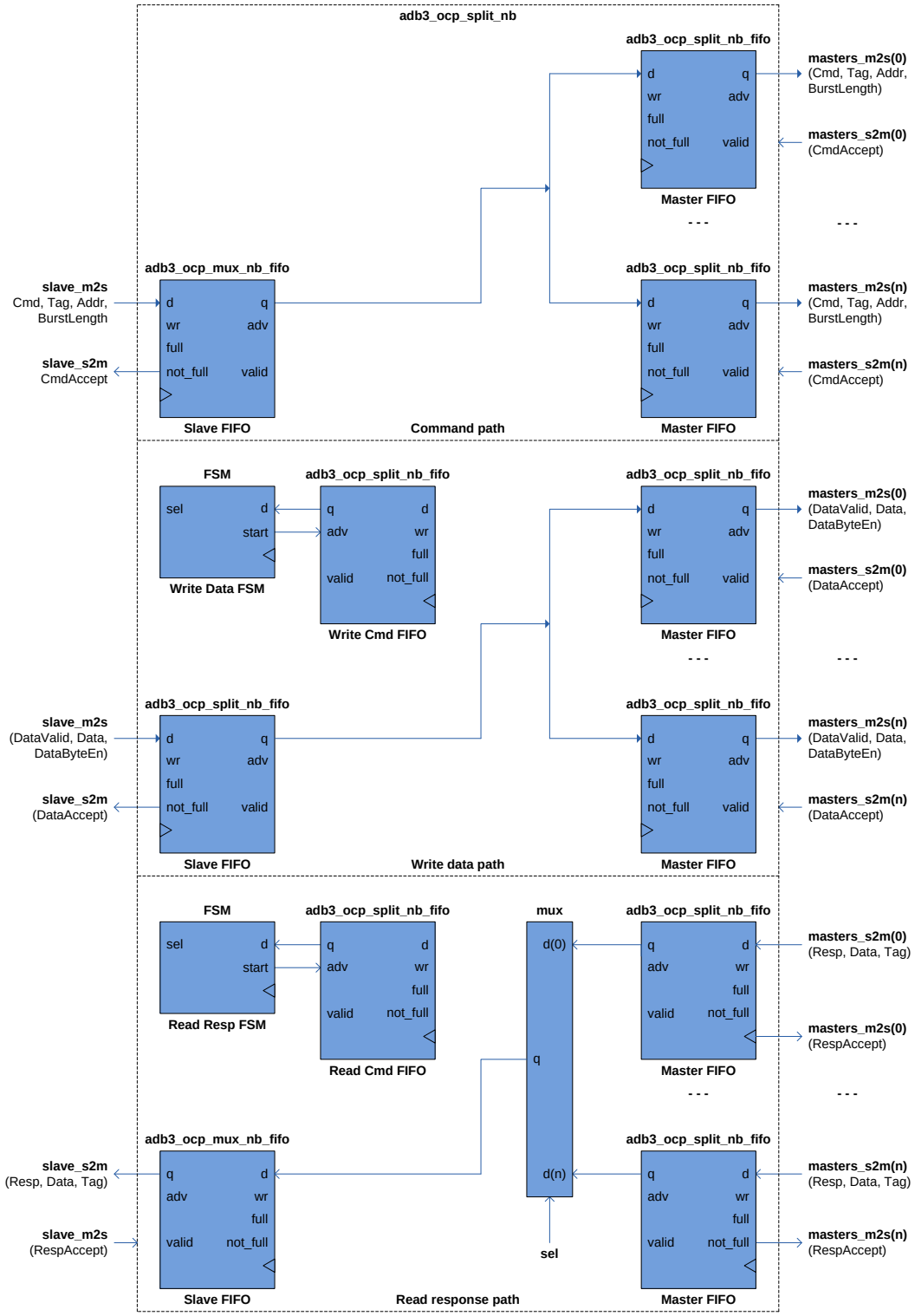

Figure 29: ADB3 OCP Library adb3_ocp_split_nb Block Diagram

6.1.3.8.3.1 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.

6.1.3.8.3.2 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.

6.1.3.8.3.3 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.

6.2 MPTL Library

The MPTL library is located in the **hdl/vhdl/common/mptl** directory and contains the following elements:

- **MPTL library components**
- **MPTL interface components**

6.2.1 MPTL Library Components

6.2.1.1 Bridge MPTL Interface Wrapper (mptl_if_bridge_wrap)

6.2.1.1.1 Introduction

This is a component in the **MPTL** library. It is used by example FPGA testbenches to convert between stimulus OCP transactions and Bridge MPTL interface data. It is located in the **hdl/vhdl/common/mptl** directory. The MPTL interface that is instantiated depends on the board selected and the design use.

The board selected is indicated by the value of the **BOARD_TYPE** constant. The design use is indicated by the value of the **TARGET_USE** constant. Both are defined in the **adb3_target_inc_pkg** package.

6.2.1.1.2 Interface

The mptl_if_bridge_wrap component interface is shown in **Figure 30** below and described in **Table 81**.

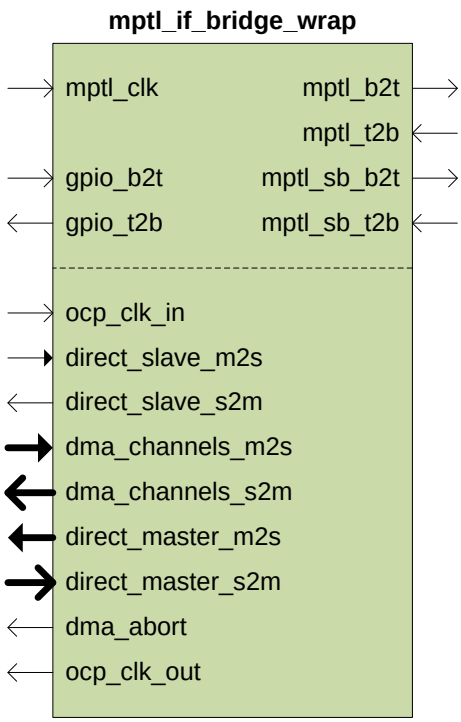

Figure 30: MPTL Library mptl_if_bridge_wrap Component Interface

Signal	Type	Description
		OCP Interface
ocp_clk_in	Input	Independent OCP clock source (from testbench).
direct_slave_m2s	Input	Direct slave OCP channel master (Transferred to target via MPTL interface).
direct_slave_s2m	Output	Direct slave OCP channel slave (Transferred from target via MPTL interface.
dma_channels_m2s	Input	DMA OCP channels master (Transferred to target via MPTL interface).
dma_channels_s2m	Output	DMA OCP channels slave (Transferred from target via MPTL interface).
direct_masters_m2s	Output	Direct mater OCP channels master (Transferred from target via MPTL interface).
direct_masters_s2m	Input	Direct master OCP channels slave (Transferred 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 data signals connected to target MPTL interface.
mptl_b2t	Output	MPTL interface data signals connected to target MPTL interface.
mptl_clk	Input	MPTL interface clock (from testbench).
mptl_sb_t2b	Input	MPTL interface sideband signals connected to target MPTL interface.
mptl_sb_b2t	Output	MPTL interface sideband signals connected to target MPTL interface.
gpio_b2t	Input	General purpose i/o (Transferred to target via MPTL interface).
gpio_t2b	Output	General purpose i/o (Transferred from target via MPTL interface).

Table 81: MPTL Library mptl_if_bridge_wrap Component Interface

6.2.1.1.3 Description

6.2.1.1.3.1 OCP-Only Simulation

During OCP-only simulation (selected by TARGET_USE = SIM_OCP), the **mptl_if_bridge_wrap** component instantiates the simulation MPTL interface **mptl_if_bridge_sim**.

Refer to **Section 6.2.2.1** for a functional description.

6.2.1.1.3.2 Full MPTL Simulation

During full MPTL simulation (selected by TARGET_USE = SIM_MPTL), the **mptl_if_bridge_wrap** component instantiates the full MPTL interface appropriate to the board in use. The MPTL interface consists of the actual wrapped logic supplied as a Xilinx™ HDL netlist file (**.vhd**).

Refer to **Section 6.2.2.3** for a functional description.

6.2.1.2 Target MPTL Interface Wrapper (mptl_if_target_wrap)

6.2.1.2.1 Introduction

This is a component in the **MPTL** library. It is used by the example FPGA designs to convert between Target MPTL interface data and OCP transactions. It is located in the **hdl/vhdl/common/mptl** directory. The type of Target MPTL interface that is instantiated depends upon which variant of the **adb3_target_inc_pkg** is in use, through the **BOARD_TYPE** and **TARGET_USE** constants.

6.2.1.2.2 Interface

The mptl_if_target_wrap component interface is shown in **Figure 31** below and described in **Table 82**.

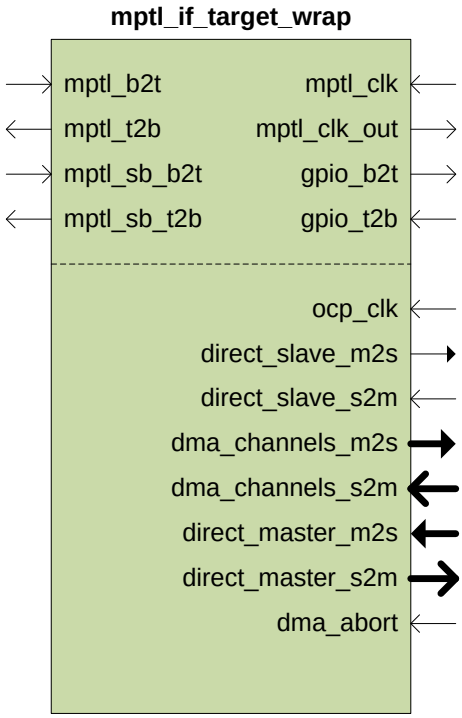

Figure 31: MPTL Library mptl_if_target_wrap Component Interface

Signal	Type	Description
		OCP Interface
ocp_clk	Input	OCP clock (from target FPGA).
direct_slave_m2s	Output	Direct slave OCP channel master (Transferred from bridge via MPTL interface).
direct_slave_s2m	Input	Direct slave OCP channel slave (Transferred to bridge via MPTL interface).
dma_channels_m2s	Output	DMA OCP channels master (Transferred from bridge via MPTL interface).
dma_channels_s2m	Input	DMA OCP channels slave (Transferred to bridge via MPTL interface).
direct_masters_m2s	Input	Direct mater OCP channels master (Transferred to bridge via MPTL interface).
direct_masters_s2m	Output	Direct master OCP channels slave (Transferred from bridge via MPTL interface).

Table 82: MPTL Library mptl_if_target_wrap Component Interface (continued on next page)

Signal	Type	Description
dma_abort	Input	DMA abort request (from target FPGA).
		MPTL Interface
mptl_t2b	Output	MPTL interface data signals connected to bridge MPTL interface.
mptl_b2t	Input	MPTL interface data signals connected to bridge MPTL interface.
mptl_clk	Input	MPTL interface clock (from target FPGA).
mptl_clk_out	Output	Unused.
ocp_ready	Input	OCP channels ready (from target FPGA).
mptl_sb_t2b	Output	MPTL interface sideband signals connected to bridge MPTL interface.
mptl_sb_b2t	Input	MPTL interface sideband signals connected to bridge MPTL interface.
gpio_b2t	Output	General purpose i/o (Transferred from bridge via MPTL interface).
gpio_t2b	Input	General purpose i/o (Transferred to bridge via MPTL interface).

Table 82: MPTL Library mptl_if_target_wrap Component Interface

6.2.1.2.3 Description

6.2.1.2.3.1 OCP-Only Simulation

During OCP-only simulation (selected by TARGET_USE = SIM_OCP), the **mptl_if_target_wrap** component instantiates the simulation MPTL interface **mptl_if_target_sim**.

Refer to **Section 6.2.2.2** for a functional description.

6.2.1.2.3.2 Full MPTL Simulation

During full MPTL simulation (selected by TARGET_USE = SIM_MPTL), the **mptl_if_target_wrap** component instantiates the full MPTL interface appropriate to the board in use. The MPTL interface consists of the actual wrapped logic supplied as a Xilinx™ HDL netlist file (**.vhd**).

Refer to **Section 6.2.2.4** for a functional description.

6.2.1.2.3.3 Synthesis

During synthesis (selected by TARGET_USE = SYN_NGC), the **mptl_if_target_wrap** component instantiates the full MPTL interface appropriate to the board in use. The MPTL interface consists of the actual wrapped logic supplied as a Xilinx™ ISE netlist file (**.ngc**).

Refer to **Section 6.2.2.5** for a functional description.

6.2.2 MPTL Interface Components

6.2.2.1 Bridge MPTL Interface For OCP-Only Simulation (mptl_if_bridge_sim)

6.2.2.1.1 Introduction

This component consists of an OCP-only simulation version of the bridge MPTL interface.

6.2.2.1.2 Interface

This component's interface is the sames as the mptl_if_bridge_wrap component. Refer to **Figure 30** and **Table 81**.

6.2.2.1.3 Description

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interface blocks. They are of type **mptl_pins_t** which is defined in the **adb3_target_inc_pkg** package appropriate for OCP-only simulation of the board in use. For example **adb3_target_inc_sim_ocp_6t1_pkg.vhd** located in **hdl/vhdl/common/adb3_target/admxrc6t1/** for the ADM-XRC-6T1. During OCP-only simulation these signals transfer OCP transactions directly between the bridge and target MPTL interface blocks.

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_sb_t2b.mptl_target_configured_l** input is ignored.
- The **mptl_sb_t2b.mptl_target_gtp_online_l** input is ignored.

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-6T1 board, the **mptl_t2b.dma_abort** input from the target MPTL interface drives the DMA abort request output **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**.

6.2.2.2 Target MPTL Interface For OCP-Only Simulation (mptl_if_target_sim)

6.2.2.2.1 Introduction

This component consists of an OCP-only simulation version of the target MPTL interface.

6.2.2.2.2 Interface

This component's interface is the sames as the mptl_if_target_wrap component. Refer to **Figure 31** and **Table 82**.

6.2.2.2.3 Description

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interface blocks. They are of type **mptl_pins_t** which is defined in the **adb3_target_inc_pkg** package appropriate for OCP-only simulation of the board in use. For example **adb3_target_inc_sim_ocp_6t1_pkg.vhd** located in **hdl/vhdl/common/adb3_target/admxrc6t1/** for the ADM-XRC-6T1. During OCP-only simulation these signals transfer OCP transactions directly between the bridge and target MPTL interface blocks.

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 direct master OCP channels slave output **direct_masters_s2m** is driven by the **mptl_b2t.direct_masters_s2m** input from the bridge MPTL interface. The **mptl_t2b.direct_masters_m2s** output to the bridge MPTL interface is driven by the direct master OCP channels master input **direct_masters_m2s**.
- 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-6T1 board, the **dma_abort** input from the target FPGA drives the DMA abort request output **mptl_t2b.dma_abort**.

- On the ADM-XRC-6TL board, the inverted **dma_abort** input from the target FPGS drives the DMA abort request output **mptl_sb_t2b.mptl_dma_abort_l**.

6.2.2.3 Bridge MPTL Interface For Full MPTL Simulation

6.2.2.3.1 Introduction

This component instantiates an HDL netlist of the bridge MPTL interface during full MPTL simulation. The component used depends on the board selected for simulation. For example, for the ADM-XRC-6T1, the block hierarchy is:

- MPTL interface wrapper (**mptl128_interface_bridge_6t1_top**)
- MPTL interface top level (**mptl128_interface_bridge_6t1**)
- MPTL interface netlist (**mptl128_interface_bridge_6t1_slv**)

These components can be found in the following locations:

- hdl/vhdl/common/mptl/admxrc6t1/mptl128_interface_bridge_6t1_top.vhd
- hdl/vhdl/common/mptl/admxrc6t1/mptl128_interface_bridge_6t1_sim.vhd
- hdl/vhdl/common/mptl/admxrc6t1/mptl128_interface_bridge_6t1_slv.vhd

6.2.2.3.2 Interface

This component's interface is the sames as the mptl_if_bridge_wrap component. Refer to **Figure 30** and **Table 81**.

6.2.2.3.3 Description

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interface blocks. They are of type **mptl_pins_t** which is defined in the **adb3_target_inc_pkg** package appropriate for full MPTL simulation of the board in use. For example **adb3_target_inc_sim_mptl_6t1_pkg.vhd** located in **hdl/vhdl/common/adb3_target/admxrc6t1/** for the ADM-XRC-6T1. During full MPTL simulation these signals transfer MPTL data between the bridge and target MPTL interface blocks.

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 interface wrapper direct master OCP channels input (**direct_masters_s2m**) is processed by the **make_defined_s2m** function to ensure that it only contains '0' or '1' data. Other data values may cause the simulation of the MPTL interface to fail.

The remainder of the MPTL interface wrapper signals are connected to their equivalents on the MPTL interface top level.

6.2.2.4 Target MPTL Interface For Full MPTL Simulation

6.2.2.4.1 Introduction

This component instantiates an HDL netlist of the target MPTL interface during Full MPTL simulation. The component used depends on the board selected for simulation. For example, for the ADM-XRC-6T1, the block hierarchy is:

- MPTL interface wrapper (mptl128_interface_target_6t1_top)
- MPTL interface top level (mptl128_interface_target_6t1)
- MPTL interface netlist (mptl128_interface_target_6t1_slv)

These components can be found in the following locations:

- hdl/vhdl/common/mptl/admxrc6t1/mptl128_interface_target_6t1_top.vhd
- hdl/vhdl/common/mptl/admxrc6t1/mptl128_interface_target_6t1_sim.vhd
- hdl/vhdl/common/mptl/admxrc6t1/mptl128_interface_target_6t1_slv.vhd

6.2.2.4.2 Interface

This component's interface is the sames as the mptl_if_target_wrap component. Refer to **Figure 31** and **Table 82**.

6.2.2.4.3 Description

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interface blocks. They are of type **mptl_pins_t** which is defined in the **adb3_target_inc_pkg** package appropriate for full MPTL simulation of the board in use. For example **adb3_target_inc_sim_mptl_6t1_pkg.vhd** located in **hdl/vhdl/common/adb3_target/admxrc6t1/** for the ADM-XRC-6T1. During full MPTL simulation these signals transfer MPTL data between the bridge and target MPTL interface blocks.

OCP Interface

- The MPTL interface wrapper direct slave OCP channel input (**direct_slave_s2m**) and DMA OCP chammels input (**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 interface wrapper signals are connected to their equivalents on the MPTL interface top level.

6.2.2.5 Target MPTL Interface For Synthesis

6.2.2.5.1 Introduction

This component instantiates a target MPTL interface core during synthesis. The component used depends on the board selected for synthesis. For example, for the ADM-XRC-6T1, the block hierarchy is:

- MPTL interface wrapper (mptl128_interface_target_6t1_top)
- MPTL interface top level (mptl128_interface_target_6t1)

These components can be found in the following locations:

- hdl/vhdl/common/mptl/admxrc6t1/mptl128_interface_target_6t1_top.vhd
- hdl/vhdl/common/mptl/admxrc6t1/v6lxt/mptl128_interface_target_6t1.ngc

6.2.2.5.2 Interface

This component's interface is the sames as the mptl_if_target_wrap component. Refer to **Figure 31** and **Table 82**.

6.2.2.5.3 Description

The MPTL interface signals **mptl_t2b** and **mptl_b2t** connect the bridge and target MPTL interface blocks. They are of type **mptl_pins_t** which is defined in the **adb3_target_inc_pkg** package appropriate for synthesis of the board in use. For example **adb3_target_inc_syn_ngc_6t1_pkg.vhd** located in **hdl/vhdl/common/adb3_target/admxrc6t1/** for the ADM-XRC-6T1. During synthesis these signals transfer MPTL data between the bridge and target MPTL interface blocks.

The MPTL interface wrapper signals are connected to their equivalents on the MPTL interface top level.

6.3 ADB3 Target Library

The ADB3 target library 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 testbench package (adb3_target_tb_pkg)**

6.3.1 ADB3 Target Types Definition Package (adb3_target_types_pkg)

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

Types are defined as follows:

- **board_type_t**. An enumerated type containing an element for each board supported by the SDK, for example **ADM_XRC_6T1** for the ADM-XRC-6T1 board.
- **target_use_t**. An enumerated type containing an element for each end use supported by the SDK, for example **SIM_OCP** for OCP-only simulation.

Maximum value constants (covering all boards supported by the SDK) are defined as follows:

- **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 banks.
- **MAX_MPTL_SER_WIDTH**: width of MPTL serial data interface.
- **MAX_XRM_GPIO_WIDTH**: width of the XRM GPIO interface.
- **MAX_XRM_MGT_WIDTH**: width of the XRM MGT interface.
- **MAX_PN4_GPIO_WIDTH**: width of the Pn4 GPIO interface.
- **MAX_PN6_GPIO_WIDTH**: width of the Pn6 GPIO interface.
- **MAX_PN6_MGT_WIDTH**: width of the Pn6 MGT interface.

6.3.2 ADB3 Target Include Package (adb3_target_inc_pkg)

The **adb3_target_inc_pkg** package defines constants and types which characterise the board selected, and whether synthesis or simulation is being performed. This enables a simulation to perform "lightweight" versions of certain lengthy initialisation sequence. 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. For example, the package for OCP-only simulation of the ADM-XRC-6T1 board is contained in **hdl/vhdl/common/adb3_target/adb3_target_inc_sim_ocp_6t1_pkg.vhd**. **Table 83** lists the available variants of the **adb3_target_inc_pkg** package:

Model	TARGET_USE	Filename relative to hdl/vhdl/common/adb3_target/
ADM-XRC-6TL	SIM_MPTL	admxrc6tl/adb3_target_inc_sim_mptl_6tl_pkg.vhd
	SIM_OCP	admxrc6tl/adb3_target_inc_sim_ocp_6tl_pkg.vhd
	SYN_NGC	admxrc6tl/adb3_target_inc_syn_ngc_6tl_pkg.vhd
ADM-XRC-6T1	SIM_MPTL	admxrc6t1/adb3_target_inc_sim_mptl_6t1_pkg.vhd
	SIM_OCP	admxrc6t1/adb3_target_inc_sim_ocp_6t1_pkg.vhd
	SYN_NGC	admxrc6t1/adb3_target_inc_syn_ngc_6t1_pkg.vhd

Table 83: Available variants of the adb3_target_inc_pkg package

The following definitions are available in this package:

Usage Definitions

- **BOARD_TYPE**. Defines the board in use according to the **board_type_t** enumerated type; for example, **ADM_XRC_6T1** for the ADM-XRC-6T1 board.
- **TARGET_USE**. Defines the usage according to the **target_use_t** enumerated type; for example, **SIM_OCP** for OCP-only simulation.

Clock Definitions

- **CLKS_IN_REF_CLK_VALID**. Indicates presence of **ref_clk** clock input on this board.
- **CLKS_IN_LCLK_VALID**. Indicates presence of **lclk** clock input on this board.
- **CLKS_IN_XRM_GCLK_M2C_VALID**. Indicates presence of **xrm_gclk_m2c** clock input on this board.
- **CLKS_OUT_XRM_MGTCLK_C2M_VALID**. Indicates presence of **xrm_mgtclk_c2m** clock outputput on this board.
- **REF_CLK_FREQ_HZ**. The frequency in Hz of the reference clock input used by the target FPGA design.

GPIO Definitions

- **XRM_GPIO_VALID**. Indicates the presence of the XRM GPIO interface on this board.
- **XRM_GPIO_WIDTH**. Indicates the width of the XRM GPIO interface on this board.
- **XRM_MGT_WIDTH**. Indicates the width of the XRM MGT interface on this board.
- **PN4_GPIO_VALID**. Indicates the presence of the Pn4 GPIO interface on this board.
- **PN4_GPIO_WIDTH**. Indicates the width of the Pn4 GPIO interface on this board.
- **PN6_GPIO_VALID**. Indicates the presence of the Pn6 GPIO interface on this board.
- **PN6_GPIO_WIDTH**. Indicates the width of the Pn6 GPIO interface on this board.
- **PN6_MGT_WIDTH**. Indicates the width of the Pn6 MGT interface on this board.

On-Board Memory Definitions

- **DDR3_VALID**. Indicates the presence of DDR3 SDRAM on this board.
- **DDR3_BANKS**. Indicates the number of banks of DDR3 SDRAM on this board.
- **DDR3_BANK_ROW_WIDTH**. Indicates the width of the DDR3 SDRAM row address interface on this board.
- **DDR3_BANK_DATA_WIDTH**. Indicates the width of the DDR3 SDRAM data interface on this board
- **DDR3_BYTE_ADDR_WIDTH**. Indicates the width of the DDR3 SDRAM byte address interface on this board.
- **DDR3_16_BYTE_ADDR_WIDTH**. Indicates the width of the DDR3 SDRAM 16-byte address interface on this board.
- **MEM_VALID**. Indicates the presence of on-board memory on this board.
- **MEM_BANKS**. Indicates the number of banks of on-board memory on this board.

MPTL 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_SER_WIDTH**. Indicates the width of the MPTL serial data interface that exists on this board.
- **std_logic_dbl_t**. Type defining a general-purpose differential std_logic signal.
- **mptl_pins_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.

6.3.3 ADB3 Target Package (adb3_target_pkg)

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

Function definitions

- ds_base_conv.
- ds_mask_conv.
- dma_base_conv.
- dma_mask_conv.
- mask_vec_width.
- make_defined.
- make_defined_s2m.

Component definitions

- **mptl_if_target_wrap**
- **mptl_if_target_sim**
- **mptl64par_interface_target_6tl_top**
- **mptl128_interface_target_6t1_top**
- **mptl64par_interface_target_6tl**
- **mptl128_interface_target_6t1**

6.3.4 ADB3 Target Testbench Package (adb3_target_tb_pkg)

The package **adb3_target_tb_pkg** defines functions, procedures, and components which relate to target example FPGA testbenches.

Function definitions

- conv_byte_vector.
- conv_byte_enable.
- conv_vector.
- conv_string_hex.
- conv_string.

Procedure definitions

- adb3_target_sim_read_reg32.
- adb3_target_sim_read_reg64.
- adb3_target_sim_read.
- adb3_target_sim_read_cmd.
- adb3_target_sim_read_resp.
- adb3_target_sim_write_reg32.
- adb3_target_sim_write_reg64.
- adb3_target_sim_write.
- adb3_wait_cycles.

Component definitions

- **mptl_if_bridge_wrap**
- **mptl_if_bridge_sim**
- **mptl64par_interface_bridge_6tl_top**
- **mptl64par_interface_bridge_6tl**
- **mptl128_interface_bridge_6t1_top**
- **mptl128_interface_bridge_6t1**

6.4 ADB3 Probe Library

The ADB3 Probe library is located in the **hdl/vhdl/common/adb3_probe/** directory and contains the following elements:

- **ADB3 probe library package (adb3_probe_pkg)**
- **ADB3 probe library components**

6.4.1 ADB3 Probe Library Package (adb3_probe_pkg)

The package **adb3_probe_pkg** defines constants and types which are used by the ADB3 probe library components.

6.4.2 ADB3 Probe Library Components

6.4.2.1 adb3_ocp_transaction_probe

6.4.2.1.1 Introduction

This is a component in the ADB3 probe library. Its function is to monitor an OCP channel and produce warnings/errors if specific conditions occur. It is used by target example FPGA testbenches.

6.4.2.1.2 Interface

The adb3_ocp_transaction_probe component interface is shown in **Figure 32** below and described in **Table 84**.

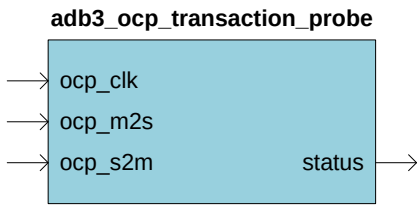

Figure 32: ADB3 Probe Library adb3_ocp_transaction_probe Component Interface

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 84: ADB3 Probe Library adb3_ocp_transaction_probe Component Interface

6.4.2.1.3 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 DataByteEn value.
- Invalid command detection.
- Invalid address detection (**addr_width_max** generic).
- Invalid burst length detection (**data_burst_max** generic).
- Non-valid response detection.

6.5 Memory Interface Library

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

- **Memory interface library package (mem_if_pkg)**
- **Memory interface library components**

6.5.1 Memory Interface Library Package (mem_if_pkg)

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

Defininitions 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 physical interface types

- **mem_addr_out_t**. A record type containing address elements for all memory banks (outputs).
- **mem_ctrl_out_t**. A record type containing bank control elements for all memory banks (outputs).
- **mem_data_inout_t**. A record type containing data elements for all memory banks (bi-dir).
- **mem_clk_out_t**. A record type containing clock elements for all memory banks (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.

DDR3 SDRAM MIG V3.6 core types

- **mig_v3_6_clocks_t**. A record type containing MIG core clock generic elements.
- **mig_v3_6_common_t**. A record type containing MIG core bank generic elements.
- **mig_v3_6_bank01_t**. A record type containing MIG core bank 0 and 1 generic elements.
- **mig_v3_6_bank2_t**. A record type containing MIG core bank 2 generic elements.
- **mig_v3_6_bank3_t**. A record type containing MIG core bank 3 generic elements.

Component definitions

- **ddr3_if_bank_v3_6**

6.5.2 Memory Interface Library Components

6.5.2.1 DDR3 SDRAM Memory Interface Bank (ddr3_if_bank_v3_6)

6.5.2.1.1 Introduction

This is a component in the memory interface library. Its function is to convert on-board memory bank OCP channel transactions to DDR3 SDRAM MIG core user interface transactions and instantiate a single bank Xilinx™ DDR3 SDRAM MIG core.

6.5.2.1.2 Interface

The ddr3_if_bank_v3_6 component interface is shown in **Figure 33** below and described in **Table 85**.

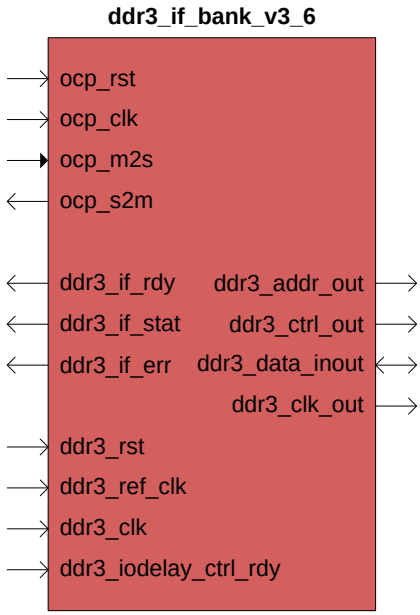

Figure 33: Memory Interface Library ddr3_if_bank_v3_6 Component Interface

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.

Table 85: Memory Interface Library ddr3_if_bank_v3_6 Component Interface (continued on next page)

Signal	Type	Description
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 85: Memory Interface Library ddr3_if_bank_v3_6 Component Interface

6.5.2.1.3 Description

This component converts on-board memory bank OCP channel transactions to DDR3 SDRAM MIG core user interface transactions and instantiates a single bank Xilinx™ DDR3 SDRAM MIG core. It is implemented in **ddr3_if_bank_v3_6.vhd** which is located in **hdl/vhdl/common/mem_if/ddr3_sdram/**. It includes the following components:

- OCP to DDR3 SDRAM MIG core (**adb3_ocp_ocp2ddr3_nb**)
- Xilinx™ DDR3 SDRAM MIG core

6.5.2.1.3.1 OCP To DDR3 SDRAM MIG Core (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 library component **adb3_ocp_ocp2ddr3_nb**.

6.5.2.1.3.2 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 **Section 6.5.2.1.4** for details of the generation procedure.

> **Note:** Currently version 3.6 of the Xilinx™ DDR3 SDRAM MIG core is supported. This is available in ISE version 12.3 or 12.4.

The component instantiated depends on the bank selected by the **bank** generic. For example **c0_memc_ui_top.vhd** located in **hdl/vhdl/common/mem_if/ddr3_sdram/mig_v3_6/rtl/ip_top/** is used for bank 0.

6.5.2.1.4 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** script. Examples are as follows:

To generate HDL files for Virtex-6 6VLX240T devices using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\common\mem_if\ddr3_sdram\mig_v3_6
gen_mem_if.bat 6vlx240t
```

To generate HDL files for Virtex-6 6vsx315t devices using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/common/mem_if/ddr3_sdram/mig_v3_6
./gen_mem_if.bash 6vsx315t
```

For further information, refer to the Xilinx™ documentation included with the generated Xilinx™ DDR3 SDRAM MIG core. After generation of the core, the documentation can be found in **hdl/vhdl/common/mem_if/ddr3_sdram/mig_v3_6/mig_temp/mig_v3_6/docs/**.

The VHDL source files can be found in **hdl/vhdl/common/mem_if/ddr3_sdram/mig_v3_6/rtl/**.

6.6 Memory Application Library

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

- **Memory application library components**

6.6.1 Memory Application Library Components

6.6.1.1 Memory Test Block (blk_mem_test)

6.6.1.1.1 Introduction

This is a component in the memory application library. Its function is to generate test stimulus, and analyse test responses on a single ADB3 OCP channel.

6.6.1.1.2 Interface

The blk_mem_test component interface is shown in **Figure 34** below and described in **Table 86**.

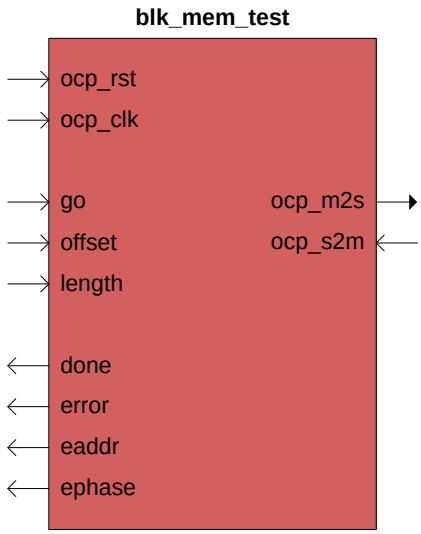

Figure 34: Memory Application Library blk_mem_test Component Interface

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.
		OCP Port
ocp_rst	Input	OCP asynchronous reset.
ocp_clk	Input	OCP clock.

Table 86: Memory Application Library blk_mem_test Component Interface (continued on next page)

Signal	Type	Description
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 86: Memory Application Library blk_mem_test Component Interface

6.6.1.1.3 Description

TBD

6.7 Memory Model Library

The Memory model library is located in the **hdl/vhdl/common/mem_tb/** directory and contains the following elements:

- **DDR3 SDRAM memory model**

6.7.1 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**

6.7.1.1 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.

Defininitions 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.

Supported speed_grade_cl_cwl_t constants

- **MT41J_187E_CL_CWL_MIN**. Micron MT41J64M16_187E (minimum values).
- **MT41J_187E_CL_CWL_MAX**. Micron MT41J64M16_187E (maximum values).

Supported speed_grade_t constants

- **MT41J_187E**. Micron MT41J64M16_187E.

Supported part_t constants

- **MT41J64M16_187E**. Micron MT41J64M16_187E (1Gib part).
- **MT47J128M16_187E**. Micron MT47J128M16_187E (2Gib part).

Component definitions

- **ddr3_sdram**

6.7.1.2 DDR3 SDRAM Model Components

6.7.1.2.1 DDR3 SDRAM Model (ddr3_sdram)

6.7.1.2.1.1 Introduction

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

6.7.1.2.1.2 Interface

The ddr3_sdram component interface is shown in **Figure 35** below and described in **Table 87**.

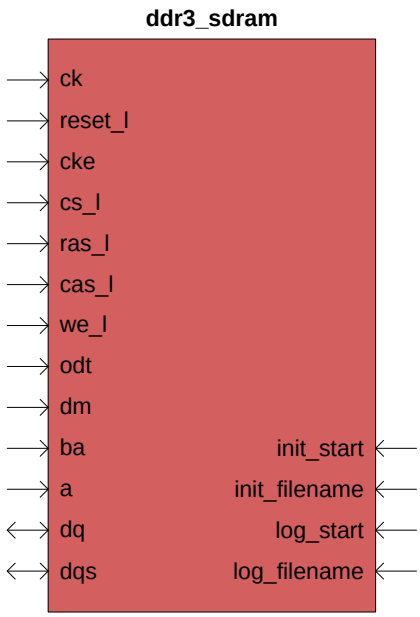

Figure 35: Memory Model Library ddr3_sdram Component Interface

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).
we_l	Input	Write enable (active low).
odt	Input	On-die termination.
dm	Input	Input data mask.

Table 87: Memory Model Library ddr3_sdram Component Interface (continued on next page)

Signal	Type	Description
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 87: Memory Model Library ddr3_sdram Component Interface

6.7.1.2.1.3 Description

TBD

6.7.1.2.1.3.1 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.

6.7.1.2.1.3.2 Part Selection

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

6.7.1.2.1.3.3 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'

6.7.1.2.1.3.4 Memory Contents Initalisation

Loading of data from a file into the model is initiated by a 'true' value on the **init_start** input signal.

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).
- 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
```

6.7.1.2.1.3.5 Memory Contents Logging

Saving of data to a file from the model is initiated by a 'true' value on the **log_start** input signal. Only memory data that has been modified is output to the log file.

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).
- 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
```

6.8 Clock Frequency Measurement Library

The clock frequency measurement library is located in the **hdl/vhdl/examples/uber/common/** directory and contains the following elements:

- **Clock frequency measurement library components**

6.8.1 Clock Frequency Measurement Library Components

6.8.1.1 Clock Frequency Measurement Block (blk_clock_freq)

6.8.1.1.1 Introduction

This is a component in the clock frequency measurement library. Its function is to count the number of edges present on a sample clock in a measurement period.

6.8.1.1.2 Interface

The blk_clock_freq component interface is shown in **Figure 36** below and described in **Table 88**.

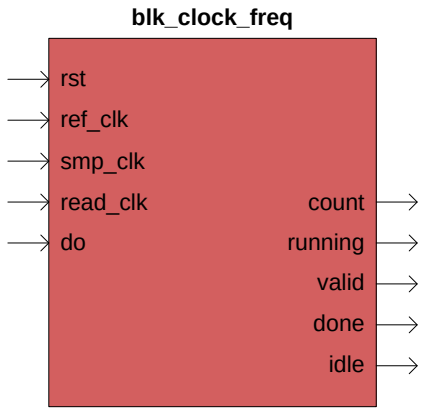

Figure 36: Clock Frequency Measurement Library blk_clock_freq Component Interface

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 88: Clock Frequency Measurement Library blk_clock_freq Component Interface

6.8.1.1.3 Description

TBD

6.8.1.1.3.1 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**.

6.9 ChipScope™ Library

The ChipScope™ library is located in the **hdl/vhdl/common/ChipScope™/** directory and contains the following elements:

- **Xilinx™ ChipScope™ interface (ICON/ILA)**
- **ChipScope™ library components**

6.9.1 Xilinx™ ChipScope™ Interface (ICON/ILA)

Prior to the initial bitstream build of a design using a Xilinx™ ChipScope™ interface, its .NGC files will need to be generated using the **gen_ChipScope™** script. Examples are as follows:

To generate .NGC files for Virtex-6 6VLX240T devices using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\common\ChipScope™
gen_ChipScope™.bat 6vlx240t
```

To generate .NGC files for a Virtex-6 6VSX315T device using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/common/ChipScope™
./gen_ChipScope™.bash 6vsx315t
```

Once generated, the Xilinx™ ChipScope™ interface .NGC files are located in **hdl/vhdl/common/ChipScope™/cgp/**.

6.9.2 ChipScope™ Library Components

6.9.2.1 ChipScope™ Block (blk_ChipScope™)

6.9.2.1.1 Introduction

This is a component in the ChipScope™ library. Its function is to instantiate up to 3 Xilinx™ ChipScope™ interfaces, each connected to an ADB3 OCP channel.

6.9.2.1.2 Interface

The **blk_ChipScope™** component interface is shown in **Figure 37** below and described in **Table 89**.

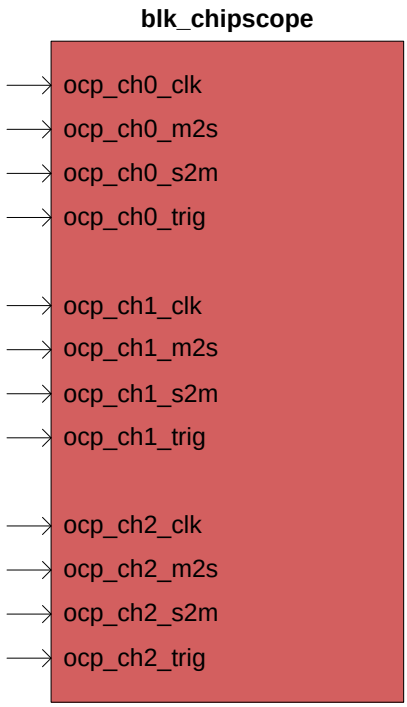

Figure 37: ChipScope™ Library blk_ChipScope™ Component Interface

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.

Table 89: ChipScope™ Library blk_ChipScope™ Component Interface (continued on next page)

Signal	Type	Description
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 89: ChipScope™ Library blk_ChipScope™ Component Interface

6.9.2.1.3 Description

For each ChipScope™ channel, a Xilinx™ **chipscope_ila** component is instantiated with connections as follows (when **instantiate** = true):

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** component is also instantiated (when **instantiate** = true).

6.9.2.2 ChipScope™ Simulation Block (blk_chipscope_sim)

6.9.2.2.1 Introduction

This is a component in the ChipScope™ library. Its function is to instantiate a simulation only version of the **blk_ChipScope™** component.

6.9.2.2.2 Interface

This component's interface is the sames as the **blk_ChipScope™** component. Refer to **Figure 37** and **Table 89**.

6.9.2.2.3 Description

Signals are generated as for the **blk_ChipScope™** component, but no **chipscope_ila** and **chipscope_icon** components are instantiated.

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 (Cmd) signal however the definition of other sideband signals is fairly flexible. The main groupings of signals used in the ADB3 OCP protocol are a Command Group, synchronous to the Cmd signal, and Data transfer groups both from Master to Slave (Write) and Slave to Master (Read Response). Each of these groupings is acknowledged independently allowing the flow to be controlled.

The MPTL interface provides the user with a bank of OCP ports through which the data is passed as Read or Write transactions.

- Master Port - instigates all transfers, can have multiple requests active at any one time if the slave can also handle multiple requests.
- Slave Port - responds to Master request, does not instigate any requests

The MPTL Interface in the User FPGA provides an OCP Master Port for direct reads and writes from the Host via Bars 2/3 and 4/5 (64 bit bars) and a Master Port for each DMA engine in the Bridge.

Each OCP Link operates as follows:

1) The Master Port outputs a command along with the address, byte enables and burst length for the transaction.
2) The Slave port responds by accepting the command (and the other information).
3) For write transactions:
   I) The Master Port outputs the data to be written along with a data valid flag.
   II) The Slave Port accepts the data as and when it is able to. Responding with a data accept flag for each data transfer.
   III) Once all the data has been transferred the Master may start the next transaction.

   For Read transactions:
   I) The Slave Port retrieves the data that has been requested.
   II) The Slave Port outputs the data as and when it is available along with a data valid flag.
   III) The Master Port accepts the data . Responding with a data accept flag for each data transfer.
   IV) Once all the data has been transferred the Master Port is free to start the next transaction.

All OCP ports operate independently and with multiple DMA engines the user can instigate multiple data streams into and out of the application design.

For advanced systems where the user application has a requirement for direct access from the Application to the Host an MPTL interface can be provided that has an extra Slave Port. This allows the application to make memory access requests direct to the Host System.

7.1.2 ADB3 OCP Signal Definitions

Signal	Group	Type	Description
Cmd	Command	OCP Cmd	Idle,Write or Read
Addr	Command	64 bit std_logic_vector	Address
BurstLength	Command	12 bit std_logic_vector	Length of transfer
Data	Data	128 bit std_logic_vector	Write Data to Slave
DataByteEn	Data	16 bit std_logic_vector	Byte Enables for Data
DataValid	Data	std_logic	Qualifier for Data
RespAccept	Response	std_logic	Flow Control for response
Tag	Command	8 bit std_logic_vector	Tag for Read response data

Table 90: ADB3 OCP Master Signals

Signal	Group	Type	Description
CmdAccept	Command	std_logic	Flow Control for commands
DataAccept	Data	std_logic	Flow Control for write Data
Data	Response	128 bit std_logic_vector	Response Data to Master
Resp	Response	OCP Resp	Qualifier for Response Data
Tag	Response	8 bit std_logic_vector	Tag for Read response data

Table 91: ADB3 OCP Slave Signals

7.1.3 Example OCP Transfer Waveform Diagrams

This section contains timing diagrams for most common transactions and highlight the main operation of the protocol.

> **Note:** These waveforms show different transfer sequences, all are valid OCP requests. This is to show the different timing sequences of commands and data transfers. **Figure 42** shows how a single OCP slave port handles the two different write requests as shown in **Figure 38**.

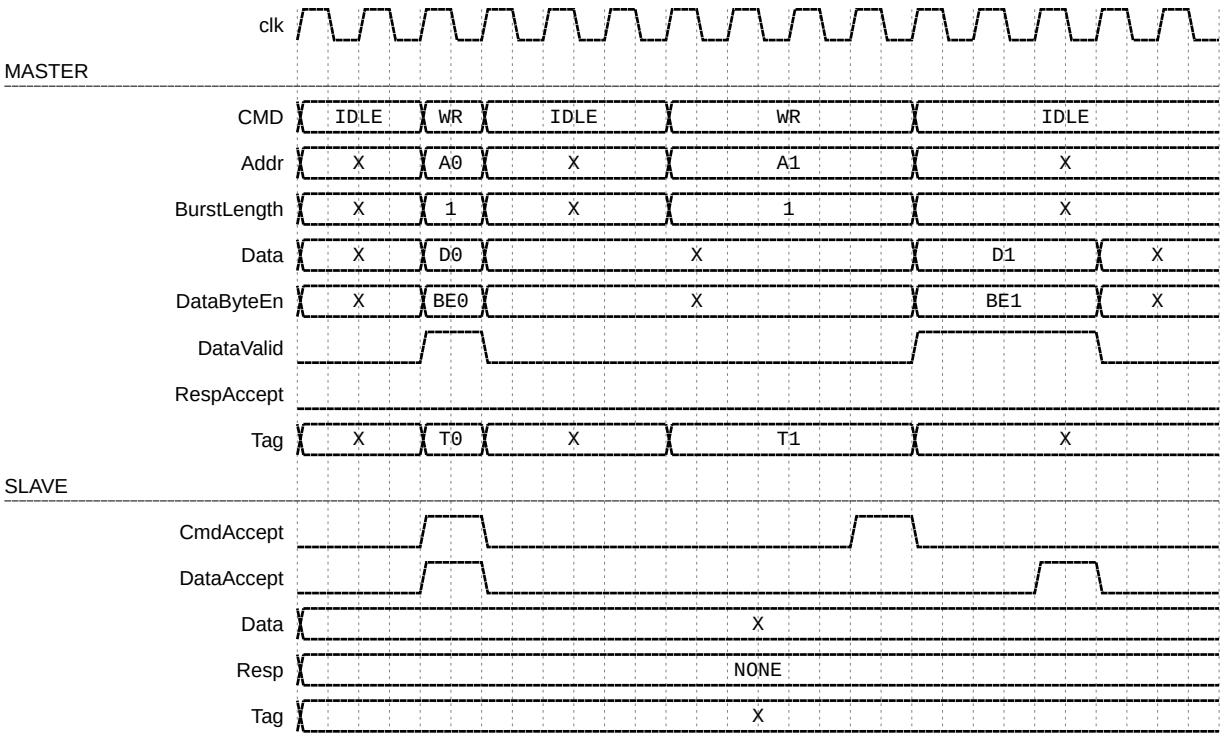

Figure 38: Single Beat Write

Figure 38 shows 2 single beat write commands. The address, burst length and tag are all presented at the same time as the Cmd is set to Write. The Cmd is acknowledged within 1 clock cycle in the first case and so the Cmd is returned to Idle after a single clock cycle. In the first case, the Data and Byte Enables are asserted and accepted also in the same clock cycle. In the second case, the Write command is not accepted until the 4th cycle after it is asserted (possible due to the Slave being busy). The master in this case also does not assert the Data Valid signal until after the Cmd. The data accept is also not accepted immediately and therefore the Data Valid must remain high until the data beat is accepted. All these cases constitute legal OCP transfers with the protocol.

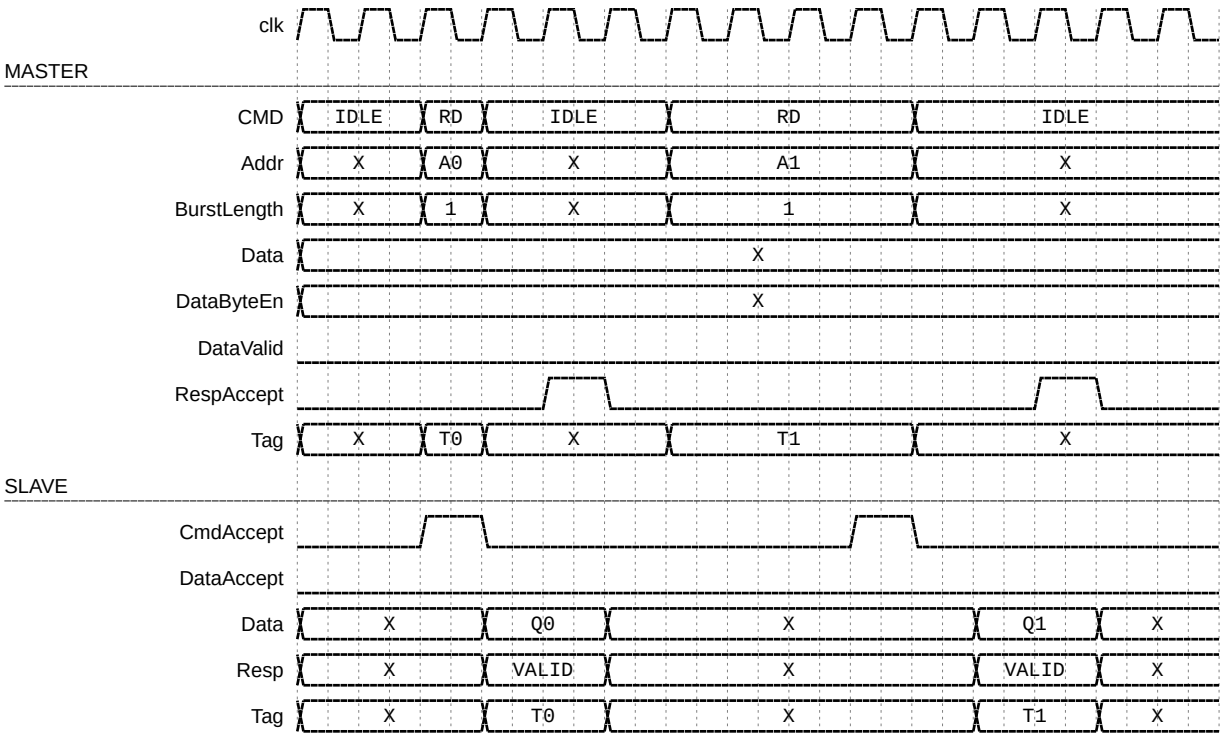

Figure 39: Single Beat Read

Figure 39 shows 2 single beat read commands. in the first case the read request is immediately accepted. The slave responds with a response (Q0) on the following clock cycle. The Tag send with the read command is returned with the response. The second example shows a delayed command accept, a delayed response and a delayed response accept, all of which are legal with the protocol.

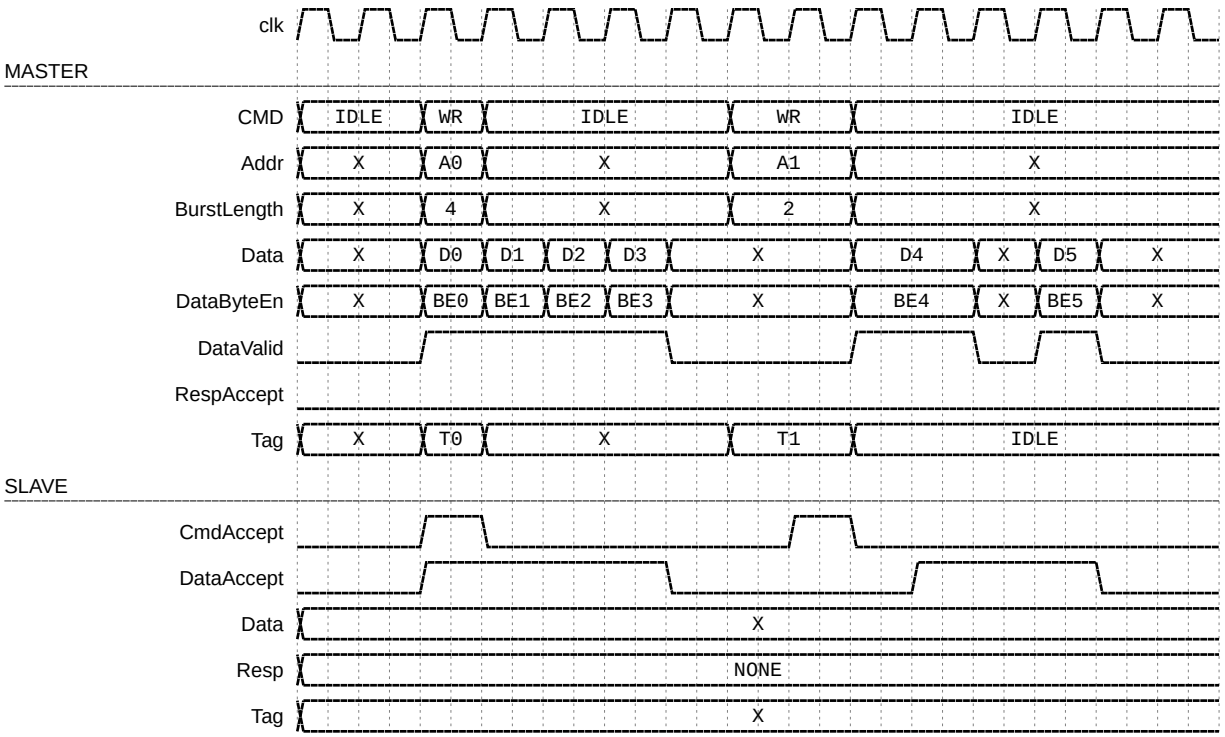

Figure 40: Burst Write

Figure 40 shows 2 burst writes. A single command is issued for multiple data word transfers. The command protocol operates in exactly the same manner as for single beat transfers. Multiple data transfers occur for each command. Data transfers only occur when both DataValid and DataAccept are asserted. The master must wait on DataAccept being asserted before presenting the next data word. The slave must check that DataValid is asserted when receiving data. The slave may assert DataAccept even if DataValid is not asserted.

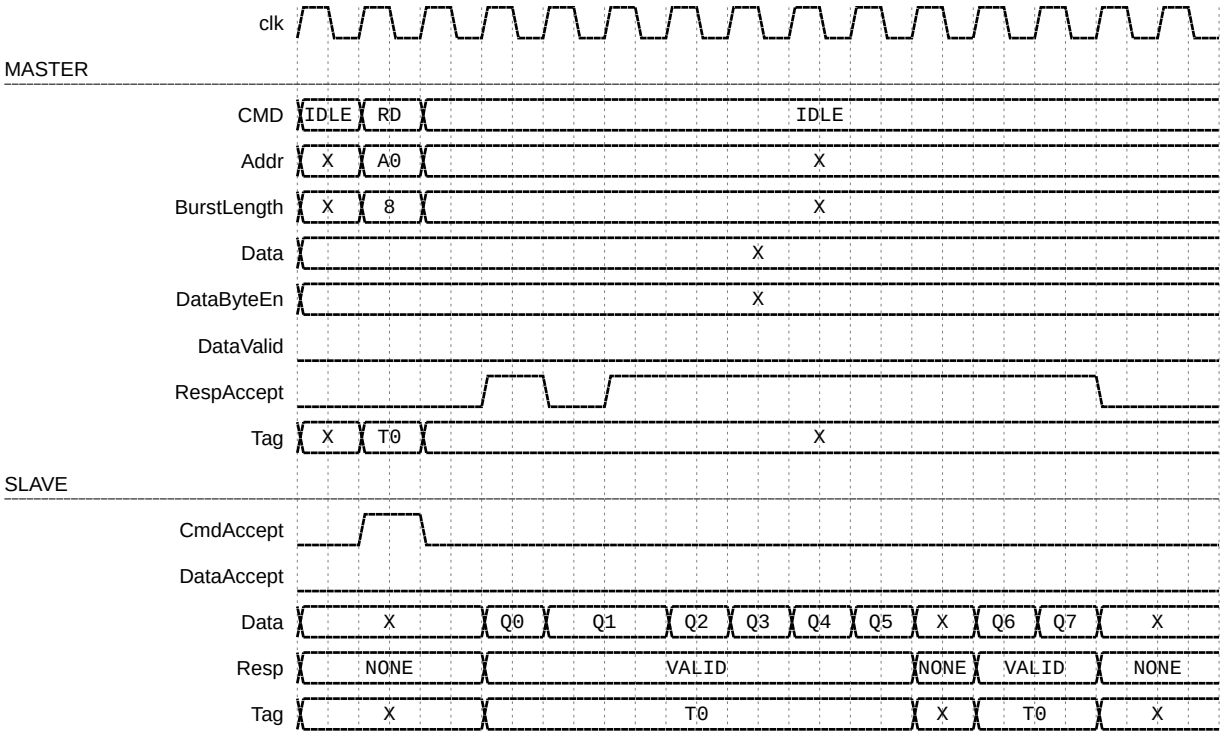

Figure 41: Burst Read

Figure 41 shows a read burst. The response should be held valid and the read tag returned by the slave for all data transfers. Each data transfer required the Response to be Valid and RespAccect to be asserted.

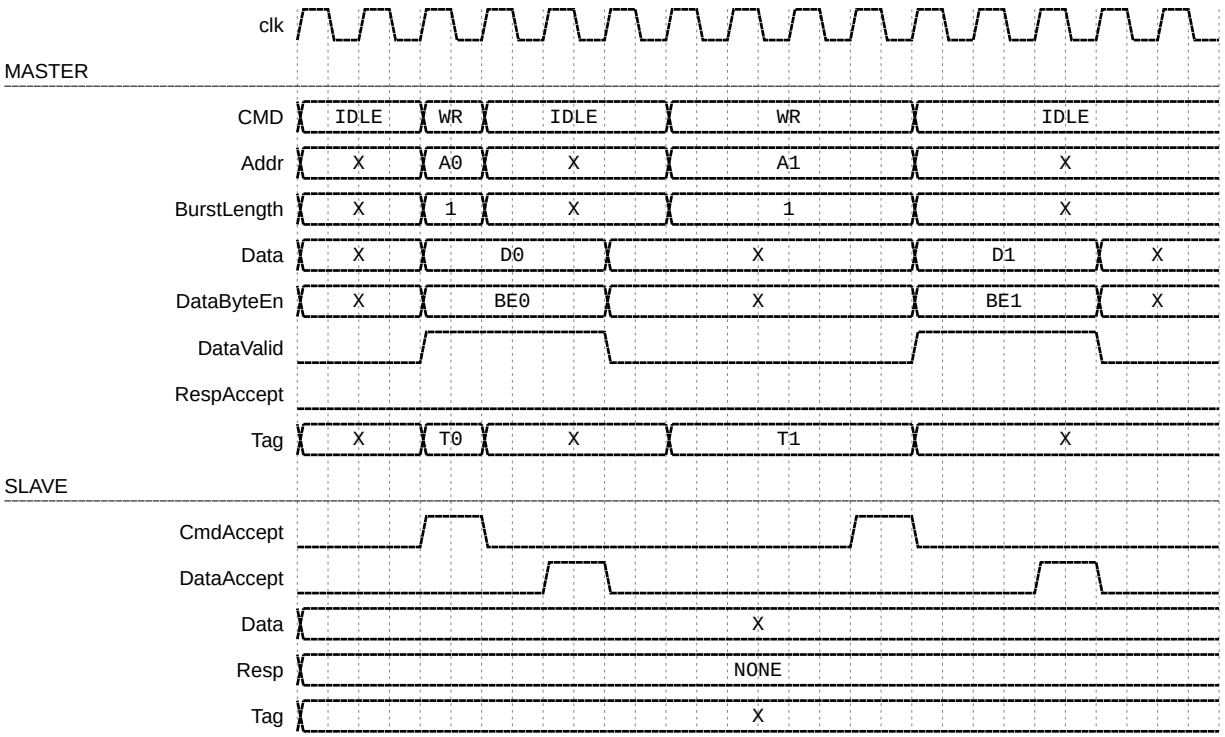

Figure 42: OCP Slave Controlled Transfers

Figure 42 shows the OCP slave port delaying accepting the write data until it has accepted the write command, notice how the OCP master port must keep the data valid during this time. Compare this to the original sequence in **Figure 38**.

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**.

Page Intentionally left blank.

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.

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