DesignWare DW_apb Databook

DW_apb* – *Product Code

3.02a
July 2018

Synopsys, Inc.
690 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
3.02a	July 2018	**Updated:** ■ Version changed for 2018.07a release ■ "Performance" on page 110 ■ "Parameter Descriptions" on page 37, "Signal Descriptions" on page 41, and "Internal Parameter Descriptions" on page 77 are auto-extracted with change bars from the RTL **Removed:** ■ Chapter 2, "Building and Verifying a Component or Subsystem" and added the contents in the newly created user guide.
3.01a	October 2016	■ Version changed for 2016.10a release ■ Added "Back-to-Back Transfer Support on an APB Interface" on page 33 ■ "Parameter Descriptions" on page 37 auto-extracted from the RTL ■ Removed the "Running Leda on Generated Code with coreConsultant" section, and reference to Leda directory in Table 2-1 ■ Removed the "Running Leda on Generated Code with coreAssembler" section, and reference to Leda directory in Table 2-4 ■ Recreated Figure 2-20 on page 35 ■ Added "Running VCS XPROP Analyzer" ■ Moved "Internal Parameter Descriptions" to Appendix
3.00a	June 2015	■ Added "Running SpyGlass® Lint and SpyGlass® CDC" ■ Added "Running SpyGlass on Generated Code with coreAssembler" ■ "Signal Descriptions" on page 41 auto-extracted from the RTL ■ Added "Internal Parameter Descriptions" on page 77 ■ Added "APB4 Protocol Feature" on page 35 ■ Updated area and power numbers in "Performance" on page 58

(Continued)

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
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 APB Protocol Specification v2.0-compliant Advanced Peripheral Bus component. The DW_apb is a part of the DesignWare Synthesizable Components for AMBA APB Protocol Specification v2.0. 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, "Functional Description" describes the functional operation of the DW_apb.
- Chapter 3, "Parameter Descriptions" identifies the configurable parameters supported by the DW_apb.
- Chapter 4, "Signal Descriptions" provides a list and description of the DW_apb signals.
- Chapter 5, "Verification" provides information on verifying the configured DW_apb.
- Chapter 6, "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, "Internal Parameter Descriptions" provides a list of internal parameter descriptions that might be indirectly referenced in expressions in the Signals chapter.
- Appendix C, "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 APB Protocol Specification v2.0, see 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 requested information, using one of the following methods:
    - *For fastest response*, use the SolvNet website. If you fill in your information as explained, 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 **Open A Support Case** 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 earlier) 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:
    https://www.synopsys.com/support/global-support-centers.html

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 APB4 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
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

Component Name	Description
DW_axi_x2x	Flexible bridge between multiple AMBA 3 AXI components or buses

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

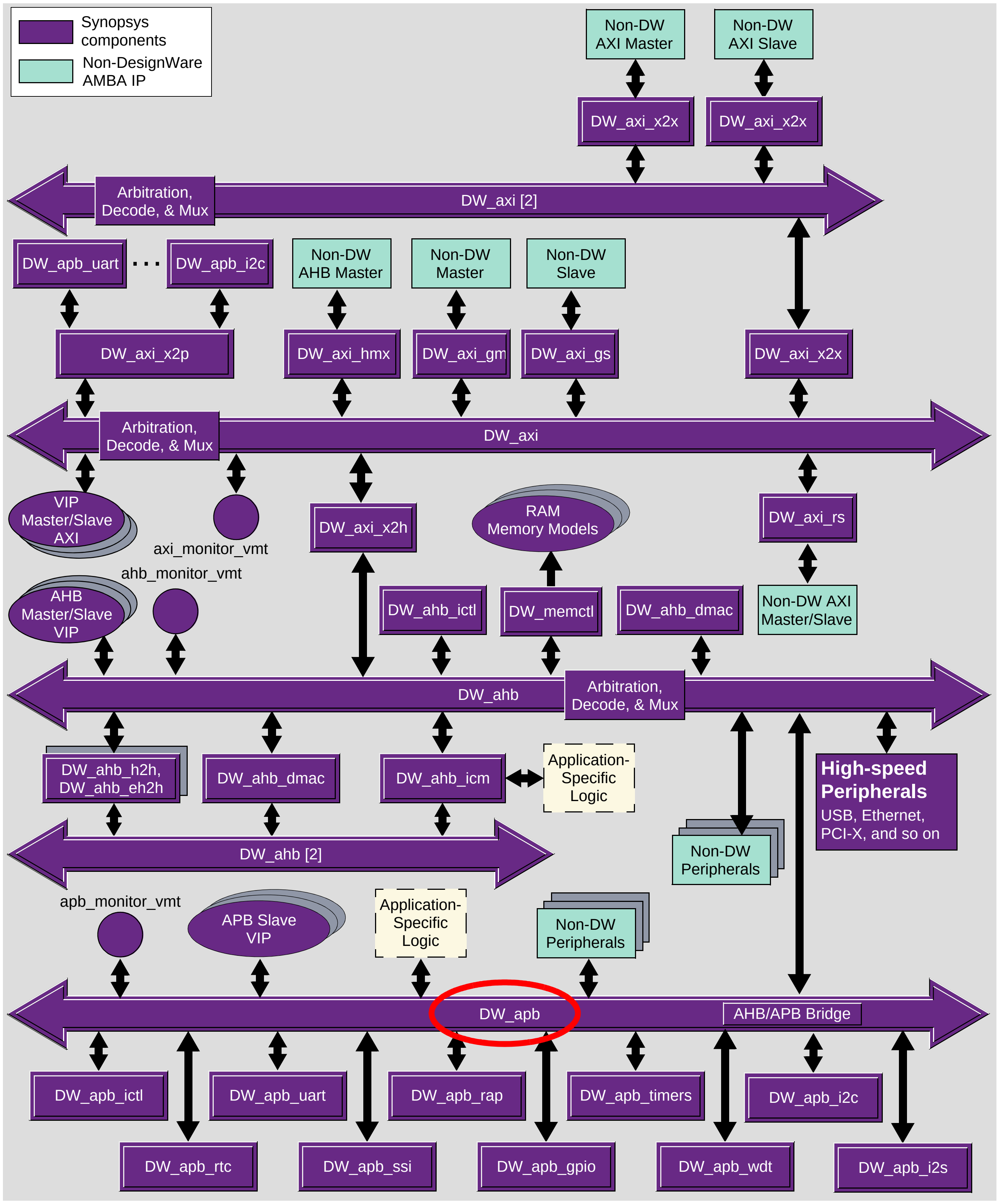
The Synopsys DesignWare Synthesizable Components environment is a parameterizable bus system containing components for the following:

- AMBA version 2.0-compliant AHB (Advanced High-performance Bus)
- AMBA APB Protocol Specification v2.0 (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.

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*.

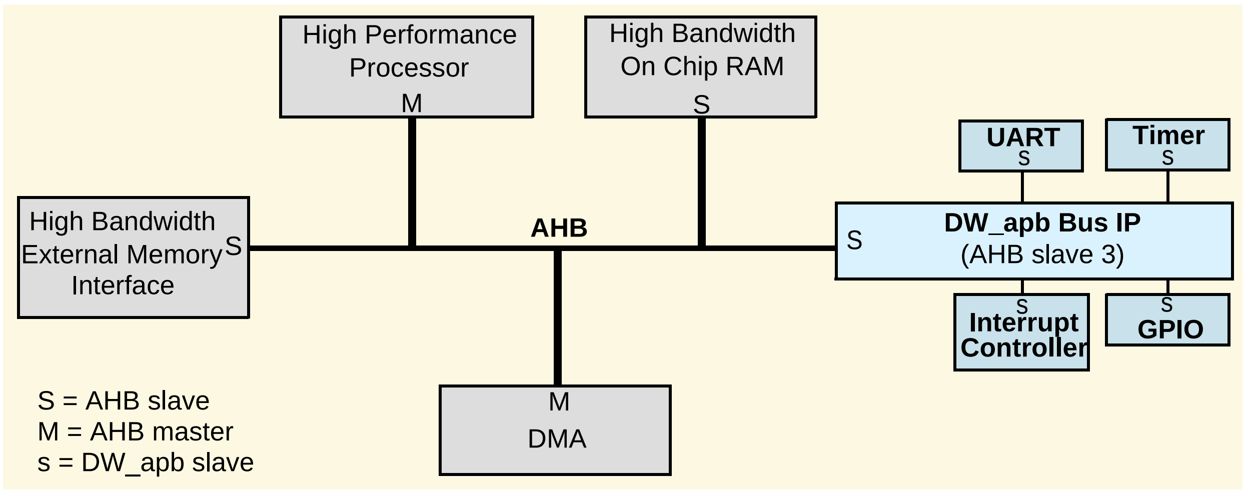
If you want to configure, synthesize, and verify a single component such as the DW_apb component, you might prefer to use coreConsultant, documentation for which is available in the *coreConsultant User Guide*.

1.2 General Product Description

The DW_apb is a parameterizable, synthesizable, and programmable component that implements the APB functionality of the *AMBA APB Protocol Specification v2.0* 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



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®
- Compliance with the *AMBA APB Protocol Specification, v2.0* from Arm®

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
- Big-endian APB peripherals

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

1.4 Standards Compliance

The DW_apb component conforms to the *AMBA Specification*, *Revision 2.0*, AMBA 3 APB Protocol Specification v1.0, and *AMBA APB Protocol Specification v2.0* 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 51 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, see “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, see *Guide to coreTools Documentation*.

For more information about configuring, synthesizing, and verifying just your DW_apb component, see “Overview of the coreConsultant Configuration and Integration Process” in DesignWare Synthesizable Components for AMBA 2 User Guide.

For more information about implementing your DW_apb component within a DesignWare subsystem using coreAssembler, see “Overview of the coreAssembler Configuration and Integration Process” in DesignWare Synthesizable Components for AMBA 2 User Guide.

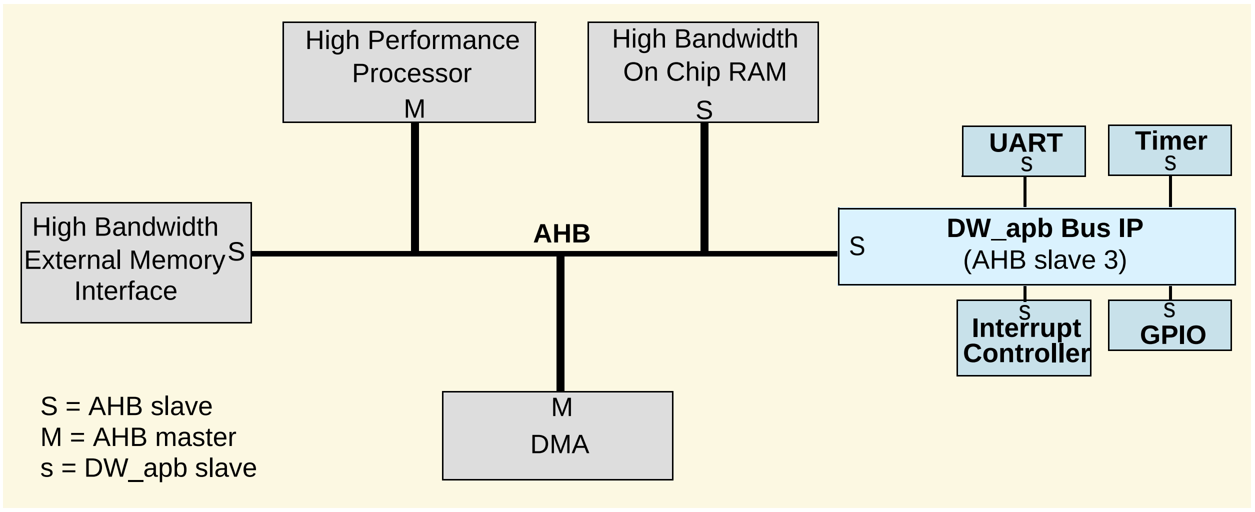
2 Functional Description

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

2.1 Overview

The DW_apb provides the interconnect fabric to connect an AHB bus to APB peripherals, which are compliant with *AMBA 2 APB Specification*, *AMBA 3 APB Protocol Specification v1.0*, or *AMBA APB Protocol Specification v2.0*. The interconnect fabric is referred to as the APB Bridge in the AMBA 2 APB specification and *AMBA APB Protocol Specification v2.0*, and simply as APB in the *AMBA 3 APB Protocol Specification v1.0*. 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 2-1.

Figure 2-1 DW_apb in an Example System



2.1.1 Block Diagram

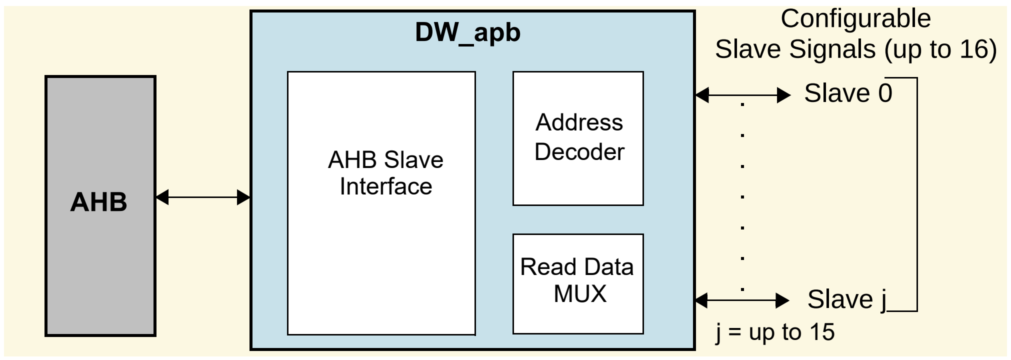
The DW_apb is configurable, synthesizable, and performs the following functions:

- 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 2-2.

Figure 2-2 DW_apb Block Diagram



2.2 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 APB4 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, see “Timing Diagrams” on page 21.

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

2.2.1 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.

2.3 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, see the text associated with Figure 2-8 on page 24.

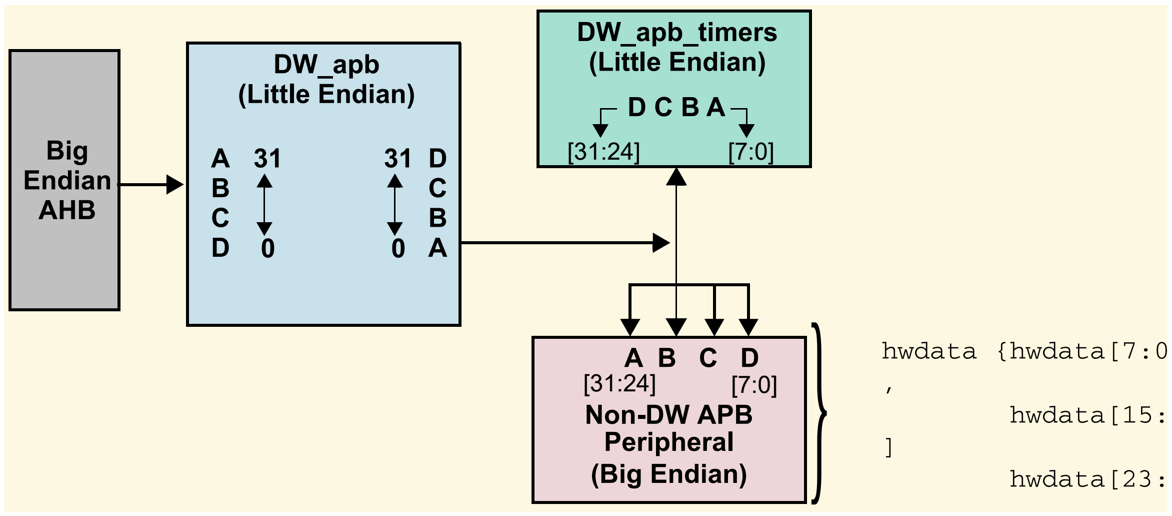
2.4 Optional External Decoder

During configuration of DW_apb, you can choose to have an external decoder. By having the decoder external to DW_apb, you 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.

2.5 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. You have to manually perform this process by swapping the bytes as illustrated in Figure 2-3.

Figure 2-3 Converting Big-Endian AHB to Big-Endian APB Peripheral

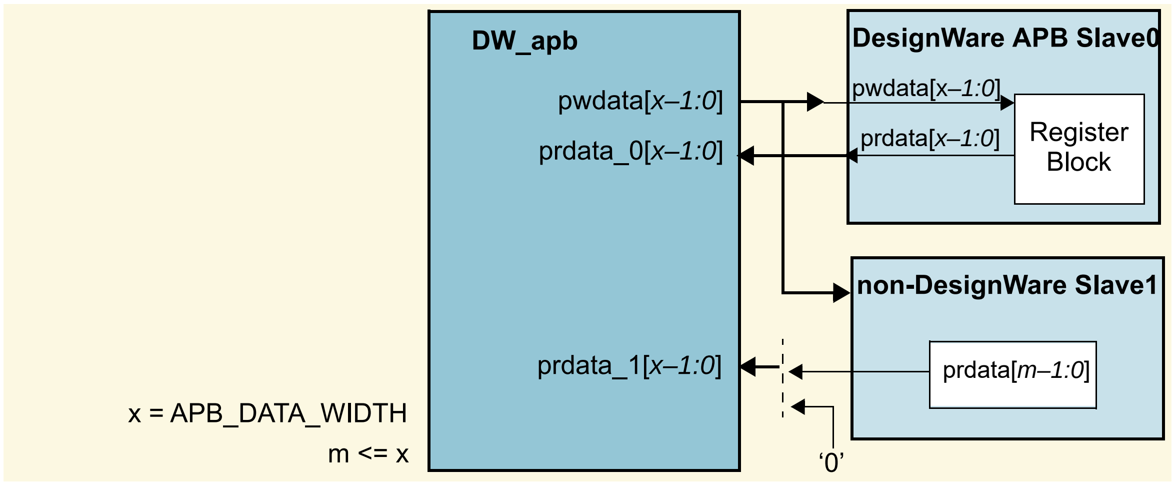


2.6 APB Slave Interface

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, you 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 2-4 shows the relationship between DesignWare and non-DesignWare APB slaves.

Figure 2-4 DW_apb and APB Slave Data Widths

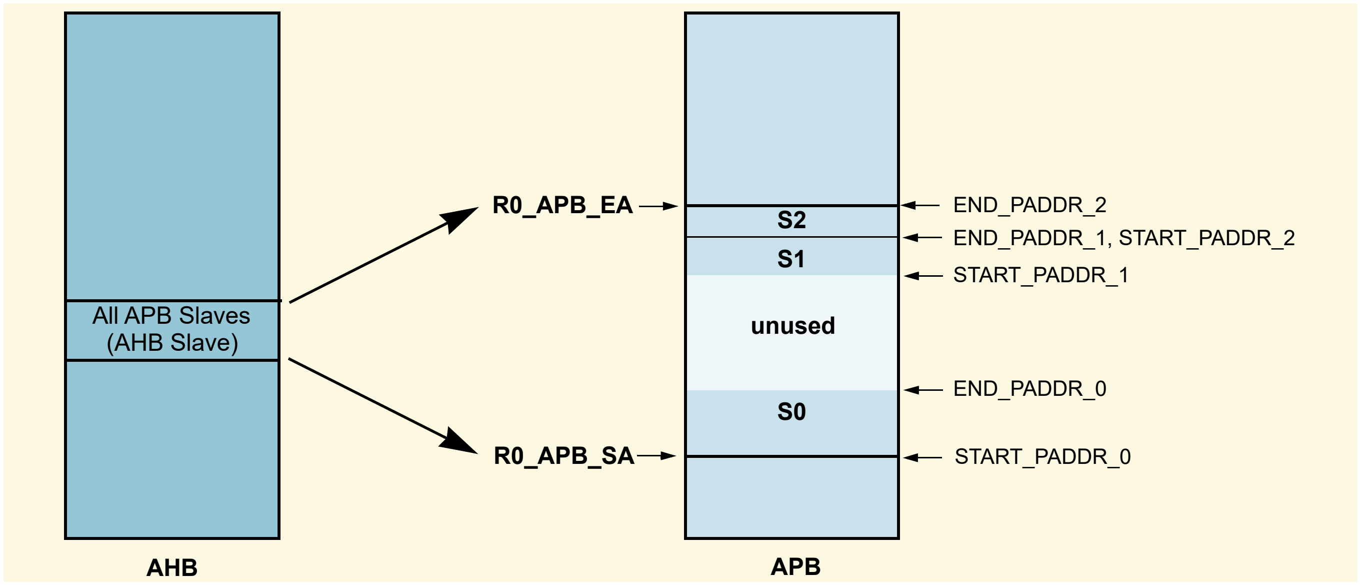


For more information about the APB data width and how it relates to DesignWare and non-DesignWare APB slaves, see "Integration Considerations" on page 57.

2.7 Memory Map

Figure 2-5 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 2-5 DW_apb Memory Map



2.8 Backward Compatibility with AMBA 2 APB and AMBA 3 APB

The AMBA 3 APB protocol has added the signals ready (pready) and error (pslverr) to the previous protocol, and the AMBA APB Protocol Specification v2.0 protocol has added the signals write-strobing (pstrb) and protection (pprot). However, APB slaves attached to the DW_apb can support either the AMBA APB Protocol Specification v2.0, AMBA 3 APB or AMBA 2 APB protocol. For each APB slave, you can use the APB_INTERFACE_TYPE_SLAVE_*i* configuration parameter to specify whether the attached component supports AMBA APB Protocol Specification v2.0, AMBA 3 APB or AMBA 2 APB. This configuration determines whether or not the freshly introduced signals are added on the I/O of the DW_apb instance. For more information on configuration parameters, see "Parameter Descriptions" on page 37.

2.9 Timing Diagrams

For timing, refer to the following diagrams:

- Read Transfer from AHB to AMBA 2 APB Slave (hclk = pclk): Figure 2-6
- Read Transfer from AHB to AMBA 3 APB Slave (hclk = pclk): Figure 2-7
- Read Transfer from AHB to AMBA 2 APB Slave(hclk != pclk): Figure 2-8
- Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk): Figure 2-9
- Write Transfer from AHB to AMBA 2 APB Slave (hclk = pclk): Figure 2-10

- Write Transfer from AHB to AMBA 3 APB Slave (hclk = pclk):Figure 2-11
- Write Transfer from AHB to AMBA 2 APB Slave (hclk != pclk): Figure 2-12
- Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk): Figure 2-13
- Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) completed with an error: Figure 2-14
- Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) completed with an error: Figure 2-15
- Back-to-back write transfer (hclk = pclk): Figure 2-16
- Back-to-back write transfer (hclk != pclk): Figure 2-17

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

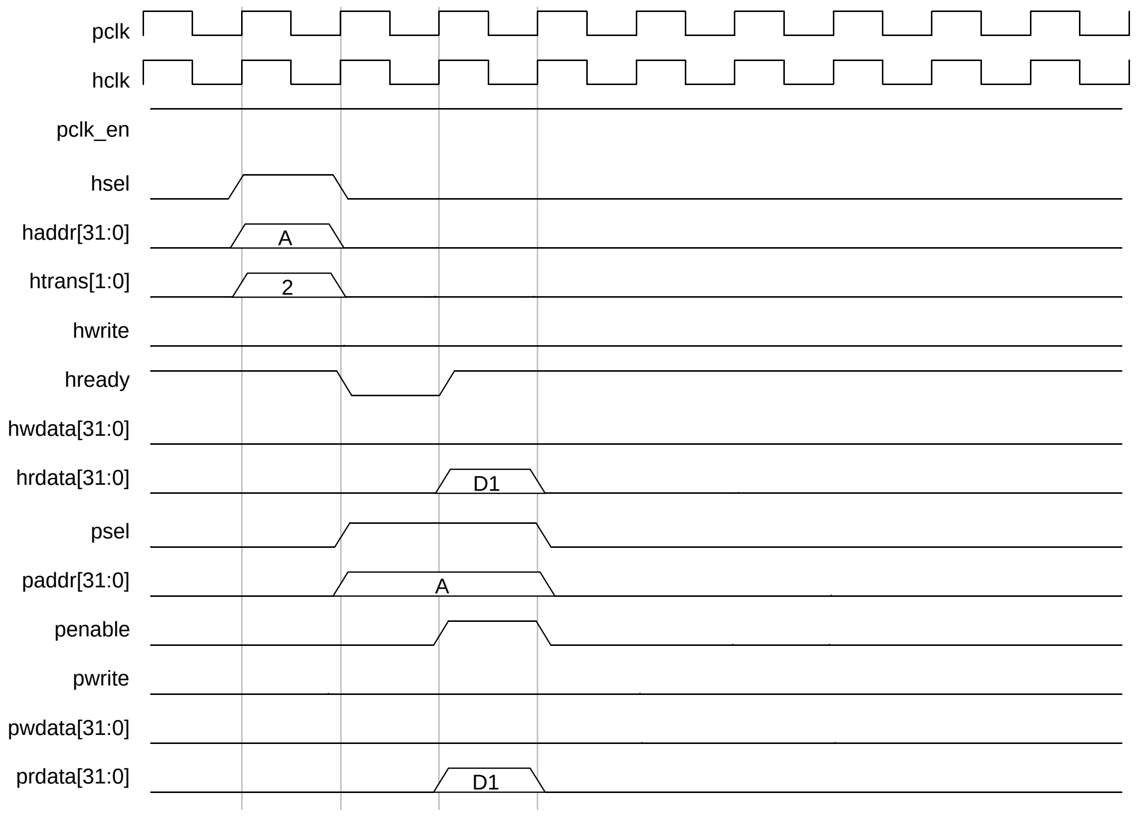


Figure 2-7 DW_apb Read Transfer from from AHB to AMBA 3 APB Slave (hclk = pclk)

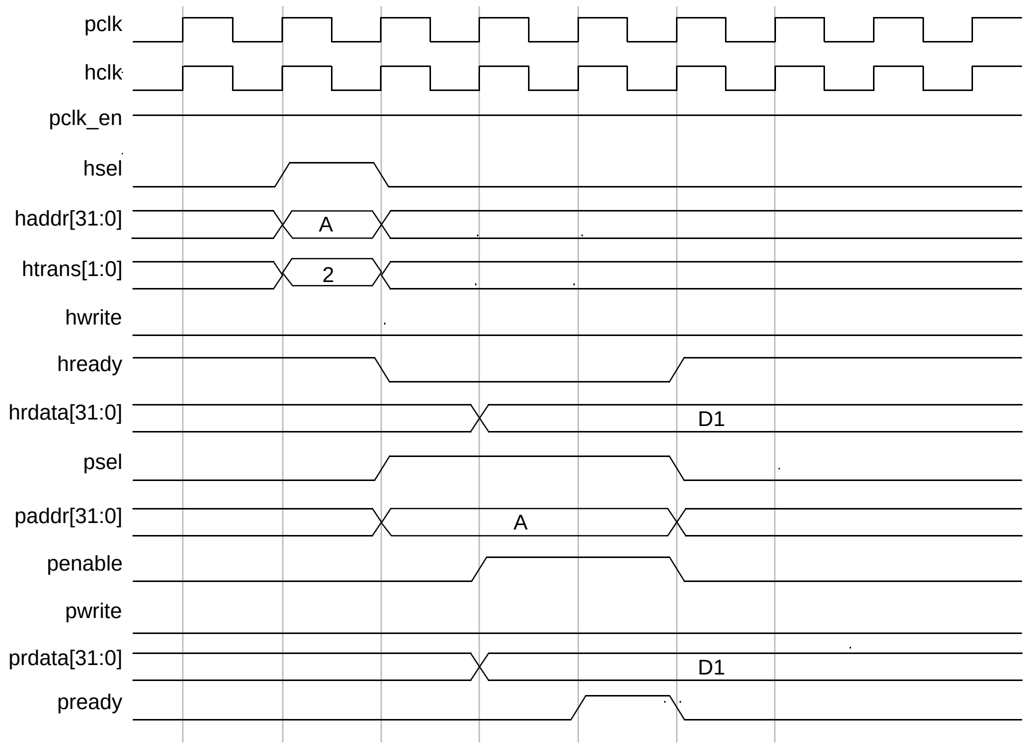
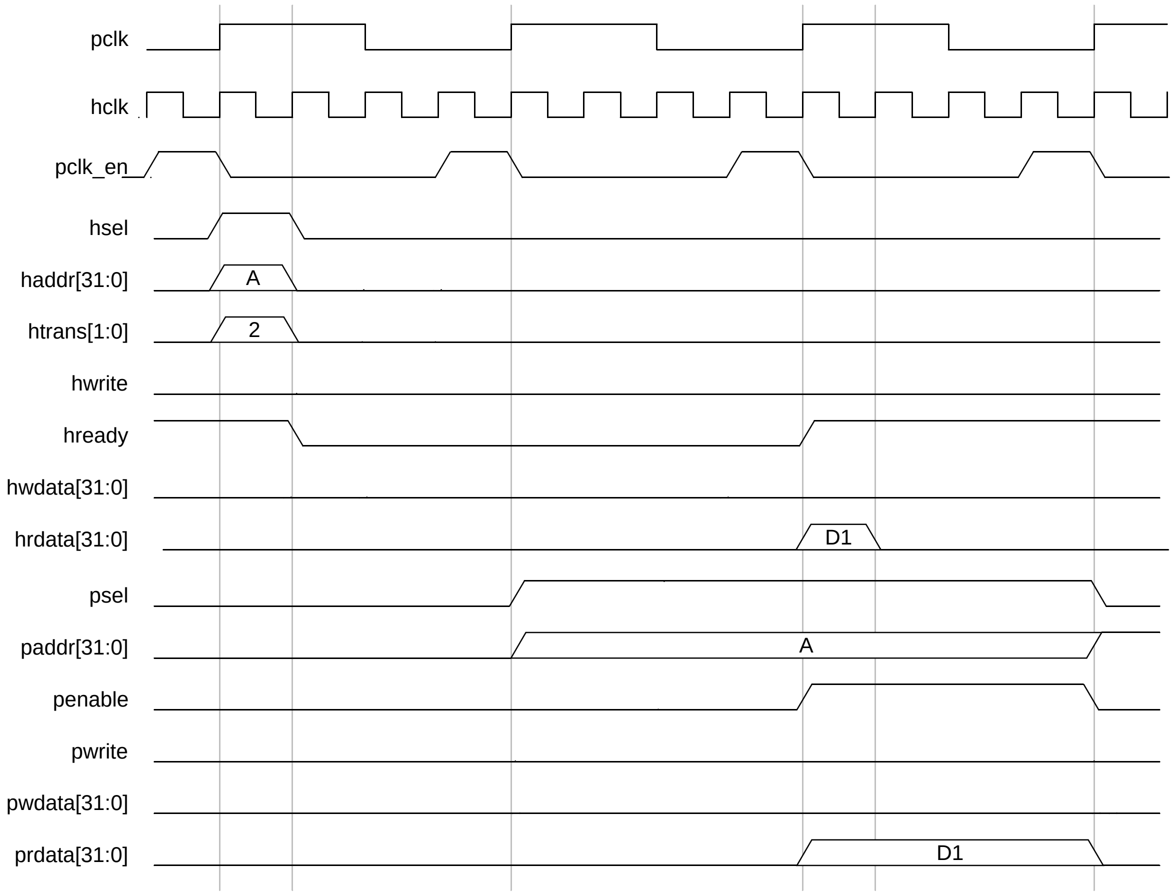


Figure 2-8 DW_apb Read Transfer from AHB to AMBA 2 APB Slave(hclk != pclk)



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 your responsibility to ensure if synthesis is done outside of coreConsultant or coreAssembler.

Figure 2-9 DW_apb Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk)

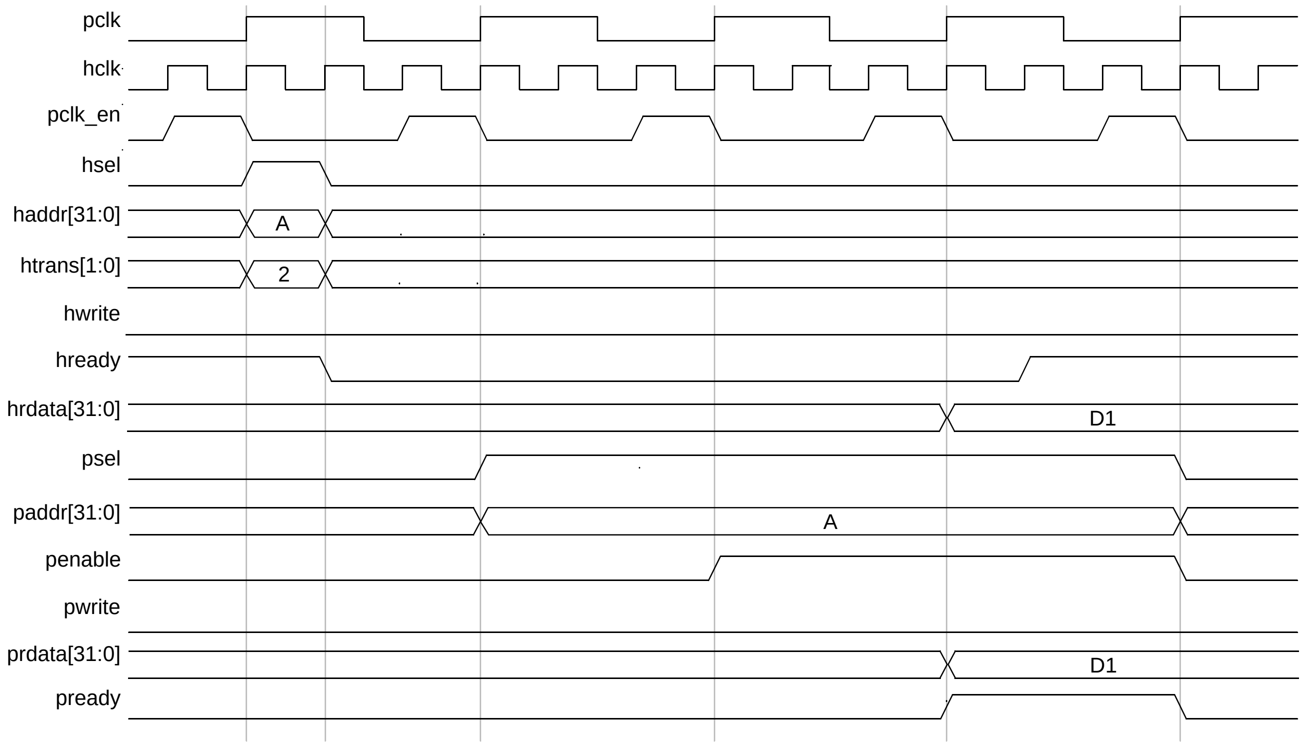


Figure 2-10 DW_apb Write Transfer from AHB to AMBA 2 APB Slave (hclk = pclk)

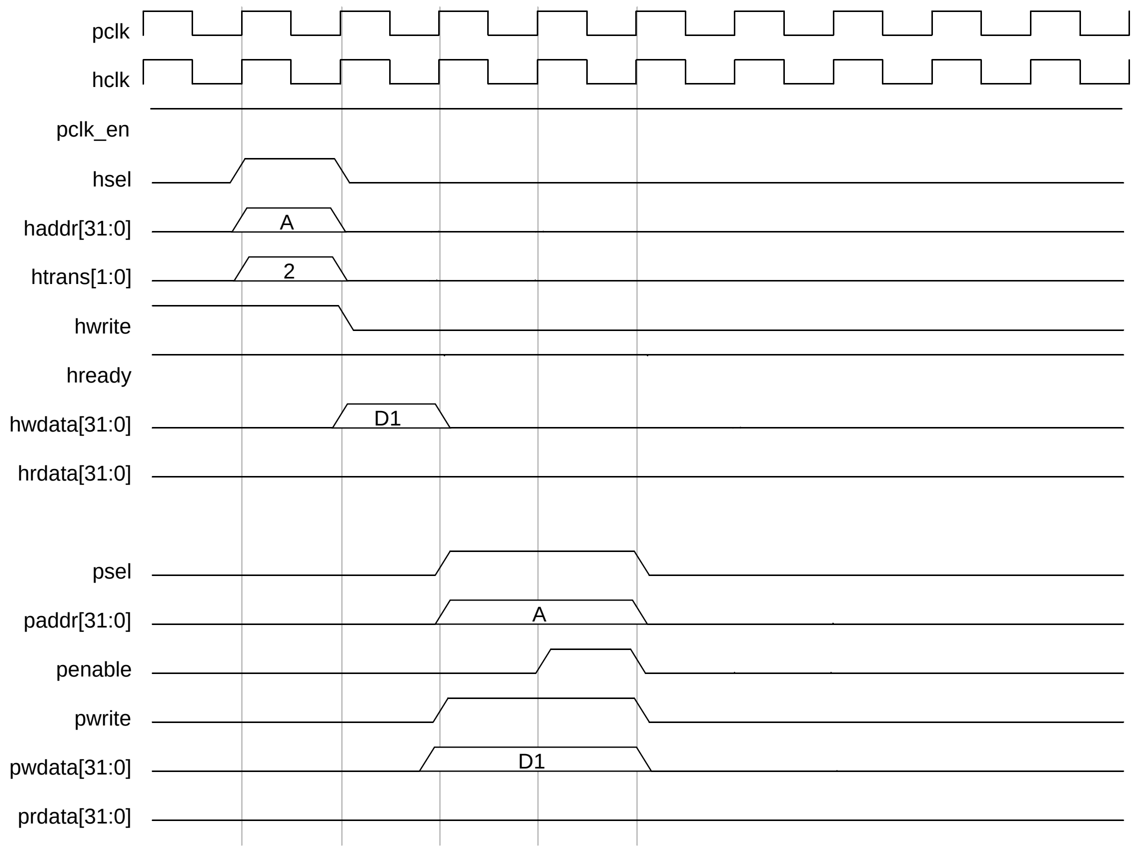


Figure 2-11 DW_apb Write Transfer from AHB to AMBA 3 APB Slave (hclk = pclk)

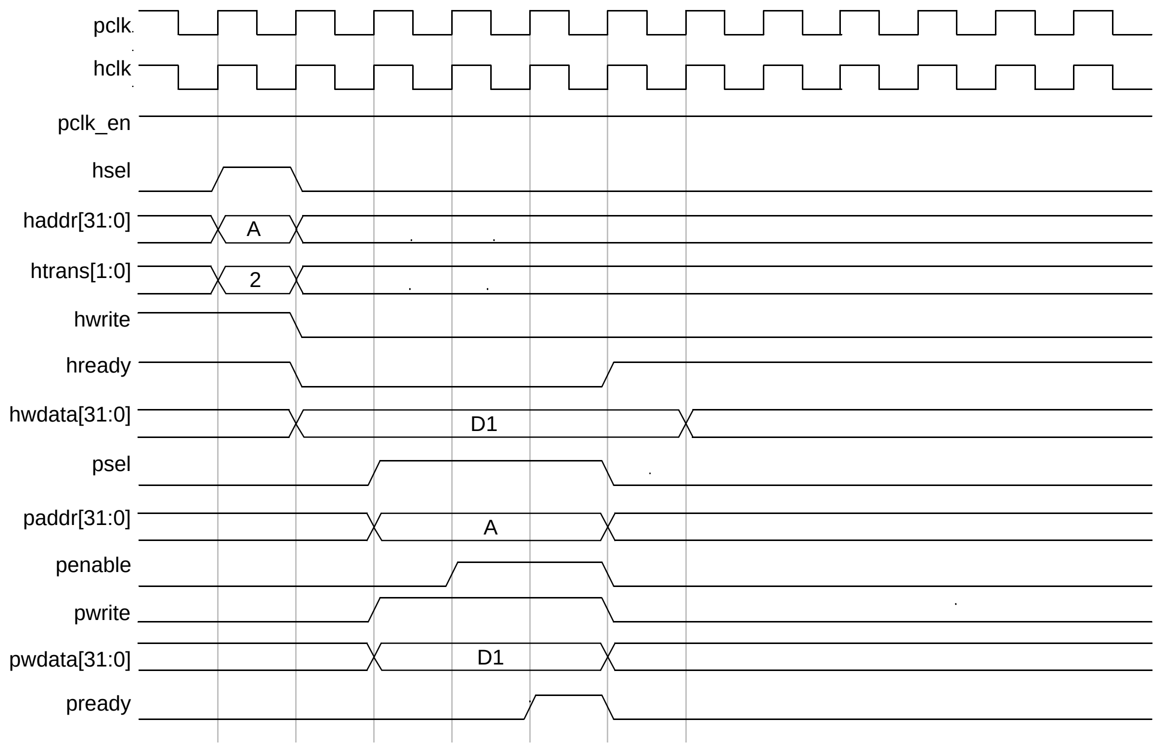


Figure 2-12 DW_apb Write Transfer from AHB to AMBA 2 APB Slave (hclk != pclk)

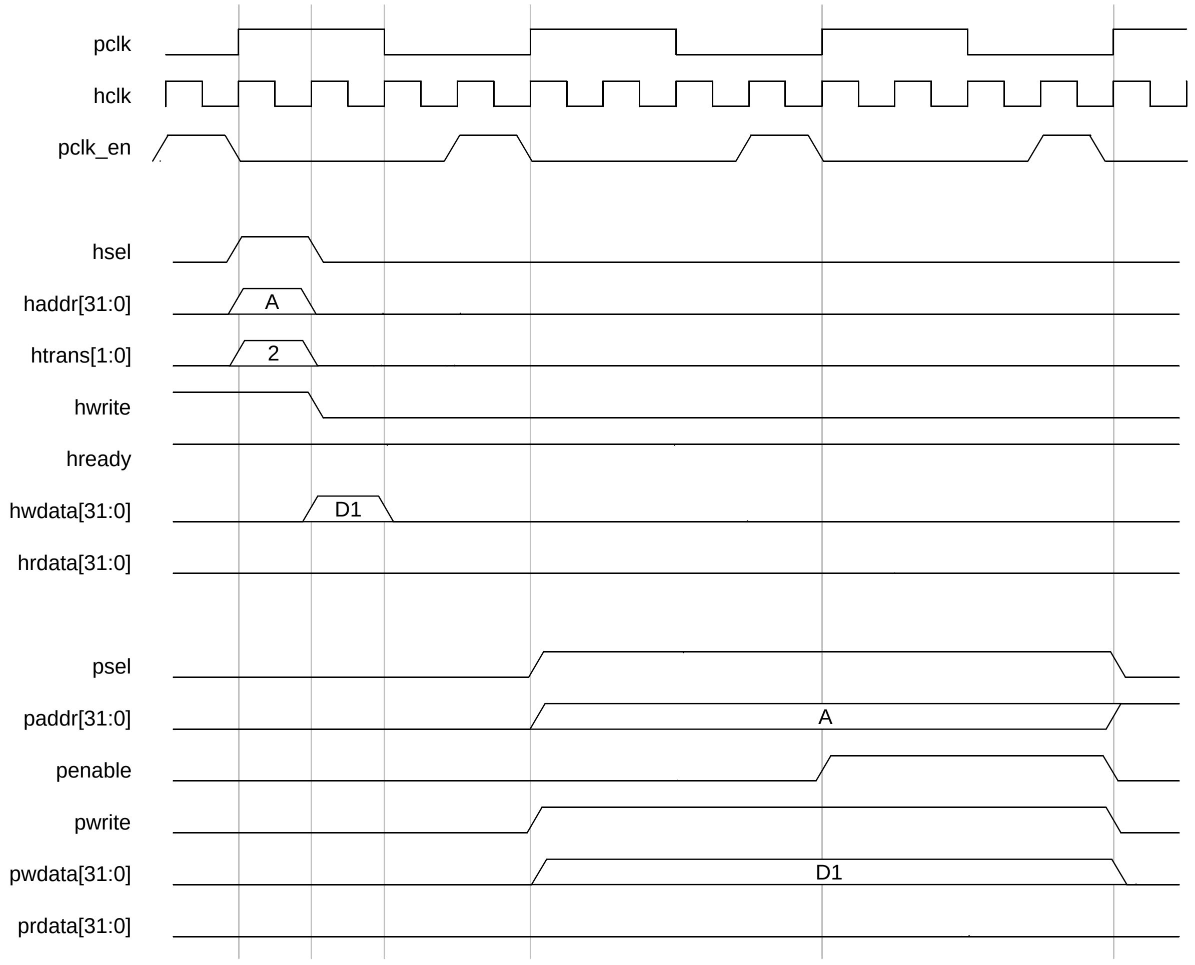


Figure 2-13 DW_apb Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk)

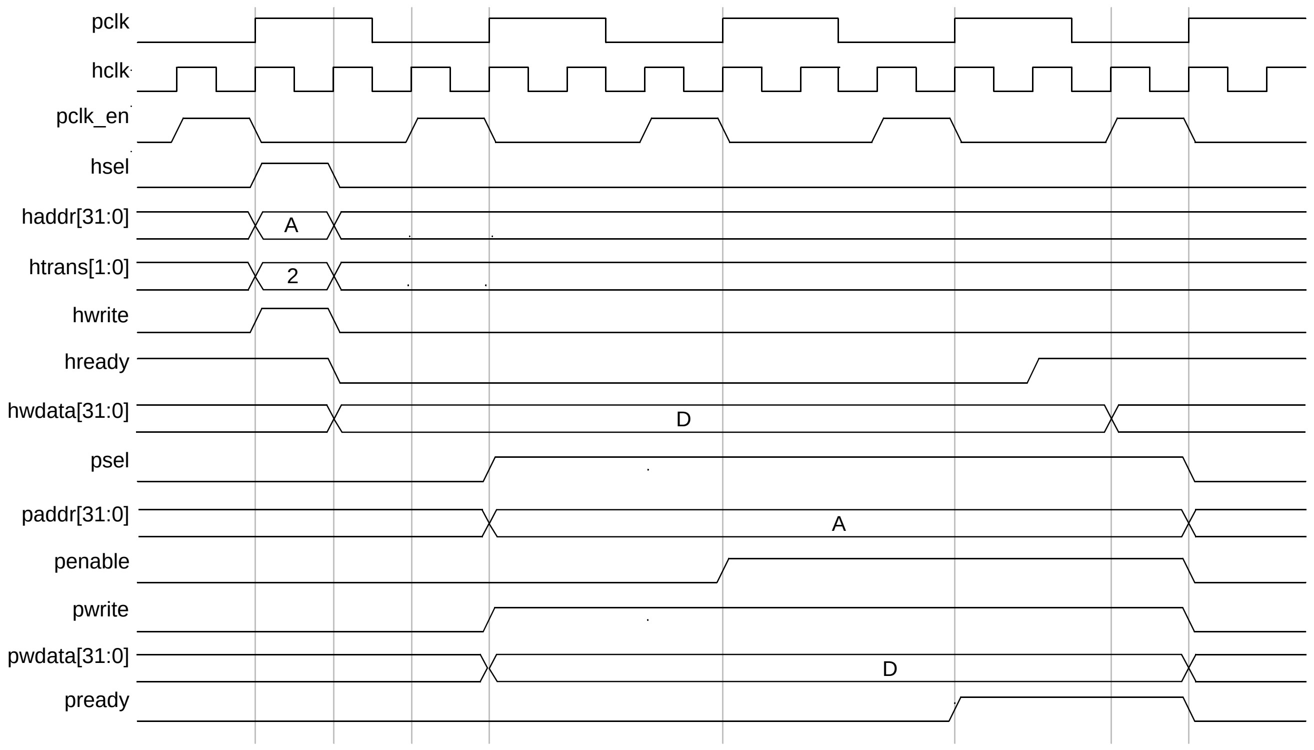


Figure 2-14 DW_apb Read Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) Completed with Error

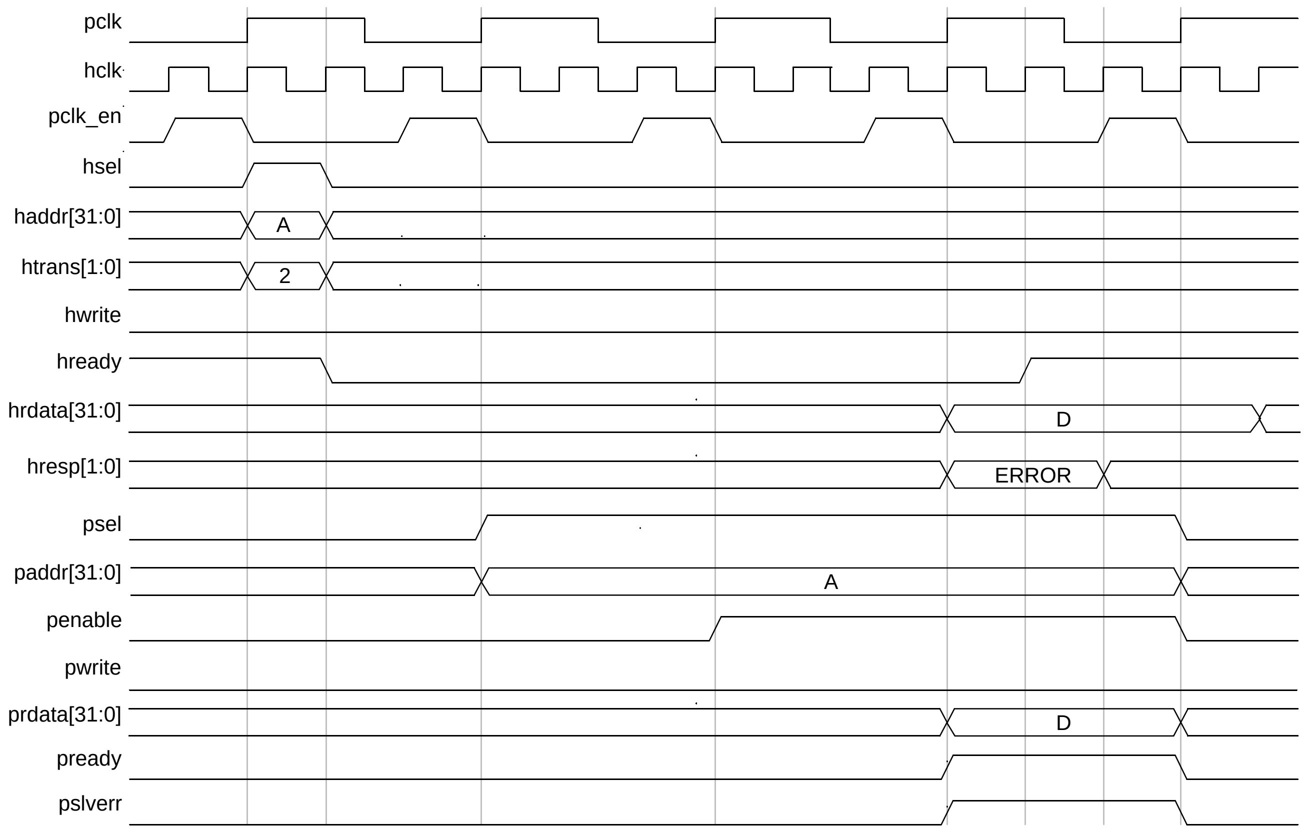
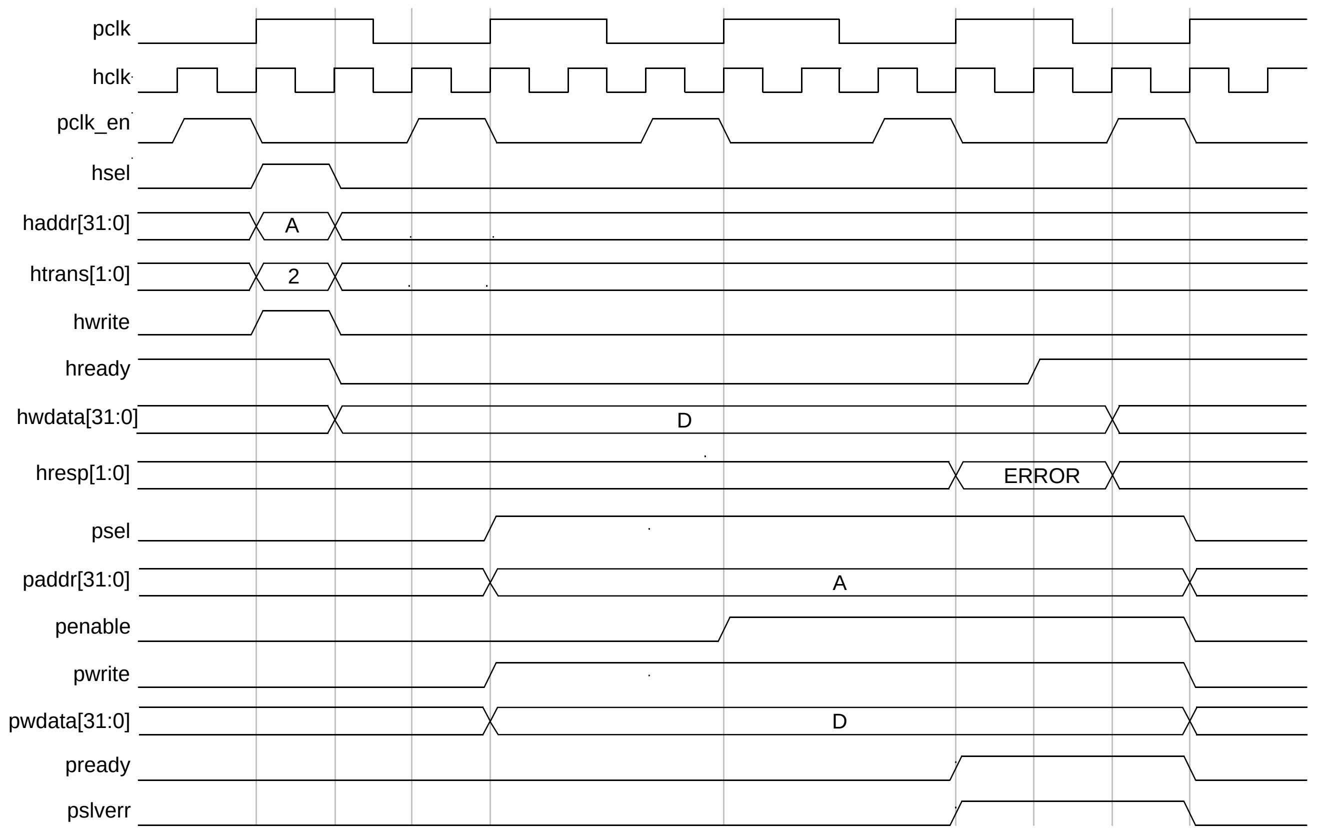


Figure 2-15 DW_apb Write Transfer from AHB to AMBA 3 APB Slave (hclk != pclk) Completed with Error



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_s*X* to hready_resp are always registered.

Figure 2-16 Back-to-Back Write Transfer (hclk = pclk)

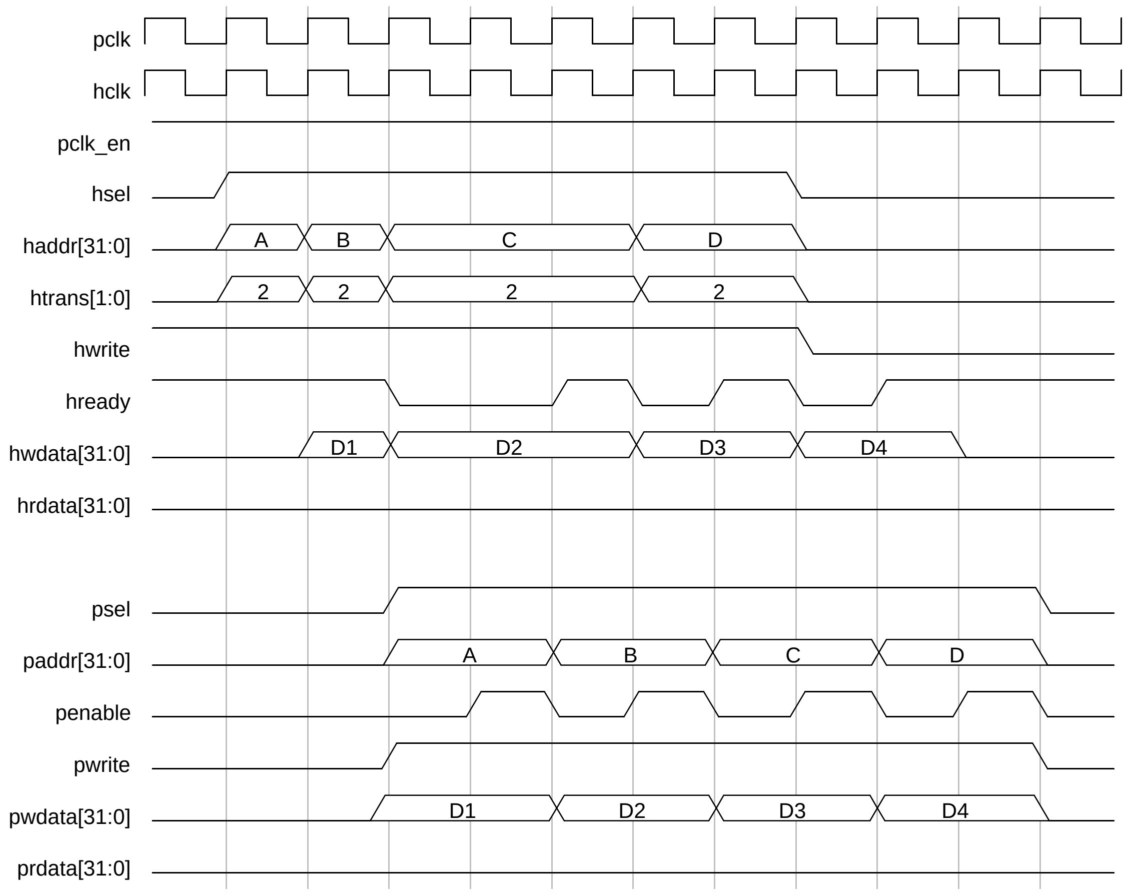
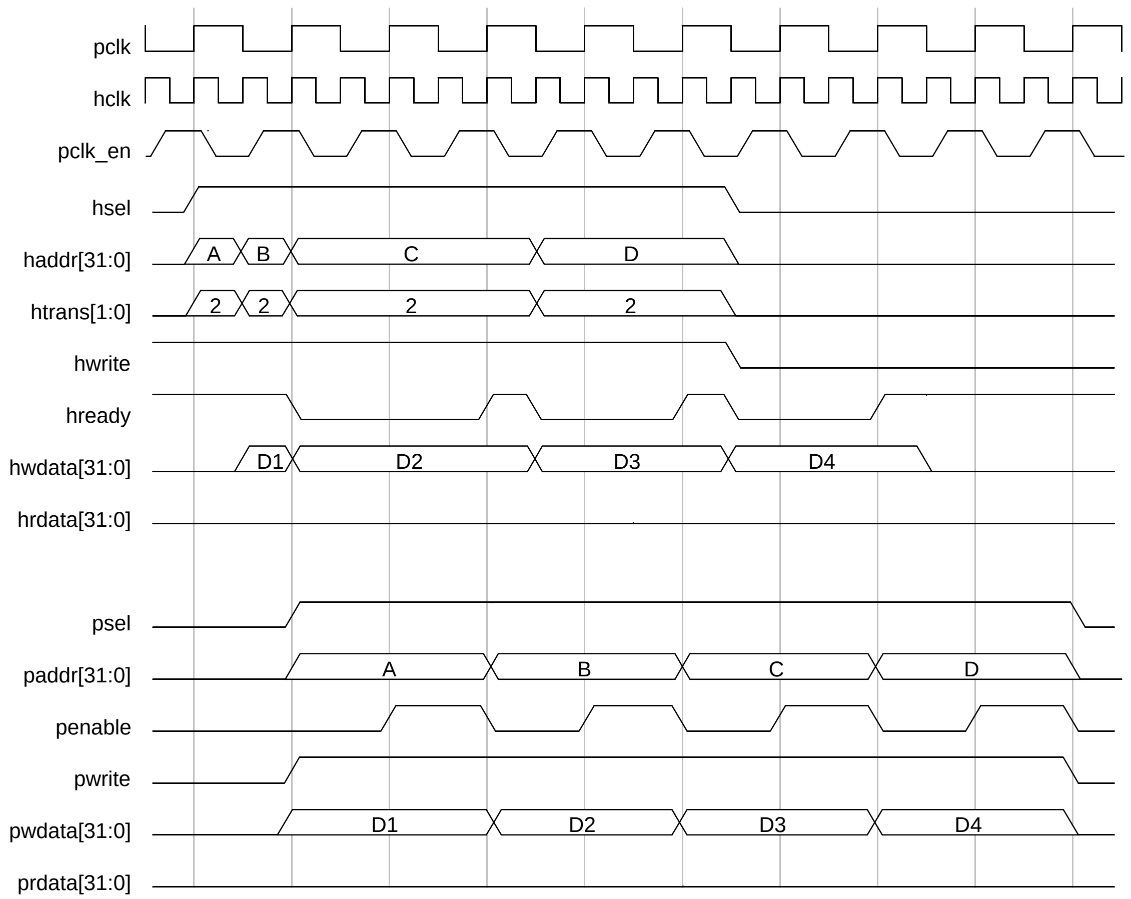


Figure 2-17 Back-to-Back Write Transfer (hclk != pclk)



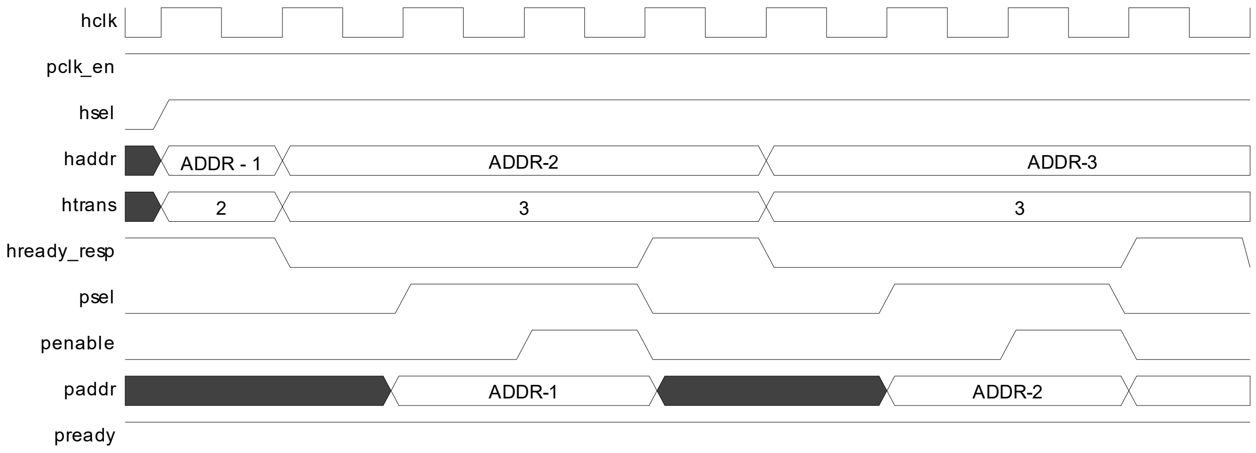
Note Figure 2-16 and Figure 2-17 show the AHB issuing consecutive write transfers on the DW_apb, which are targeting AMBA 2 APB slaves. If any transfer targets an AMBA 3 APB slave, the bus brings hready low and the systems stalls until each transfer completes on the APB bus.

2.10 Back-to-Back Transfer Support on an APB Interface

By default, DW_apb waits for an APB transfer to complete to provide the response to an AHB interface and thereby introduces an additional cycle (wait cycle) on the APB interface. The additional cycle reduces the performance on the APB interface. DW_apb supports the APB_ENH_THROUGHPUT parameter to enable back-to-back transfers such that the wait cycle is removed and a back-to-back transfer occurs on an APB interface.

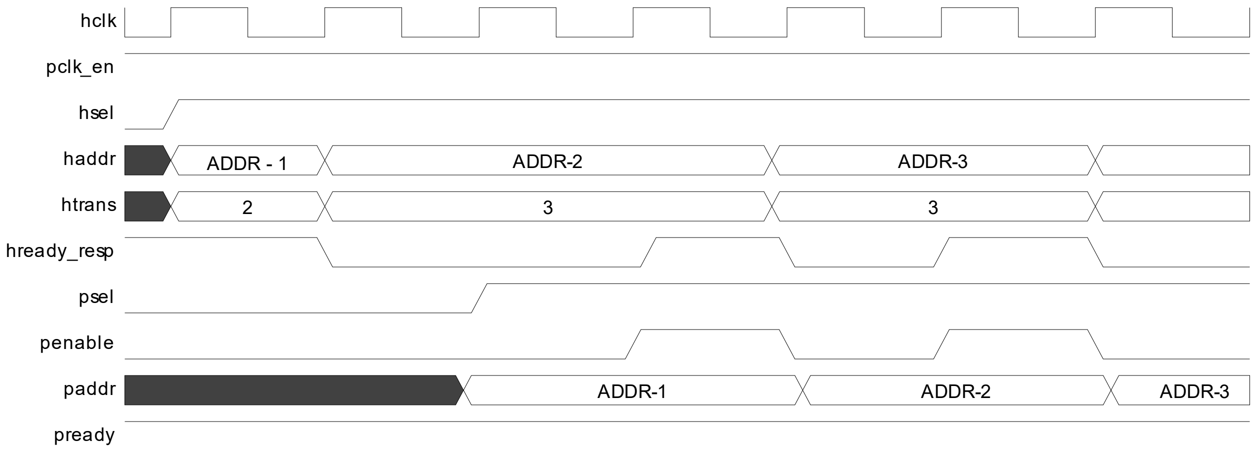
Figure 2-18 shows the three transfers on an APB when APB_ENH_THROUGHPUT is not set to 1 and clock ratio of pclk and hclk is 1. DW_apb issues next transfer only after the current transfer is over and there is always a 2-cycle delay between every APB transfer.

Figure 2-18 APB 3 Transfer When APB_ENH_THROUGHPUT_EN = 0



When APB_ENH_THROUGHPUT is set to 1, the 2-cycle delay between every APB transfer is removed and back-to-back transfer occurs on the APB interface that includes the overall bandwidth. Figure 2-19 shows the APB 3 transfer when APB_ENH_THROUGHPUT is set to 1 and clock ratio of pclk and hclk is 1.

Figure 2-19 APB 3 Transfer When APB_ENH_THROUGHPUT_EN = 1



Note APB transactions are not back-to-back when:

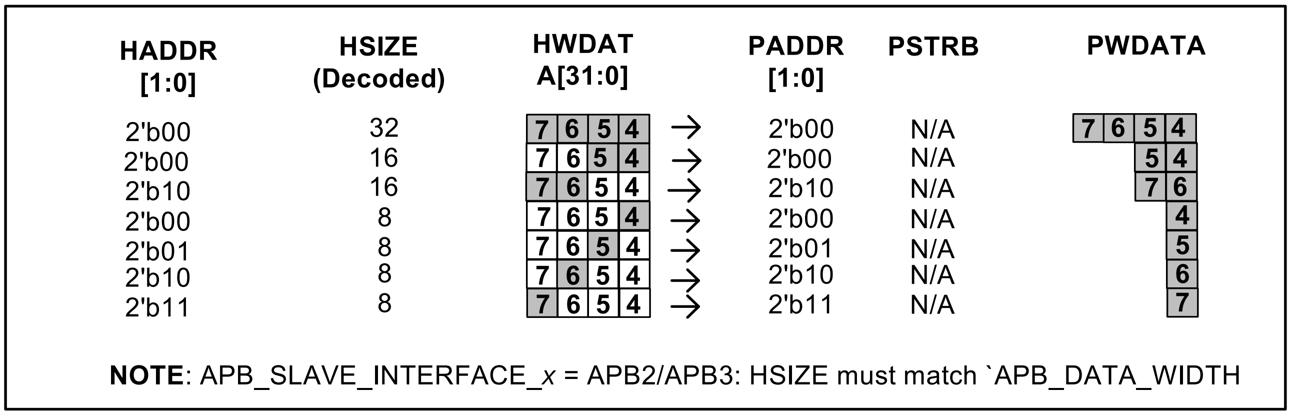
- APB transfers return with an ERROR response.
- AHB Master inserts the next transfer after an APB slave receives a READY response for the current transfer.

2.11 APB4 Protocol Feature

2.11.1 Write Strobing

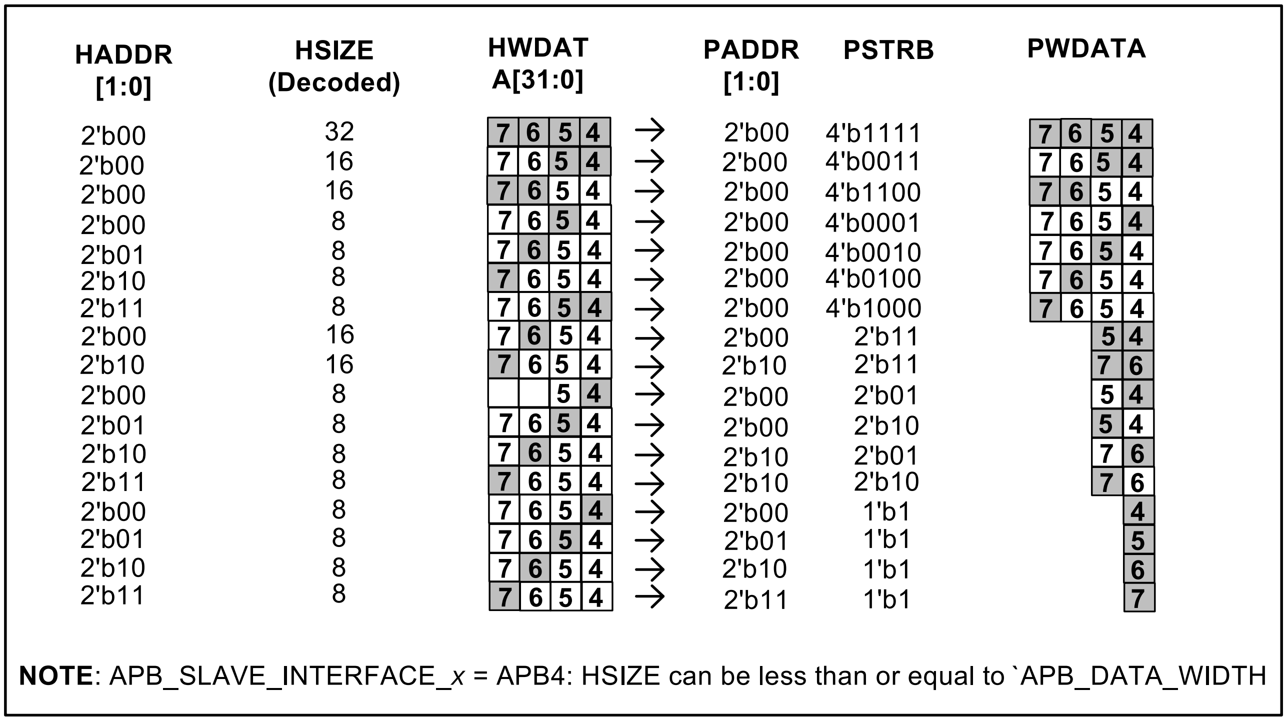
Write strobing allows AHB master to send data with width lesser than that defined by APB_DATA_WIDTH. The active data bytes are communicated to the APB peripheral by the pstrb signal. The mapping between hwdata and pwdata in case of varying haddr, hsize, and APB_DATA_WIDTH is shown independently in Figure 2-20, for APB2 / APB3 and APB4 modes.

Figure 2-20 Mapping Between hwdata and pwdata



HADDR [1:0]	HSIZE (Decoded)	HWDAT A[31:0]		PADDR [1:0]	PSTRB	PWDATA
2'b00	32	7 6 5 4	→	2'b00	N/A	7 6 5 4
2'b00	16	7 6 5 4	→	2'b00	N/A	5 4
2'b10	16	7 6 5 4	→	2'b10	N/A	7 6
2'b00	8	7 6 5 4	→	2'b00	N/A	4
2'b01	8	7 6 5 4	→	2'b01	N/A	5
2'b10	8	7 6 5 4	→	2'b10	N/A	6
2'b11	8	7 6 5 4	→	2'b11	N/A	7

NOTE: APB_SLAVE_INTERFACE_*x* = APB2/APB3: HSIZE must match `APB_DATA_WIDTH



HADDR [1:0]	HSIZE (Decoded)	HWDAT A[31:0]		PADDR [1:0]	PSTRB	PWDATA
2'b00	32	7 6 5 4	→	2'b00	4'b1111	7 6 5 4
2'b00	16	7 6 5 4	→	2'b00	4'b0011	7 6 5 4
2'b00	16	7 6 5 4	→	2'b00	4'b1100	7 6 5 4
2'b00	8	7 6 5 4	→	2'b00	4'b0001	7 6 5 4
2'b01	8	7 6 5 4	→	2'b00	4'b0010	7 6 5 4
2'b10	8	7 6 5 4	→	2'b00	4'b0100	7 6 5 4
2'b11	8	7 6 5 4	→	2'b00	4'b1000	7 6 5 4
2'b00	16	7 6 5 4	→	2'b00	2'b11	5 4
2'b10	16	7 6 5 4	→	2'b10	2'b11	7 6
2'b00	8	5 4	→	2'b00	2'b01	5 4
2'b01	8	7 6 5 4	→	2'b00	2'b10	5 4
2'b10	8	7 6 5 4	→	2'b10	2'b01	7 6
2'b11	8	7 6 5 4	→	2'b10	2'b10	7 6
2'b00	8	7 6 5 4	→	2'b00	1'b1	4
2'b01	8	7 6 5 4	→	2'b01	1'b1	5
2'b10	8	7 6 5 4	→	2'b10	1'b1	6
2'b11	8	7 6 5 4	→	2'b11	1'b1	7

NOTE: APB_SLAVE_INTERFACE_*x* = APB4: HSIZE can be less than or equal to `APB_DATA_WIDTH

2.11.2 Protection

Setting APB_INTERFACE_TYPE_SLAVE_x to APB4 adds a three-bit pprot signal to its interface. Values to the signal are mapped in the following manner:

- If EXT_PROT_EN parameter is enabled:

  Enabling the parameter includes the hsize signal to the AHB interface, which supplies values to pprot.

  pprot[0] <= hprot[1];

  pprot[1] <= 1'b0;

  pprot[2] <= hprot[0];

  In any case, hprot[3:2] (cacheable, bufferable bits) are unused in design.

- If EXT_PROT_EN parameter is disabled:

  All three bits in pprot signal are assigned a default value (1'b0)

3 Parameter Descriptions

This chapter details all the configuration parameters. **You can use the coreConsultant GUI configuration reports to determine the actual configured state of the controller.** Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

The parameter descriptions in this chapter include the **Enabled:** attribute which indicates the values required to be set on other parameters before you can change the value of this parameter.

These tables define all of the user configuration options for this component.

- Top Level Parameters on page 38
- Address Map on page 40

3.1 Top Level Parameters

Table 3-1 Top Level Parameters

Label	Description
	Top Level Parameters
AHB System Address Width	The address width of the AHB system. **Values:** ■ 32 (32) ■ 64 (64) **Default Value:** 32 **Enabled:** Always **Parameter Name:** HADDR_WIDTH
APB System Address Width	The address width of the APB system. **Values:** ■ 32 (32) ■ 64 (64) **Default Value:** 32 **Enabled:** Always **Parameter Name:** PADDR_WIDTH
AHB Data Bus Width	The data width of the AHB bus. **Values:** 32, 64, 128, 256 **Default Value:** 32 **Enabled:** Always **Parameter Name:** AHB_DATA_WIDTH
AHB Endianness	The endianness of the AHB system. The APB subsystem is always little-endian. **Values:** ■ Little-Endian (0) ■ Big-Endian (1) **Default Value:** Little-Endian **Enabled:** Always **Parameter Name:** BIG_ENDIAN
APB Data Bus Width	The data width of the APB bus. **Values:** 8, 16, 32 **Default Value:** 32 **Enabled:** Always **Parameter Name:** APB_DATA_WIDTH

Table 3-1 Top Level Parameters (Continued)

Label	Description
Number of APB Slave Ports	The number of APB slave ports. **Values:** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 **Default Value:** 4 **Enabled:** Always **Parameter Name:** NUM_APB_SLAVES
External Decoder?	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 during configuration. An external decoder allows users to connect to any decoder. **Values:** ■ false (0) ■ true (1) **Default Value:** false **Enabled:** Always **Parameter Name:** APB_HAS_XDCDR
Include HPROT signal to interface?	Enabling this parameter includes the HPROT signal to the AHB interface. **Values:** 0, 1 **Default Value:** 0 **Enabled:** APB_INTERFACE_TYPE_SLAVE(x) configured for APB4 on any Slave **Parameter Name:** EXT_PROT_EN
Enable enhanced throughput on APB bus?	If configured in this mode, DW_apb performs back-to-back transfers on the APB bus, if AHB master is providing back-to-back transfers. This increases the overall throughput. **Values:** ■ No (0) ■ Yes (1) **Default Value:** No **Enabled:** Always **Parameter Name:** APB_ENH_THROUGHPUT_EN

3.2 Address Map Parameters

Table 3-2 Address Map Parameters

Label	Description
	Slave x
Start Address of APB Slave x (for x = 0; x <= NUM_APB_SLAVES-1)	The Start Address for APB Slave x. This is an absolute address value. **Values:** 0x00000000, ..., 0xfffffc00 **Default Value:** For N=0: 0x00000400; for N=1: 0x00000800; for N=2: 0x00000c00; for N=3: 0x00001000; for N=4: 0x00001400; for N=5: 0x00001800; for N=6: 0x00001c0; for N=7: 0x00002000; for N=8: 0x00002400; for N=9: 0x00002800; for N=10: 0x00002c00; for N=11: 0x00003000; for N=12: 0x00003400; for N=13: 0x00003800; for N=14: 0x00003c00; for N=15: 0x000040000 **Enabled:** NUM_APB_SLAVES > x && !APB_HAS_XDCDR **Parameter Name:** START_PADDR_(x)
End Address of APB Slave x (for x = 0; x <= NUM_APB_SLAVES-1)	The End Address for APB Slave x. This is an absolute address value. **Values:** 0x000003ff, ..., 0xffffffff **Default Value:** For N=0: 0x000007ff; for N=1: 0x00000bff; for N=2: 0x00000fff; for N=3: 0x000013ff; for N=4: 0x000017ff; for N=5: 0x00001bff; for N=6: 0x00001fff; for N=7: 0x000023ff; for N=8: 0x000027ff; for N=9: 0x00002bff; for N=10: 0x00002fff; for N=11: 0x000033ff; for N=12: 0x000037ff; for N=13: 0x00003bff; for N=14: 0x00003fff; for N=15: 0x000043ff **Enabled:** NUM_APB_SLAVES > x && !APB_HAS_XDCDR **Parameter Name:** END_PADDR_(x)
APB Slave Interface Type (for x = 0; x <= NUM_APB_SLAVES-1)	Select between AMBA 4 APB slave (APB4), AMBA 3 APB slave (APB3), and AMBA 2 APB slave (APB2) for Slave x. If AMBA 3 APB Slave is selected, the additional ports PREADY and PSLVERR are included. If AMBA 4 APB Slave is selected, the additional ports PSTRB and PPROT are included. **Values:** ■ APB2 (0) ■ APB3 (1) ■ APB4 (2) **Default Value:** APB2 **Enabled:** NUM_APB_SLAVES > x && !APB_HAS_XDCDR **Parameter Name:** APB_INTERFACE_TYPE_SLAVE_(x)

4 Signal Descriptions

This chapter details all possible I/O signals in the controller. For configurable IP titles, your actual configuration might not contain all of these signals.

Inputs are on the left of the signal diagrams; outputs are on the right.

Attention: For configurable IP titles, do not use this document to determine the exact I/O footprint of the controller. It is for reference purposes only.

When you configure the controller in coreConsultant, you must access the I/O signals for your actual configuration at workspace/report/IO.html or workspace/report/IO.xml after you have completed the report creation activity. That report comes from the exact same source as this chapter but removes all the I/O signals that are not in your actual configuration. This does not apply to non-configurable IP titles. In addition, all parameter expressions are evaluated to actual values. Therefore, the widths might change depending on your actual configuration.

Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>)** that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

In addition to describing the function of each signal, the signal descriptions in this chapter include the following information:

Active State: Indicates whether the signal is active high or active low. When a signal is not intended to be used in a particular application, then this signal needs to be tied or driven to the inactive state (opposite of the active state).

Registered: Indicates whether or not the signal is registered directly inside the IP boundary without intervening logic (excluding simple buffers). A value of No does not imply that the signal is not synchronous, only that there is some combinatorial logic between the signal's origin or destination register and the boundary of the controller. A value of N/A indicates that this information is not provided for this IP title.

Synchronous to: Indicates which clock(s) in the IP sample this input (drive for an output) when considering all possible configurations. A particular configuration might not have all of the clocks listed. This clock might not be the same as the clock that your application logic should use to clock (sample/drive) this pin. For more details, consult the clock section in the databook.

Exists: Name of configuration parameter(s) that populates this signal in your configuration.

Validated by: Assertion or de-assertion of signal(s) that validates the signal being described.

Attributes used with Synchronous To

- Clock name - The name of the clock that samples an input or drive and output.
- None - This attribute may be used for clock inputs, hard-coded outputs, feed-through (direct or combinatorial), dangling inputs, unused inputs and asynchronous outputs.
- Asynchronous - This attribute is used for asynchronous inputs and asynchronous resets.

The I/O signals are grouped as follows:

- Clocks and Resets on page 43
- AHB Slave Interface Signals on page 44
- APB Interface Signals on page 47

4.1 Clocks and Resets Signals

hclk
hresetn
pclk_en

Table 4-1 Clocks and Resets Signals

Port Name	I/O	Description
hclk	I	AHB Clock Signal. This clock times all bus transfers. All signal timings are related to the rising edge of hclk. **Exists:** Always **Synchronous To:** None **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hresetn	I	AHB Reset Signal. This signal is used to reset the system and the bus on the DesignWare interface. 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. **Exists:** Always **Synchronous To:** Asynchronous **Registered:** N/A **Power Domain:** SINGLE_DOMAIN **Active State:** Low
pclk_en	I	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. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High

4.2 AHB Slave Interface Signals

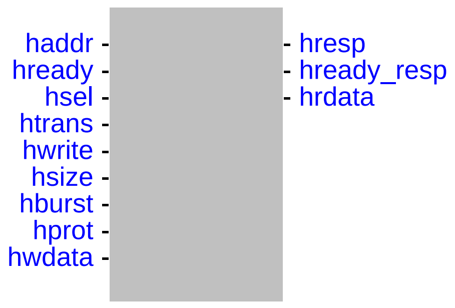


Table 4-2 AHB Slave Interface Signals

Port Name	I/O	Description
haddr[(HADDR_WIDTH-1):0]	I	Address bus from AHB master. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hready	I	Ready response for DW_apb. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
hsel	I	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. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
htrans[1:0]	I	Transfer type from selected master. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-2 AHB Slave Interface Signals (Continued)

Port Name	I/O	Description
hwrite	I	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). **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
hsize[2:0]	I	Transfer size. Indicates the size of the transfer. This signal is used in the component only when the component is configured to have an APB4 enabled interface. In this case, hsize can supply values which decode to a width less than or equal to APB_DATA_WIDTH. Otherwise, each transfer requires a hsize of APB_DATA_WIDTH, and the input value is assumed to decode to the same value. The signal is left unconnected in this case. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hburst[(HBURST_WIDTH-1):0]	I	Burst type indication from selected AHB master. This signal is left unconnected on the interface because it is not required. **Exists:** Always **Synchronous To:** None **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hprot[3:0]	I	AHB Protection type signal. HPROT values are mapped to relevant pprot signal if the input signal is included. Otherwise, default values (3'b000) are copied to the pprot signals upon access. **Exists:** (EXT_PROT_EN==1) **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-2 AHB Slave Interface Signals (Continued)

Port Name	I/O	Description
hresp[1:0]	O	Transfer Response. This signal is always an OKAY response. When hready_resp is High, this shows the transfer has completed successfully(Connected directly to 1 or 0). **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hready_resp	O	Response from DW_apb. Asserted when current transfer has completed. **Exists:** Always **Synchronous To:** hclk **Registered:** APB_ENH_THROUGHPUT_EN==0 ? Yes : (APB_HAS_APB3==0 ? Yes : No) **Power Domain:** SINGLE_DOMAIN **Active State:** High
hwdata[(AHB_DATA_WIDTH-1):0]	I	Write data bus from selected AHB master. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
hrdata[(AHB_DATA_WIDTH-1):0]	O	Transfer read data. The read data bus is used to transfer data from DW_apb to the bus master during read operations. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

4.3 APB Interface Signals

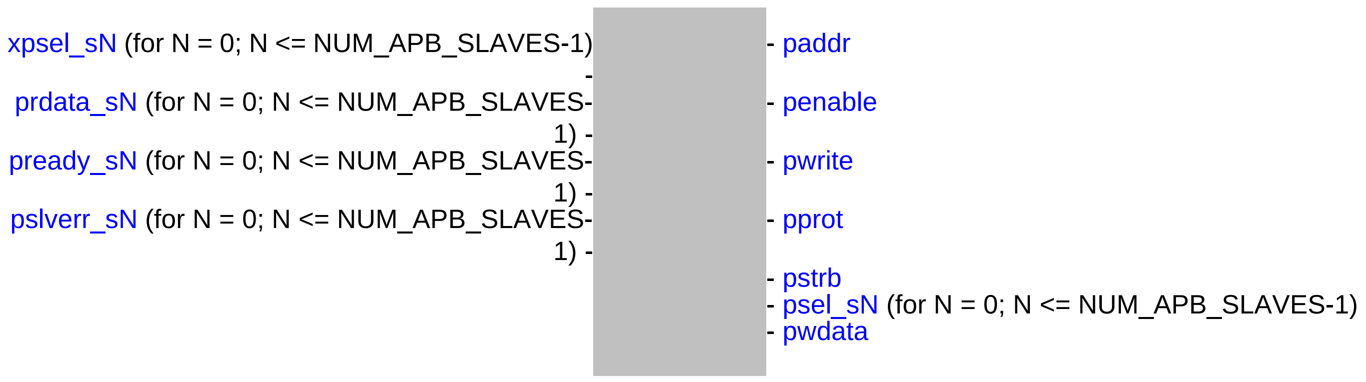

xpsel_sN (for N = 0; N <= NUM_APB_SLAVES-1) -
prdata_sN (for N = 0; N <= NUM_APB_SLAVES-1) -
pready_sN (for N = 0; N <= NUM_APB_SLAVES-1) -
pslverr_sN (for N = 0; N <= NUM_APB_SLAVES-1) -
- paddr
- penable
- pwrite
- pprot
- pstrb
- psel_sN (for N = 0; N <= NUM_APB_SLAVES-1)
- pwdata

Table 4-3 APB Interface Signals

Port Name	I/O	Description
paddr[(PADDR_WIDTH-1):0]	O	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. **Exists:** Always **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
penable	O	Enable Strobe. Asserted to validate APB transfer. It is always driven low at the end of each APB access. **Exists:** Always **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
pwrite	O	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). **Exists:** Always **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High

Table 4-3 APB Interface Signals (Continued)

Port Name	I/O	Description
pprot[2:0]	O	APB4 Protection type signal. HPROT values are mapped to relevant pprot signal if the input signal is included. Else, default values are copied to the pprot signals upon access. **Exists:** APB_HAS_APB4==1 **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
pstrb[((APB_DATA_WIDTH/8)-1):0]	O	APB4 Write strobe bus. HSIZE of an incoming transaction may now be lower than APB_DATA_WIDTH. This value along with address offset is used to generate strobe signals on APB side to selectively write to certain bytes in the apb data bus. **Exists:** APB_HAS_APB4==1 **Synchronous To:** hclk **Registered:** Yes **Power Domain:** SINGLE_DOMAIN **Active State:** High
psel_sN (for N = 0; N <= NUM_APB_SLAVES-1)	O	Select lines for APB slaves (one per slave) **Exists:** N < NUM_APB_SLAVES **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
xpsel_sN (for N = 0; N <= NUM_APB_SLAVES-1)	I	Optional. Slave select line for APB. **Exists:** N < NUM_APB_SLAVES **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A
prdata_sN[(APB_DATA_WIDTH-1):0] (for N = 0; N <= NUM_APB_SLAVES-1)	I	Read Data from APB slaves. The width of each bus is APB_DATA_WIDTH. **Exists:** N < NUM_APB_SLAVES **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

Table 4-3 APB Interface Signals (Continued)

Port Name	I/O	Description
pready_sN (for N = 0; N <= NUM_APB_SLAVES-1)	I	Indicates whether a request cycle was accepted. Exists only on slave interfaces configured as APB3 or APB4. **Exists:** N < NUM_APB_SLAVES **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
pslverr_sN (for N = 0; N <= NUM_APB_SLAVES-1)	I	Flag for the slave error response from APB. Exists only on slave interfaces configured as APB3 or APB4. **Exists:** N < NUM_APB_SLAVES **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** High
pwdata[(APB_DATA_WIDTH-1):0]	O	APB transfer write data bus shared by all slaves. **Exists:** Always **Synchronous To:** hclk **Registered:** No **Power Domain:** SINGLE_DOMAIN **Active State:** N/A

5 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). 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.*

5.1 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.

5.1.1 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

5.1.2 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 occurs 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

5.1.3 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

5.2 Overview of DW_apb Testbench

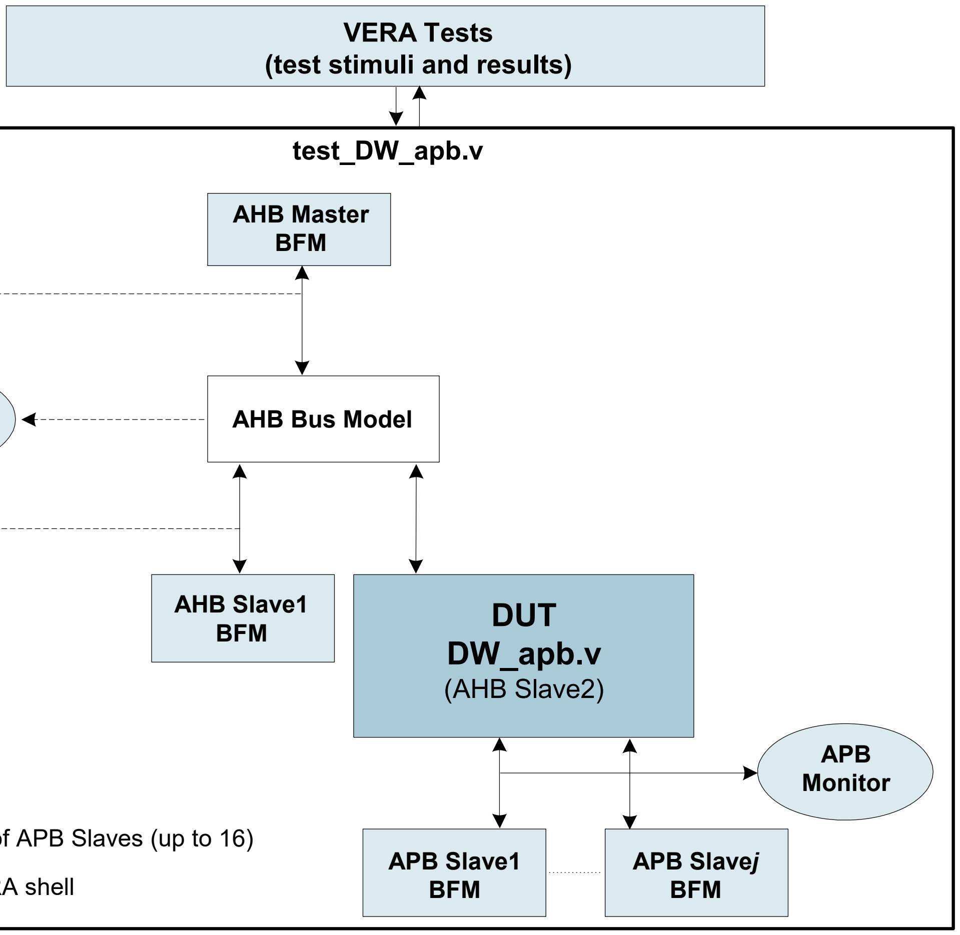
As illustrated in Figure 5-1, the DW_apb testbench is a Verilog testbench that includes an instantiation of the design under test (DUT) and a Vera shell, which consists of the following components:

- 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 5-1 DW_apb Testbench



5.3 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.

5.3.1 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.

6 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.

6.1 Performance

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

6.1.1 Power Consumption, Frequency, and Area Results

Table 6-1 provides information about the synthesis results (power consumption, frequency, and area) of the DW_apb using the industry standard 28nm technology library and how it affects performance.

Table 6-1 Power Consumption, Frequency, and Area Results for DW_apb Using 28nm Technology Library

Configuration	Operating Frequency	Gate Count	Static Power Consumption	Dynamic Power Consumption
Default Configuration	hclk: 300 MHz	1832 gates	32 nW	6.008 uW
Minimum Configuration: NUM_APB_SLAVES=1	hclk:300 MHz	1683 gates	29.4 nW	5.794 uW

Maximum Configuration:
NUM_APB_SLAVES=16
BIG_ENDIAN=1
	hclk:300 MHz	2376 gates	38.9 nW	6.239 uW

Maximum Configuration with APB4 Enabled:
NUM_APB_SLAVES=16
BIG_ENDIAN=1
All Slaves support APB4
	hclk:300 MHz	2818 gates	46.2 nW	7.868 uW

6.2 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>
```

6.3 Reading and Writing from an APB Slave

When writing to and reading from DesignWare APB slaves, you should consider the following:

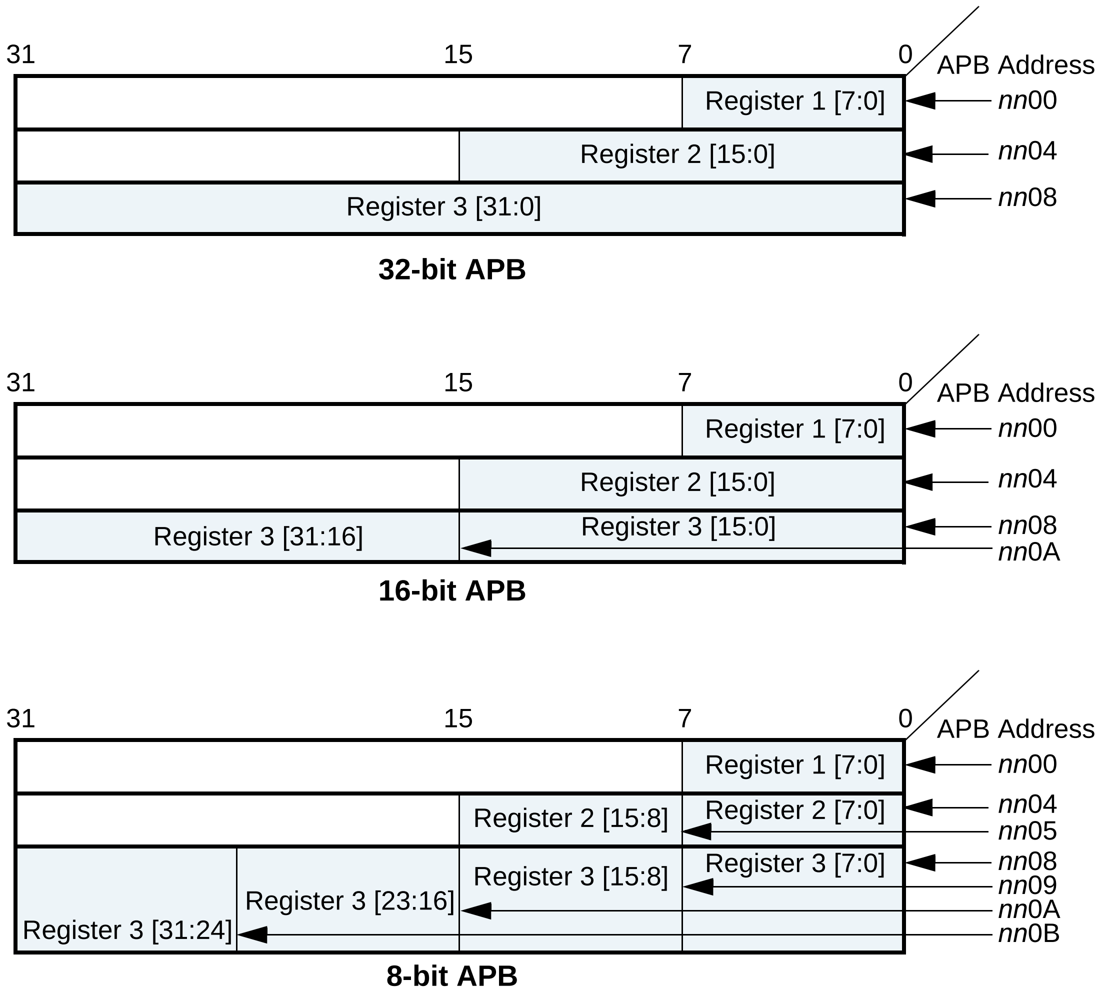
- The size of the APB peripheral should always be set equal to the size of the APB data bus, if possible.
- The APB bus has no concept of a transfer size or a byte lane, unlike the DW_ahb.

- The APB slave subsystem is little endian; the DW_apb performs the conversion from a big-endian AHB to the little-endian APB.
- All APB slave programming registers are aligned on 32-bit boundaries, irrespective of the APB bus size.
- The maximum APB_DATA_WIDTH is 32 bits. Registers larger than this occupies more than one location in the memory map.
- The DW_apb does not return any SPLIT or RETRY response; it always returns an OKAY response to the AHB.
- For all bus widths:
    - In the case of a read transaction, registers less than the full bus width returns zeros in the unused upper bits.
    - Writing to bit locations larger than the register width does not have any effect. Only the pertinent bits are written to the register.
- The APB slaves do not need the full 32-bit address bus, paddr. The slaves include the lower bits even though they are not actually used in a 32- or 16-bit system.

6.3.1 Reading From Unused Locations

Reading from an unused location or unused bits in a particular register always returns zeros. The following sections show the relationship between the register map and the read / write operations for the three possible APB_DATA_WIDTH values: 8-, 16-, and 32-bit APB buses.

Figure 6-1 Read/Write Locations for Different APB Bus Data Widths



6.3.2 32-bit Bus System

For 32-bit bus systems, all programming registers can be read or written with one operation, as illustrated in the previous figure.

Because all registers are on 32-bit boundaries, paddr[1:0] is not actually needed in the 32-bit bus case. But these bits still exist in the configured code for usability purposes.

Note If you write to an address location not on a 32-bit boundary, the bottom bits are ignored/not used.

6.3.3 16-bit Bus System

For 16-bit bus systems, two scenarios exist, as illustrated in the previous picture:

1. The register to be written to or read from is less than or equal to 16 bits

   In this case, the register can be read or written with one transaction. In the case of a read transaction, registers less than 16 bits wide returns zeros in the un-used bits. Writing to bit locations larger than the register width causes nothing to happen, i.e. only the pertinent bits are written to the register.

2. The register to be written to or read from is >16 and <= 32 bits

   In this case, two AHB transactions are required, which in turn creates two APB transactions, to read or write the register. The first transaction should read / write the lower two bytes (half-word) and the second transaction the upper half-word.

Because the bus is reading a half-word at a time, paddr[0] is not actually needed in the 16-bit bus case. But these bits still exist in the configured code for connectivity purposes.

Note If you write to an address location not on a 16-bit boundary, the bottom bits are ignored/not used.

6.3.4 8-bit Bus System

For 8-bit bus systems, three scenarios exist, as illustrated in the previous picture:

1. The register to be written to or read from is less than or equal to 8 bits

   In this case, the register can be read or written with one transaction. In the case of a read transaction, registers less than 8 bits wide returns zeros in the unused bits. Writing to bit locations larger than the register width causes nothing to happen, that is, only the pertinent bits are written to the register.

2. The register to be written to or read from is >8 and <=16 bits

   In this case, two AHB transactions are required, which in turn creates two APB transactions, to read or write the register. The first transaction should read / write the lower byte and the second transaction the upper byte.

3. The register to be written to or read from is >16 and <=32 bits

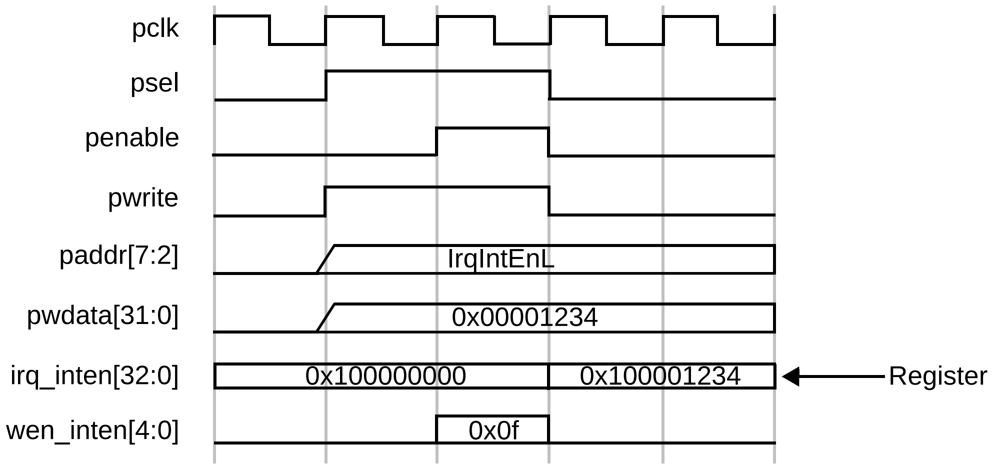
   In this case, four AHB transactions are required, which in turn creates four APB transactions, to read or write the register. The first transaction should read / write the lower byte and the second transaction the second byte, and so on.

Because the bus is reading a byte at a time, all lower bits of paddr are decoded in the 8-bit bus case.

6.4 Write Timing Operation

A timing diagram of an APB write transaction for an APB peripheral register (an earlier version of the DW_apb_ictl) is shown in the following figure. Data, address, and control signals are aligned. The APB frame lasts for two cycles when psel is high, unless an APB3 enabled slave delays the transfer completion by pulling pready low.

Figure 6-2 APB Write Transaction



A write can occur after the first phase with penable low, or after the second phase when penable is high. The second phase is preferred and is used in all APB slave components. The timing diagram is shown with the write occurring after the second phase. Whenever the address on paddr matches a corresponding address from the memory map and provided psel, pwrite, and penable are high, then the corresponding register write enable is generated.

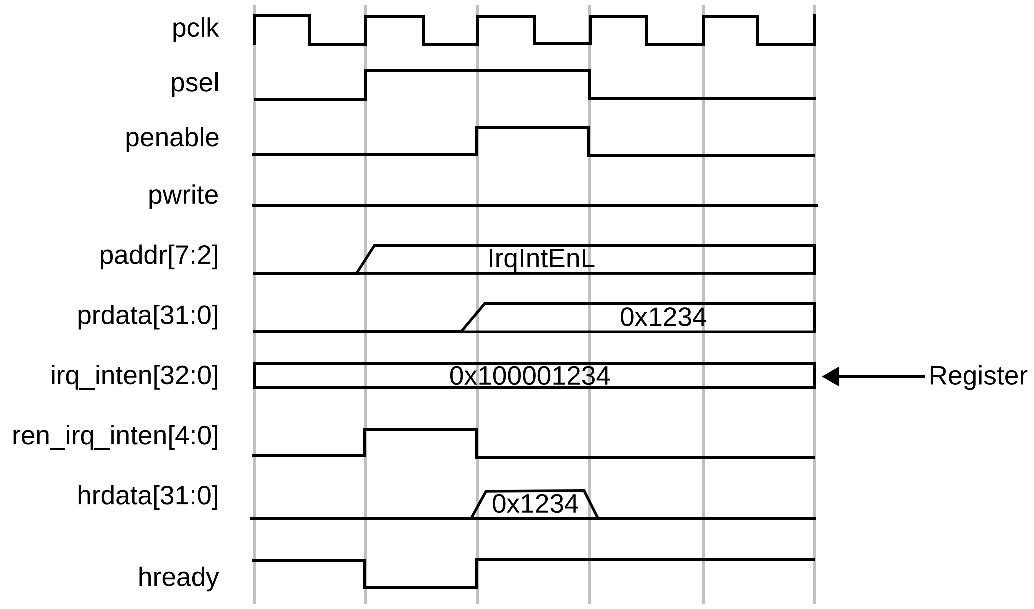
A write from the AHB to the APB does not require the AHB system bus to stall until the transfer on the APB has completed. A write to the APB can be followed by a read transaction from another AHB peripheral (not the DW_apb).

The timing example is a 33-bit register and a 32-bit APB data bus. To write this, 5 byte enables would be generated internally. The example shows writing to the first 32 bits with one write transaction.

6.5 Read Timing Operation

A timing diagram of an APB read transaction for an APB peripheral (an earlier version of the DW_apb_ictl) is shown in the following figure. The APB frame lasts for two cycles, when psel is high, unless an APB3 enabled slave delays the transfer completion by pulling pready low.

Figure 6-3 APB Read Transaction



Whenever the address on paddr matches the corresponding address from the memory map—psel is high, pwrite and penable are low—then the corresponding read enable is generated. The read data is registered within the peripheral before passing back to the master through the DW_apb and DW_ahb.

The qualification of the read-back data with hready from the bridge is shown in the timing diagram, but this does not form part of the APB interface. The read happens in the first APB cycle and is passed straight back to the AHB master in the same cycles as it passes through the bridge. By returning the data immediately to the AHB bus, the bridge can release control of the AHB data bus faster. This is important for systems where the APB clock is slower than the AHB clock.

Once a read transaction has started, it is completed and the AHB bus is held either until the data is returned from the slave, or until it responds with an ERROR message.

Note If a read enable is not active, then the previously read data is maintained on the read-back data bus.

6.6 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.

6.6.1 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 6-2 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

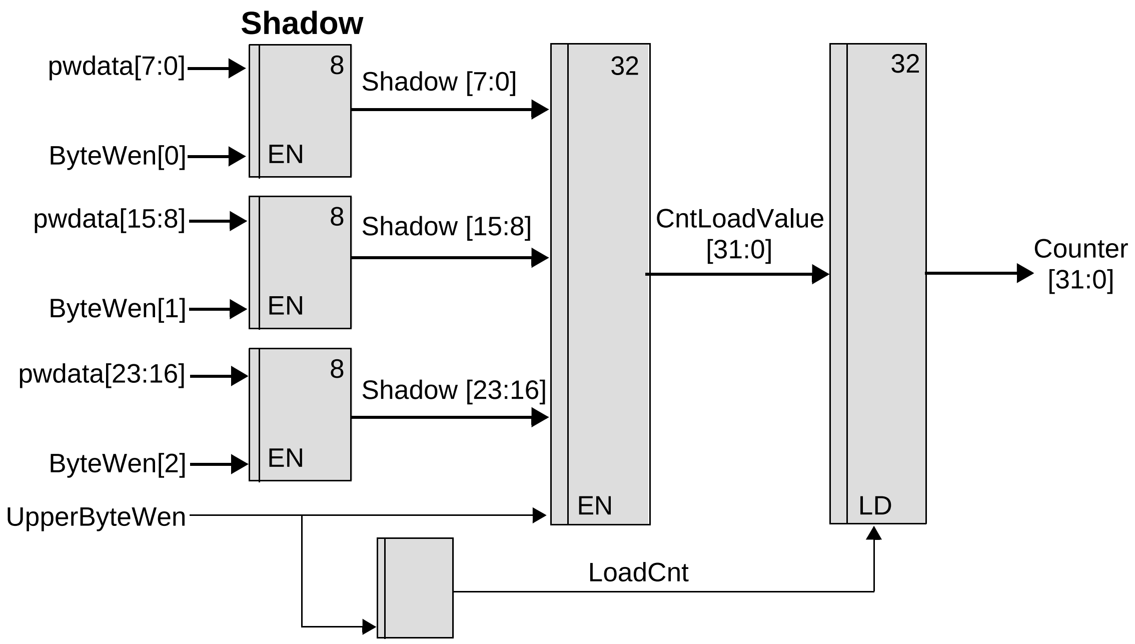
There are three relationship cases to be considered for the processor and peripheral clocks:

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

6.6.1.1 Identical Clocks

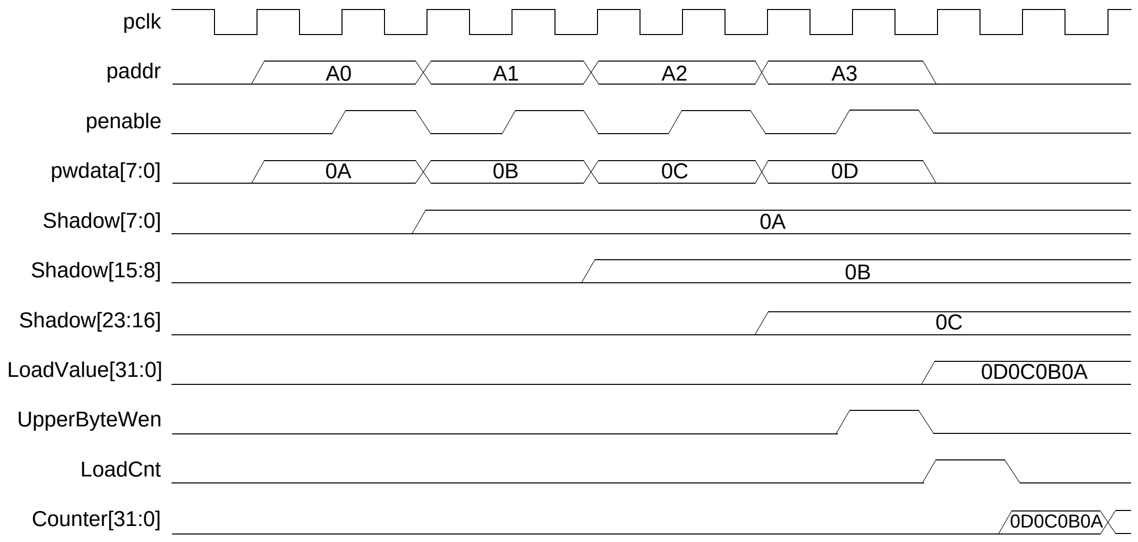
The following figure illustrates an RTL diagram for the circuitry required to implement the coherent write transaction when the APB bus clock and peripheral clocks are identical.

Figure 6-4 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 6-5 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.

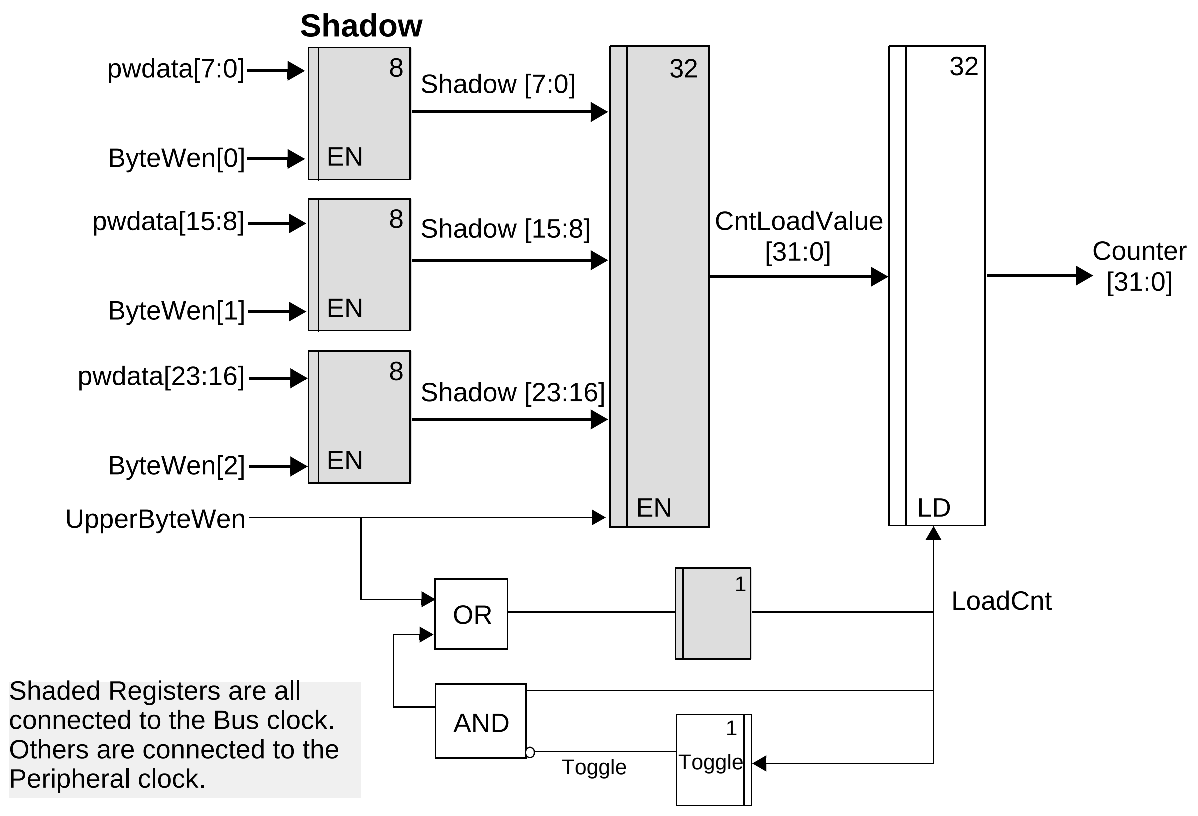
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this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

6.6.1.2 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.

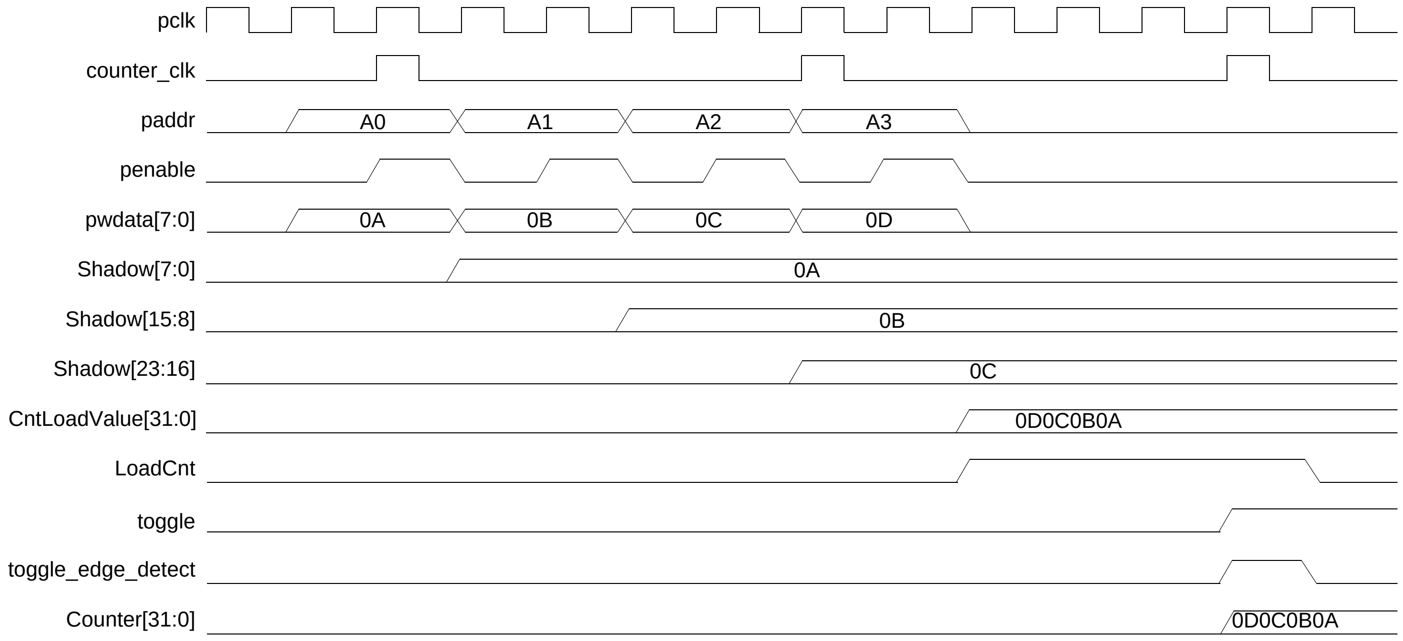
The following figure shows an RTL diagram for the circuitry required to implement the coherent write when the bus and peripheral clocks are synchronous.

Figure 6-6 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.

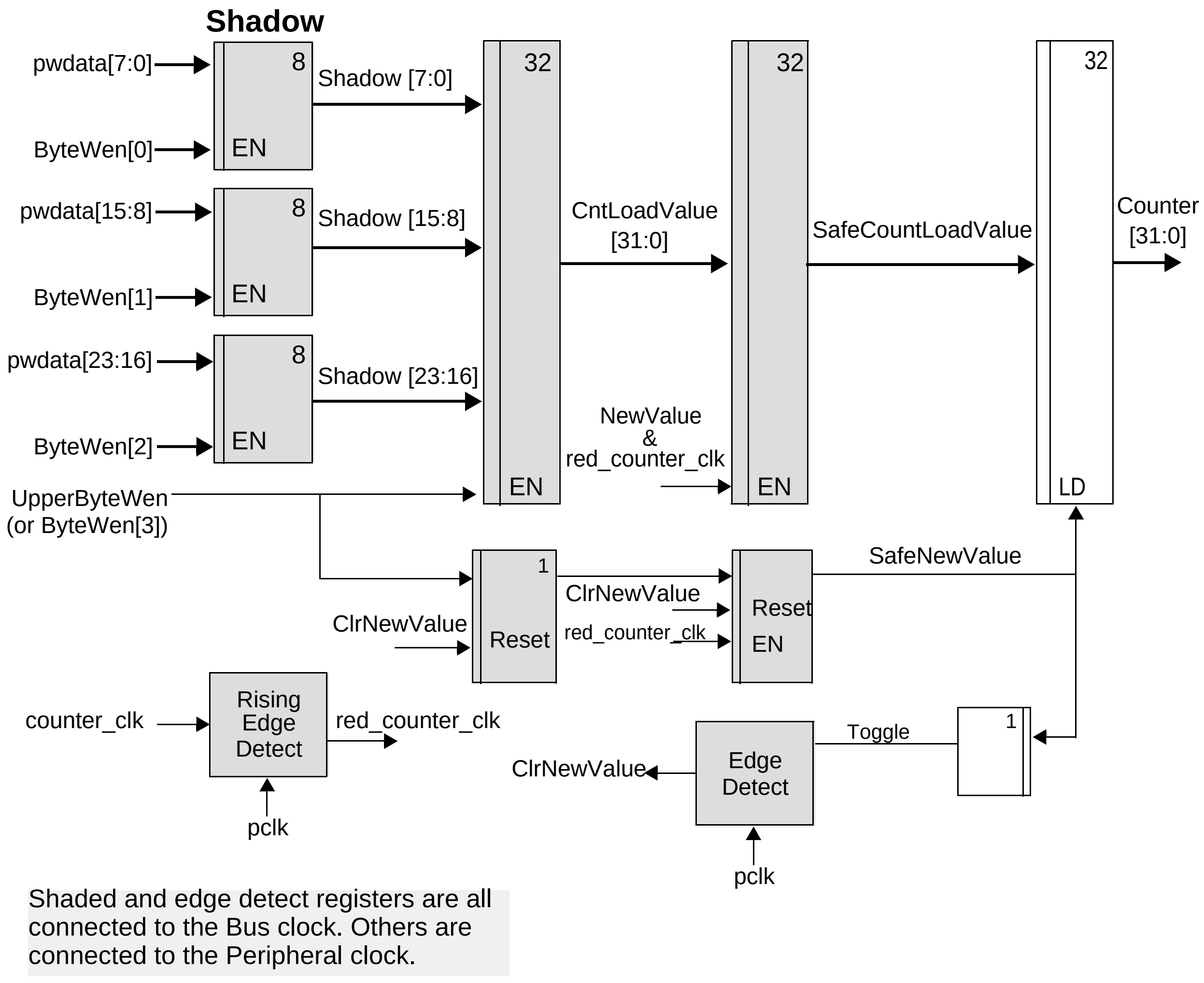
Figure 6-7 Coherent Loading – Synchronous Clocks



6.6.1.3 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 6-8 Coherent Loading – Asynchronous Clocks

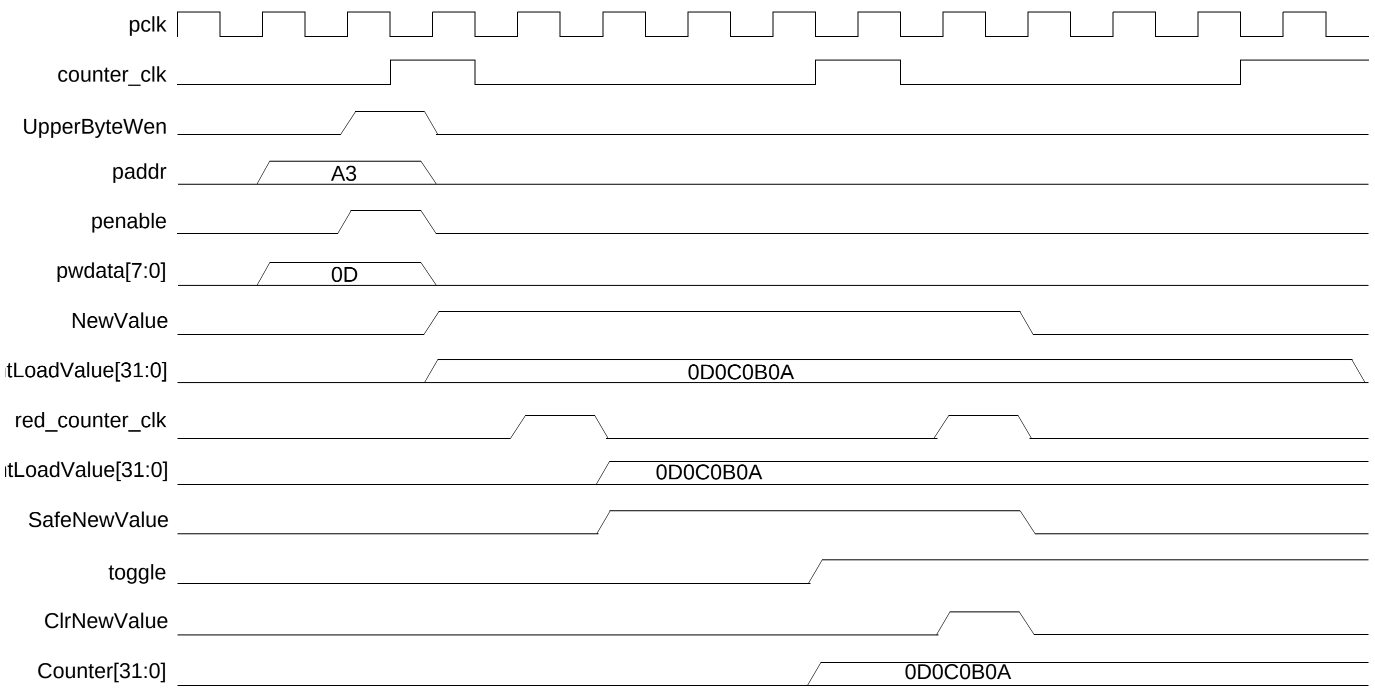


When the clocks are asynchronous, you need to transfer the contents of the register from one clock domain to another. It is not desirable to transfer the entire register through meta-stability registers, as coherency is not guaranteed with this method. The circuitry needed requires the processor clock 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 6-9 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.

6.6.2 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 6-3 Lower Byte Generation

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

Table 6-3 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

6.6.2.1 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 6-10 Coherent Registering – Synchronous Clocks

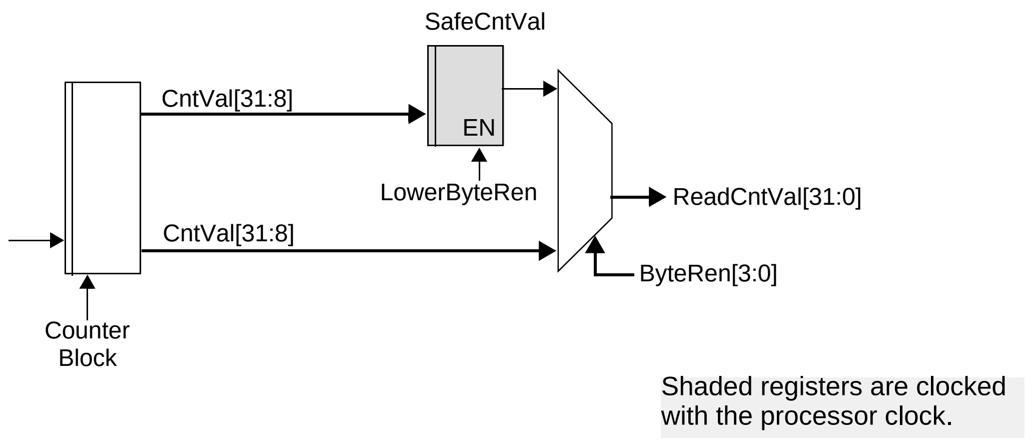
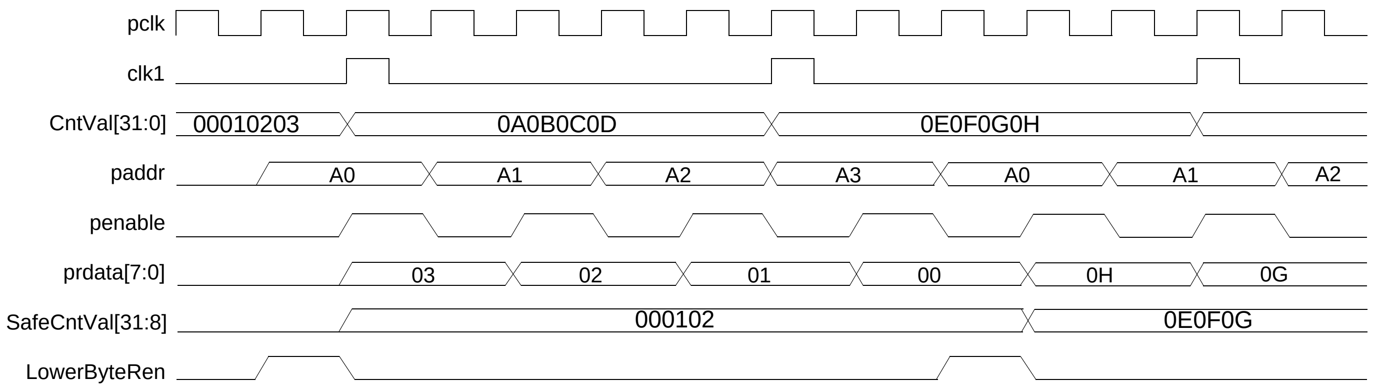


Figure 6-11 Coherent Registering – Synchronous Clocks



6.6.2.2 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.

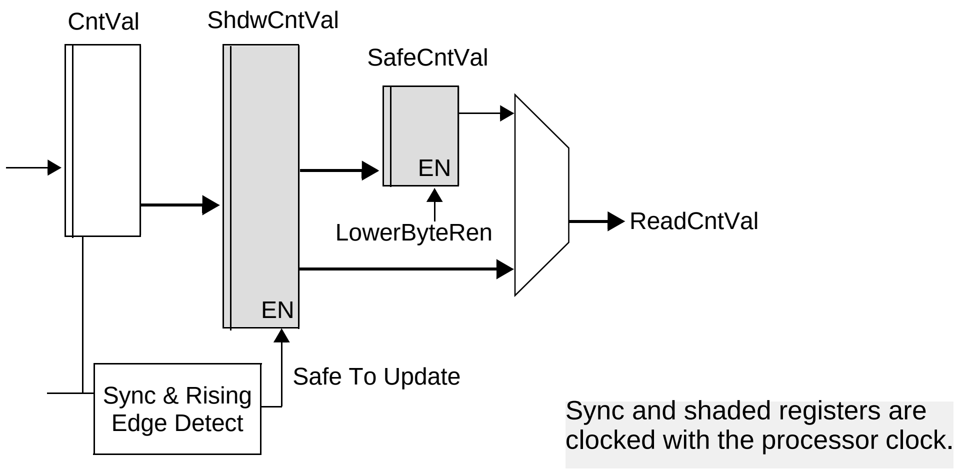
Note You must read LSB to MSB when the bus width is narrower than the counter width.

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 6-12 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
Internal Parameter Descriptions

Provides a description of the internal parameters that might be indirectly referenced in expressions in the Signals, Parameters, or Registers chapters. These parameters are not visible in the coreConsultant GUI and most of them are derived automatically from visible parameters. **You must not set any of these parameters directly.**

Some expressions might refer to TCL functions or procedures (sometimes identified as **function_of**) that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the core in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

Table B-1 Internal Parameters

Parameter Name	Equals To
APB_HAS_APB3	=(APB_INTERFACE_TYPE_SLAVE_0 ! =0 || APB_INTERFACE_TYPE_SLAVE_1 ! =0 || APB_INTERFACE_TYPE_SLAVE_2 ! =0 || APB_INTERFACE_TYPE_SLAVE_3 ! =0 || APB_INTERFACE_TYPE_SLAVE_4 ! =0 || APB_INTERFACE_TYPE_SLAVE_5 ! =0 || APB_INTERFACE_TYPE_SLAVE_6 ! =0 || APB_INTERFACE_TYPE_SLAVE_7 ! =0 || APB_INTERFACE_TYPE_SLAVE_8 ! =0 || APB_INTERFACE_TYPE_SLAVE_9 ! =0 || APB_INTERFACE_TYPE_SLAVE_10 ! =0 || APB_INTERFACE_TYPE_SLAVE_11 ! =0 || APB_INTERFACE_TYPE_SLAVE_12 ! =0 || APB_INTERFACE_TYPE_SLAVE_13 ! =0 || APB_INTERFACE_TYPE_SLAVE_14 ! =0 || APB_INTERFACE_TYPE_SLAVE_15 ! =0)

Table B-1 Internal Parameters (Continued)

Parameter Name	Equals To
APB_HAS_APB4	=(APB_INTERFACE_TYPE_SLAVE_0 ==2 || APB_INTERFACE_TYPE_SLAVE_1 ==2 || APB_INTERFACE_TYPE_SLAVE_2 ==2 || APB_INTERFACE_TYPE_SLAVE_3 ==2 || APB_INTERFACE_TYPE_SLAVE_4 ==2 || APB_INTERFACE_TYPE_SLAVE_5 ==2 || APB_INTERFACE_TYPE_SLAVE_6 ==2 || APB_INTERFACE_TYPE_SLAVE_7 ==2 || APB_INTERFACE_TYPE_SLAVE_8 ==2 || APB_INTERFACE_TYPE_SLAVE_9 ==2 || APB_INTERFACE_TYPE_SLAVE_10 ==2 || APB_INTERFACE_TYPE_SLAVE_11 ==2 || APB_INTERFACE_TYPE_SLAVE_12 ==2 || APB_INTERFACE_TYPE_SLAVE_13 ==2 || APB_INTERFACE_TYPE_SLAVE_14 ==2 || APB_INTERFACE_TYPE_SLAVE_15 ==2)
IDLE	2'b00
OKAY	2'b00

C 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