DesignWare DW_apb Databook

DW_apb – Product Code

2.03a
June 2014

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com

Contents

Revision History

This section tracks the significant documentation changes that occur from release-to-release and during a release from version 1.02d onward.

Version	Date	Description
2.03a	June 2014	■ Version change for 2014.06a release ■ Updated “Performance” section in the “Integration Considerations” chapter ■ Corrected Default Input/Output Delays in Signals chapter
2.02c	May 2013	■ Version change for 2013.05a release ■ Updated the template
2.02b	Oct 2012	Added the product code on the cover and in Table 1-1
2.02b	Oct 2011	Version change for 2011.10a release
2.02a	Jun 2011	Updated: ■ Figure 3-14 to reflect current hrdata functionality ■ System diagram in Figure 1-1 ■ “Related Documents” section in Preface.
2.01a	May 2011	Corrected Figures 3-7 and 3-9.
2.01a	Apr 2011	Version change for 2011.03a release.
2.00a	Dec 2010	Version change for 2010.12a release.
1.04a	Sep 2010	■ Corrected names of include files and vcs command used for simulation ■ Included additional information about AMBA 3 APB protocol
1.03a	Dec 2009	Updated databook to new template for consistency with other IIP/VIP/PHY databooks
1.03a	Jul 2009	Enhanced with PRDATA sample timing
1.03a	May 2009	Removed references to QuickStarts, as they are no longer supported
1.03a	Oct 2008	Version change for 2008.10a release
1.02e	Jul 2008	Added “Burst Transfers” subsection
1.02e	Jun 2008	Version change for 2008.06a release

(Continued)

Version	Date	Description
1.02d	Dec 2007	■ Updated for revised installation guide and consolidated release notes titles ■ Changed references of “Designware AMBA” to simply “DesignWare”
1.02d	Jun 2007	Description added under Figure 11

Preface

This databook provides information about the DW_apb, which is an AMBA® version 2.0-compliant Advanced Peripheral Bus component. The DW_apb is a part of the DesignWare Synthesizable Components for AMBA 2. The databook also supplies descriptions of tests used to verify the DW_apb component, synthesis information, and user options unique to the DW_apb.

This databook is intended for designers who plan to use the DW_apb with Synopsys tools and supported third-party simulators. Readers are assumed to be familiar with the *AMBA Specification, Revision 2.0* from ARM.

Organization

The chapters of this databook are organized as follows:

- Chapter 1, "Product Overview" provides a system overview, a component block diagram, basic features, and an overview of the verification environment.
- Chapter 2, "Building and Verifying a Component or Subsystem" introduces you to using the DW_apb within the coreAssembler and coreConsultant tools.
- Chapter 3, "Functional Description" describes the functional operation of the DW_apb.
- Chapter 4, "Parameters" identifies the configurable parameters supported by the DW_apb.
- Chapter 5, "Signals" provides a list and description of the DW_apb signals.
- Chapter 6, "Verification" provides information on verifying the configured DW_apb.
- Chapter 7, "Integration Considerations" includes information you need to integrate the configured DW_apb into your design.
- Appendix A, "DesignWare Constants" includes the contents of the DesignWare Synthesizable Components bus constants file.
- Appendix B, "Glossary" provides a glossary of general terms.

Related Documentation

- *Using DesignWare Library IP in coreAssembler* – Contains information on getting started with using DesignWare SIP components for AMBA 2 and AMBA 3 AXI components within coreTools
- *coreAssembler User Guide* – Contains information on using coreAssembler
- *coreConsultant User Guide* – Contains information on using coreConsultant

To see a complete listing of documentation within the DesignWare Synthesizable Components for AMBA 2, refer to the *Guide to Documentation for DesignWare Synthesizable Components for AMBA 2 and AMBA 3 AXI*.

Web Resources

- DesignWare IP product information: http://www.designware.com
- Your custom DesignWare IP page: http://www.mydesignware.com
- Documentation through SolvNet: http://solvnet.synopsys.com (Synopsys password required)
- Synopsys Common Licensing (SCL): http://www.synopsys.com/keys

Customer Support

To obtain support for your product:

- First, prepare the following debug information, if applicable:
    - For environment setup problems or failures with configuration, simulation, or synthesis that occur within coreConsultant or coreAssembler, use the following menu entry:

        File > Build Debug Tar-file

      Check all the boxes in the dialog box that apply to your issue. This menu entry gathers all the Synopsys product data needed to begin debugging an issue and writes it to the file <*core tool startup directory*>/debug.tar.gz.
    - For simulation issues outside of coreConsultant or coreAssembler:
        - Create a waveforms file (such as VPD or VCD)
        - Identify the hierarchy path to the DesignWare instance
        - Identify the timestamp of any signals or locations in the waveforms that are not understood
- Then, contact Support Center, with a description of your question and supplying the above information, using one of the following methods:
    - *For fastest response*, use the SolvNet website. If you fill in your information as explained below, your issue is automatically routed to a support engineer who is experienced with your product. The **Sub Product** entry is critical for correct routing.

      Go to http://solvnet.synopsys.com/EnterACall and click on the link to enter a call. Provide the requested information, including:
        - **Product:** DesignWare Library IP
        - **Sub Product:** AMBA
        - **Tool Version:** <product version number>
        - **Problem Type:**
        - **Priority:**
        - **Title:** DW_apb
        - **Description:** For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood

      After creating the case, attach any debug files you created in the previous step.

- Or, send an e-mail message to support_center@synopsys.com (your email will be queued and then, on a first-come, first-served basis, manually routed to the correct support engineer):
  - Include the Product name, Sub Product name, and Tool Version number in your e-mail (as identified above) so it can be routed correctly.
  - For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood
  - Attach any debug files you created in the previous step.
- Or, telephone your local support center:
  - North America:
    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:
    http://www.synopsys.com/Support/GlobalSupportCenters

Product Code

Table 1-1 lists all the components associated with the product code for DesignWare AMBA Fabric.

Table 1-1 DesignWare AMBA Fabric – Product Code: 3768-0

Component Name	Description
DW_ahb	High performance, low latency interconnect fabric for AMBA 2 AHB
DW_ahb_eh2h	High performance, high bandwidth AMBA 2 AHB to AHB bridge
DW_ahb_h2h	Area efficient, low bandwidth AMBA 2 AHB to AHB Bridge
DW_ahb_icm	Configurable multi-layer interconnection matrix
DW_ahb_ictl	Configurable vectored interrupt controllers for AHB bus systems
DW_apb	High performance, low latency interconnect fabric & bridge for AMBA 2 APB for direct connect to AMBA 2 AHB fabric
DW_apb_ictl	Configurable vectored interrupt controllers for APB bus systems
DW_axi	High performance, low latency interconnect fabric for AMBA 3 AXI
DW_axi_a2x	Configurable bridge between AXI and AHB components or AXI and AXI components.
DW_axi_gm	Simplify the connection of third party/custom master controllers to any AMBA 3 AXI fabric
DW_axi_gs	Simplify the connection of third party/custom slave controllers to any AMBA 3 AXI fabric
DW_axi_hmx	Configurable high performance interface from and AHB master to an AXI slave
DW_axi_rs	Configurable standalone pipelining stage for AMBA 3 AXI subsystems

Table 1-1 DesignWare AMBA Fabric – Product Code: 3768-0 (Continued)

Component Name	Description
DW_axi_x2h	Bridge from AMBA 3 AXI to AMBA 2.0 AHB, enabling easy integration of legacy AHB designs with newer AXI systems
DW_axi_x2p	High performance, low latency interconnect fabric and bridge for AMBA 2 & 3 APB for direct connect to AMBA 3 AXI fabric
DW_axi_x2x	Flexible bridge between multiple AMBA 3 AXI components or busses

1 Product Overview

This chapter describes the DesignWare APB, which provides a bridge between the AHB bus and a set of APB peripherals.

1.1 DesignWare System Overview

The Synopsys DesignWare Synthesizable Components environment is a parameterizable bus system containing components for the following:

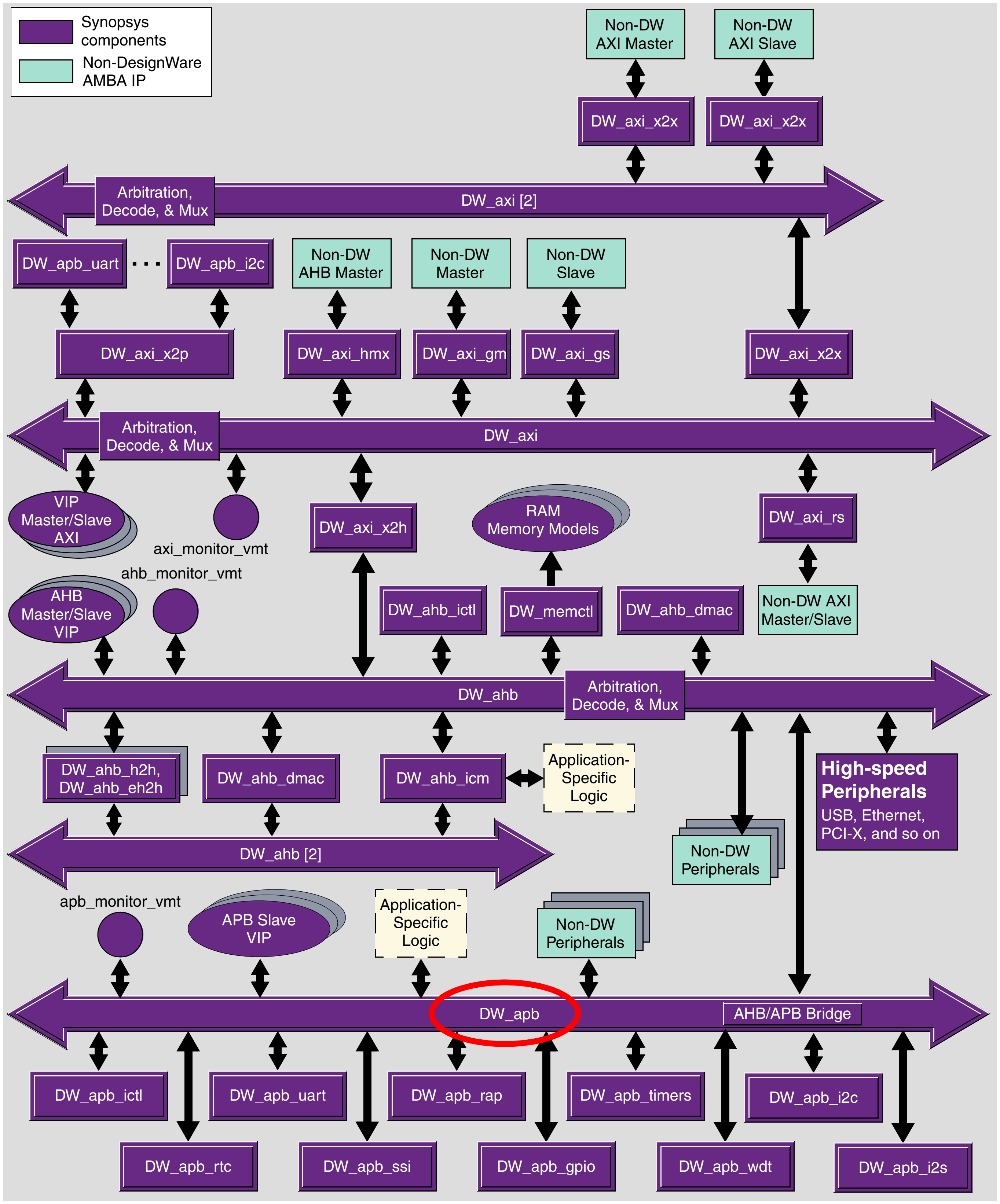
- AMBA version 2.0-compliant AHB (Advanced High-performance Bus)
- AMBA 3 APB version 1.0-compliant APB (Advanced Peripheral Bus)
- AMBA version 3.0-compliant AXI (Advanced eXtensible Interface)

1.1.1 DesignWare System Block Diagram

Figure 1-1 illustrates one example of this environment, including the AXI bus, the AHB bus, and an APB bus. Included in this subsystem are synthesizable IP for AXI/AHB/APB peripherals, bus bridges, and an AXI interconnect and AHB bus fabric. Also included are verification IP for AXI/AHB/APB master/slave models and bus monitors. In order to display the databook for a DW_* component, click on the corresponding component object in the illustration.

Attention Links resolve only if you are viewing this databook from your $DESIGNWARE_HOME tree, and to only those components that are installed in the tree.

Figure 1-1 Example of DW_apb in a Complete System



You can connect, configure, synthesize, and verify the DW_apb within a DesignWare subsystem using coreAssembler, documentation for which is available on the web in the *coreAssembler User Guide*.

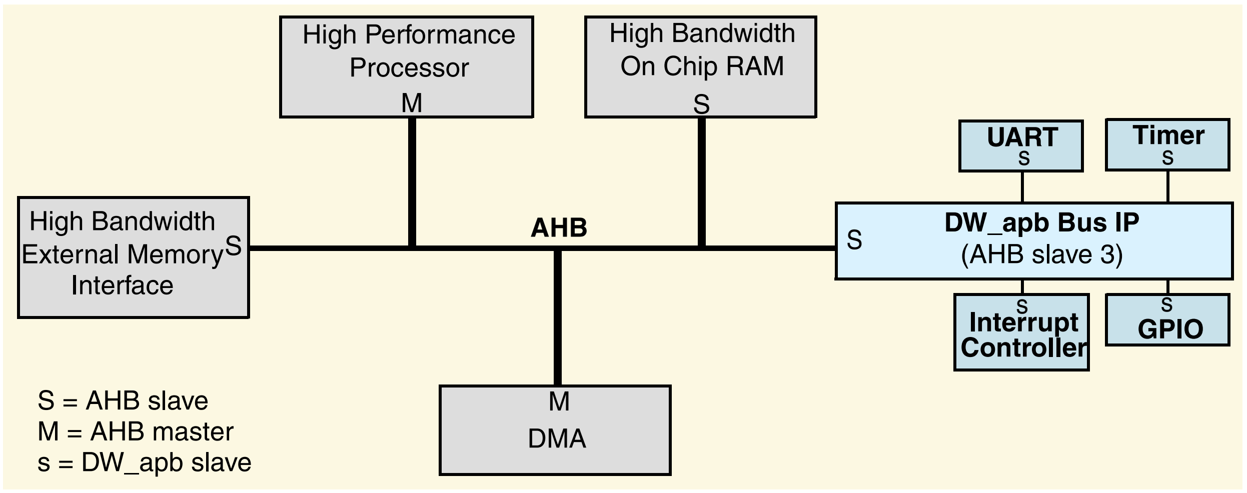
If you want to configure, synthesize, and verify a single component such as the DW_apb component, you might prefer to use coreConsultant, documentation for which is available in the *coreConsultant User Guide*.

1.2 General Product Description

The DW_apb is a parameterizable, synthesizable, and programmable component that implements the APB functionality of the *AMBA 3 APB Specification, Revision 1.0* from ARM.

The DW_apb provides a bridge between the AHB bus and a set of APB peripherals. All communication between masters on the AHB and slaves on the APB pass through the DW_apb. From the point of view of the AHB system, the DW_apb appears as a slave, as illustrated in Figure 1-2.

Figure 1-2 DW_apb in an Example System


High Performance Processor M
High Bandwidth On Chip RAM S
UART s
Timer s
High Bandwidth External Memory Interface S
AHB
S DW_apb Bus IP (AHB slave 3)
s Interrupt Controller
s GPIO
M DMA
S = AHB slave
M = AHB master
s = DW_apb slave

1.3 Features

The DW_apb includes the following features:

- Compliance with the *AMBA Specification, Revision 2.0* from ARM
- Compliance with the *AMBA 3 APB Specification, Revision 1.0* from ARM
- AHB slave

Support for the following:

- Up to 16 APB slaves
- Big- and little-endian AHB systems
- Little-endian APB slaves
- 32, 64, 128, and 256-bit AHB data buses
- 8, 16, and 32-bit APB data buses
- Single and burst AHB transfers

- Synchronous hclk/pclk; hclk is an integer multiple of pclk
- Optional external decoder

1.3.1 Notes and Restrictions

- Slave numbers are configured consecutively—0, 1, 2, 3; not 0, 3, 5, 9.
- All slaves must have their address spaces aligned to a 1 KB boundary.
- Minimum address space allocated to a configured slave is 1 KB.
- There is support for only little-endian APB slaves.
- The APB data bus width must be less than or equal to the AHB data bus width.
- The APB clock must be equal to, or a submultiple of and synchronous to, the AHB clock.

1.3.2 Features Not Supported

The following features are not supported in this release:

- Independent AHB clock (*hclk*) and APB clock (*pclk*) (APB bus must be synchronous with AHB bus)
- No support for the following AHB features when an AHB slave:
  - SPLIT transfers
  - RETRY responses
  - ERROR responses
- Big-endian APB peripherals

Source code for this component is available on a per-project basis as a DesignWare Core. Please contact your local sales office for the details.

1.4 Standards Compliance

The DW_apb component conforms to the *AMBA Specification, Revision 2.0* and *AMBA 3 APB Specification, Revision 1.0* from ARM. Readers are assumed to be familiar with these specifications.

1.5 Verification Environment Overview

The DW_apb includes an extensive verification environment, which sets up and invokes your selected simulation tool to execute tests that verify the functionality of the configured component. You can then analyze the results of the simulation.

The "Verification" on page 59 chapter discusses the specific procedures for verifying the DW_apb.

1.6 Licenses

Before you begin using the DW_apb, you must have a valid license. For more information, refer to "Licenses" in the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide.*

1.7 Where To Go From Here

At this point, you may want to get started working with the DW_apb component within a subsystem or by itself. Synopsys provides several tools within its coreTools suite of products for the purposes of configuration, synthesis, and verification of single or multiple synthesizable IP components—coreConsultant and coreAssembler. For information on the different coreTools, refer to *Guide to coreTools Documentation*.

For more information about configuring, synthesizing, and verifying just your DW_apb component, refer to "Overview of the coreConsultant Configuration and Integration Process" on page 20.

For more information about implementing your DW_apb component within a DesignWare subsystem using coreAssembler, refer to "Overview of the coreAssembler Configuration and Integration Process" on page 23.

2 Building and Verifying a Component or Subsystem

DesignWare Synthesizable IP (SIP) components for AMBA 2 and AMBA 3 AXI are packaged using Synopsys coreTools, which enable the user to configure, synthesize, and run simulations on a single SIP title, or to build a configured AMBA subsystem. You do this by generating a workspace view using one of the following coreTools applications:

- coreConsultant – Used for configuration, RTL generation, synthesis, and execution of packaged verification for a single SIP title. The *coreConsultant User Guide* provides complete information on using coreConsultant.
- coreAssembler – Used for building and configuration of a subsystem that connects multiple SIP titles, RTL generation, synthesis, and creation of a template subsystem testbench. The *coreAssembler User Guide* provides complete information on using coreAssembler.

A workspace is your working version of a DesignWare SIP component or subsystem. In fact, you can create several workspaces to experiment with different design alternatives.

Hint If you are unfamiliar with coreTools—which is comprised of the coreAssembler, coreConsultant, and coreBuilder tools—you can go to *Using DesignWare Library IP in coreAssembler* to “get started” learning how to work with DesignWare SIP components.

2.1 Setting up Your Environment

The DW_apb is included in a release of DesignWare SIP components. It is assumed that you have already downloaded and installed the release. If you have not, you can download and install the latest versions of required tools using the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

You also need to set up your environment correctly using specific environment variables, such as DESIGNWARE_HOME, VERA_HOME, PATH, and SYNOPSYS. If you are not familiar with these requirements and the necessary licenses, refer to “Setting up Your Environment” in the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide.*

2.2 Overview of the coreConsultant Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare SIP components and then set up your environment, you can begin work on the DW_apb using coreConsultant.

2.2.1 coreConsultant Usage

Figure 2-1 illustrates some general directories and files in a coreConsultant workspace.

Figure 2-1 coreConsultant Usage Flow

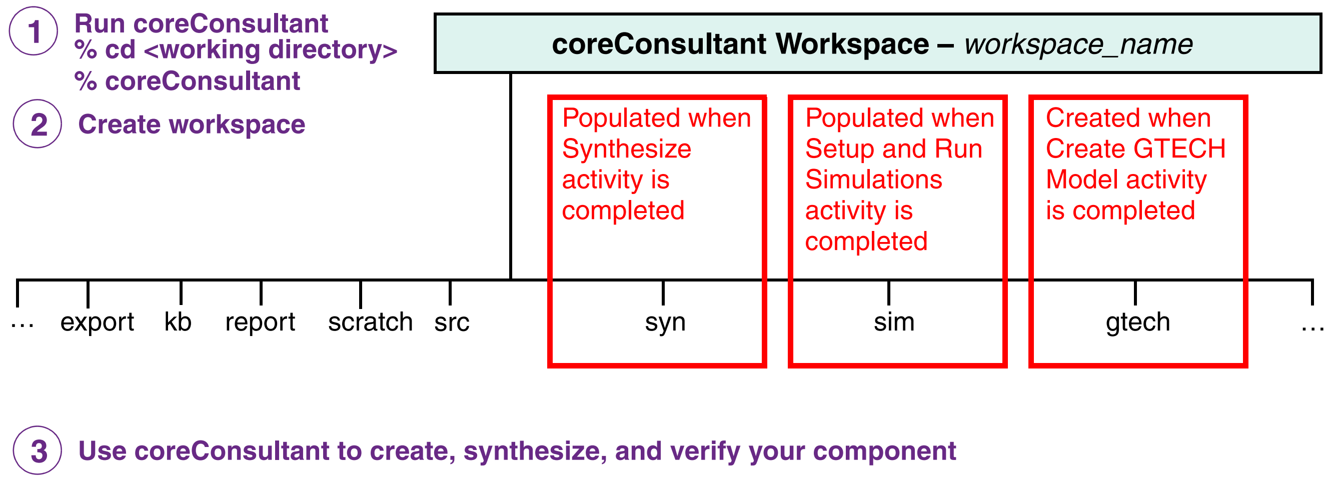

1 Run coreConsultant
% cd <working directory>
% coreConsultant
2 Create workspace
coreConsultant Workspace – workspace_name
Populated when Synthesize activity is completed
Populated when Setup and Run Simulations activity is completed
Created when Create GTECH Model activity is completed
…
export
kb
report
scratch
src
syn
sim
gtech
…
3 Use coreConsultant to create, synthesize, and verify your component

Table 2-1 provides a description of the implementation workspace directory and subdirectories.

Table 2-1 coreConsultant Implementation Workspace Directory Contents

Directory/Subdirectory	Description
auxiliary	Scripts and text files used by coreConsultant. Generated upon first creating workspace.
doc	Contains local copies of component-specific databooks. Generated upon first creating workspace.
export	Contains files used to integrate results from the completed source configuration and synthesis activities into your design (outside coreConsultant). Generated upon first creating workspace; populated during Specify Configuration activity.
gtech	Contains synthesis scripts and output netlists from gtech generation; also used for RTL simulation of encrypted source code. Generated during Generate GTECH Model activity.
kb	Contains knowledge base information used by coreConsultant. These are binary files containing the state of the design. Generated upon first creating workspace; populated and updated throughout activities.

Table 2-1 coreConsultant Implementation Workspace Directory Contents (Continued)

Directory/Subdirectory	Description
leda	Contains Leda configuration files for the component. Generated upon first creating workspace; updated during Run Leda Coding Checker activity.
pkg	Contains RTL preprocessor scripts. Generated during Specify Configuration activity.
report	Contains all of the reports created by coreConsultant during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated upon first creating workspace; populated and updated throughout activities.
scratch	Contains temp files used during the coreConsultant processes. Generated upon first creating workspace; populated and updated throughout activities.
sim	Contains test stimulus and output files. Generated upon first creating workspace; updated during Setup and Run Simulations activity.
src	Includes the top-level RTL file, *design_name*.v. If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated upon first creating workspace; populated during Specify Configuration activity.
syn	Contains synthesis files for the component. Generated upon first creating workspace; updated during Synthesis activity and Formal Verification activity.
tcl	Contains synthesis intent scripts. Generated upon first creating workspace.

For details on some key files created during coreConsultant activities, refer to "Database Files" on page 27.

For information on using coreConsultant, refer to the *coreConsultant User Guide*.

2.2.2 Configuring the DW_apb within coreConsultant

The "Parameters" chapter on page 49 describes the DW_apb hardware configuration parameters that you configure using the coreConsultant GUI.

The "Creating the RTL View of a Core" chapter in the *coreConsultant User Guide* discusses how to specify a configuration for an individual component like the DW_apb.

2.2.3 Creating Gate-Level Netlists within coreConsultant

The "Creating the Gate-Level Netlist for a Core" chapter in the *coreConsultant User Guide* discusses how to create a translation of the RTL view into a technology-specific netlist for an individual component like the DW_apb.

2.2.4 Verifying the DW_apb within coreConsultant

The "Verification" chapter on page 59 provides an overview of the testbench available for DW_apb verification using the coreConsultant GUI.

The "Verifying Your Implementation" chapter in the *coreConsultant User Guide* discusses how to simulate an individual component like the DW_apb.

2.2.5 Running Leda on Generated Code with coreConsultant

When you select **Verify Component > Run Leda Coding Checker** from the Activity List, the corresponding Activity View appears. In this Activity View you select rules configuration file and define Leda command line switches.

2.3 Overview of the coreAssembler Configuration and Integration Process

Once you have correctly downloaded and installed a release of DesignWare SIP components and then set up your environment, you can begin work on your DesignWare subsystem with coreAssembler.

2.3.1 coreAssembler Usage

Figure 2-2 illustrates some general directories and files in a coreAssembler workspace.

Figure 2-2 coreAssembler Usage Flow

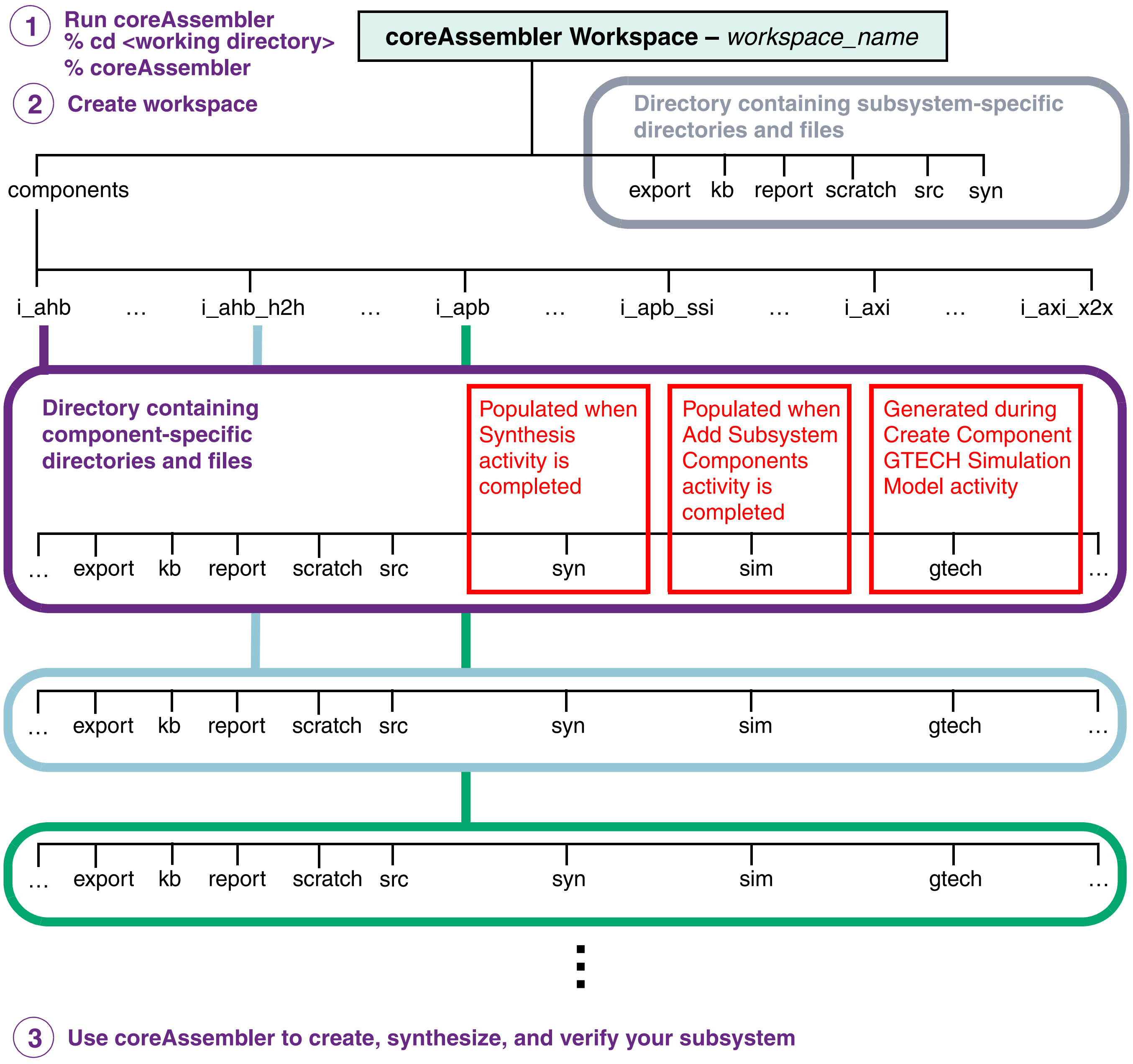

1
Run coreAssembler
% cd <working directory>
% coreAssembler
2
Create workspace
coreAssembler Workspace – *workspace_name*
Directory containing subsystem-specific directories and files
export
kb
report
scratch
src
syn
components
i_ahb
…
i_ahb_h2h
…
i_apb
…
i_apb_ssi
…
i_axi
…
i_axi_x2x
Directory containing component-specific directories and files
Populated when Synthesis activity is completed
Populated when Add Subsystem Components activity is completed
Generated during Create Component GTECH Simulation Model activity
…
export
kb
report
scratch
src
syn
sim
gtech
…
3
Use coreAssembler to create, synthesize, and verify your subsystem

Table 2-2 provides a description of the implementation workspace directory and subdirectories.

Table 2-2 coreAssembler Implementation Workspace Directory Contents

Directory/Subdirectory	Description
components	Contains a directory for each IP component instance connected in the subsystem. Generated and populated with separate component directories upon first adding components; populated and updated throughout activities.
i_*component*/auxiliary	Scripts and text files used by coreAssembler. Generated during Add Subsystem Components activity.
i_*component*/doc	Contains local copies of component-specific databooks. Generated during Add Subsystem Components activity.
i_*component*/export	Contains files used to integrate results from the completed source configuration and synthesis activities into your design (outside coreAssembler). Generated during Add Subsystem Components activity; populated during Configure Components activity.
i_*component*/gtech	Contains synthesis scripts and output netlists from gtech generation; also used for RTL simulation of encrypted source code. Generated during Create Component GTECH Simulation Model activity.
i_*component*/kb	Contains knowledge base information used by coreAssembler. These are binary files containing the state of the design. Generated during Add Subsystem Components activity; populated and updated throughout activities.
i_*component*/leda	Contains Leda configuration files for the component. Generated during Add Subsystem Components activity; populated during Run Leda Coding Checker (for /i_*component*) activity.
i_*component*/pkg	Contains RTL preprocessor scripts. Generated during Configure Components activity.
i_*component*/report	Contains all of the reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated during Add Subsystem Components activity; populated and updated throughout activities.
i_*component*/scratch	Contains temp files used during the coreAssembler processes. Generated during Add Subsystem Components activity; populated and updated throughout activities.
i_*component*/sim	Contains test stimulus and output files. Generated during Add Subsystem Components activity; updated during Setup and Run Simulations (for /i_*component*) activity.

Table 2-2 coreAssembler Implementation Workspace Directory Contents (Continued)

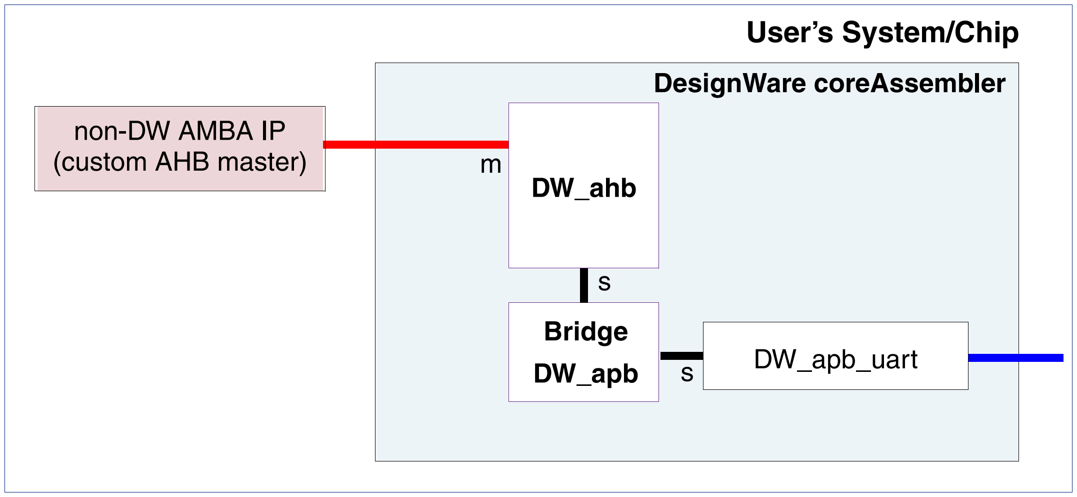
Directory/Subdirectory	Description
i_*component*/src	Includes the top-level RTL file, *design_name*.v. If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated during Add Subsystem Components activity; populated during Specify Configuration activity.
i_*component*/syn	Contains synthesis files for the component. Generated during Add Subsystem Components activity; updated during Synthesis activity.
i_*component*/tcl	Contains synthesis intent scripts. Generated during Add Subsystem Components activity.
export	Contains subsystem files used to integrate the results from the completed source configuration and synthesis activities into your design (outside coreAssembler). Generated upon first creating workspace; populated starting with Memory Map Specification activity.
kb	Contains subsystem knowledge base information used by coreAssembler. These are binary files containing the state of the design. Generated upon first creating workspace; populated and updated throughout activities.
report	Contains subsystem reports created by coreAssembler during build, configuration, test and synthesis phases. An index.html file in this directory links to many of these generated reports. Generated upon first creating workspace; populated and updated throughout activities.
scratch	Contains subsystem temp files used during the coreAssembler processes. Generated upon first creating workspace; populated and updated throughout activities.
src	Includes the RTL related to the subsystem. If you have a source license, this will contain plain-text RTL; if you only have a designware license, this will contain encrypted RTL. Generated upon first creating workspace; populated starting with Generate Subsystem RTL activity.
syn	Contains synthesis files for the subsystem. Generated upon first creating workspace; updated during Synthesize activity and Formal Verification activity.

For details on some key files created during coreAssembler activities, refer to "Database Files" on page 27.

For information on using coreAssembler, refer to the *coreAssembler User Guide*. For information on getting started with using DesignWare SIP components for AMBA 2 and AMBA 3 AXI components within coreTools, refer to *Using DesignWare Library IP in coreAssembler*.

Figure 2-3 illustrates the DW_apb in a simple subsystem.

Figure 2-3 DW_apb in Simple Subsystem



🟥 manually exported interfaces to non-DesignWare IP
🟦 automatically exported interfaces from DesignWare peripheral IP

The subsystem in Figure 2-3 contains the following components that you may want to use as you learn to use coreAssembler:

- DW_apb
- DW_ahb
- DW_apb_uart
- AHB Master

The AHB Master is meant to be exported out of the design and then replaced by a real AHB Master—such as a CPU—later in the design process; at least one exported AHB master is required in a subsystem if you intend to do a basic simulation that tests connections.

2.3.2 Configuring the DW_apb within a Subsystem

The "Parameters" chapter on page 49 describes the DW_apb hardware configuration parameters that you configure using the coreAssembler GUI. Corresponding databooks for the other components in a subsystem contain "Parameters" chapters that describe their respective configuration parameters.

The "Creating the RTL View of a Subsystem" chapter in the *coreAssembler User Guide* discusses how to configure subsystem components and automatically connect them using the coreAssembler GUI.

2.3.3 Creating Gate-Level Netlists within coreAssembler

The "Creating the Gate-Level Netlist for a Subsystem" chapter in the *coreAssembler User Guide* discusses how to create a translation of the RTL view into a technology-specific netlist for a subsystem.

2.3.4 Verifying the DW_apb within coreAssembler

The "Verification" chapter on page 59 provides an overview of the testbench available for DW_apb verification using the coreAssembler GUI.

The "Verifying Subsystems and Components" chapter in the *coreAssembler User Guide* discusses how to simulate a subsystem.

2.3.5 Running Leda on Generated Code with coreAssembler

When you select **Verify Component > Run Leda Coding Checker for /i_*component*)** from the Activity List, the corresponding Activity View appears. In this Activity View you select rules configuration file and define Leda command line switches.

2.4 Database Files

The following subsections describe some key files created in coreConsultant and coreAssembler activities.

2.4.1 Design/HDL Files

The following sections describe the design and HDL files that are produced by coreConsultant and coreAssembler when configuring and verifying a DesignWare Synthesizable Component. The following files are created in different directories by coreConsultant and coreAssembler:

- coreConsultant – *workspace*/ directory
- coreAssembler – *workspace*/components/i_*component*/ directory

2.4.1.1 RTL-Level Files

The following table describes the RTL files that are generated by the Create RTL activity. They are encrypted except where otherwise noted. Any Synopsys synthesis tool or simulator can read encrypted RTL files.

Table 2-3 RTL-Level Files

Files	Encrypted?	Purpose
./src/*component*_cc_constants.v	No	Includes definitions and values of all configuration parameters that you have specified for the component.

./src/*component*.v	No	Top-level HDL file. Include the DesignWare libraries by using the following options in your simulator invocation:
		`+libext+.v+.V`
		`-y ${SYNOPSYS}/packages/gtech/src_ver`
		`-y ${SYNOPSYS}/dw/sim_ver`

./src/*component_submodule*.v	Yes	Sub-modules of component
./src/*component*_constants.v	No	Includes the constants used internally in the design.

Table 2-3 RTL-Level Files (Continued)

Files	Encrypted?	Purpose
./src/*component*_undef.v		Includes an undef for each of the definitions found in the *component*_cc_constants.v file; compiled in after the last file listed in ./src/*components*.lst when compiling multiple instances of the same IP.
./src/*component*.lst	No	Lists the order in which the RTL files should be read into tools, such as simulators or dc_shell. For example, use the following option to read the design into VCS: `vcs +v2k -f component.lst`

2.4.1.2 Simulation Model Files

The following table includes files generated for the component during the Generate GTECH Simulation activity. These files are needed when you are using a non-Synopsys simulator (when you can not use the encrypted RTL).

Table 2-4 Simulation Model Files

Files	Encrypted?	Purpose

./gtech/final/db/*component*.v	No	Simulation model of the component for use with non-Synopsys simulators. A technology-independent, gate-level netlist; VHDL and Verilog versions are generated. Include the DesignWare libraries by using the following options in your simulator invocation:
		`+libext+.v+.V`
		`-y ${SYNOPSYS}/packages/gtech/src_ver`
		`-y ${SYNOPSYS}/dw/sim_ver`

2.4.2 Synthesis Files

The following table includes files generated after the Create Gate-Level Netlist activity is performed on a component.

Table 2-5 Synthesis Files

Files	Encrypted?	Purpose
./syn/auxScripts	No	Auxiliary files for synthesis.
./syn/final/db/*component*.db	Binary format	Synopsys .db files (gate level) that can be read into dc_shell for further synthesis, if desired.
./syn/final/db/*component*.v	No	Gate-level netlist that is mapped to technology libraries that you specify.
./syn/constrain/script/*.*	No	Constraint files for the components.
./syn/final/report/*.*	No	Synthesis result files.

2.4.3 Verification Reference Files

Files described in the following table include information pertaining to the component's operation so that you can verify installation and configuration of the component has been successful. These files are not for re-use during system-level verification.

Table 2-6 Verification Reference Files

Files	Encrypted?	Purpose
./sim/runtest	No	Perl script that runs the Setup and Run Simulations activity from the command line.
./sim/runtest.log	No	The overall result of simulation, including pass/fail results.
./sim/test_*testname*/test.result	No	Pass/fail of individual test.
./sim/test_*testname*/test.log	No	Log file for individual test.

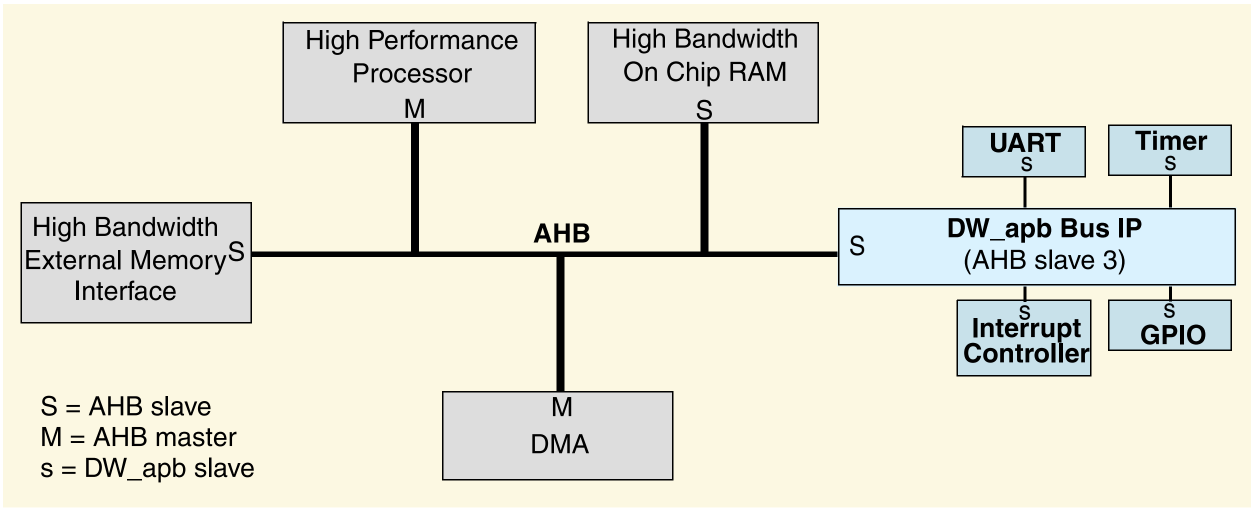
3 Functional Description

The DW_apb is a parameterizable, synthesizable, and programmable component that implements the APB functionality of the *AMBA Specification (Rev. 2.0)*.

3.1 Overview

The DW_apb provides the interconnect fabric to connect AMBA 2.0-compliant or AMBA 3.0-compliant APB peripherals to an AHB bus. It is referred to as the APB Bridge in the *AMBA Specification (Rev. 2.0)* and simply as APB in the *Amba 3.0 APB Protocol Specification (Rev 1.0)*. The bridge is the only master on the APB. From the point of view of the AHB system, the DW_apb appears as a slave, as illustrated in Figure 3-1.

Figure 3-1 DW_apb in an Example System



3.1.1 Block Diagram

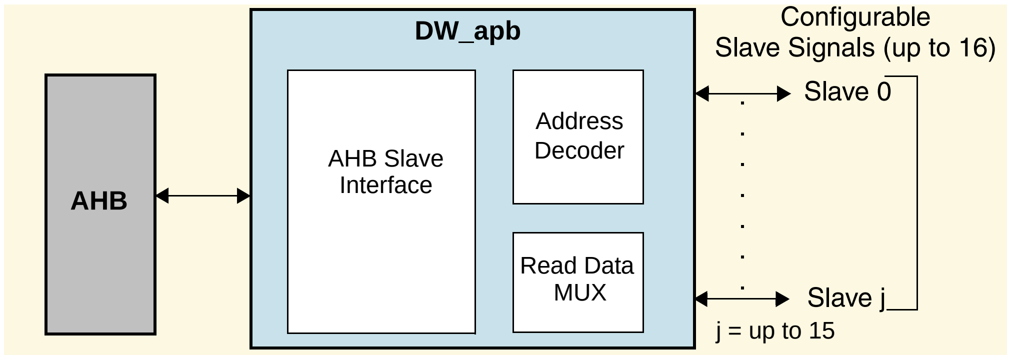
The DW_apb is configurable, synthesizable, and performs the following functions:

- Monitors and responds to AHB transactions for the DW_apb
- Generates APB control, address, and write data signals
- Generates AHB address and APB peripheral select lines

- Frames APB peripheral control signals
- Matches wide AHB write data bus to narrow APB write data bus
- Converts big-endian AHB write data to little-endian APB write data
- Matches narrow APB read data buses to wide AHB read data bus
- Converts little-endian APB read data to big-endian AHB read data

A block diagram is illustrated in Figure 3-2.

Figure 3-2 DW_apb Block Diagram



3.2 Transfers

If an AHB master wants to communicate with an APB slave, it does this by selecting the DW_apb and driving the necessary address, data, and control information to it. The DW_apb presents the data it receives from the APB peripherals onto the AHB data bus. The DW_apb cannot initiate any transfers on the AHB itself; it responds to only requests from AHB masters.

A write transfer on the APB has the address, control, and data signals aligned, unlike the AHB where data and addresses are pipelined. The transfer on the APB takes a minimum of two cycles to complete. A write transfer from the AHB to the APB does not require the AHB system bus to stall until the transfer on the APB has completed. This means a write to the APB can be followed directly by a read from an AHB peripheral (not DW_apb).

While the APB transfer is being aligned, started, and executed, a read from an AHB peripheral can be performed. If the system were held until the write is completed, then for a system with a very slow APB, it would be the APB that would control the system performance. If another write occurs to the APB immediately following the first, the address and control is taken, the instruction is pipelined, and other transfers are stalled by bringing hready low. When the pipeline is cleared, any additional instructions for the APB are then processed. However if the first write transfer targets an AMBA 3 APB slave, the AHB cannot issue any new transfer while the first does not complete on DW_apb.

Regarding reads, once a read is started, it is completed and the AHB bus held (by bringing hready low) until the data is returned from the slave. For more information about read and write transfers to or from the APB, refer to "Timing Diagrams" on page 35.

Note If a transfer is initiated with a BUSY or IDLE transfer, DW_apb ignores the transfer.

3.2.1 Burst Transfers

The DW_apb supports all AHB burst accesses. Since the DW_apb is a relatively simple slave, it processes all AHB beats on a cycle-by-cycle basis. Since an AHB master is required to generate an address for every beat of a burst, the DW_apb can support AHB bursts without internally sampling the hburst signal. The hburst is necessary for only more advanced slaves that do prefetching, cache line fills, and so on.

The hburst input is still included in the DW_apb for I/O signal compatibility with later releases that may include functionality that uses the hburst information.

3.3 PCLK versus HCLK

The DW_apb uses only hclk and pclk_en, and it treats a rising edge of hclk and pclk_en = 1 as an indication of a rising edge on pclk. This means that if pclk_en is active, then the next rising edge on hclk is also a rising edge of pclk. The design of the DW_apb assumes that the clocks hclk and pclk are synchronous; they do not have to be the same frequency. The pclk_en should be generated from an hclk register.

When pclk is the same as hclk, pclk_en must be always high. (The data rate on the APB is half that on the AHB, due to the how the AMBA standard is defined.)

When pclk is not the same as hclk, the data rate on the APB depends on the frequency of the pclk_en signal, which pulses once every *n* hclk cycles. When addresses and data come from an AHB master, they are saved. Only when pclk_en is high are addresses and data presented to the APB slave.

APB peripherals use the pclk signal as the clock, whereas the APB bridge uses hclk and the pclk_en signal in order to gauge pclk in relation to hclk.

Note When pclk is not equal to hclk, prdata is sampled on the first positive hclk edge after assertion of penable, *not* on the first pclk edge after assertion of penable. For more details, refer to the text associated with Figure 3-8 on page 38.

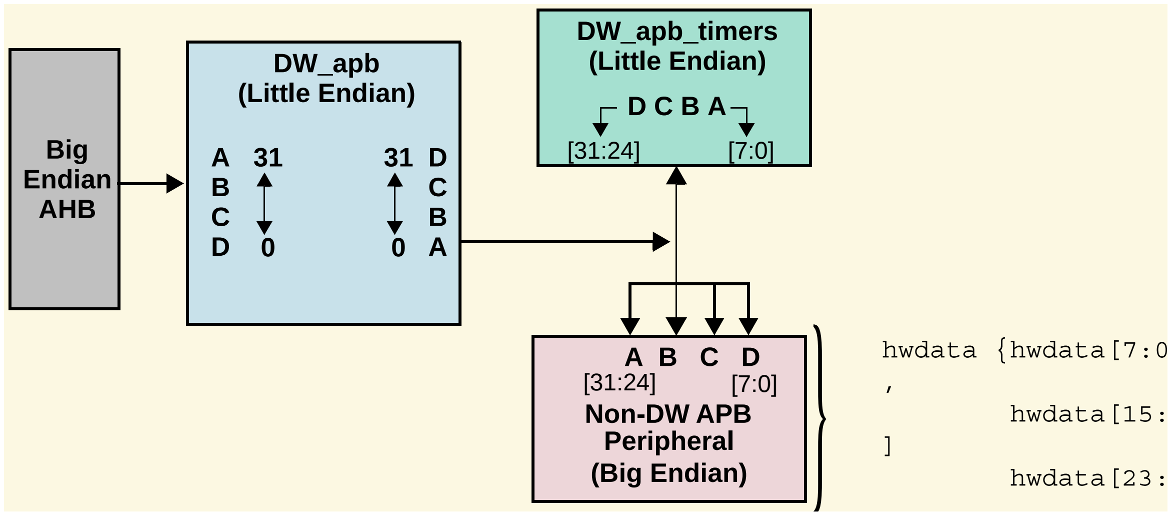
3.4 Optional External Decoder

During configuration of DW_apb, users can choose to have an external decoder. By having the decoder external to DW_apb, users can connect any decoder with any number of remap options. When this option is chosen, the internal decoder is not included. There are inputs for the peripheral selects from the external decoder, which pass though the bridge and drive the peripheral select outputs of DW_apb.

3.5 Endianness

APB slave subsystems are little-endian; the DW_apb performs the conversion from a big-endian AHB to the little-endian APB peripheral by swapping the bytes. However, there is no support for converting a big-endian AHB to a big-endian APB slave peripheral. The user has to manually perform this process by swapping the bytes as illustrated in Figure 3-3.

Figure 3-3 Converting Big-Endian AHB to Big-Endian APB Peripheral

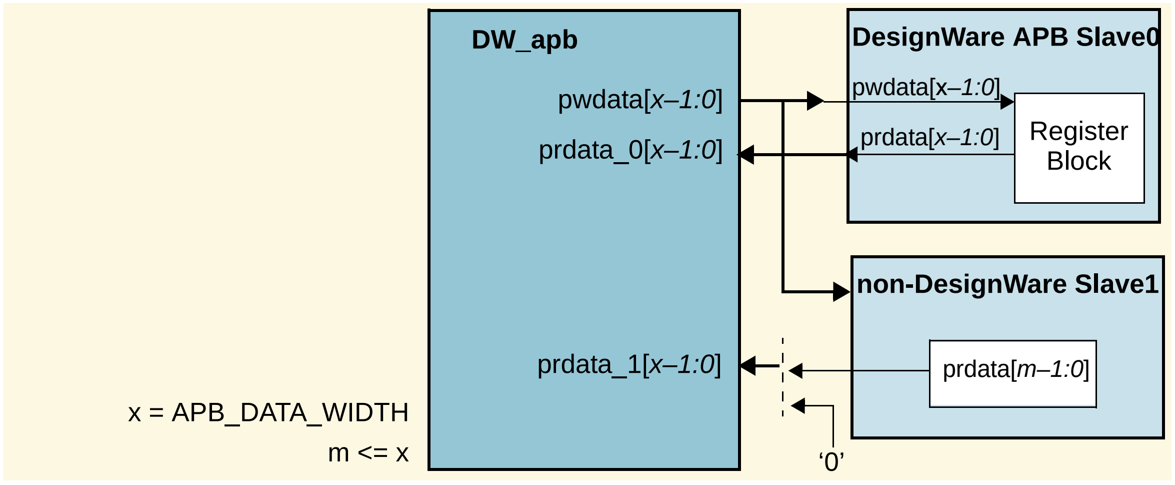


3.6 APB Slave Interface

The DW_apb and DesignWare APB slaves have only one data width for both the read and write APB data buses (APB_DATA_WIDTH). The DW_apb expects each read data bus to be *APB_DATA_WIDTH* bits wide. For non-DesignWare APB slaves, the user must pad the upper bits with zeros to make the bus *APB_DATA_WIDTH* bits wide. No APB slave can have a read data bus width greater than APB_DATA_WIDTH.

Figure 3-4 shows the relationship between DesignWare and non-DesignWare APB slaves.

Figure 3-4 DW_apb and APB Slave Data Widths

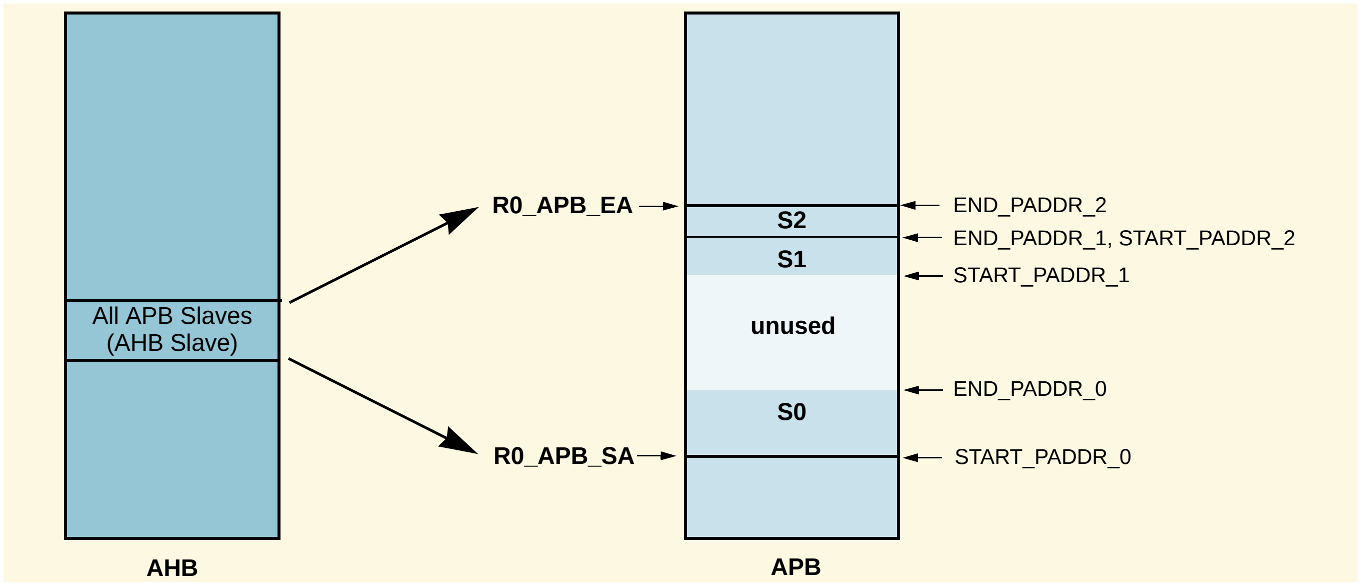


For more information about the APB data width and how it relates to DesignWare and non-DesignWare APB slaves, refer to "Integration Considerations" on page 65.

3.7 Memory Map

Figure 3-5 illustrates a DW_apb memory map for a system with three slaves. Notice that the starting and ending address space (R0_APB_SA, R0_APB_EA) of the APB corresponds to an address space on the AHB for all APB slaves.

Figure 3-5 DW_apb Memory Map



3.8 Backward Compatibility with AMBA 2 APB

The AMBA 3 APB protocol has added the signals ready (pready) and error (pslverr) to the previous protocol. However, APB slaves attached to the DW_apb can support either the AMBA 3 APB or AMBA 2 APB protocol. For each APB slave, you can use the APB_IS_APB3_*j* configuration parameter to specify whether the attached component supports AMBA 3 APB or AMBA 2 APB. This configuration determines whether or not the pready and pslverr signals from that APB slave exist on the I/O of the DW_apb instance. For more information on configuration parameters, refer to "Parameters" on page 49.

3.9 Timing Diagrams

For timing, refer to the following diagrams:

- Read Transfer from AHB to AMBA 2 APB Slave (hclk = pclk): Figure 3-6 on page 36
- Read Transfer from AHB to AMBA 3 APB Slave (hclk = pclk): Figure 3-7 on page 37
- Read Transfer from AHB to AMBA 2 APB Slave(hclk != pclk): Figure 3-8 on page 38
- Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk): Figure 3-9 on page 39
- Write Transfer from AHB to AMBA 2 APB Slave (hclk = pclk): Figure 3-10 on page 40
- Write Transfer from AHB to AMBA 3 APB Slave (hclk = pclk):Figure 3-11 on page 41

- Write Transfer from AHB to AMBA 2 APB Slave (hclk != pclk): Figure 3-12 on page 42
- Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk): Figure 3-13 on page 43
- Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) completed with an error: Figure 3-14 on page 44
- Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) completed with an error: Figure 3-15 on page 45
- Back-to-back write transfer (hclk = pclk): Figure 3-16 on page 46
- Back-to-back write transfer (hclk != pclk): Figure 3-17 on page 47

Figure 3-6 DW_apb Read Transfer from AHB to AMBA 2 APB Slave (hclk = pclk)

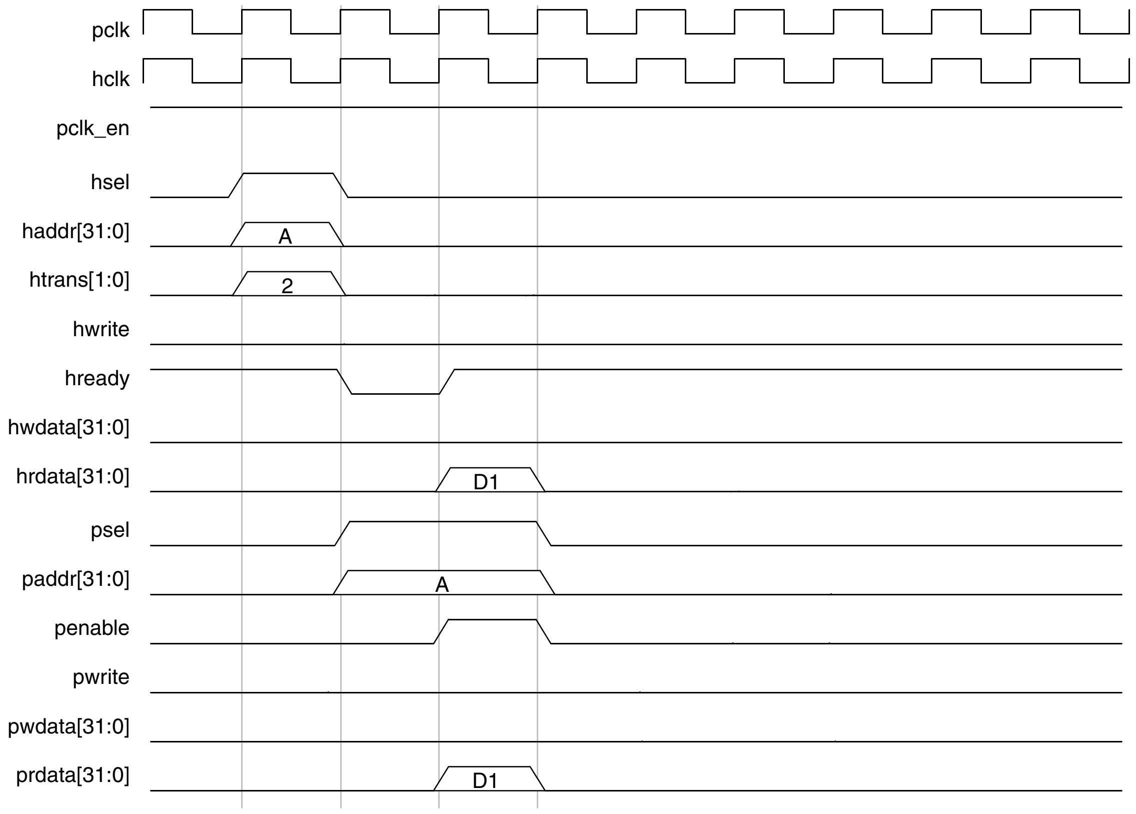


Figure 3-7 DW_apb Read Transfer from from AHB to AMBA 3 APB Slave (hclk = pclk)

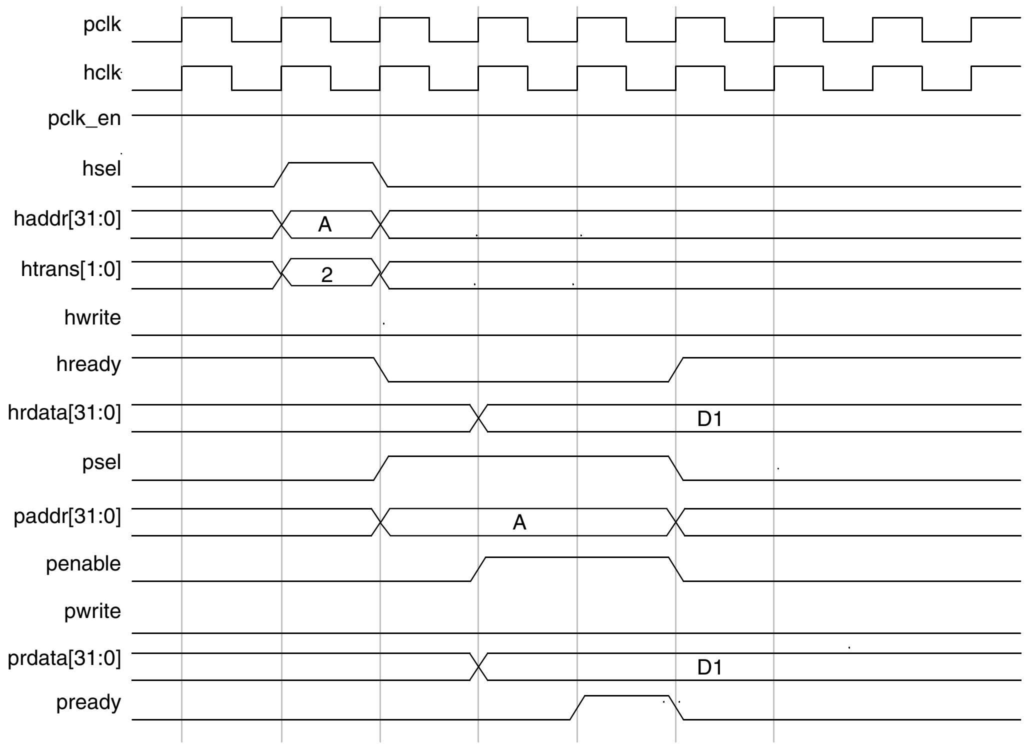
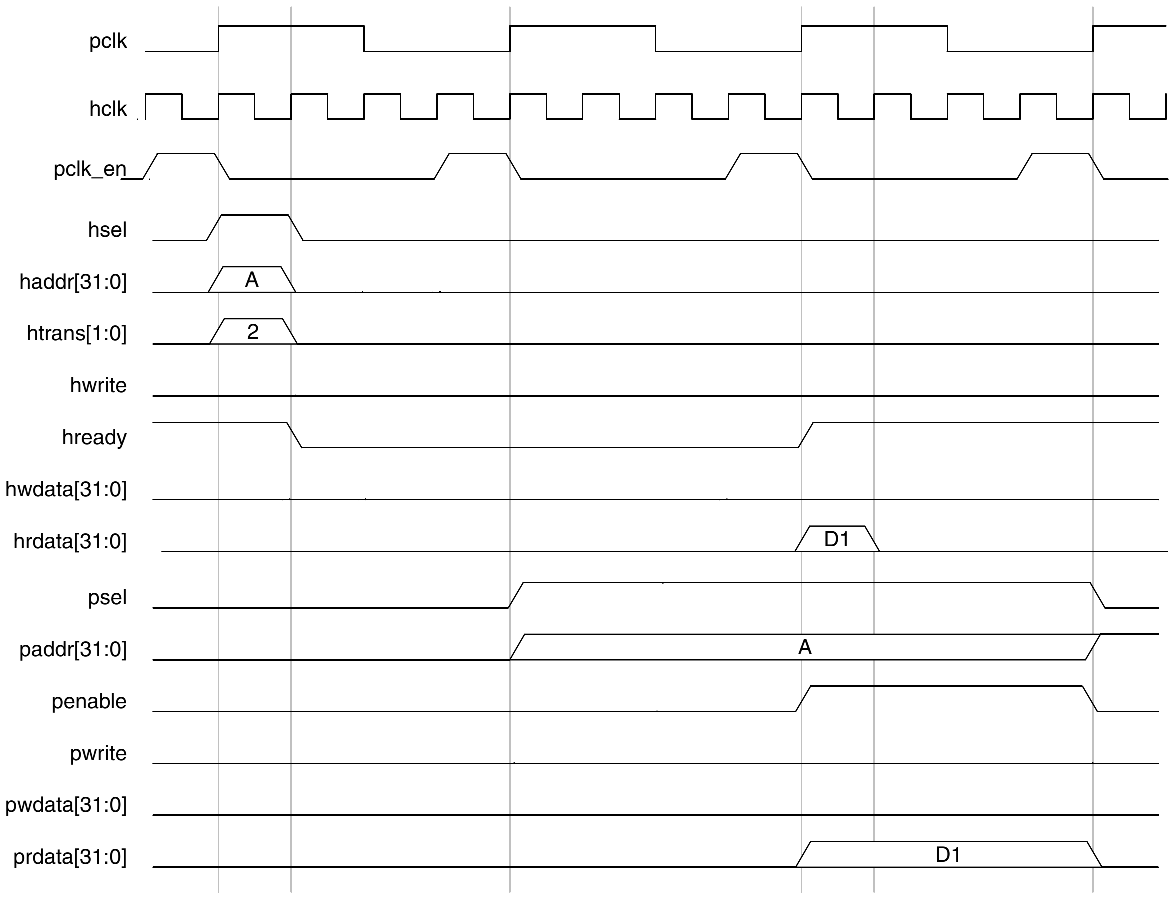


Figure 3-8 DW_apb Read Transfer from AHB to AMBA 2 APB Slave(hclk != pclk)



The DW_apb registers the hready_resp output to prevent long combinatorial paths in the AHB bus system. The pclk_en signal is used to ensure that the hready_resp output can be registered from the DW_apb, regardless of the frequency ratio between hclk and pclk.

This implementation results in read data from the APB slave being sampled by the AHB master one hclk cycle after being driven by the APB slave.

The DW_apb bridge expects the prdata input from the APB slave to be registered. As the prdata input to the DW_apb bridge is driven from a register, it returns the read data to the AHB master before the end of the PENABLE phase without negatively affecting the timing closure of the system.

This architecture results in a high performance AMBA-compliant APB bridge.

The AMBA protocol specification gives designers two choices when interfacing APB and AHB; refer to 5-15 of the *AMBA Specification, Revision 2.0*.

1. Route prdata directly to the AHB (hclk domain).
2. Register prdata at the end of the ENABLE cycle.

Because option 1 does not require a wait state for APB reads, and since prdata is assumed to come from a register, this is the best option for high performance. This is how the DW_apb bridge is implemented.

In systems where pclk is not equal to hclk, this means that prdata is sampled on the first hclk edge after penable is asserted. This requires that the prdata signals from the APB slaves—attached to the prdata_s(j) ports—must be constrained to be stable one hclk after transitioning. This is already taken care of in the packaged synthesis intent of the DW_apb, but is the responsibility of the user to ensure if synthesis is done outside of coreConsultant or coreAssembler.

Figure 3-9 DW_apb Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk)

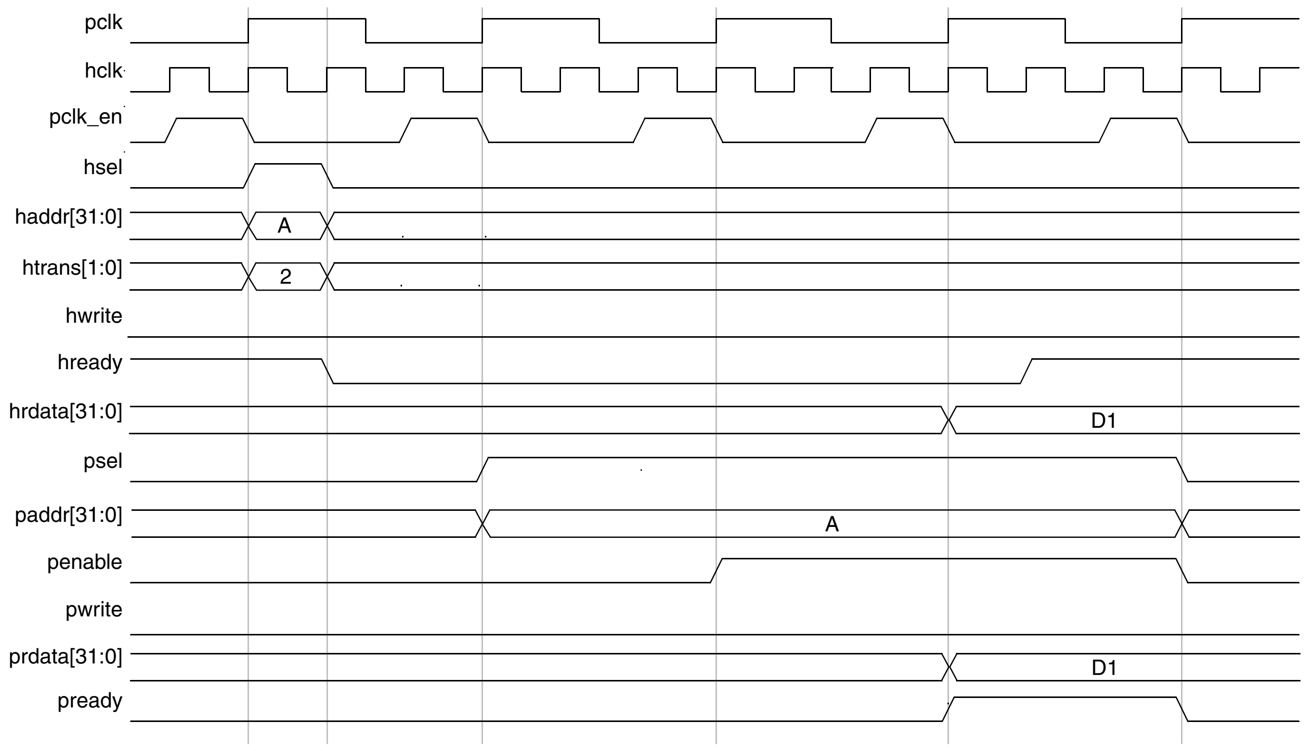


Figure 3-10 DW_apb Write Transfer from AHB to AMBA 2 APB Slave (hclk = pclk)

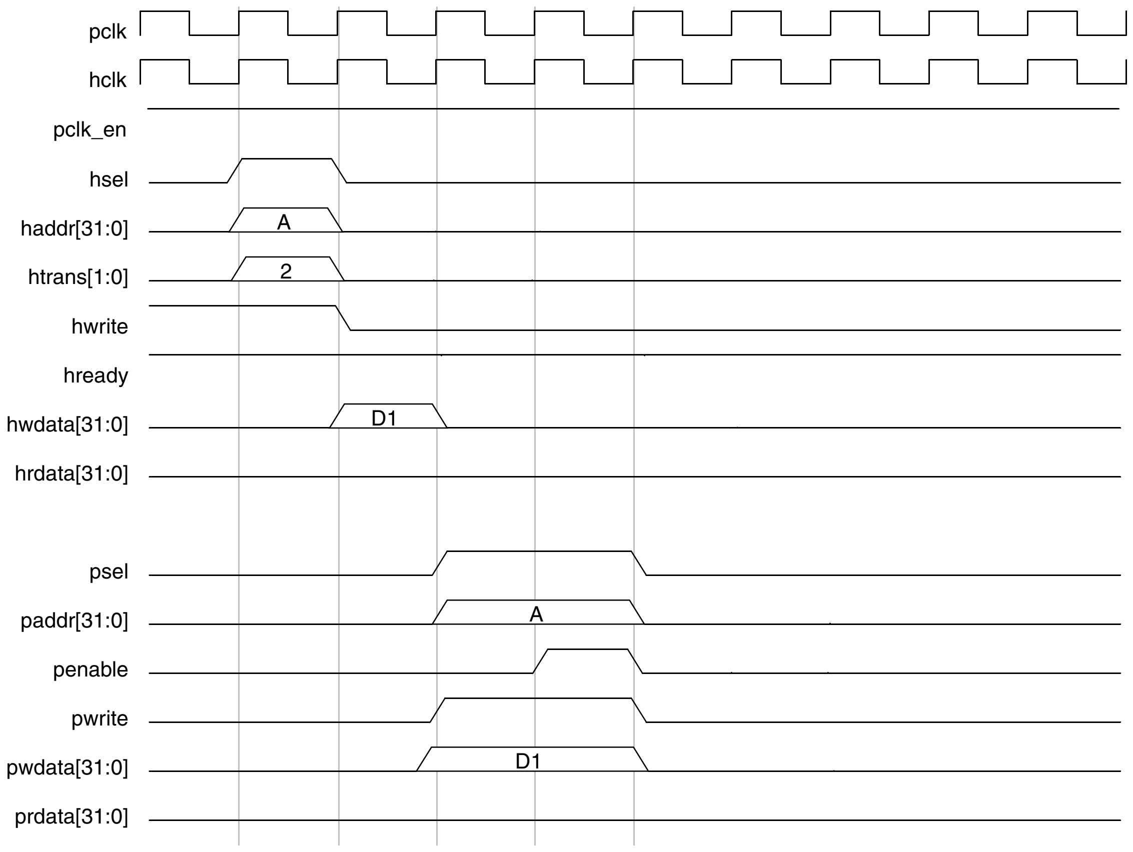


Figure 3-11 DW_apb Write Transfer from AHB to AMBA 3 APB Slave (hclk = pclk)

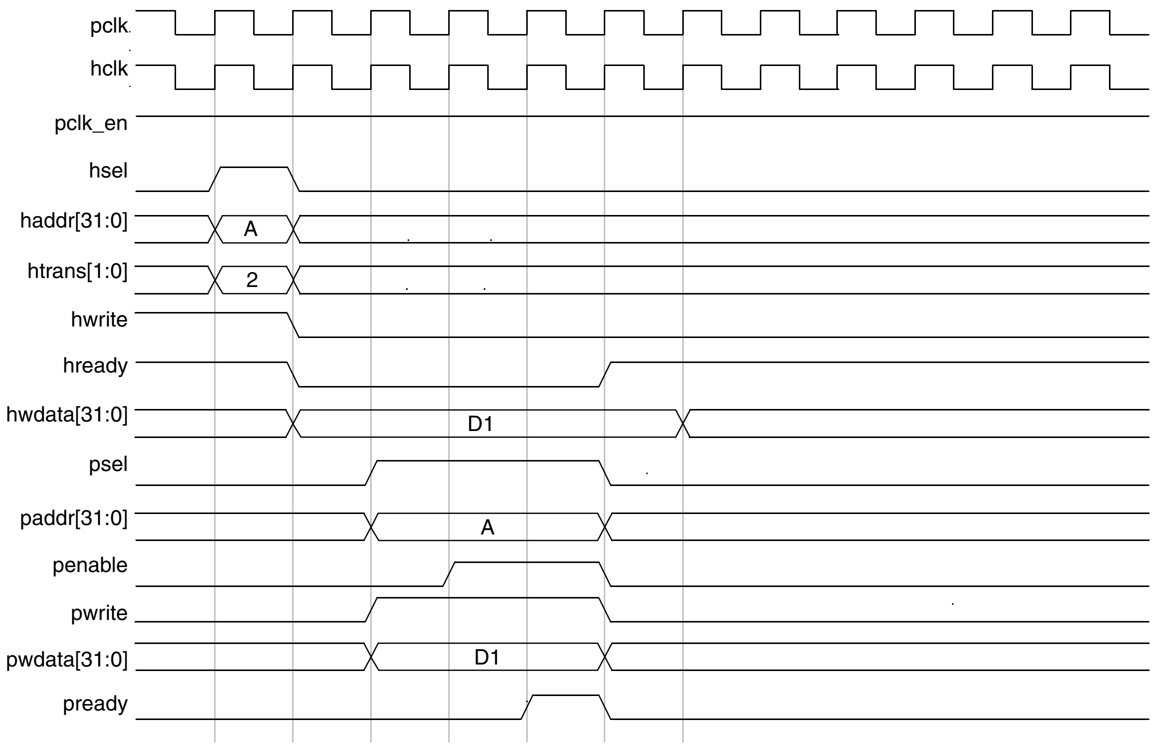


Figure 3-12 DW_apb Write Transfer from AHB to AMBA 2 APB Slave (hclk != pclk)

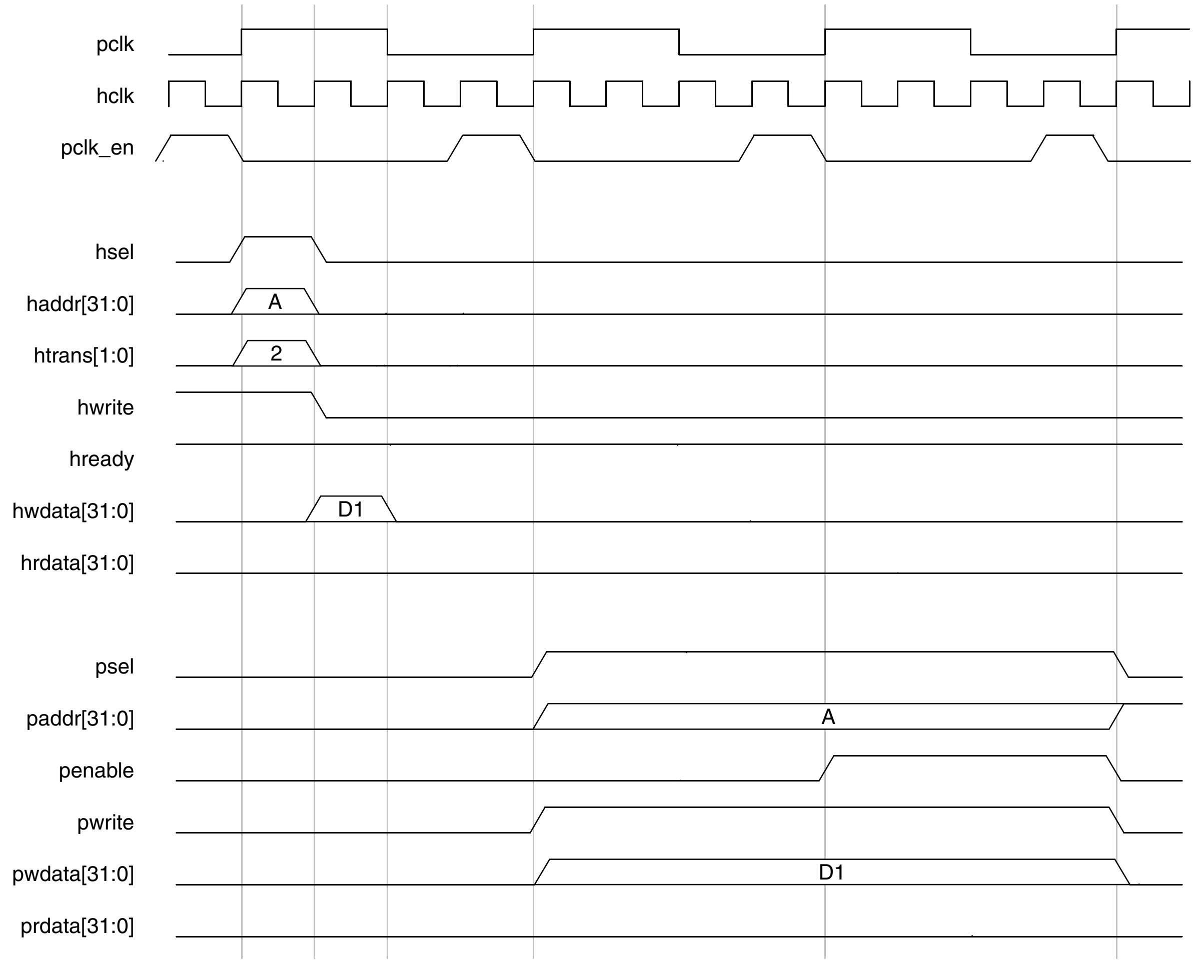


Figure 3-13 DW_apb Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk)

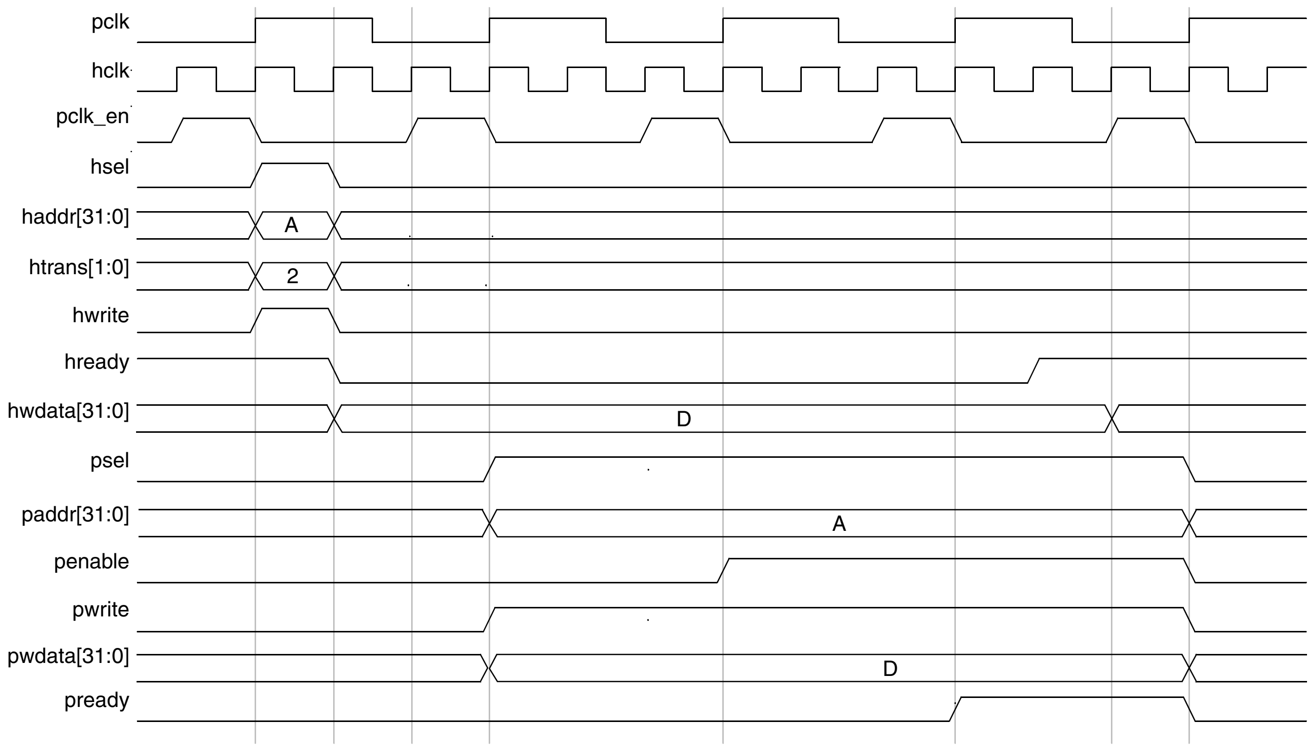


Figure 3-14 DW_apb Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) Completed with Error

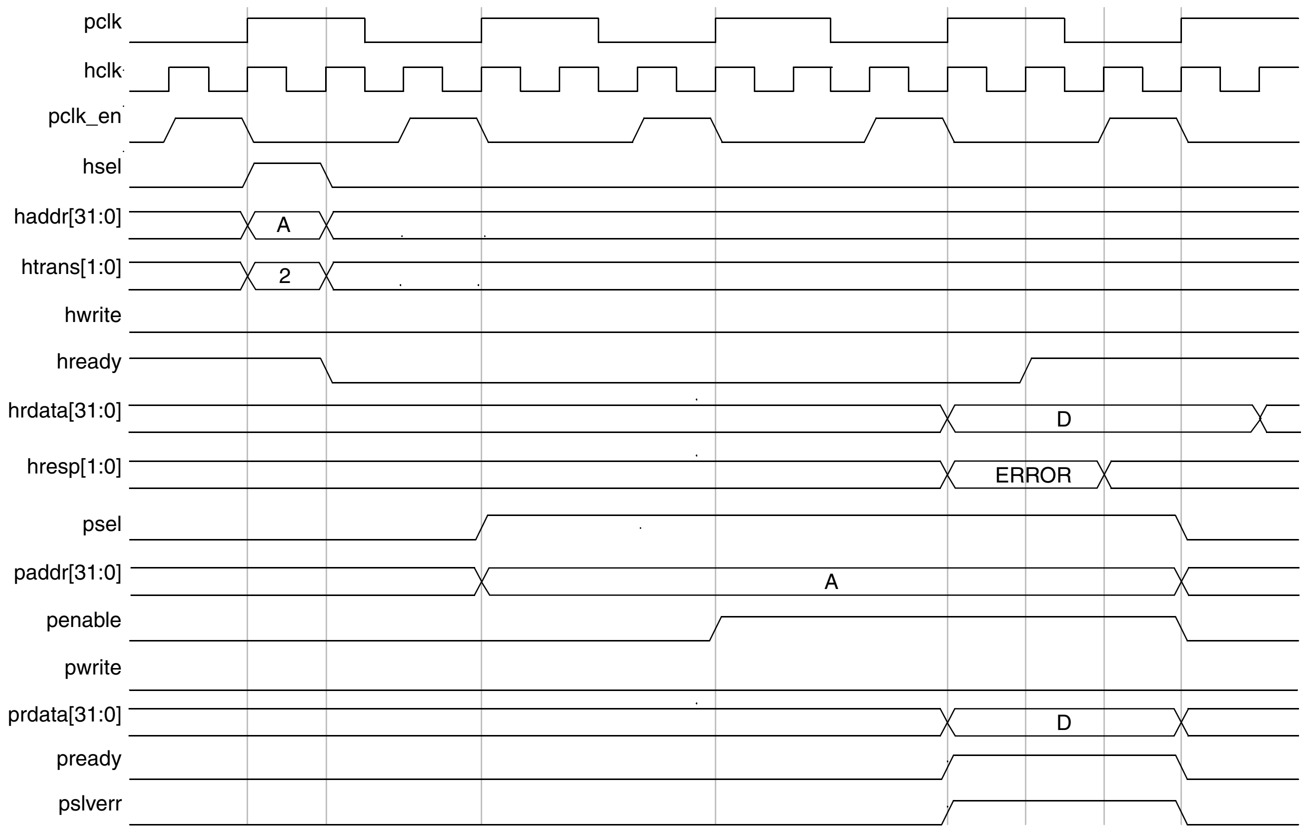
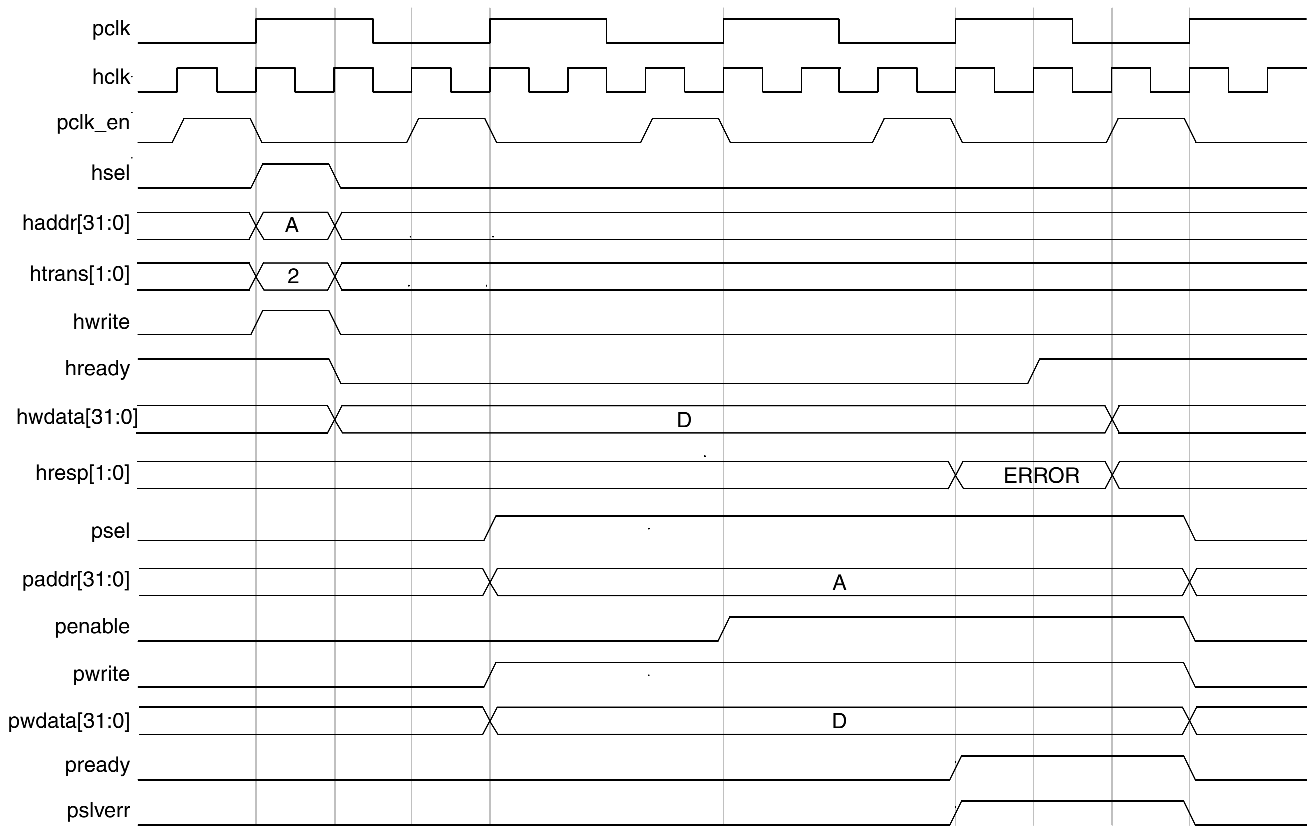


Figure 3-15 DW_apb Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) Completed with Error



The DW_apb bridge expects the pslverr input from the APB slave to be registered. As the pslverr input to the DW_apb bridge is driven from a register, it returns hresp to the AHB master before the end of the PENABLE phase without negatively affecting the timing closure of the system. Thus pslverr is directly routed to the AHB (hclk domain) by mapping to hresp=ERROR (when pready is high) as suggested in the *AMBA 3 APB Specification, Revision 1.0*. Note that the paths from pready_sX to hready_resp are always registered.

Figure 3-16 Back-to-Back Write Transfer (hclk = pclk)

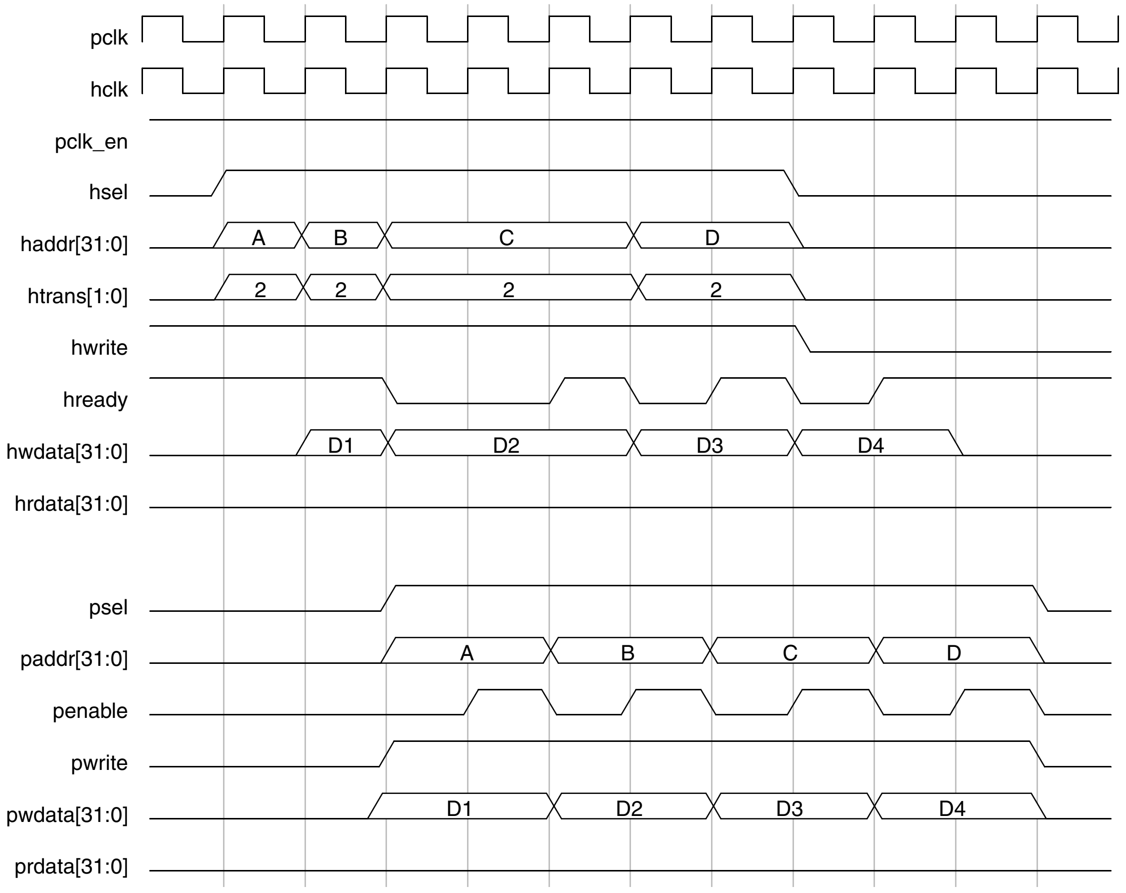
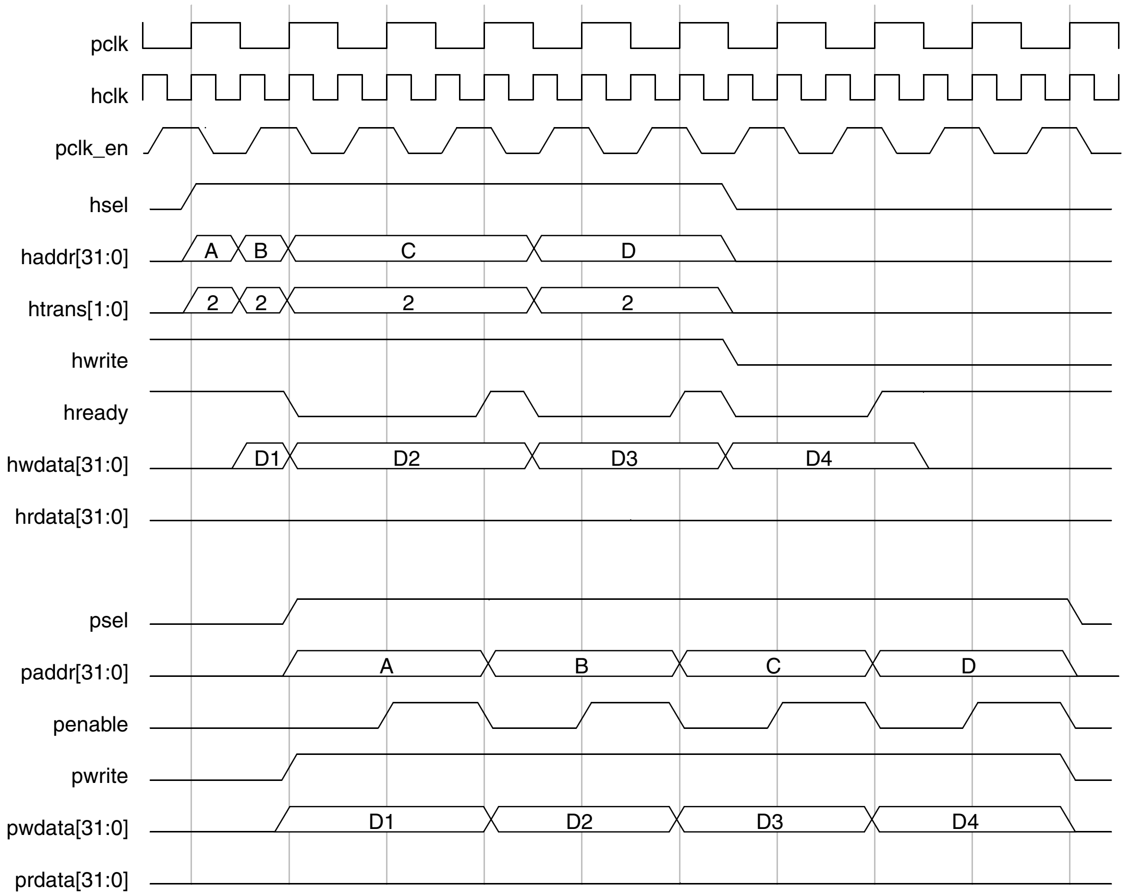


Figure 3-17 Back-to-Back Write Transfer (hclk != pclk)



Note

Figure 3-16 and Figure 3-17 show the AHB issuing consecutive write transfers on the DW_apb, which are targeting AMBA 2 APB slaves. If any transfer targets an AMBA 3 APB slave, the bus brings hready low and the systems stalls until each transfer completes on the APB bus.

4 Parameters

This chapter describes the configuration parameters used by the DW_apb. The settings of the configuration parameters determine the I/O signal list of the DW_apb. You use coreConsultant or coreAssembler to configure the following parameters and generate the configured code.

⚠ **Attention**	When using coreConsultant or coreAssembler, you can right-click on a parameter label to access a "What's This" popup dialog that will tell you the details for that particular parameter. The information in each What's This dialog essentially matches the information in the parameter descriptions below.

4.1 Parameter Descriptions

In the following tables, the values 0 and 1 occasionally appear in parentheses in the descriptions for the parameters. These are the logical values for parameter settings that appear in the coreTools GUIs as check boxes, dropdown lists, a multiple selection, and so on.

4.1.1 DW_apb Top-level Parameters

The information in Table 4-1 shows the coreTools field name and the parameter definition for the DW_apb top-level parameters.

Table 4-1 DW_apb Top-Level Parameters

Field Label	Parameter Definition
AHB System Address Width	**Parameter Name:** HADDR_WIDTH **Legal Values:** 32, 64 **Default Value:** 32 **Dependencies:** None **Description:** The address width of the AHB system.

Table 4-1 DW_apb Top-Level Parameters (Continued)

Field Label	Parameter Definition
APB System Address Width	**Parameter Name:** PADDR_WIDTH **Legal Values:** 32, 64 **Default Value:** 32 **Dependencies:** None **Description:** The address width of the APB system.
AHB Data Bus Width	**Parameter Name:** AHB_DATA_WIDTH **Legal Values:** 32, 64, 128, 256 **Default Value:** 32 **Dependencies:** None **Description:** The data width of the AHB bus.
AHB Endianness	**Parameter Name:** BIG_ENDIAN **Legal Values:** Little-endian (0) or Big-endian (1) **Default Value:** Little-endian (0) **Dependencies:** None **Description**: Define the endianness of the AHB system. The APB subsystem is always little-endian.
APB Data Bus Width	**Parameter Name:** APB_DATA_WIDTH **Legal Values:** 8, 16, 32 **Default Value:** 32 **Dependencies:** None **Description:** The data width of the APB bus.
Number of APB Slave Ports	**Parameter Name:** NUM_APB_SLAVES **Legal Values:** 1 to 16 **Default Value:** 4 **Dependencies:** None **Description**: Number of APB slave ports
External Decoder?	**Parameter Name:** APB_HAS_XDCDR **Legal Values:** True (1) or False (0) **Default Value:** False **Dependencies:** None **Description**: If this parameter is set to True (1), the decoder is external to DW_apb. If False (0), the decoder is internal to DW_apb. For an internal decoder, the addresses needs to be supplied by DW_apb at configuration. An external decoder allows users to connect to any decoder.

4.1.2 DW_apb Slave Parameters

The information in Table 4-2 shows the coreConsultant field name and the parameter definition for the DW_apb slaves. You will need the parameter name if you want to run coreConsultant in batch mode.

Table 4-2 DW_apb Slave Parameters

Field Label	Parameter Definition
APB Slave Addresses	
Start Address of APB Slave *N* (*N* = 0 to 15)	**Parameter Name:** START_PADDR_*N* **Legal Values:** 0x00000000 to 0xfffffc00 **Default Value:** *N*=0: 0x00000400 *N*=1: 0x00000800 *N*=2: 0x00000c00 *N*=3: 0x00001000 *N*=4: 0x00001400 *N*=5: 0x00001800 *N*=6: 0x00001c00 *N*=7: 0x00002000 *N*=8: 0x00002400 *N*=9: 0x00002800 *N*=10: 0x00002c00 *N*=11: 0x00003000 *N*=12: 0x00003400 *N*=13: 0x00003800 *N*=14: 0x00003c00 *N*=15: 0x00004000 **Dependencies:** The decoder must be configured as internal when APB_HAS_XDCDR = 0. **Description**: Start address for slave *N*.

Table 4-2 DW_apb Slave Parameters (Continued)

Field Label	Parameter Definition
End Address of APB Slave*N* (*N* = 0 to 15)	**Parameter Name:** END_PADDR_*N* **Legal Values:** 0x000003ff to 0xffffffff **Default Value:** *N*=0: 0x000007ff *N*=1: 0x00000bff *N*=2: 0x00000fff *N*=3: 0x000013ff *N*=4: 0x000017ff *N*=5: 0x00001bff *N*=6: 0x00001fff *N*=7: 0x000023ff *N*=8: 0x000027ff *N*=9: 0x00002bff *N*=10: 0x00002fff *N*=11: 0x000033ff *N*=12: 0x000037ff *N*=13: 0x00003bff *N*=14: 0x00003fff *N*=15: 0x000043ff **Dependencies:** The decoder must be configured as internal when APB_HAS_XDCDR = 0. **Description**: End address for slave *N*.
AMBA Version	**Parameter Name:** APB_IS_APB3_*n*, where *n* = 0 to 15 **Legal Values:** True (1) or False (0) ■ 0 = AMBA 2 ■ 1 = AMBA 3 **Default Value:** False (0) **Dependencies:** The decoder must be configured as internal when APB_HAS_XDCDR = 0. **Description**: Select between AMBA 3 APB slave and AMBA 2 APB slave. If AMBA 3 APB slave is selected, the additional ports PREADY and PSLVERR are included.

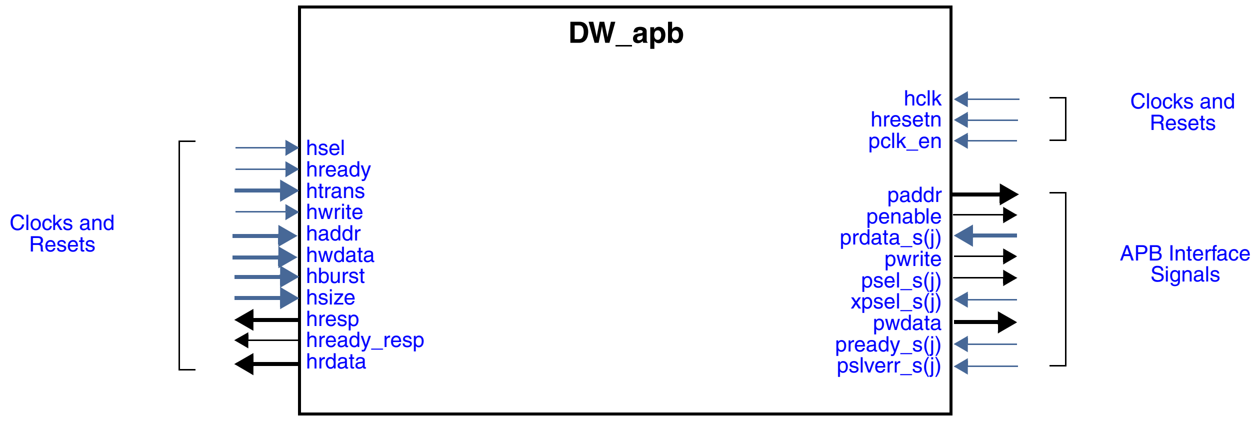
5 Signals

The following subsections describe the DW_apb I/O signals.

5.1 DW_apb Interface Diagram

Figure 5-1 illustrates the input and output signals for the DesignWare APB Bus IP.

Figure 5-1 DW_apb Interface Diagram



5.2 DW_apb Signal Descriptions

Table 5-1 provides descriptions for the DW_apb input and output signals.

Note The Description column in Table 5-1 provides detailed information about each signal.

- In the **Registered** field, a "Yes" indicates whether an I/O signal is directly connected to an internal register and nothing else. An I/O signal is also considered to be registered if the signal is connected to one or more inverters or buffers between the I/O port and internal register, but not connected to any logic that involves another signal.
- The **Input/Output Delay** field provides the percentage of the clock cycle assumed to be used by logic outside this design. The given value is used to automatically define the default synthesis constraints for input/output delay. You can override these default values in the Specify Port Constraints activity in coreConsultant or coreAssembler.

Table 5-1 DW_apb Top-level Signal Descriptions

Name	Width	I/O	Description
Clocks and Resets			
hclk	1 bit	In	AHB Clock Signal. This clock times all bus transfers. All signal timings are related to the rising edge of hclk. **Active State:** N/A **Registered:** N/A **Synchronous to:** N/A **Default Input Delay:** N/A
hresetn	1 bit	In	AHB Reset Signal. This signal is used to reset the system and the bus on the DesignWare interface. **Active State:** Low **Registered:** N/A **Synchronous to:** The reset must be synchronously deasserted after the rising edge of hclk. Since DW_apb does not contain logic to perform this synchronization, it must be provided externally. During reset, masters must ensure that address and control signals are at valid levels, and that htrans indicates the IDLE state. **Default Input Delay:** N/A
pclk_en	1 bit	In	APB Clock Enable Strobe. This signal is of one hclk cycle duration and identifies the hclk rising edge that corresponds with a pclk rising edge. Tied high by the user if pclk = hclk. **Active State:** High **Registered:** No **Synchronous to:** hclk **Default Input Delay:** 66%
AHB Slave Interface Signals			
hsel	1 bit	In	When asserted, the signal indicates that the DW_apb has been selected. Each AHB slave has its own hsel line. This is driven by the AHB decoder block. **Active State:** High **Registered:** No **Synchronous to:** hclk **Default Input Delay:** 66%
hready	1 bit	In	Ready response for DW_apb. **Active State:** High **Registered:** No **Synchronous to:** hclk **Default Input Delay:** 66%

Table 5-1 DW_apb Top-level Signal Descriptions (Continued)

Name	Width	I/O	Description
htrans	2 bits	In	Transfer type from selected master. **Active State:** N/A **Registered:** No **Synchronous to:** hclk **Default Input Delay:** 66%
hwrite	1 bit	In	When hwrite is high, there is a write transfer and the master broadcasts data on the write data bus (hwdata). When hwrite is low, a read transfer is performed, and the DW_apb must generate the data on the read data bus (hrdata). **Active State:** High **Registered:** No **Synchronous to:** hclk **Default Input Delay:** 66%
haddr	32 bits	In	Address bus from AHB master. **Active State:** N/A **Registered:** Yes **Synchronous to:** hclk **Default Input Delay:** 66%
hwdata	[*w*–1:0]	In	Write data bus from selected AHB master. **Width:** Where *w* is the width of the AHB data bus. Maximum value is 256 bits. **Active State:** N/A **Registered:** No **Synchronous to:** hclk **Default Input Delay:** 66%
hburst	3 bits	In	Burst type indication from selected AHB master. This signal is left unconnected on the interface because it is not required. **Active State:** N/A **Registered:** No **Synchronous to:** N/A **Default Input Delay:** N/A
hsize	3 bits	In	Transfer size. Indicates the size of the transfer (refer to "DesignWare Constants" on page 83 for the sizes). This signal is left unconnected on the interface because it is not required. **Active State:** N/A **Registered:** No **Synchronous to:** N/A **Default Input Delay:** N/A

Table 5-1 DW_apb Top-level Signal Descriptions (Continued)

Name	Width	I/O	Description
hresp	2 bits	Out	Transfer Response. This signal is always an OKAY response. When hready_resp is High, this shows the transfer has completed successfully. **Active State:** N/A **Registered:** N/A (connected directly to 1 or 0) **Synchronous to:** hclk **Default Output Delay:** 66%
hready_resp	1 bit	Out	Response from DW_apb. Asserted when current transfer has completed. **Active State:** High **Registered:** Yes **Synchronous to:** hclk **Default Output Delay:** 66%
hrdata	[*w*–1:0]	Out	Transfer read data. The read data bus is used to transfer data from DW_apb to the bus master during read operations. **Width:** Where *w* is the width of the AHB data bus. Maximum value is 256 bits. **Active State:** N/A **Registered:** No **Synchronous to:** hclk **Default Output Delay:** 66%
APB Interface Signals (Where *j* is the number associated with each APB slave [0 to 15])			
paddr	32 bits	Out	APB address bus. Can change on only a pclk_en active edge. It retains its last value, even though there may be no activity on the APB bus, until it is overwritten by a new address. **Active State:** High **Registered:** Yes **Synchronous to:** hclk **Default Output Delay:** 66%
penable	1 bit	Out	Enable Strobe. Asserted to validate APB transfer. It is always driven low at the end of each APB access. **Active State:** High **Registered:** Yes **Synchronous to:** hclk **Default Output Delay:** 30%
prdata_s*(j)*	[*r*–1:0]	In	Read Data from APB slaves. The width of each bus is APB_DATA_WIDTH. **Width:** Where *r* is the width of the APB data bus. Maximum value is 32 bits. **Active State:** N/A **Registered:** No **Synchronous to:** hclk **Default Input Delay:** 10%

Table 5-1 DW_apb Top-level Signal Descriptions (Continued)

Name	Width	I/O	Description
pwrite	1 bit	Out	APB Read/Write Signal. When pwrite is high, there is a write transfer and data is broadcast on the write data bus (pwdata). When pwrite is low, a read transfer is performed, and the slave must generate the data on its read data bus (prdata). **Active State:** High **Registered:** Yes **Synchronous to:** hclk **Default Output Delay:** 30%
psel_s*(j)*	1 bit	Out	Select lines for APB slaves (one per slave). **Active State:** N/A **Registered:** No **Synchronous to:** hclk **Default Output Delay:** 60%
xpsel_s*(j)*	1 bit	In	*Optional*. Slave select line for APB. **Active State:** N/A **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 66% Dependencies: This signal is included if DW_apb is configured to use an external decoder (APB_HAS_XDCDR = 1).
pwdata	[*w*–1:0]	Out	APB transfer write data bus shared by all slaves. **Width:** Where *w* is the width of the AHB data bus. Maximum value is 256 bits. **Active State:** N/A **Registered:** No **Synchronous to:** hclk **Default Output Delay:** 30%
pready_s(*j*)	1 bit	In	Indicates whether a request cycle was accepted. Exists only for slaves configured as AMBA 3 APB compliant (APB_IS_APB3_n=1). **Width:** One bit for each input. **Active State:** High **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 25%
pslverr_s(*j*)	1 bit	In	Flag for the slave error response from APB. Exists only for slaves configured as AMBA 3 APB compliant (APB_IS_APB3_n=1). **Width:** One bit for each input. **Active State:** High **Registered:** No **Synchronous to:** pclk **Default Output Delay:** 25%

Note The HPROT, HMASTERLOCK, HMASTER, and HSPLIT signals included in the *AMBA Specification (Rev. 2.0)* are not supported or needed in the DW_apb component.

6 Verification

This chapter provides an overview of the testbench available for DW_apb verification. Once you have configured the DW_apb in coreConsultant and have set up the verification environment, you can run simulations automatically.

Note The DW_apb verification testbench is built with DesignWare Verification IP (VIP). Please make sure you have the supported version of the VIP components for this release, otherwise, you may experience some tool compatibility problems. For more information about supported tools in this release, refer to the *DesignWare Synthesizable Components for AMBA 2/AMBA 3 AXI Installation Guide*.

6.1 Overview of Vera Tests

The DW_apb verification environment performs the following set of tests, which are listed in the Tests tab of the coreConsultant Verification activity. By default, all of the tests are enabled to run. The tests have been written to verify the functionality and have also achieved maximum RTL code coverage.

6.1.1 PCLK equals HCLK

When the pclk is the same as the hclk, the data rate on the APB is half that on the AHB. Internal latency is not an issue in this mode—when data is ready on the hclk domain, it can be transferred directly to the pclk domain. To test this functionality, the following tests are performed:

- Initiate a single write transfer to the APB slave
- Initiate two consecutive write transfers to different address locations within the APB slave
- Initiate two write transfers to different address locations within the APB slave, separated by one hclk cycle
- Initiate two write transfers to different address locations within the APB slave, separated by two hclk cycles
- Initiate two write transfers to different address locations within the APB slave, separated by three or more hclk cycles
- Initiate multiple write transfers to different address locations within the APB slave, separated by a random number of hclk cycles

- Initiate a single read transfer to the APB slave
- Initiate two consecutive read transfers to the APB slave
- Initiate two read transfers to the APB slave, separated by one hclk cycle
- Initiate two read transfers to the APB slave, separated by two or more hclk cycles
- Initiate a write transfer, followed directly by a read transfer to the same address location
- Initiate a write transfer, followed by one hclk cycle later with a read transfer to the same address location
- Initiate a write transfer, followed by two hclk cycles later with a read transfer to the same address location
- Initiate a write transfer, followed by three or more hclk cycles later with a read transfer to the same address location
- Initiate a write transfer to the start address, to the end address of the slave address. Then Initiate a write transfer to addresses outside the slave address range, but within the DW_apb address range
- Initiate a read transfer to the APB slave, followed directly by a read to another AHB slave

6.1.2 PCLK Equals HCLK Divided by 2 or more

When pclk is not the same as hclk, the data saved on the hclk side needs to be held and the master held off from starting new transfers until the rising edge of pclk occurs. This way the saved data can be off-loaded and the new data stored. The data are sometimes address values; at other times they are write data values.

The following checks are needed when the first transfer occurs in any phase of the pclk domain. The transfer will occur when pclk_en is low and high. When pclk_en is high, the state machine moves on; when it is low, it waits for the rising edge of pclk.

Some of the states of the state machine are dependent on pclk_en; others are directly controlled by only hclk.

- Initiate a single write transfer to the APB slave
- Initiate two consecutive write transfers to different address locations within the APB slave
- Initiate two write transfers to different address locations within the APB slave, separated by one hclk cycle
- Initiate two write transfers to different address locations within the APB slave, separated by two hclk cycles
- Initiate two write transfers to different address locations within the APB slave, separated by three or more hclk cycles
- Initiate multiple write transfers to different address locations within the APB slave, separated by a random number of hclk cycles
- Initiate a single read transfer to the APB slave
- Initiate two consecutive read transfers to the APB slave
- Initiate two read transfers to the APB slave, separated by one hclk cycle
- Initiate two read transfers to the APB slave, separated by two or more hclk cycles

- Initiate a write transfer, followed directly by a read transfer to the same address location
- Initiate a write transfer, followed one hclk cycle later with a read transfer to the same address location
- Initiate a write transfer, followed two hclk cycles later with a read transfer to the same address location
- Initiate a write transfer, followed three or more hclk cycles later with a read transfer to the same address location
- Initiate a write transfer to the start address, to the end address of the slave address. Initiate a write transfer to addresses outside the slave address range, but within the DW_apb address range
- Initiate a read transfer to the APB slave, followed directly by a read to another AHB slave

6.1.3 Ignoring IDLE and BUSY transfers

Only for nonsequential or sequential transfer will there be any resultant APB activity. If a transfer is initiated with a busy or an idle transfer, DW_apb ignores this transfer.

- Initiate a single write that is IDLE on htrans
- Initiate a single read that is IDLE on htrans
- Initiate a single write that is BUSY on htrans
- Initiate a single read that is BUSY on htrans
- Initiate back-to-back writes, the first being a NONSEQ, followed directly by an IDLE
- Initiate back-to-back writes, the first being a NONSEQ, followed directly by a BUSY
- Initiate back-to-back reads, the first being a NONSEQ, followed directly by an IDLE
- Initiate back-to-back reads, the first being a NONSEQ, followed directly by a BUSY
- Initiate back-to-back read, followed by a write which is an IDLE
- Initiate back-to-back read, followed by a write which is a BUSY
- Initiate back-to-back write followed by read a which is an IDLE
- Initiate back-to-back write followed by read which is an IDLE

6.2 Overview of DW_apb Testbench

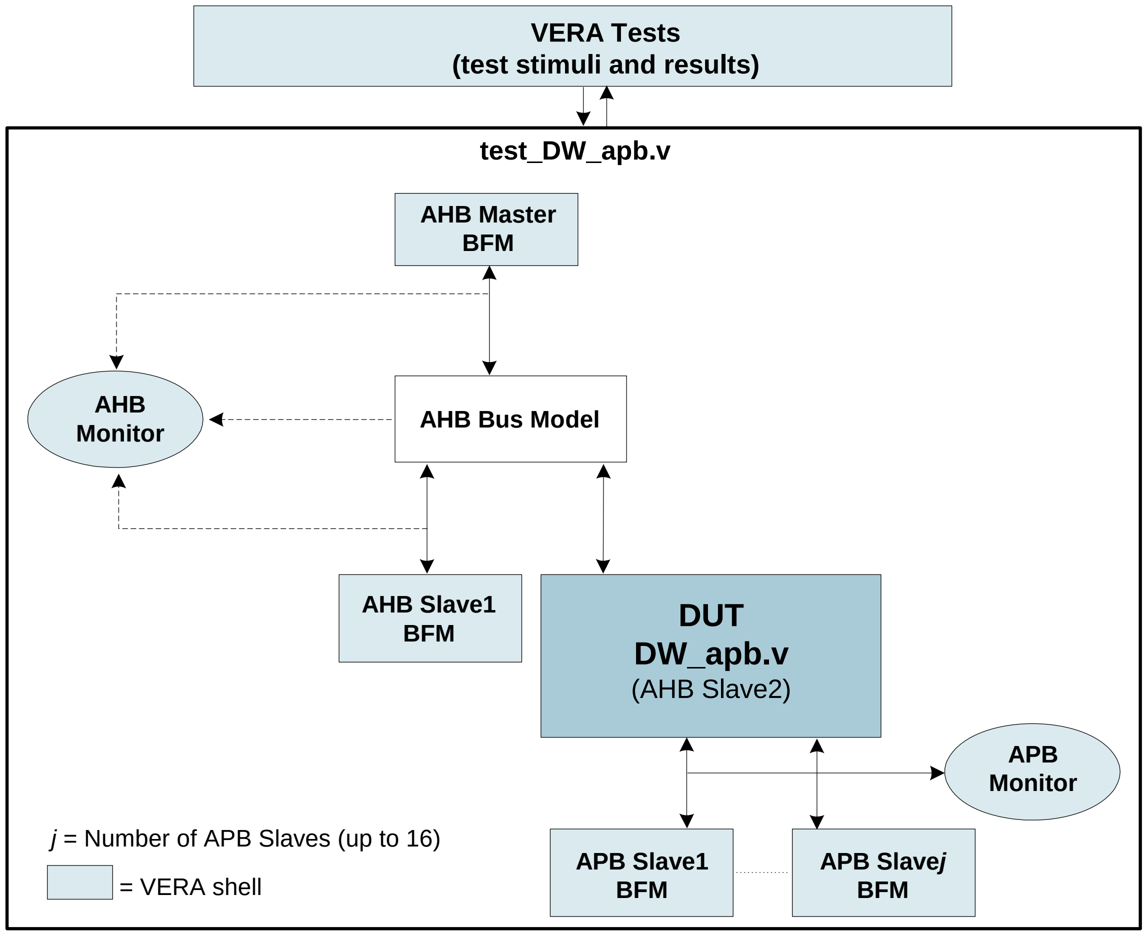
As illustrated in Figure 6-1 on page 62, the DW_apb testbench is a Verilog testbench that includes an instantiation of the design under test (DUT) and a Vera shell, which consists of the following components:

- An AHB master bus functional model (BFM)
- One AHB slave BFM – the DW_apb
- An AHB monitor
- APB slave BFMs
- An APB monitor

- Test stimuli
- Test results

The AHB monitor monitors activity from the AHB master to the AHB slave; the APB monitor oversees activity to and from the APB slave BFMs. The testbench verifies all possible user configurations specified in the Specify Configuration task of coreConsultant. The testbench also tests that the component is AMBA-compliant and self-checking, displaying pass or fail results.

Figure 6-1 DW_apb Testbench



6.3 Running Simulations from the Command Line

To run simulations from a UNIX command line, a simulation model must be generated through the coreConsultant GUI. In addition, all tests and test options must be configured in the Verification tab of the GUI. Then, simulations can be run as follows:

- To run all tests selected in the GUI, change your working directory to DW_apb/sim and then execute the following command:

```
runtest.sh
```

- To run single tests, change the working directory to DW_apb/sim and run the following:

  ```
  runtest --simulator selected_simulator --test test_name
  ```

The *selected_simulator* is the one chosen in the GUI (does not work if not configured in the GUI). The *test_name* is the name of the selected test and the sub directory where the test is located. For example, to run the simple register write/read test using VCS, run the following:

```
runtest --simulator vcs --test test_reg_wr_rd
```

The results of running tests through the command line are available only in the test.log file in each test directory.

6.3.1 Command Line Output Files

The runtest.log file in *workspace*/sim includes all of the results of the simulation and presents them in the following categories:

- Summary of All Results – Provides the final result either PASSED or FAILED
- Verification Activity Log – Shows a log of the simulation activity
- Testbench Preparation – Provides a list of runtest options that were executed during the simulation
- Simulation Execution – Provides the output of the simulator; this information is also saved to test.log in *workspace*/sim/test_apb
- Simulation Results – Includes the time the simulation completed, the path to test.log, how many errors were encountered, and the overall result (PASSED/FAILED)

The *workspace*/sim/test_apb directory includes the various logs that are included in runtest.log. The individual log files in *workspace*/sim/test_apb are:

- test.log – Output of the testbench; includes specifics about the simulators used, the tests used to verify the core, and the simulation results.
- summary – Post-processed file that includes the following sections:
  - Testbench Preparation
  - Simulation Execution
  - Profiling Report
  - Test Report
  - Simulation Results
- test.result – Testbench automatically compares the simulation results with the expected results during simulation. If the simulation results match expected results, the simulation completes successfully and the simulation status in the test.result file is PASSED. If the simulation results do not match expected results, the simulation terminates and the simulation status in the test.result file is FAILED.

7 Integration Considerations

After you have configured, tested, and synthesized your component with the coreTools flow, you can integrate the component into your own design environment.

7.1 Performance

This section discusses performance and the hardware configuration parameters that affect the performance of the DW_apb.

7.1.1 Area

This section provides information to help you configure area for your configuration.

The following table includes synthesis results that have been generated using the TSMC 65nm technology library.

Table 7-1 Synthesis Results Using TSMC 65nm Technology Library

Configuration	Operating Frequency	Gate Count
Default Configuration	166.67 MHz	1780 gates
Minimum Configuration: NUM_APB_SLAVES=1	166.67 MHz	1651 gates

Maximum Configuration:
NUM_APB_SLAVES=16
BIG_ENDIAN=1
	166.67 MHz	2362 gates

The following table includes synthesis results that have been generated using the TSMC 28nm technology library.

Table 7-2 Synthesis Results Using TSMC 28nm Technology Library

Configuration	Operating Frequency	Gate Count
Default Configuration	166.67 MHz	1790 gates
Minimum Configuration: NUM_APB_SLAVES=1	166.67 MHz	1669 gates

Maximum Configuration:
NUM_APB_SLAVES=16
BIG_ENDIAN=1
	166.67 MHz	2317 gates

7.1.2 Power Consumption

The following table provides information about the power consumption of the DW_apb using the TSMC 65nm technology library and how it affects performance.

Table 7-3 DW_apb Power Consumption Using TSMC 65nm Technology Library

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Default Configuration	166.67 MHz	499.1066 nW	222.4857 µW
Minimum Configuration: NUM_APB_SLAVES=1	166.67 MHz	457.2467 nW	220.6523 µW

Maximum Configuration:
NUM_APB_SLAVES=16
BIG_ENDIAN=1
	166.67 MHz	677.6034 nW	228.7330 µW

The following table provides information about the power consumption of the DW_apb using the TSMC 28nm technology library and how it affects performance.

Table 7-4 DW_apb Power Consumption Using TSMC 28nm Technology Library

Configuration	Operating Frequency	Static Power Consumption	Dynamic Power Consumption
Default Configuration	166.67 MHz	186.3901µW	159.7825 µW
Minimum Configuration: NUM_APB_SLAVES=1	166.67 MHz	175.2981µW	158.4800 µW

Maximum Configuration:
NUM_APB_SLAVES=16
BIG_ENDIAN=1
	166.67 MHz	239.9482 µW	163.3700 µW

7.2 Addressing

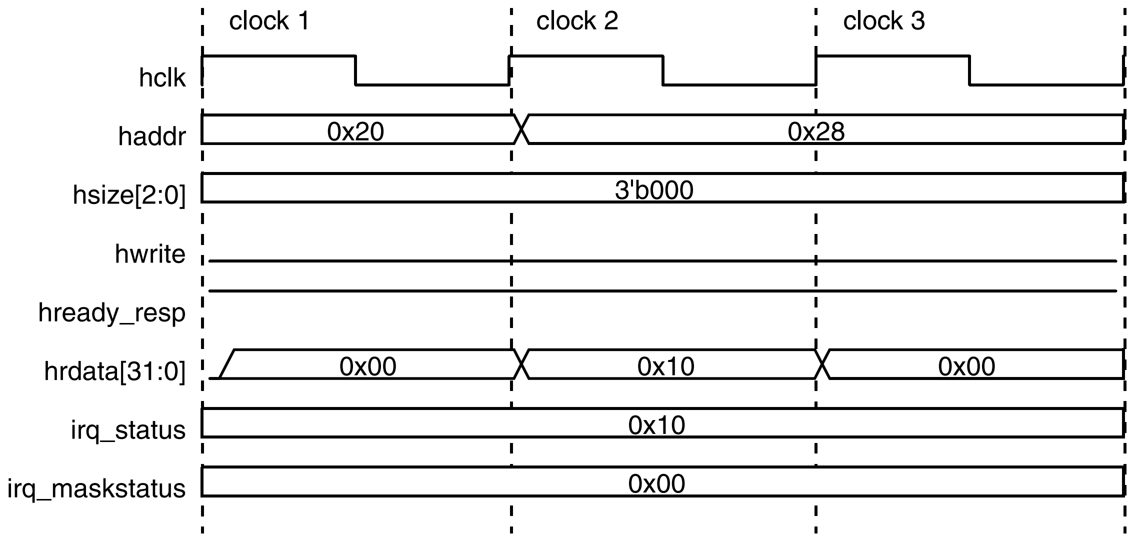
There is mixture of 32 and 64-bit registers in the AHB slave interface. All the registers are 64 bits wide, except the fiq registers, which are 32 bits wide. A register may not be accessed with a hsize greater than its width. Notwithstanding this hsize restriction, all other types of hsize accesses are supported as per the AMBA specification.

When performing a transfer on the AHB bus, the address must be aligned with the value of hsize. For example, if hsize = 32, then the address must be aligned to 32 bits, such as the two least significant bits = 'b0. This condition is not checked in the slave interface but is a requirement of the AMBA specification. Also, hsize must not specify a transfer width greater than the slave's AHB data width; this is also a protocol violation.

7.3 Read Accesses

For reads, registers less than the full access width return zeros in the unused upper bits. An AHB read takes two hclk cycles. The two cycles can be thought of as a control and data cycle, respectively. As shown in the following figure, the address and control is driven from clock 1 (control cycle); the read data for this access is driven by the slave interface onto the bus from clock 2 (data cycle) and is sampled by the master on clock 3. The operation of the AHB bus is pipelined, so while the read data from the first access is present on the bus for the master to sample, the control for the next access is present on the bus for the slave to sample.

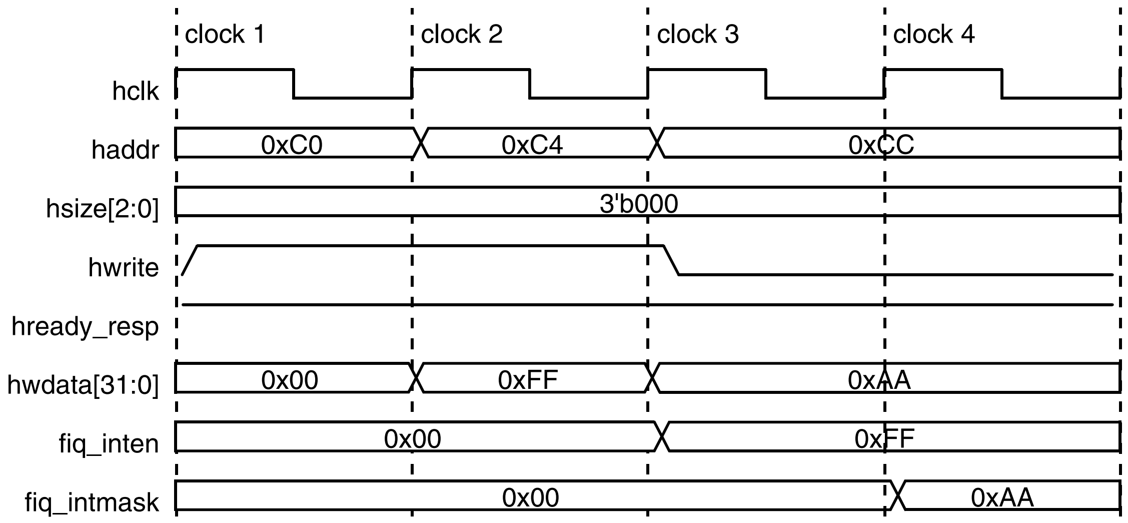
Figure 7-1 AHB Read



7.4 Write Accesses

When writing to a register, bit locations larger than the register width or allocation are ignored. Only pertinent bits are written to the register. Similar to the read case, a write access may be thought of as comprising a control and data cycle. As illustrated in the following figure, the address and control is driven from clock1 (control cycle), and the write data is driven by the bus from clock 2 (data cycle) and sampled by the destination register on clock 3.

Figure 7-2 AHB Write

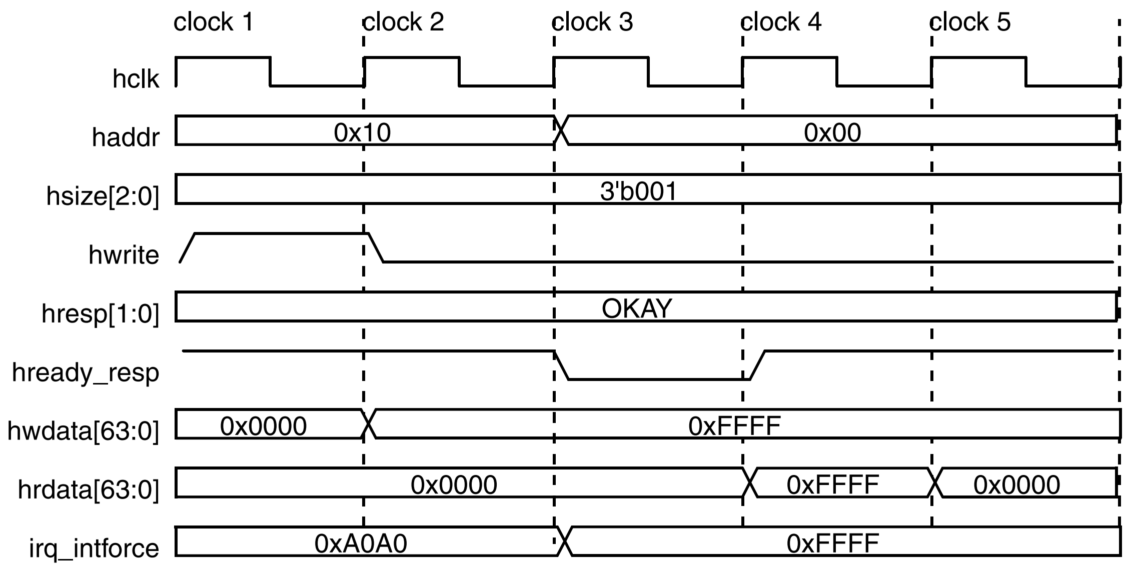


The operation of the AHB bus is pipelined, so while the write data for the first write is present on the bus for the slave to sample, the control for the next write is present on the bus for the slave to sample.

7.5 Consecutive Write-Read

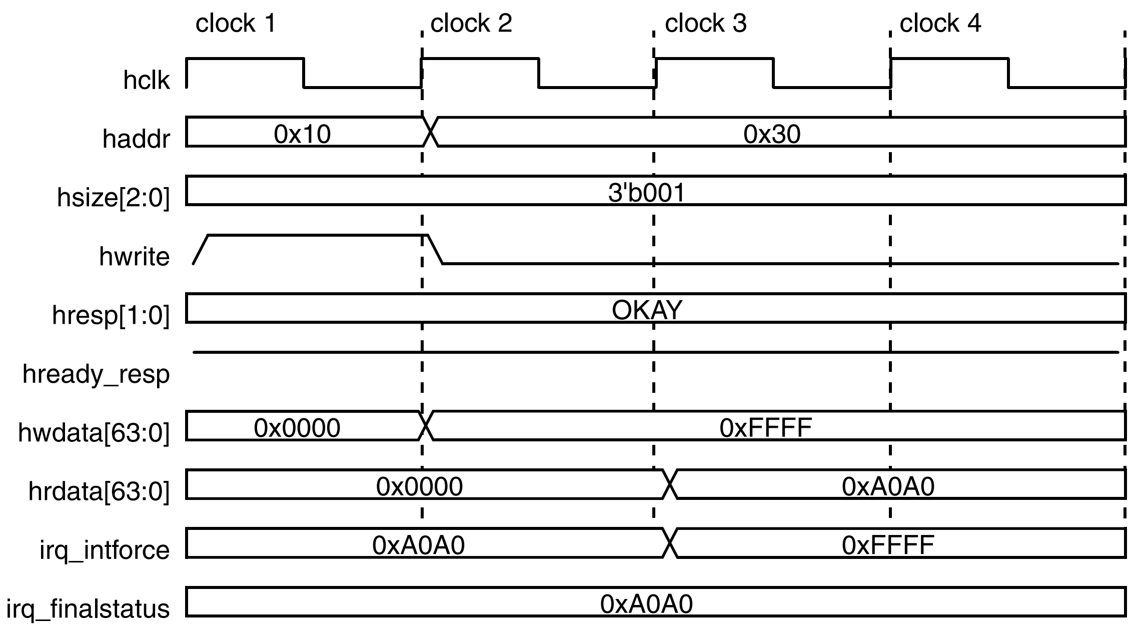
This is a specific case for the AHB slave interface. The AMBA specification says that for a read after a write to the same address, the newly written data must be read back, not the old data. To comply with this, the slave interface in the DW_ahb_ictl inserts a "wait state" when it detects a read immediately after a write to the same address. As shown in the following figure, the control for a write is driven on clock 1, followed by the write data and the control for a read from the same address on clock 2.

Figure 7-3 AHB Wait State Read/Write



Sensing the read after a write to the same address, the slave interface drives hready_resp low from clock 3; hready_resp is driven high on clock 4 when the new write data can be read; and the bus samples hready_resp high on clock 5 and reads the newly written data. The following figure shows a normal consecutive write-read access.

Figure 7-4 AHB Consecutive Read/Write



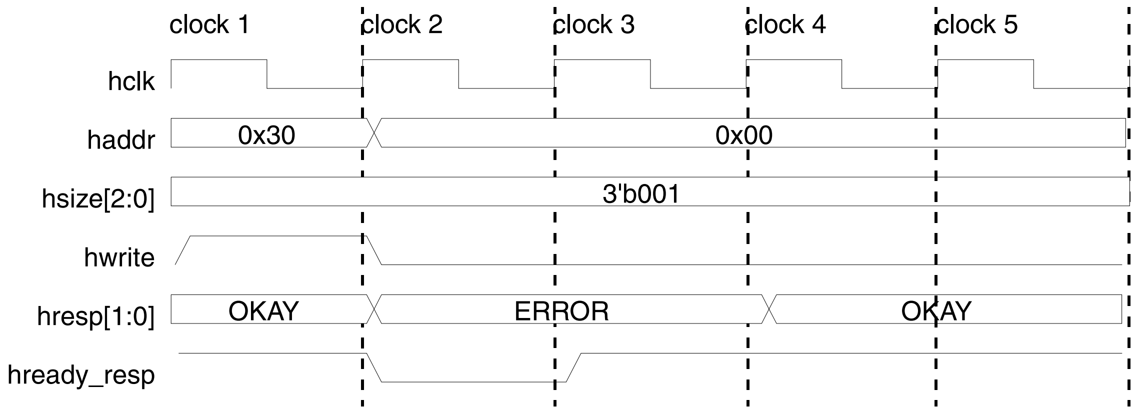
7.6 Error Responses

The following are the cases where the slave interface of the DW_ahb_ictl issues an ERROR response.

- Accessing the slave with an address that does not decode to a register in the design. (The DW_ahb_ictl slave interface decodes for a 1 KB address space [bits 0 to 9 of haddr]. If an address inside this boundary does not refer to a register in the design, an ERROR is given.)
- Attempting to write to a read only register location.
- Accesses to a 64 bit register location with hsize greater than 64.
- Accesses to a 32 bit register location with hsize greater than 32.
- Accesses to the irq_vector register with hsize < 32. We impose this restriction to remove the need for shadow registers to guarantee coherency of irq_vector for reads with a hsize of less than 32 bits.

The following figure shows an error response. The control for the errant access is driven from clock 1. In this case, you are attempting to write to the irq_finalstatus register, which is read only. From clock 2, hready_resp is driven low, and hresp is driven to ERROR. From clock 3, hready_resp is driven to high and hresp remains at ERROR. From clock 4, hresp is driven to OKAY again and the ERROR response is complete.

Figure 7-5 AHB Error Response



7.7 Accessing Top-level Constraints

To get SDC constraints out of coreConsultant, you need to first complete the synthesis activity and then use the “write_sdc” command to write out the results:

1. This cC command sets synthesis to write out scripts only, without running DC:

```
set_activity_parameter Synthesize ScriptsOnly 1
```

2. This cC command autocompletes the activity:

```
autocomplete_activity Synthesize
```

3. Finally, this cC command writes out SDC constraints:

```
write_sdc <filename>
```

7.8 Coherency

Coherency is where bits within a register are logically connected. For instance, part of a register is read at time 1 and another part is read at time 2. Being coherent means that the part read at time 2 is at the same value it was when the register was read at time 1. The unread part is stored into a shadow register and this is read at time 2. When there is no coherency, no shadow registers are involved.

A bus master may need to be able to read the contents of a register, regardless of the data bus width, and be guaranteed of the coherency of the value read. A bus master may need to be able to write a register coherently regardless of the data bus width and use that register only when it has been fully programmed. This may need to be the case regardless of the relationship between the clocks.

Coherency enables a value to be read that is an accurate reflection of the state of the counter, independent of the data bus width, the counter width, and even the relationship between the clocks. Additionally, a value written in one domain is transferred to another domain in a seamless and coherent fashion.

Throughout this appendix the following terms are used:

- **Writing**. A bus master programs a configuration register. An example is programming the load value of a counter into a register.
- **Transferring**. The programmed register is in a different clock domain to where it is used, therefore, it needs to be transferred to the other clock domain.
- **Loading**. Once the programmed register is transferred into the correct clock domain, it needs to be loaded or used to perform its function. For example, once the load value is transferred into the counter domain, it gets loaded into the counter.

7.8.1 Writing Coherently

Writing coherently means that all the bits of a register can be written at the same time. A peripheral may have programmable registers that are wider than the width of the connected APB data bus, which prevents all the bits being programmed at the same time unless additional coherency circuitry is provided.

The programmable register could be the load value for a counter that may exist in a different clock domain. Not only does the value to be programmed need to be coherent, it also needs to be transferred to a different clock domain and then loaded into the counter. Depending on the function of the programmable register, a qualifier may need to be generated with the data so that it knows when the new value is currently transferred and when it should be loaded into the counter.

Depending on the system and on the register being programmed, there may be no need for any special coherency circuitry. One example that requires coherency circuitry is a 32-bit timer within an 8-bit APB system. The value is entirely programmed only after four 8-bit wide write transfers. It is safe to transfer or use the register when the last byte is currently written. An example where no coherency is required is a 16-bit wide timer within a 16-bit APB system. The value is entirely programmed after a single 16-bit wide write transfer.

Coherency circuitry enables the value to be loaded into the counter only when fully programmed and crossed over clock domains if the peripheral clock is not synchronous to the processor clock. While the load register is being programmed, the counter has access to the previous load value in case it needs to reload the counter.

Coherency circuitry is only added in cores where it is needed. The coherency circuitry incorporates an upper byte method that requires users to program the load register in LSB to MSB order when the peripheral width is smaller than the register width. When the upper byte is programmed, the value can be transferred and loaded into the load register. When the lower bytes are being programmed, they need to be stored in shadow registers so that the previous load register is available to the counter if it needs to reload. When the upper byte is programmed, the contents of the shadow registers and the upper byte are loaded into the load register.

The upper byte is the top byte of a register. A register can be transferred and loaded into the counter only when it has been fully programmed. A new value is available to the counter once this upper byte is written into the register. The following table shows the relationship between the register width and the peripheral bus width for the generation of the correct upper byte. The numbers in the table represent bytes, Byte 0 is the LSB and Byte 3 is the MSB. NCR means that no coherency circuitry is required, as the entire register is written with one access.

Table 7-5 Upper Byte Generation

	Upper Byte Bus Width		
Load Register Width	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	1	NCR	NCR
17 - 24	2	2	NCR
25 - 32	3	2 (or 3)	NCR

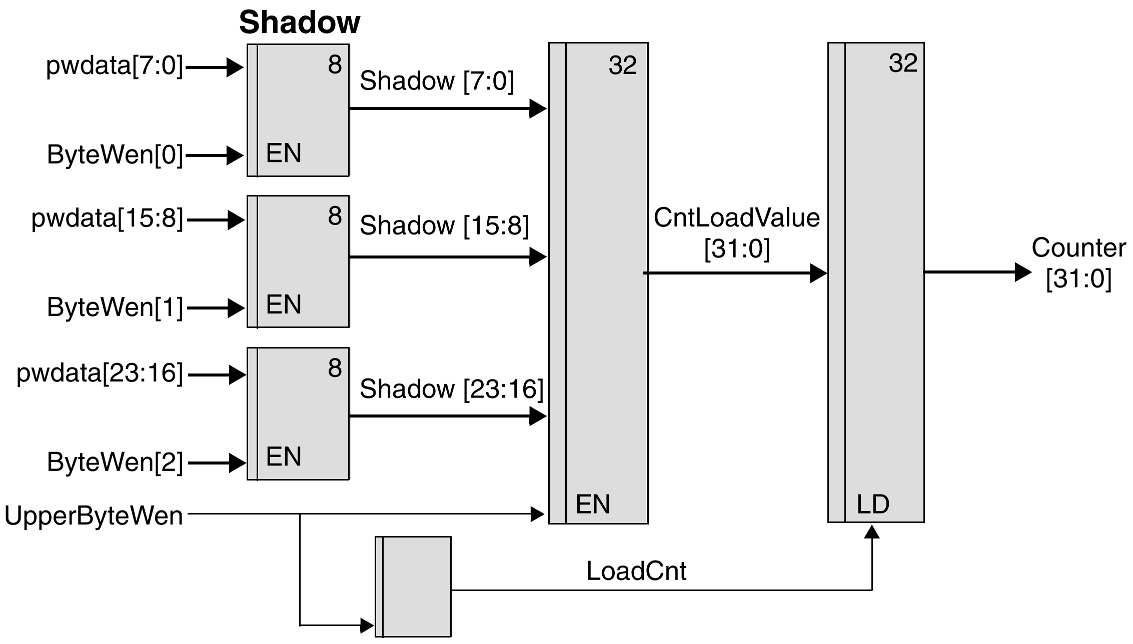
There are three relationship cases to be considered for the processor and peripheral clocks:

- Identical
- Synchronous (phase coherent but of an integer fraction)
- Asynchronous

7.8.1.1 Identical Clocks

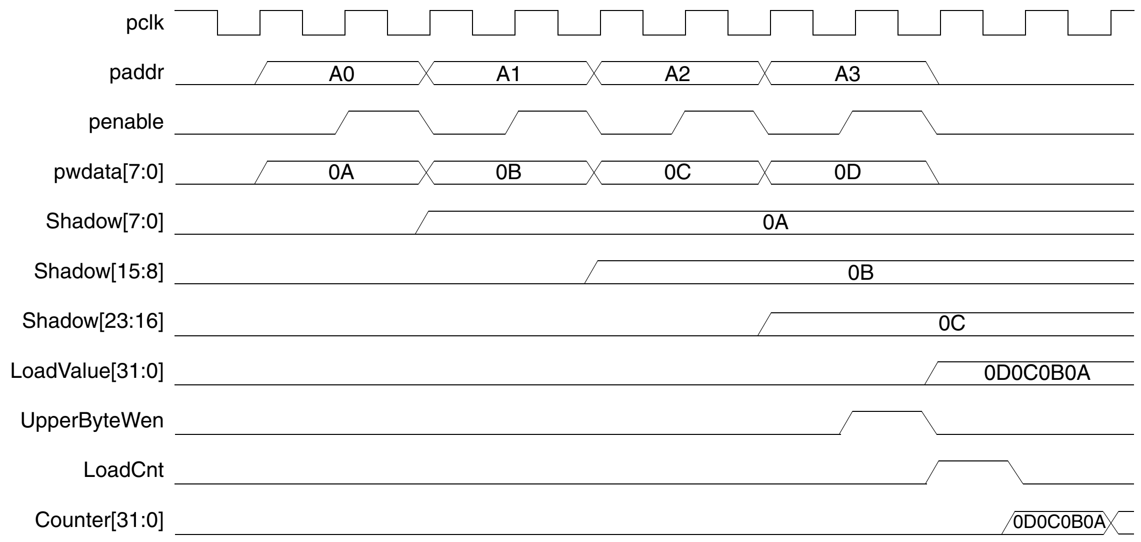
The following figure illustrates an RTL diagram for the circuitry required to implement the coherent write transaction when the APB bus clock and peripheral clocks are identical.

Figure 7-6 Coherent Loading – Identical Synchronous Clocks



The following figure shows a 32-bit register that is written over an 8-bit data bus, as well as the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal lasts for one cycle and is used to load the counter with CntLoadValue.

Figure 7-7 Coherent Loading – Identical Synchronous Clocks



Each of the bytes that make up the load register are stored into shadow registers until the final byte is written. The shadow register is up to three bytes wide. The contents of the shadow registers and the final byte are transferred into the CntLoadValue register when the final byte is written. The counter uses this register to load/initialize itself. If the counter is operating in a periodic mode, it reloads from this register each time the count expires.

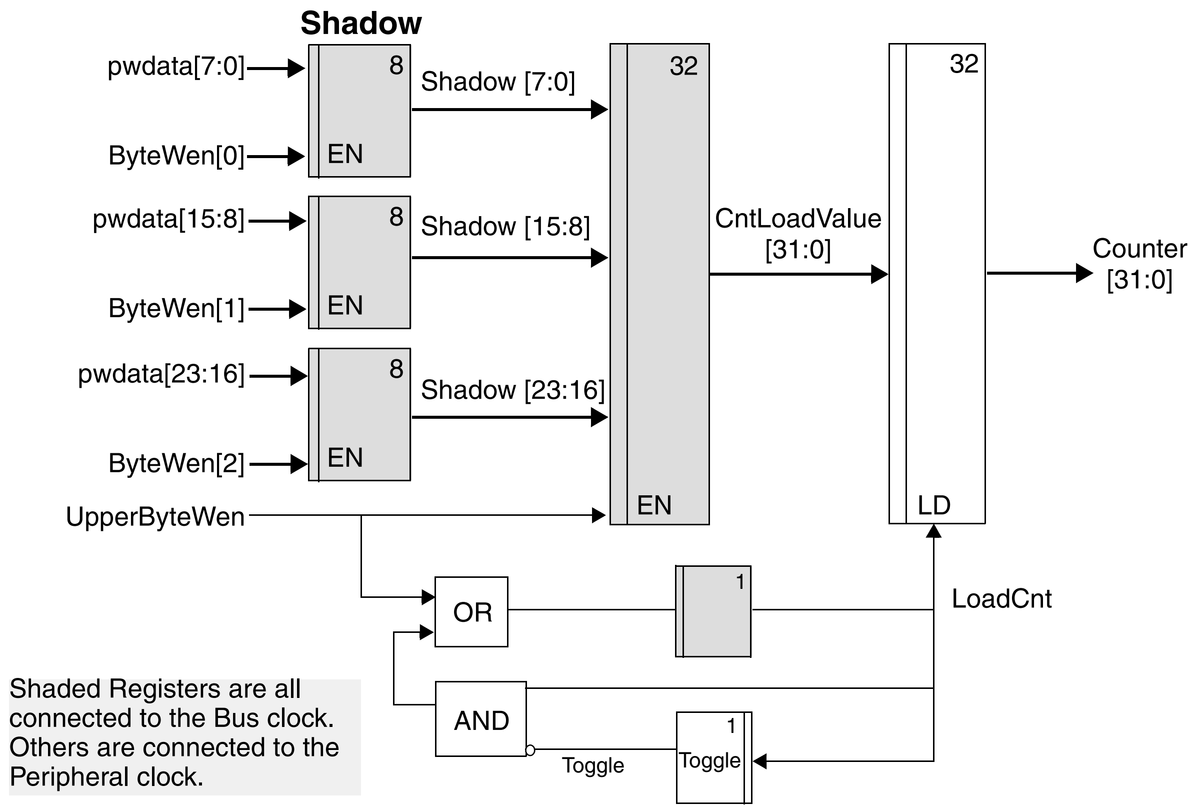
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this is signaled by LoadCnt = 1. After the upper byte is written, the LoadCnt goes to zero.

7.8.1.2 Synchronous Clocks

When the clocks are synchronous but do not have identical periods, the circuitry needs to be extended so that the LoadCnt signal is kept high until a rising edge of the counter clock occurs. This extension is necessary so that the value can be loaded, using LoadCnt, into the counter on the first counter clock edge. At the rising edge of the counter clock if LoadCnt is high, then a register clocked with the counter clock toggles, otherwise it keeps its current value. A circuit detecting the toggling is used to clear the original LoadCnt by looking for edge changes. The value is loaded into the counter when a toggle has been detected. Once it is loaded, the counter should be free to increment or decrement by normal rules.

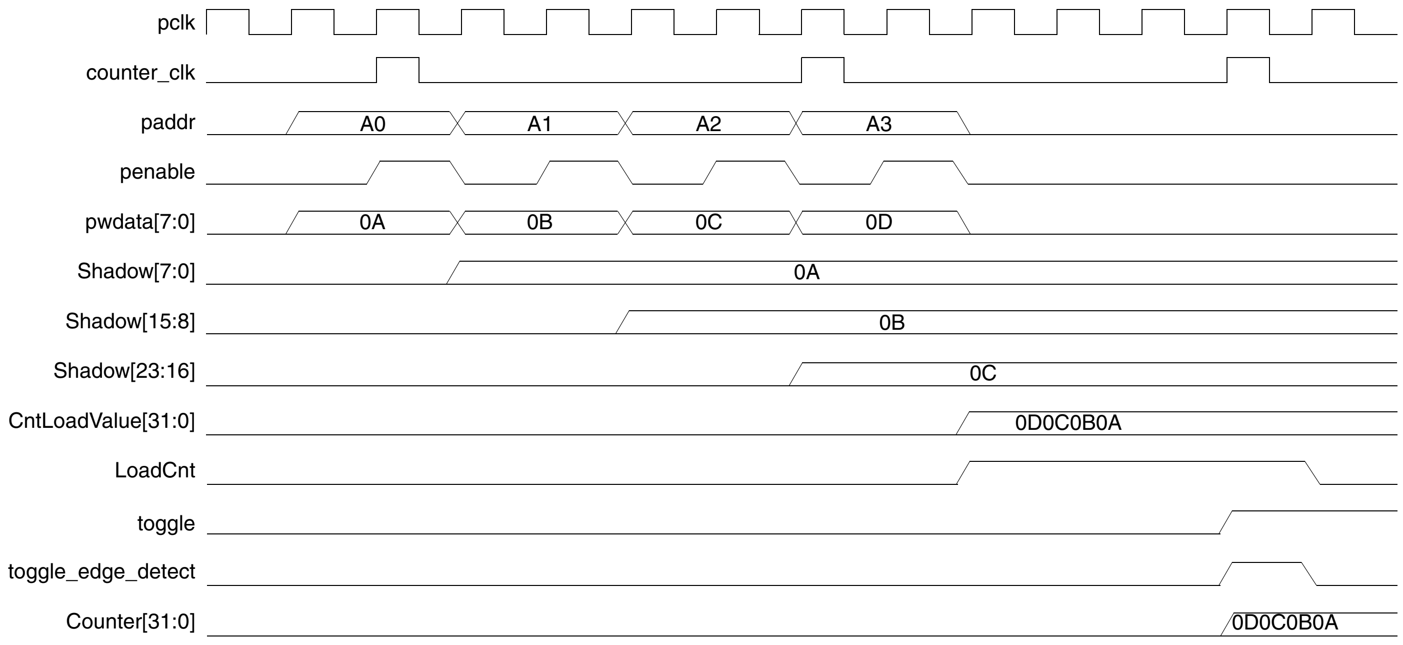
The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are synchronous.

Figure 7-8 Coherent Loading – Synchronous Clocks



The following figure shows a 32-bit register being written over an 8-bit data bus, as well as the shadow registers being loaded and then loaded into the counter when fully programmed. The LoadCnt signal is extended until a change in the toggle is detected and is used to load the counter.

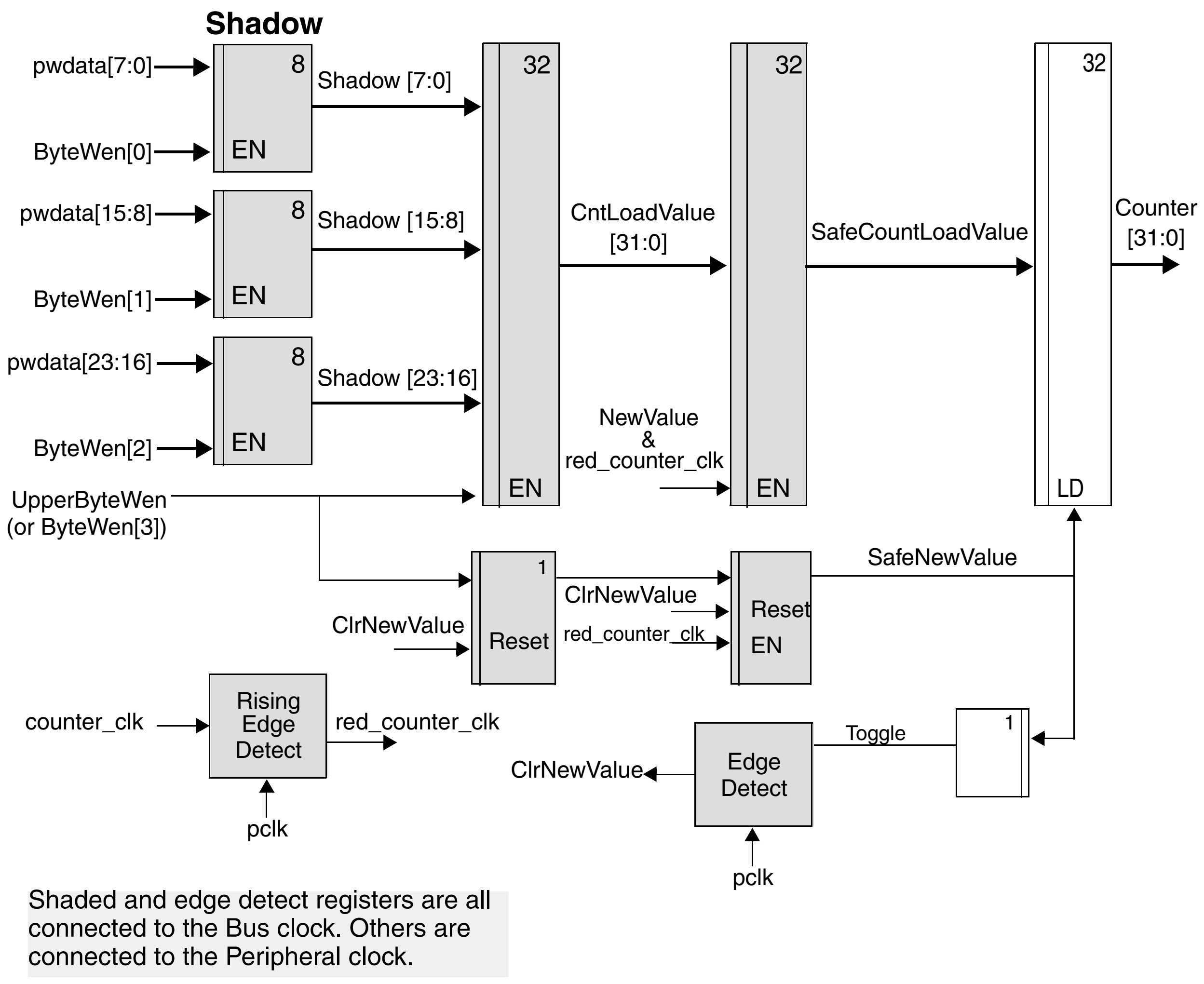
Figure 7-9 Coherent Loading – Synchronous Clocks



7.8.1.3 Asynchronous Clocks

When the clocks are asynchronous, the processor clock needs to be three-times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock. The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are asynchronous.

Figure 7-10 Coherent Loading – Asynchronous Clocks

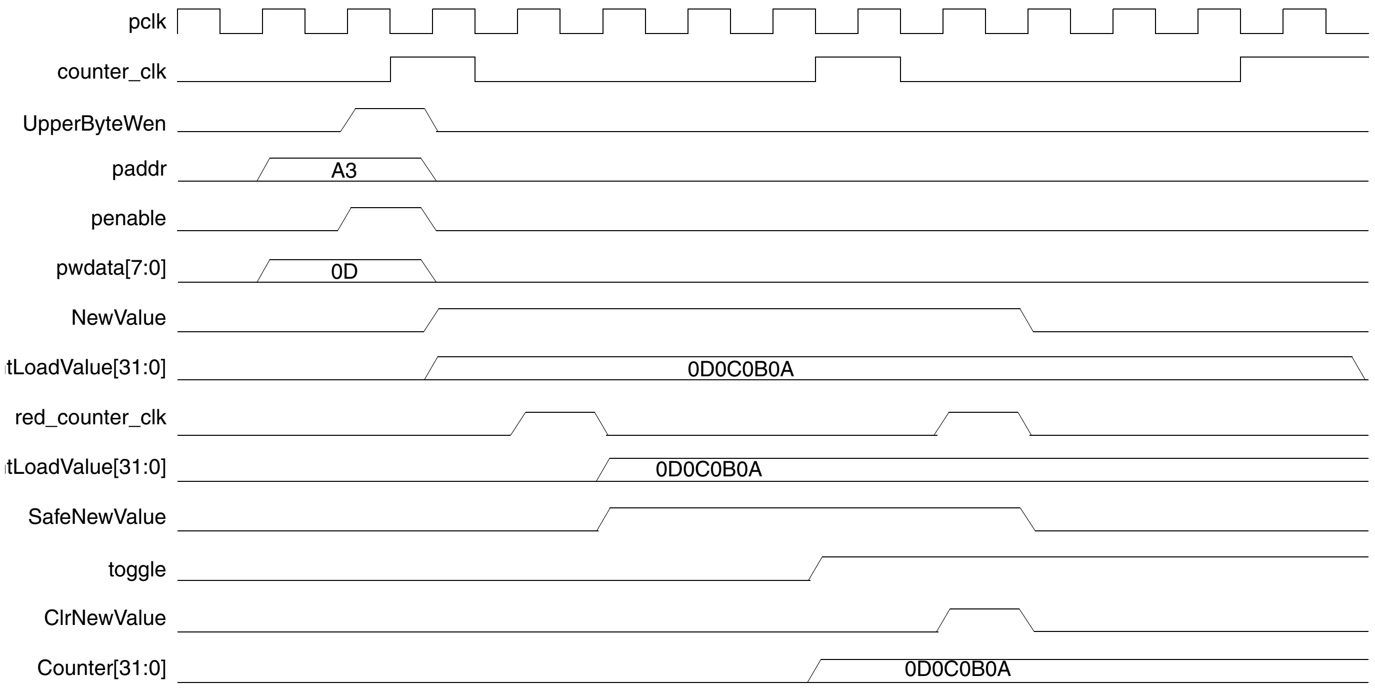


When the clocks are asynchronous, you need to transfer the contents of the register from one clock domain to another. It is not desirable to transfer the entire register through meta-stability registers, as coherency is not guaranteed with this method. The circuitry needed requires the processor clock to be used to re-time the peripheral clock. Upon a rising edge of the re-timed clock, the new value signal, NewValue, is transferred into a safe new value signal, SafeNewValue, which happens after the edge of the peripheral clock has occurred.

Every time there is a rising edge of the peripheral clock detected, the CntLoadValue is transferred into a SafeCntLoadValue. This value is used to transfer the load value across the clock domains. The SafeCntLoadValue only changes a number of bus clock cycles after the peripheral clock edge changes. A counter running on the peripheral clock is able to use this value safely. It could be up to two peripheral clock periods before the value is loaded into the counter. Along with this loaded value, there also is a single bit transferred that is used to qualify the loading of the value into the counter.

The timing diagram depicted in the following figure does not show the shadow registers being loaded. This is identical to the loading for the other clock modes.

Figure 7-11 Coherent Loading – Asynchronous Clocks



The NewValue signal is extended until a change in the toggle is detected and is used to update the safe value. The SafeNewValue is used to load the counter at the rising edge of the peripheral clock. Each time a new value is written the toggle bit is flipped and the edge detection of the toggle is used to remove both the NewValue and the SafeNewValue.

7.8.2 Reading Coherently

For writing to registers, an upper-byte concept is proposed for solving coherency issues. For read transactions, a lower-byte concept is required. The following table provides the relationship between the register width and the bus width for the generation of the correct lower byte.

Table 7-6 Lower Byte Generation

	Lower Byte Bus Width		
Counter Register Width	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	0	NCR	NCR

Table 7-6 Lower Byte Generation

	Lower Byte Bus Width		
17 - 24	0	0	NCR
25 - 32	0	0	NCR

Depending on the bus width and the register width, there may be no need to save the upper bits because the entire register is read in one access, in which case there is no problem with coherency. When the lower byte is read, the remaining upper bytes within the counter register are transferred into a holding register. The holding register is the source for the remaining upper bytes. Users must read LSB to MSB for this solution to operate correctly. NCR means that no coherency circuitry is required, as the entire register is read with one access.

There are two cases regarding the relationship between the processor and peripheral clocks to be considered as follows:

- Identical and/or synchronous
- Asynchronous

7.8.2.1 Synchronous Clocks

When the clocks are identical and/or synchronous, the remaining unread bits (if any) need to be saved into a holding register once a read is started. The first read byte must be the lower byte provided in the previous table, which causes the other bits to be moved into the holding register, SafeCntVal, provided that the register cannot be read in one access. The upper bytes of the register are read from the holding register rather than the actual register so that the value read is coherent. This is illustrated in the following figure and in the timing diagram after it.

Figure 7-12 Coherent Registering – Synchronous Clocks

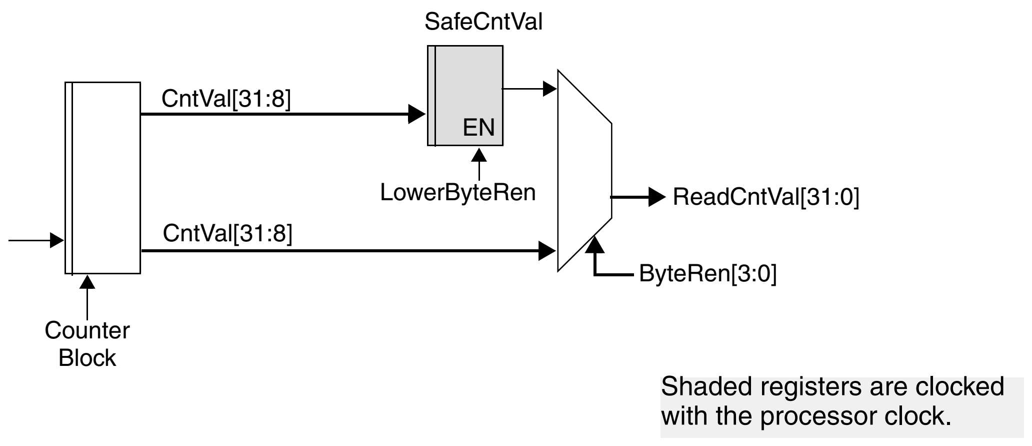
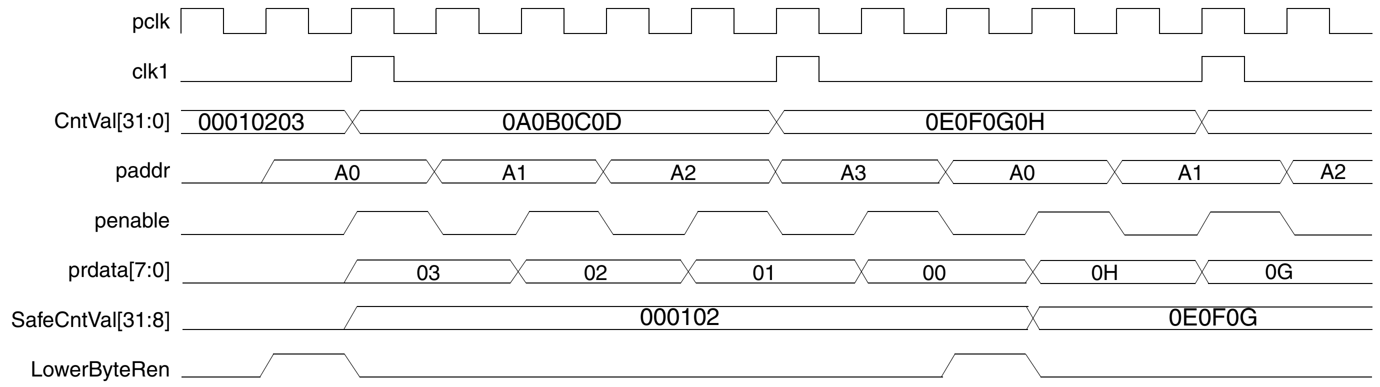


Figure 7-13 Coherent Registering – Synchronous Clocks



7.8.2.2 Asynchronous Clocks

When the clocks are asynchronous, the processor clock needs to be three times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock.

To safely transfer a counter value from the counter clock domain to the bus clock domain, the counter clock signal should be transferred to the bus clock domain. When the rising edge detect of this re-timed counter clock signal is detected, it is safe to use the counter value to update a shadow register that holds the current value of the counter.

While reading the counter contents it may take multiple APB transfers to read the value.

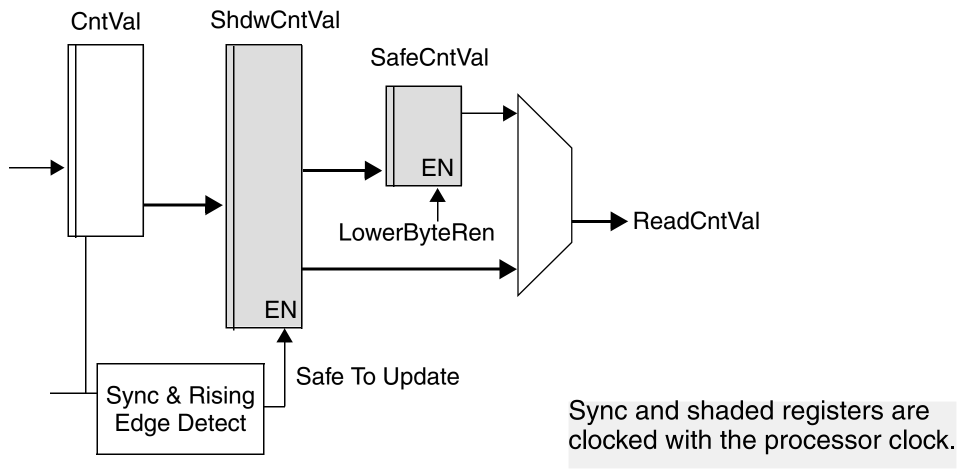
Note You must read LSB to MSB when the bus width is narrower than the counter width.

Once a read transaction has started, the value of the upper register bits need to be stored into a shadow register so that they can be read with subsequent read accesses. Storing these upper bits preserves the coherency of the value that is being read. When the processor reads the current value it actually reads the contents of the shadow register instead of the actual counter value. The holding register is read when the bus width is narrower than the counter width. When the LSB is read, the value comes from the shadow register; when the remaining bytes are read they come from the holding register. If the data bus width is wide enough to read the counter in one access, then the holding registers do not exist.

The counter clock is registered and successively pipelined to sense a rising edge on the counter clock. Having detected the rising edge, the value from the counter is known to be stable and can be transferred into the shadow register. The coherency of the counter value is maintained before it is transferred, because the value is stable.

The following figure illustrates the synchronization of the counter clock and the update of the shadow register.

Figure 7-14 Coherency and Shadow Registering – Asynchronous Clocks


CntVal
ShdwCntVal
SafeCntVal
EN
LowerByteRen
ReadCntVal
EN
Safe To Update
Sync & Rising Edge Detect
Sync and shaded registers are clocked with the processor clock.

A
DesignWare Constants

Table A-1 provides the DesignWare bus constant definitions. These definitions can also be found in DW_amba_constants.v file in the src directory of your DW_apb coreKit.

Table A-1 DesignWare Bus Constant Definitions

DesignWare Constant	Value
HBURST_WIDTH	3
HMASTER_WIDTH	4
HPROT_WIDTH	4
HRESP_WIDTH	2
HSIZE_WIDTH	3
HSPLIT_WIDTH	16
HTRANS_WIDTH	2
HBURST Values	
SINGLE	3‘b000
INCR	3‘b001
WRAP4	3‘b010
INCR4	3‘b011
WRAP8	3‘b100
INCR8	3‘b101
WRAP16	3‘b110
INCR16	3‘b111
HRESP Values	
OKAY	2‘b00

Table A-1 DesignWare Bus Constant Definitions (Continued)

DesignWare Constant	Value
ERROR	2‘b01
RETRY	2‘b10
SPLIT	2‘b11
HSIZE Values	
BYTE	3‘b000
HWORD	3‘b001
WORD	3‘b010
LWORD	3‘b011
DWORD	3‘b100
WORD4	3‘b101
WORD8	3‘b110
WORD16	3‘b111
HTRANS Values	
IDLE	2‘b00
BUSY	2‘b01
NONSEQ	2‘b10
SEQ	2‘b11
HWRITE/PWRITE Values	
READ	1‘b0
WRITE	1‘b1
Generic Definitions	
TRUE	1‘b1
FALSE	1‘b0
zero8	8‘b0
zero16	16‘b0
zero32	32‘b0
KBYTE	1024

B Glossary

active command queue	Command queue from which a model is currently taking commands; see also command queue.
activity	A set of functions in coreConsultant that step you through configuration, verification, and synthesis of a selected core.
AHB	Advanced High-performance Bus — high-performance system backbone bus. AHB supports the efficient connection of processors, on-chip memories and off-chip external memory interfaces (ARM Limited specification).
AMBA	Advanced Microcontroller Bus Architecture — a trademarked name by ARM Limited that defines an on-chip communication standard for high speed microcontrollers.
APB	Advanced Peripheral Bus — optimized for minimal power consumption and reduced interface complexity to support peripheral functions (ARM Limited specification).
APB bridge	DW_apb submodule that converts protocol between the AHB bus and APB bus.
application design	Overall chip-level design into which a subsystem or subsystems are integrated.
arbiter	AMBA bus submodule that arbitrates bus activity between masters and slaves.
BFM	Bus-Functional Model — A simulation model used for early hardware debug. A BFM simulates the bus cycles of a device and models device pins, as well as certain on-chip functions. See also Full-Functional Model.
big-endian	Data format in which most significant byte comes first; normal order of bytes in a word.
blocked command stream	A command stream that is blocked due to a blocking command issued to that stream; see also command stream, blocking command, and non-blocking command.
blocking command	A command that prevents a testbench from advancing to next testbench statement until this command executes in model. Blocking commands typically return data to the testbench from the model.

bus bridge	Logic that handles the interface and transactions between two bus standards, such as AHB and APB. See APB bridge.
command channel	Manages command streams. Models with multiple command channels execute command streams independently of each other to provide full-duplex mode function.
command stream	The communication channel between the testbench and the model.
component	A generic term that can refer to any synthesizable IP or verification IP in the DesignWare Library. In the context of synthesizable IP, this is a configurable block that can be instantiated as a single entity (VHDL) or module (Verilog) in a design.
configuration	The act of specifying parameters for a core prior to synthesis; can also be used in the context of VIP.
configuration intent	Range of values allowed for each parameter associated with a reusable core.
core	Any configurable block of synthesizable IP that can be instantiated as a single entity (VHDL) or module (Verilog) in a design. Core is the preferred term for a big piece of IIP. Anything that requires coreConsultant for configuration, as well as anything in the DesignWare Cores library, is a core.
core developer	Person or company who creates or packages a reusable core. All the cores in the DesignWare Library are developed by Synopsys.
core integrator	Person who uses coreConsultant or coreAssembler to incorporate reusable cores into a system-level design.
coreAssembler	Synopsys product that enables automatic connection of a group of cores into a subsystem. Generates RTL and gate-level views of the entire subsystem.
coreConsultant	A Synopsys product that lets you configure a core and generate the design views and synthesis views you need to integrate the core into your design. Can also synthesize the core and run the unit-level testbench supplied with the core.
coreKit	An unconfigured core and associated files, including the core itself, a specified synthesis methodology, interfaces definitions, and optional items such as verification environment files and core-specific documentation.
cycle command	A command that executes and causes HDL simulation time to advance.
decoder	Software or hardware subsystem that translates from and “encoded” format back to standard format.
design context	Aspects of a component or subsystem target environment that affect the synthesis of the component or subsystem.
design creation	The process of capturing a design as parameterized RTL.
Design View	A simulation model for a core generated by coreConsultant.
DesignWare Synthesizable Components	The Synopsys name for the collection of AMBA-compliant coreKits and verification models delivered with DesignWare and used with coreConsultant or coreAssembler to quickly build DesignWare Synthesizable Component designs.

DesignWare cores	A specific collection of synthesizable cores that are licensed individually. For more information, refer to www.synopsys.com/designware.
DesignWare Library	A collection of synthesizable IP and verification IP components that is authorized by a single DesignWare license. Products include SmartModels, VMT model suites, DesignWare Memory Models, Building Block IP, and the DesignWare Synthesizable Components.
dual role device	Device having the capabilities of function and host (limited).
endian	Ordering of bytes in a multi-byte word; see also little-endian and big-endian.
Full-Functional Mode	A simulation model that describes the complete range of device behavior, including code execution. See also BFM.
GPIO	General Purpose Input Output.
GTECH	A generic technology view used for RTL simulation of encrypted source code by non-Synopsys simulators.
hard IP	Non-synthesizable implementation IP.
HDL	Hardware Description Language – examples include Verilog and VHDL.
IIP	Implementation Intellectual Property — A generic term for synthesizable HDL and non-synthesizable “hard” IP in all of its forms (coreKit, component, core, MacroCell, and so on).
implementation view	The RTL for a core. You can simulate, synthesize, and implement this view of a core in a real chip.
instantiate	The act of placing a core or model into a design.
interface	Set of ports and parameters that defines a connection point to a component.
IP	Intellectual property — A term that encompasses simulation models and synthesizable blocks of HDL code.
little-endian	Data format in which the least-significant byte comes first.
MacroCell	Bigger IP blocks (6811, 8051, memory controller) available in the DesignWare Library and delivered with coreConsultant.
master	Device or model that initiates and controls another device or peripheral.
model	A Verification IP component or a Design View of a core.
monitor	A device or model that gathers performance statistics of a system.
non-blocking command	A testbench command that advances to the next testbench statement without waiting for the command to complete.
peripheral	Generally refers to a small core that has a bus connection, specifically an APB interface.

RTL	Register Transfer Level. A higher level of abstraction that implies a certain gate-level structure. Synthesis of RTL code yields a gate-level design.
SDRAM	Synchronous Dynamic Random Access Memory; high-speed DRAM adds a separate clock signal to control signals.
SDRAM controller	A memory controller with specific connections for SDRAMs.
slave	Device or model that is controlled by and responds to a master.
SoC	System on a chip.
soft IP	Any implementation IP that is configurable. Generally referred to as synthesizable IP.
static controller	Memory controller with specific connections for Static memories such as asynchronous SRAMs, Flash memory, and ROMs.
subsystem	In relation to coreAssembler, highest level of RTL that is automatically generated.
synthesis intent	Attributes that a core developer applies to a top-level design, ports, and core.
synthesizable IP	A type of Implementation IP that can be mapped to a target technology through synthesis. Sometimes referred to as Soft IP.
technology-independent	Design that allows the technology (that is, the library that implements the gate and via widths for gates) to be specified later during synthesis.
Testsuite Regression Environment (TRE)	A collection of files for stand-alone verification of the configured component. The files, tests, and functionality vary from component to component.
VIP	Verification Intellectual Property — A generic term for a simulation model in any form, including a Design View.
workspace	A network location that contains a personal copy of a component or subsystem. After you configure the component or subsystem (using coreConsultant or coreAssembler), the workspace contains the configured component/subsystem and generated views needed for integration of the component/subsystem at the top level.
wrap, wrapper	Code, usually VHDL or Verilog, that surrounds a design or model, allowing easier interfacing. Usually requires an extra, sometimes automated, step to create the wrapper.
zero-cycle command	A command that executes without HDL simulation time advancing.

Index