



DW_apb_timers Databook

DW_apb_timers

Copyright Notice and Proprietary Information

Copyright © 2010 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, CRITIC, Certify, CHIPit, Design Compiler, DesignWare, Formality, HDL Analyst, HSIM, HSPICE, Identify, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Syndicated, Synplicity, the Synplicity Logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Confirma, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclipse, Encore, EPIC, Galaxy, Galaxy Custom Designer, HANEX, HAPS, HapsTrak, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping Ssystem, HSIMplus, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, StarRC, Taurus, TotalRecall, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

PCI Express is a trademark of PCI-SIG.

All other product or company names may be trademarks of their respective owners.

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

1

Product Overview

The DW_apb_timers is a programmable timers peripheral. This component is an AMBA 2.0-compliant Advanced Peripheral Bus (APB) slave device and is part of the family of DesignWare Synthesizable Components.

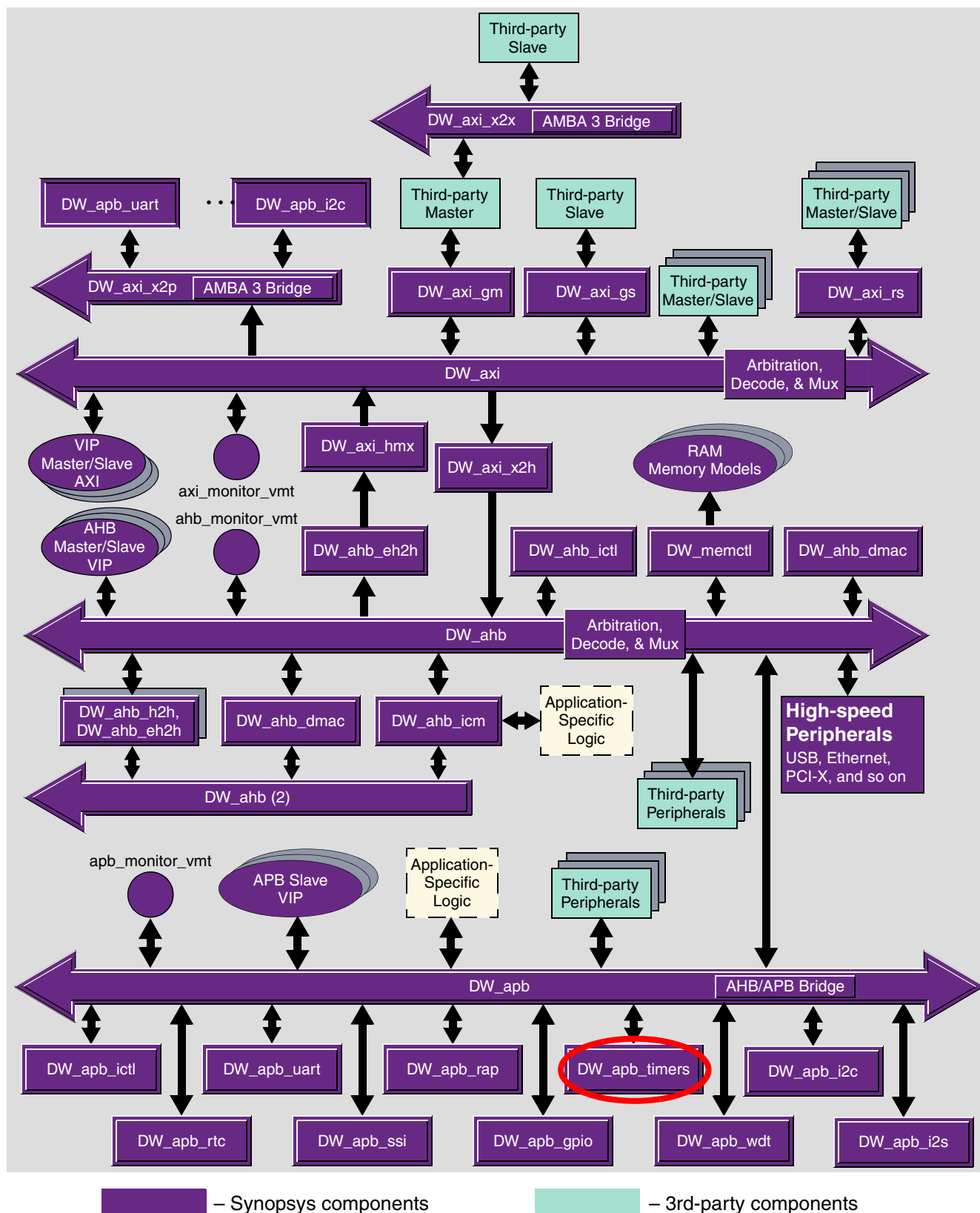
1.1 DesignWare System Overview

The Synopsys DesignWare Synthesizable Components environment is a parameterizable bus system containing AMBA version 2.0-compliant AHB (Advanced High-performance Bus) and APB (Advanced Peripheral Bus) components, and AMBA version 3.0-compliant AXI (Advanced eXtensible Interface) components.

[Figure 1-1](#) illustrates one example of this environment, including the AXI bus, the AHB bus, and the 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.



Figure 1-1 Example of DW_apb_timers in a Complete System



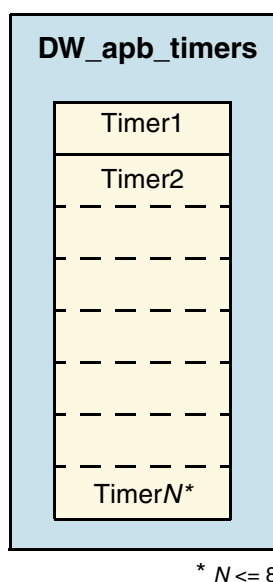
1.2 General Product Description

The Synopsys DW_apb_timers is a component of the DesignWare Advanced Peripheral Bus (DW_apb).

1.2.1 DW_apb_timers Block Diagram

Figure 1-2 shows the block diagram of the DW_apb_timers.

Figure 1-2 DW_apb_timers Block Diagram



1.3 Features

DW_apb_timers has the following features:

- ❖ Up to eight programmable timers
- ❖ Configurable timer width: 8 to 32 bits
- ❖ Support for two operation modes: free-running and user-defined count
- ❖ Support for independent clocking of timers
- ❖ Configurable polarity for each individual interrupt
- ❖ Configurable option for a single or combined interrupt output flag
- ❖ Configurable option to have read/write coherency registers for each timer
- ❖ Configurable option to include timer toggle output, which toggles whenever timer counter reloads

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

1.4 Standards Compliance

The DW_apb_timers component conforms to the *AMBA Specification, Revision 2.0* from ARM. Readers are assumed to be familiar with this specification.

1.5 Verification Environment Overview

The DW_apb_timers includes an extensive verification environment, detailed in “[Verification](#)” on page [55](#).

2

Functional Description

This chapter describes in the following sections how you can use the DW_apb_timers.

2.1 Timer Operation

The DW_apb_timers component implements up to eight identical but separately-programmable timers, which are accessed through a single AMBA APB interface.

Timers count down from a programmed value and generate an interrupt when the count reaches zero. You can use the TIM_INTR_IO parameter (Single Combined Interrupt) to create a single combined interrupt, which is active whenever any of the individual timer interrupts is active.

Each timer has an independent clock input, timer_N_clk (where N is in the range 1 to 8), that you can connect to pclk (also known as the system clock or the APB clock) or to an external clock source. You can configure the width of a timer from 8 to 32 bits using the TIMER_WIDTH_N parameter (Width of Timer N), where N is in the range 1 to NUM_TIMERS, the number of instantiated timers.

The initial value for each timer – that is, the value from which it counts down – is loaded into the timer using the appropriate load count register ([TimerNLoadCount](#)). Two events can cause a timer to load the initial count from its TimerNLoadCount register:

- ❖ Timer is enabled after being reset or disabled
- ❖ Timer counts down to 0

All interrupt status registers and end-of-interrupt registers can be accessed at any time.

2.2 DW_apb_timers Usage Flow

The procedure illustrated in [Figure 2-1](#) is a basic flow to follow when programming the DW_apb_timers. More advanced functions are discussed later in this chapter.

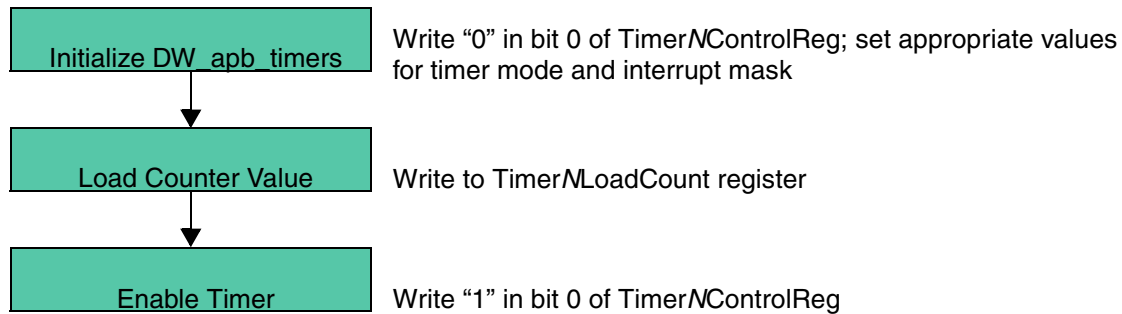
1. Initialize the timer through the [TimerNControlReg](#) register (where N is in the range 1 to 8):
 - a. Disable the timer by writing a “0” to the timer enable bit (bit 0); accordingly, the timer_en output signal is de-asserted.

**Note**

Before writing to a TimerNLoadCount register, you *must* disable the timer by writing a “0” to the timer enable bit of TimerNControlReg in order to avoid potential synchronization problems.

- b. Program the timer mode – user-defined or free-running – by writing a “0” or “1,” respectively, to the timer mode bit (bit 1).

- c. Set the interrupt mask as either masked or not masked by writing a “1” or “0,” respectively, to the timer interrupt mask bit (bit 2).
2. Load the timer counter value into the [TimerNLoadCount](#) register (where *N* is in the range 1 to 8).
3. Enable the timer by writing a “1” to bit 0 of [TimerNControlReg](#).

Figure 2-1 DW_apb_timers Usage Flow

As an example, suppose you have only timer1, and the timer_1_clk signal is asynchronous to pclk. When you disable the timer enable bit (bit 0 of [Timer1ControlReg](#)), the timer_en output signal is de-asserted and, accordingly, timer_1_clk should stop. Then when you enable the timer, the timer_en signal is asserted and timer_1_clk should start running. This is not necessary, however, as long as the timer_1_clk is synchronous to pclk; in this case, you can choose to directly tie timer_1_clk to pclk.

For more information on synchronization and metastability issues, refer to [“Controlling Clock Boundaries and Metastability”](#) on page 31.

2.3 DW_apb_timers Configuration

The following sections tell you how to set up the DW_apb_timers.

2.3.1 Choosing the Number of Timers

You can have up to eight timers in your design. There are several registers with names specific to the number of timers that you choose (where *N* is from 1 to 8):

- ❖ [TimerNLoadCount](#) – Timer*N* load count register
- ❖ [TimerNCurrentValue](#) – Timer*N* current value register
- ❖ [TimerNControlReg](#) – Timer*N* control register
- ❖ [TimerNEOI](#) – Timer*N* end-of-interrupt register
- ❖ [TimerNIntStatus](#) – Timer*N* interrupt status register

Thus you have five individual registers for each of the timers in your design. All other registers control their respective functions for all active timers, rather than for individual timers.

2.3.2 Enabling and Disabling a Timer

You use bit 0 of the [TimerNControlReg](#), where *N* is in the range 1 to 8, to either enable or disable a timer.

2.3.2.1 Enabling a Timer

If you want to enable a timer, you write a “1” to bit 0 of its [TimerNControlReg](#) register.

2.3.2.2 Disabling a Timer

To disable a timer, write a “0” to bit 0 of its `TimerNControlReg` register.

When a timer is enabled and running, its counter decrements on each rising edge of its clock signal, `timer_N_clk`. When a timer transitions from disabled to enabled, the current value of its `TimerNLoadCount` register is loaded into the timer counter on the next rising edge of `timer_N_clk`.

When the timer enable bit is de-asserted and the timer stops running, the timer counter and any associated registers in the timer clock domain, such as the toggle register, are asynchronously reset.

When the timer enable bit is asserted, then a rising edge on the `timer_en` signal is used to load the initial value into the timer counter. A “0” is always read back when the timer is not enabled; otherwise, the current value of the timer (`TimerNCurrentValue` register) is read back.

2.3.3 Configuring the Width of a Timer

You configure the width of a timer through the `TIMER_WIDTH_N` parameter; each timer can be from 8 bits to 32 bits. You do this for each timer through the Timer *N* Configuration. You should bear in mind that, if the width of the APB bus is smaller than the width of a timer—the APB data bus can be 8, 16, or 32 bits wide—there will have to be multiple APB write accesses to load the counter.

2.3.4 Loading a Timer Countdown Value

When a timer counter is enabled after being reset or disabled, the count value is loaded from the `TimerNLoadCount` register; this occurs in both free-running and user-defined count modes.

When a timer counts down to 0, it loads one of two values, depending on the timer operating mode:

- ❖ User-defined count mode – Timer loads the current value of the `TimerNLoadCount` register. Use this mode if you want a fixed, timed interrupt. Designate this mode by writing a “1” to bit 1 of `TimerNControlReg`.
- ❖ Free-running mode – Timer loads the maximum value, which is dependent on the timer width; that is, the `TimerNLoadCount` register is comprised of $2^{\text{TIMER_WIDTH_N}-1}$ bits, all of which are loaded with 1s. The timer counter wrapping to its maximum value allows time to reprogram or disable the timer before another interrupt occurs. Use this mode if you want a single timed interrupt. Designate this mode by writing a “0” to bit 1 of `TimerNControlReg`.

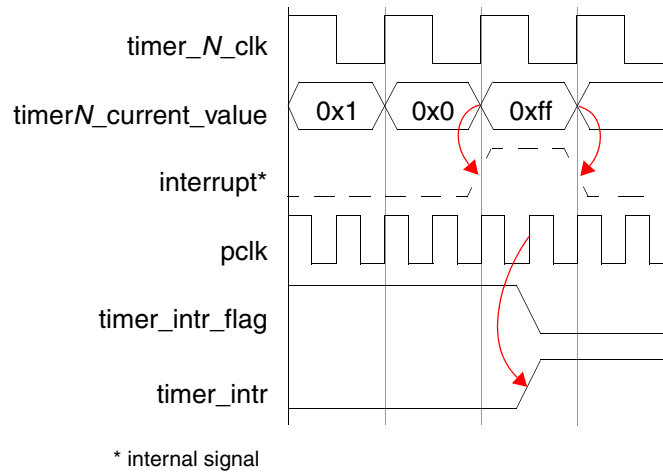
2.3.5 Working with Interrupts

The `TimerNIntStatus` and `TimerNEOI` registers handle interrupts in order to ensure safe operation of the interrupt clearing. Because of the `hclk/pclk` ratio, if `pclk` can perform a write to clear an interrupt, it could continue with another transfer on the bus without knowing whether the write has occurred. Therefore, it is much safer to clear the interrupt by a read operation.

To detect and service an interrupt, the system clock must be active. The `timer_en` output bus from this block is used to activate the necessary timer clocks and to ensure that the component is supplied with an active system clock while timers are running.

In both the free-running and user-defined count modes of operation, a timer generates an internal interrupt signal when its count changes from 0 to its maximum count value, as shown in [Figure 2-2](#).

Figure 2-2 Timer Interrupt Set – No Metastability Registers



The setting of the internal interrupt signal occurs synchronously to the timer clock domain. This internal interrupt signal is transferred to the pclk domain in order to set the timer interrupt. The internal interrupt signal and the timer interrupt are not generated if the timer is disabled; if the timer interrupt is set, then it is cleared when the timer is disabled.

If the system bus (AHB) can perform a write to clear a timer interrupt, it could continue with another transfer on the bus without knowing whether the write has occurred because of the hclk/pclk ratio. Therefore, it is much safer to clear the timer interrupt by a read operation.

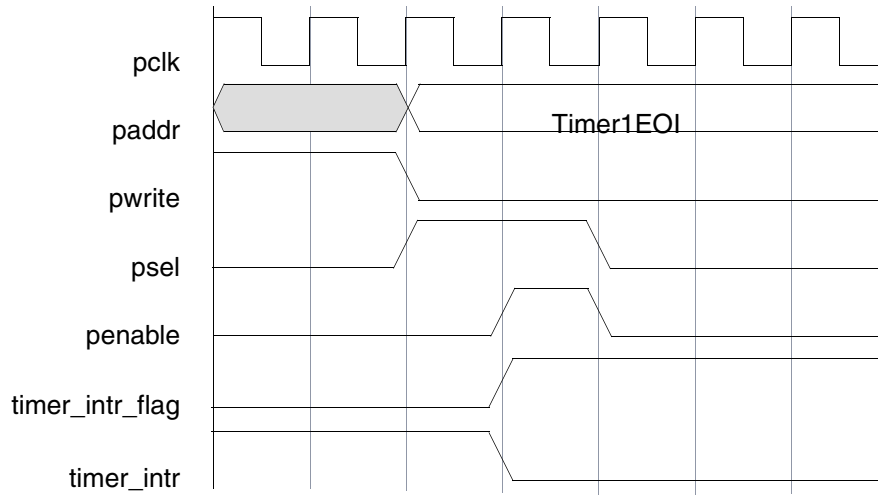
2.3.5.1 Clearing Interrupts

Provided the timer is enabled, its interrupt remains asserted until it is cleared by reading one of two end-of-interrupt registers ([TimerNEOI](#) or [TimersEOI](#), the individual and global end-of-interrupt registers, respectively). When the timer is disabled, the timer interrupt is cleared. You can clear an individual timer interrupt by reading its [TimerNEOI](#) register. You can clear all active timer interrupts at once by reading the global [TimersEOI](#) register or by disabling the timer.

When reading the [TimersEOI](#) register, timer interrupts are cleared at the rising edge of pclk and when penable is low. If an end-of-interrupt register is read during the time when the internal interrupt signal is high, the timer interrupt is set. This occurs because setting timer interrupts takes precedence over clearing them.

Figure 2-3 shows the timer interrupt timing when cleared by the TimersEOI register.

Figure 2-3 Clearing an Interrupt From DW_apb_timers



2.3.5.2 Checking Interrupt Status

You can query the interrupt status of an individual timer without clearing its interrupt by reading the [TimerNIntStatus](#) register. You can query the interrupt status of all timers without clearing the interrupts by reading the global [TimersIntStatus](#) register.

2.3.5.3 Masking Interrupts

Each individual timer interrupt can be masked using its [TimerNControlReg](#) register. To mask an interrupt, you write a “1” to bit 2 of [TimerNControlReg](#).

If all individual timer interrupts are masked, then the combined interrupt is also masked.

2.3.5.4 Setting Interrupt Polarity

The polarity of the generated timer interrupts can be configured to be either active-high or active-low using the `TIM_INTRPT_PLRITY` parameter (Interrupt Polarity). In addition to an interrupt output signal for each timer, there is also a single, global interrupt flag, `timer_intr_flag`, that is asserted if any timer asserts its interrupt. This global interrupt flag shares the same polarity characteristic with the other generated interrupts; thus, multiple interrupt service schemes can be supported.

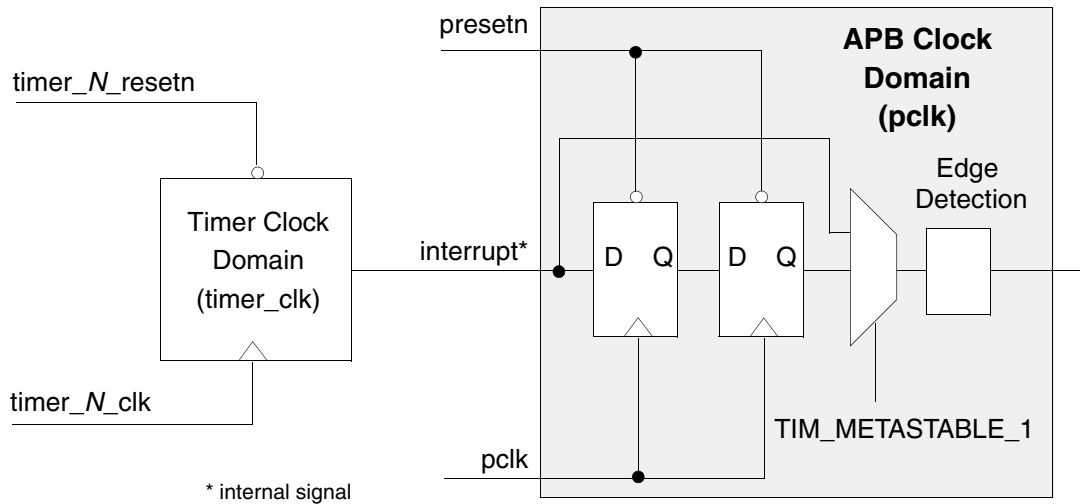
2.3.6 Controlling Clock Boundaries and Metastability

All registers in the APB interface are synchronous to `pclk`. Each of the timers has a separate clock input signal, `timer_N_clk`, that can be asynchronous or synchronous to `pclk`. It is possible to connect `timer_N_clk` to a clock other than `pclk`, but if you do that, you must take into account the possibility of synchronization and metastability issues. If a timer clock is asynchronous to `pclk`, you must ensure that the clocks are stopped whenever the timer is disabled.

The `timer_N_resn` signal resets all of the registers in the `timer_N_clk` domain, including the timer counter. For each timer, there are several factors that internally affect the boundaries between the `pclk` and timer clock domains.

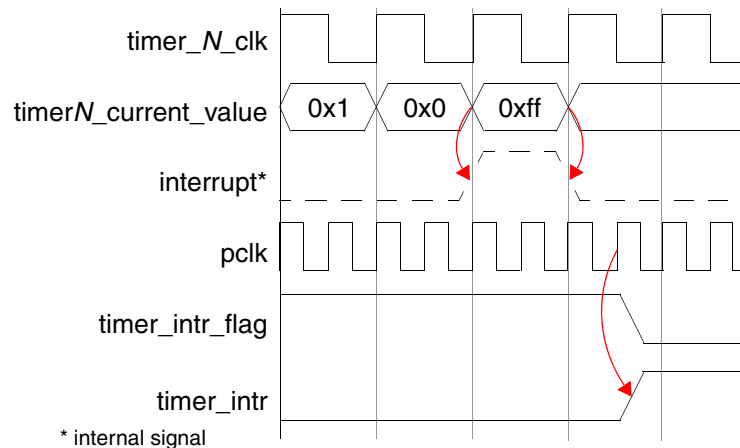
Each timer generates an internal interrupt signal that is synchronized to the pclk domain. [Figure 2-4](#) shows an internal interrupt signal affecting the clock boundaries between the two clock domains.

Figure 2-4 Boundary Between Clock Domains



The internal interrupt signal is generated in the timer clock domain when the timer counter rolls over to its maximum value. The internal interrupt signal is edge-detected in the pclk domain in order to set the timer interrupt, illustrated in [Figure 2-5](#).

Figure 2-5 Timer Interrupt Set – Metastability Registers Included



A timer_en signal is edge-detected in the timer clock domain. When it transitions from 0 to 1, the timer counter is loaded with the value of the TimerNLoadCount register. This guarantees that the timer is in a known state when enabled. If you disable a timer counter by writing a “0” to bit 0 of its TimerNControlReg register, it also synchronously disables interrupts for that timer counter in the pclk domain. This prevents spurious interrupts because of mis-sampling in the timer clock domain.

Neither the timer mode bit of TimerNControlReg nor the TimerNLoadCount register are synchronized between the pclk domain and the timer clock domain. Because of this, it is important that you disable a timer before programming its mode or load count value so that any information on these signals is always

communicated to the timer while it is inactive. Thus you must ensure that these signals are stable whenever a timer is enabled. In practice, this means that you must follow at least this basic procedure:

1. First use the `TimerNControlReg` to disable the timer, program its timer mode, and then set the interrupt mask.
2. Next, load the timer counter value into the `TimerNLoadCount` register.
3. Finally, enable the timer through `TimerNControlReg`.

For more details on this procedure, refer to “[DW_apb_timers Usage Flow](#)” on page 27.

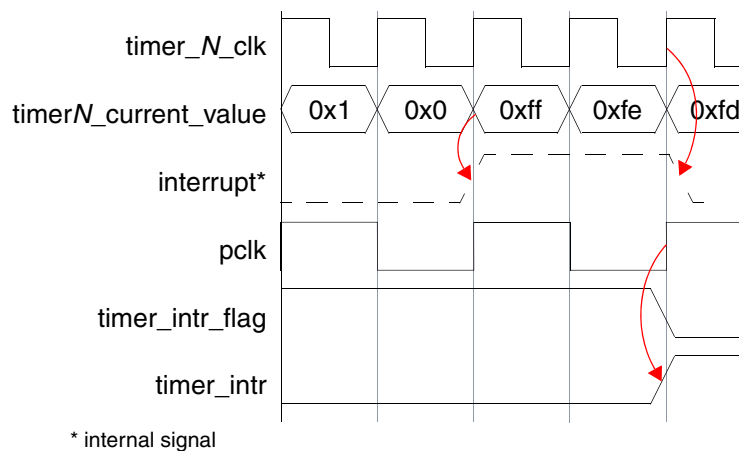
When you connect a `timer_N_clk` input to a clock source that is independent of `pclk`, metastability registers must be instantiated by setting the `TIM_METASTABLE_N` parameter (Metastability support for interrupt from Timer *N*) to “Present” (where *N* is in the range 1 to 8). By instantiating the metastability registers, an extra two `pclk` periods of latency occurs between when a timer maximum count is reached and when its interrupt goes active. To see the difference, compare the timing in [Figure 2-2](#) on page 30 (no metastability registers) to that in [Figure 2-5](#) on page 32 (metastability registers included).

The `DW_apb_timers` component supports timer clocks that are up to four times the frequency of `pclk`. If you connect a `timer_N_clk` to a clock source that is faster than `pclk`, you must extend the width of the internal interrupt signal to allow adequate time for it to be sampled in the `pclk` domain.

You extend the width of the interrupt signal up to three `timer_N_clk` clock periods by setting the `TIM_PULSE_EXTD_N` parameter (Number of clock cycles by which to extend interrupt, where *N* is in the range 1 to 8) to a non-zero value.

[Figure 2-6](#) illustrates an example of related `pclk` and `timer_N_clk`, where the frequency of `timer_N_clk` is two times that of `pclk`. To accommodate this, the `TIM_PULSE_EXTD_N` parameter is set to 1 in order to extend the internal interrupt signal by one `timer_N_clk` clock period.

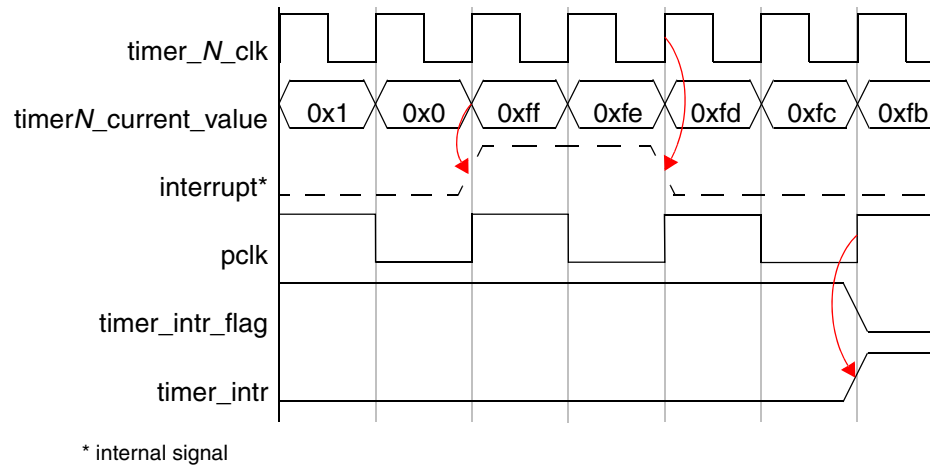
Figure 2-6 Timer Interrupt Set – Pulse Extend One Cycle, No Metastability



[Figure 2-7](#) illustrates an example where metastability registers are required because `pclk` is independent of `timer_N_clk`, and $1 < \text{frequency of timer_N_clk} < 2 \text{ times that of pclk}$. To accommodate this, the

TIM_PULSE_EXTD_N parameter is set to 1 in order to extend the internal interrupt signal by one timer_N_clk clock period.

Figure 2-7 Timer Interrupt Set – Pulse Extend One Cycle, With Metastability



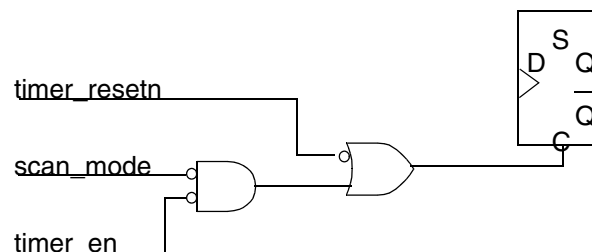
2.3.7 Generating Toggled Outputs

You can configure a timer through the TIMER_HAS_TOGGLE_N parameter (Include toggle output for timer # on I/F, see [page 36](#)) in order to generate an output that toggles whenever the timer counter reaches 0. You do this for each timer through the Timer N Configuration.

2.4 Design For Test

A scan_mode signal controls the asynchronous clear signal of some of the flip-flops during scan testing; the operation of this is shown in [Figure 2-8](#). In normal operation, in order to load a new value into a timer, the timer must be disabled. The new value is loaded into the timer on the first rising edge of the clock when the timer is re-enabled. To implement this, an asynchronous end-of-interrupt signal is supplied to some internal flip-flops. If scan_mode is asserted, this asynchronous signal is controlled by the timer reset signal. The scan_mode signal must be asserted during scan testing in order to ensure that all flip-flops in the design can be controlled and observed during scan testing; at all other times, this signal must be de-asserted.

Figure 2-8 Design For Test – Use of Scan Mode Signal



3

Parameters

This chapter describes the configuration parameters used by the DW_apb_timers. The settings of the configuration parameters determine the I/O signal list of the DW_apb_timers peripheral.



Attention

3.1 Parameter Descriptions

In the following tables, the values 0 and 1 occasionally appear in parentheses in the descriptions for the parameters.

Table 3-1 Top-Level Parameters

Field Label	Parameter Definition
APB Data Bus Width (bits)	Parameter Name: APB_DATA_WIDTH Legal Values: 8, 16, or 32 Default Value: 32 Dependencies: None. Description: Width of the APB data bus to which this component is attached.
Number of Timers to instantiate	Parameter Name: NUM_TIMERS Legal Values: 1 to 8 Default Value: 2 Dependencies: None. Description: Number of timers to instantiate in DW_apb_timers. Up to eight timers can be instantiated.

Table 3-1 Top-Level Parameters (Continued)

Field Label	Parameter Definition
Interrupt Polarity	Parameter Name: TIM_INTRPT_PLRITY Legal Values: Active-high (1) or active-low (0) Default Value: Active-high (1) Dependencies: None. Description: Polarity of interrupt signals generated by DW_apb_timers.
Single Combined Interrupt?	Parameter Name: TIM_INTR_IO Legal Values: True (1) or False (0) Default Value: False (0) Dependencies: None. Description: When set to True (1), the component generates a single interrupt combining all timer interrupts. If set to False (0), the component generates an interrupt output for each timer.


Table 3-2 Timer *i* Configuration Parameters

Field Label	Parameter Definition (<i>N</i> has the range 1 to <i>NUM_TIMERS</i>)
Width of Timer # <i>i</i>	Parameter Name: TIMER_WIDTH_ <i>N</i> , where <i>N</i> is in the range 1 to NUM_TIMERS Legal Values: 8 to 32 bits Default Value: 32 bits Dependencies: This option is disabled for the following: for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7 Description: Width of each timer.
Include toggle output for timer # <i>i</i> on I/F?	Parameter Name: TIMER_HAS_TOGGLE_ <i>N</i> , where <i>N</i> is in the range 1 to NUM_TIMERS Legal Values: True (1) or False (0) Default Value: False (0) Dependencies: This option is disabled for the following: for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7 Description: When set to True (1), the interface includes an output (timer_ <i>N</i> _toggle) that toggles each time the timer counter reloads. The output is disabled to 0 each time the timer is disabled.

Table 3-2 Timer *i* Configuration Parameters (Continued)

Field Label	Parameter Definition (<i>N</i> has the range 1 to <i>NUM_TIMERS</i>)												
Metastability support for interrupt from Timer # <i>i</i>	<p>Parameter Name: TIM_METASTABLE_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: Absent (0) or Present (1)</p> <p>Default Value: Absent (0)</p> <p>Dependencies: This option is disabled for the following: for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7</p> <p>Description: This option instantiates metastability registers to synchronize timer interrupt signals to the pclk domain. Set this to Present (1) if timer_<i>N</i>_clk is independent of pclk. If this parameter is set to Absent (0), then timer_<i>N</i>_clk is considered to be connected to or synchronous with pclk</p>												
Number of clock cycles by which to extend interrupt	<p>Parameter Name: TIM_PULSE_EXTD_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: 0 to 3</p> <p>Default Value: 0</p> <p>Dependencies: This option is disabled for the following: for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7</p> <p>Description: If this timer clock is faster than the system bus clock, you can extend the internal interrupt by up to three timer clock cycles to guarantee that it is seen in the bus clock domain. A 0 value in this field means that no pulse extension is performed. Also refer to “Controlling Clock Boundaries and Metastability” on page 31.</p> <p>Set this parameter to the following values, depending on the timer_<i>N</i>_clk/pclk frequency ratio <i>R</i>:</p> <table> <tr> <td>timer_<i>N</i>_clk/pclk frequency <i>R</i></td><td>PULSE_EXTEND_<i>N</i></td></tr> <tr> <td>$R \leq 1$</td><td>0</td></tr> <tr> <td>$1 < R \leq 2$</td><td>1</td></tr> <tr> <td>$2 < R \leq 3$</td><td>2</td></tr> <tr> <td>$3 < R \leq 4$</td><td>3</td></tr> <tr> <td>$4 < R$</td><td>Not valid</td></tr> </table>	timer_ <i>N</i> _clk/pclk frequency <i>R</i>	PULSE_EXTEND_ <i>N</i>	$R \leq 1$	0	$1 < R \leq 2$	1	$2 < R \leq 3$	2	$3 < R \leq 4$	3	$4 < R$	Not valid
timer_ <i>N</i> _clk/pclk frequency <i>R</i>	PULSE_EXTEND_ <i>N</i>												
$R \leq 1$	0												
$1 < R \leq 2$	1												
$2 < R \leq 3$	2												
$3 < R \leq 4$	3												
$4 < R$	Not valid												

Table 3-2 Timer *i* Configuration Parameters (Continued)

Field Label	Parameter Definition (<i>N</i> has the range 1 to <i>NUM_TIMERS</i>)
Include Coherency Registers for this timer?	<p>Parameter Name: TIM_COHERENCY_<i>N</i>, where <i>N</i> is in the range 1 to NUM_TIMERS</p> <p>Legal Values: True (1) or False (0)</p> <p>Default Value: False (0)</p> <p>Dependencies: You can set coherency for the timer if its width, TIMER_WIDTH_<i>N</i>, is greater than APB_DATA_WIDTH. If the TIMER_WIDTH is less than or equal to APB_DATA_WIDTH, coherency is not implemented for the specific timer.</p> <p>This option is disabled for the following:</p> <ul style="list-style-type: none"> for <i>N</i> = 2, when NUM_TIMERS = 1; for <i>N</i> = 3, when NUM_TIMERS <= 2; for <i>N</i> = 4, when NUM_TIMERS <= 3; for <i>N</i> = 5, when NUM_TIMERS <= 4; for <i>N</i> = 6, when NUM_TIMERS <= 5; for <i>N</i> = 7, when NUM_TIMERS <= 6; for <i>N</i> = 8, when NUM_TIMERS <= 7 <p>Description: When set to True (1), a bank of registers is added between this timer and the APB interface of DW_apb_timers to guarantee that the timer value read back from this block is coherent. It does not reflect ongoing changes in the timer value that take place while the read operation is in progress.</p> <div>  <p>Note Including coherency can dramatically increase the register count of the design.</p> </div>

4

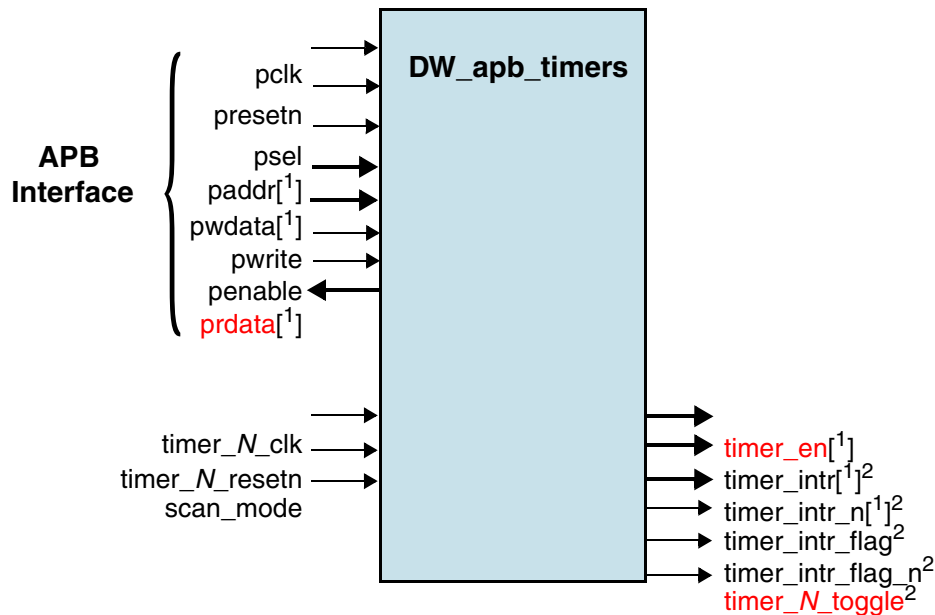
Signals

The following subsections describe the DW_apb_timers I/O signals.

4.1 DW_apb_timers Interface Diagram

Figure 4-1 shows the I/O signal diagram for DW_apb_timers (see Table 4-1 on page 40).

Figure 4-1 DW_apb_timers Interface Diagram



¹ Refer to Table 5-1 on page 40 for the sizes of these buses.

² These signals are instantiated depending on the configuration of DW_apb_timers. For more information refer to Table 5-1 on page 40 and Table 4-2 on page 36.

Signals in red indicate a registered input or output.

4.2 DW_apb_timers Signal Descriptions

All signals are synchronous to the pclk, except where otherwise specified. Table 4-1 identifies the signals that are associated with the DW_apb_timers. For an illustration of the I/O block diagram for the DW_apb_timers, refer to Figure 1-2 on page 13.

**Note**

The Description column in [Table 4-1](#) provides detailed information about each signal.

In the **Registered** field, a “Yes” indicates whether an I/O signal is directly connected to an internal register and nothing else. An I/O signal is also considered to be registered if the signal is connected to one or more inverters or buffers between the I/O port and internal register, but not connected to any logic that involves another signal.

The **Input/Output Delay** field provides the percentage of the clock cycle assumed to be used by logic outside this design. The given value is used to automatically define the default synthesis constraints for input/output delay. You can override these default values in the Specify Port Constraints.

Table 4-1 DW_apb_timers Signal Description

Name	Width	I/O	Description
ABP Interface			
pclk	1 bit	In	<p>APB clock; also known as the system clock. This clock times all bus transfers. All signal timings are related to the rising edge of pclk.</p> <p>Active State: N/A</p> <p>Registered: N/A</p> <p>Synchronous to: N/A</p> <p>External Input Delay: N/A</p>
presetn	1 bit	In	<p>APB reset. The bus reset signal is used to reset the system and the bus on the DesignWare interface.</p> <p>Active State: Low</p> <p>Registered: N/A</p> <p>Synchronous to: Asynchronous APB interface domain reset. This signal resets only the bus interface. The signal is asserted asynchronously, but is deasserted synchronously after the rising edge of pclk. DW_apb_timers does not contain logic to perform this synchronization, so it must be provided externally.</p> <p>External Input Delay: N/A</p>
psel	1 bit	In	<p>APB peripheral select</p> <p>Active State: High</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Input Delay: 40%</p>
paddr	8 bits	In	<p>APB address bus</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Input Delay: 40%</p>
pwwdata	See Description	In	<p>APB write data bus</p> <p>Width: <i>APB_DATA_WIDTH</i></p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Input Delay: 40%</p>

Table 4-1 DW_apb_timers Signal Description (Continued)

Name	Width	I/O	Description
pwrite	1 bit	In	APB write control Active State: High Registered: No Synchronous to: pclk External Input Delay: 40%
penable	1 bit	In	APB enable control that indicates the second cycle of the APB frame Active State: High Registered: No Synchronous to: pclk External Input Delay: 40%
prdata	See Description	Out	APB readback data Width: <i>APB_DATA_WIDTH</i> Registered: Yes Synchronous to: pclk External Output Delay: 30%
Timer Signals			
timer_N_clk	1 bit	In	Each timer is supplied with its own clock from this bus. The number of these signals is set by NUM_TIMERS parameter. Active State: High Registered: N/A Synchronous to: This signal can be asynchronous or synchronous to pclk. If a timer clock is asynchronous to pclk, you must ensure that the clocks are stopped whenever the timer is disabled. External Input Delay: N/A
timer_N_resetrn	1 bit	In	Asynchronous reset for each timer. The number of these signals are set by NUM_TIMERS parameter. Active State: Low Registered: N/A Synchronous to: Asynchronous assertion, synchronous de-assertion. Must be synchronously de-asserted after the rising edge of pclk. External Input Delay: 40% of timer_N_clk
scan_mode	1 bit	In	Active-high scan mode used to ensure that test automation tools can control all asynchronous flip-flop signals. This signal should be asserted – that is, driven to logic 1 – during scan testing, and should be deasserted – tied to logic 0 – at all other times. For more information, refer to “ Design For Test ” on page 34. Active State: High Registered: No Synchronous to: pclk External Input Delay: 40%

Table 4-1 DW_apb_timers Signal Description (Continued)

Name	Width	I/O	Description
timer_en	See Description	Out	<p>When asserted, activates the necessary timer clocks and ensures the component is supplied with an active pclk while timers are running. You can tie a timer clock to pclk, but if pclk is asynchronous to a timer clock, then you must stop the timer clock before programming it. Timer clock should start and stop depending on assertion and de-assertion of the timer_en output signal when the timer clock is asynchronous to pclk.</p> <p>Width: NUM_TIMERS</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 30%</p>
timer_intr	See Description	Out	<p><i>Optional.</i> Timer interrupt signals.</p> <p>Width: NUM_TIMERS</p> <p>Active State: High</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 30%</p> <p>Dependencies: Present only when active-high interrupts are configured (TIM_INTRPT_PLRITY = 1) AND single interrupts are configured (TIM_INTR_IO = 1).</p>
timer_intr_n	See Description	Out	<p><i>Optional.</i> Timer interrupt signals.</p> <p>Width: NUM_TIMERS</p> <p>Active State: Low</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 30%</p> <p>Dependencies: Present only when active-low interrupts are configured (TIM_INTRPT_PLRITY = 0) AND single interrupts are configured (TIM_INTR_IO = 1).</p>
timer_intr_flag	1 bit	Out	<p><i>Optional.</i> Interrupt flag that is set if any timer interrupt is set.</p> <p>Active State: High</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 30%</p> <p>Dependencies: Present only when active-high interrupts are configured (TIM_INTRPT_PLRITY = 1) AND combined interrupts are configured (TIM_INTR_IO = 0).</p>

Table 4-1 DW_apb_timers Signal Description (Continued)

Name	Width	I/O	Description
timer_intr_flag_n	1 bit	Out	<p><i>Optional.</i> Interrupt flag that is set if any timer interrupt is set.</p> <p>Active State: Low</p> <p>Registered: No</p> <p>Synchronous to: pclk</p> <p>External Output Delay: 30%</p> <p>Dependencies: Present only when active-low interrupts are configured (TIM_INTRPT_PLRITY = 0) AND combined interrupts are configured (TIM_INTR_IO = 0).</p>
timer_N_toggle	1 bit	Out	<p><i>Optional.</i> Signal that toggles each time the timer counter reloads. The output is disabled to 0 each time the timer is disabled.</p> <p>Active State: High</p> <p>Registered: Yes</p> <p>Synchronous to: timer_N_clk</p> <p>External Output Delay: 30%</p> <p>Dependencies: Present only when the configuration parameter TIMER_HAS_TOGGLE_N is set to 1, where N is 1 to the number of configured timers (up to 8).</p>

5

Registers

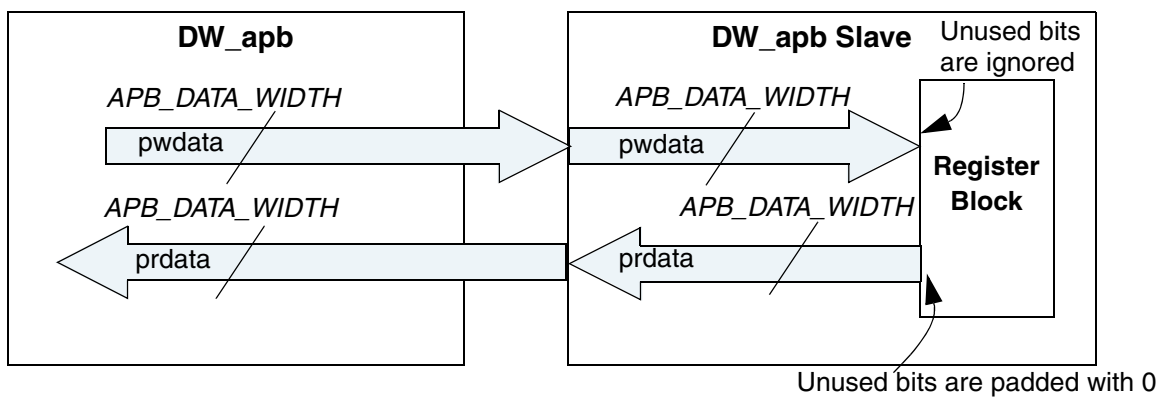
This section describes the programmable registers of the DW_apb_timers.

5.1 Bus Interface

The DW_apb_timers peripheral has a standard AMBA 2.0 APB interface for reading and writing the internal registers. This peripheral supports APB data bus widths of 8, 16, or 32 bits, which is set with the APB_DATA_WIDTH parameter.

Figure 5-1 shows the read/write buses between the DW_apb and the APB slave.

Figure 5-1 Relationship Between DW_apb and Slave Data Widths



“Integration Considerations” on page 59 provides information about reading to and writing from the APB interface.

5.2 Register Memory Map

The following tables contain information about the register memory map.

[Table 5-1](#) lists the address ranges of the registers for each timer; they are aligned to 32-bit boundaries.

Table 5-1 DW_apb_timers Address Range

Address Range (Base +)	Function
0x00 to 0x10	Timer 1 Registers
0x14 to 0x24	Timer 2 Registers
0x28 to 0x38	Timer 3 Registers
0x3c to 0x4c	Timer 4 Registers
0x50 to 0x60	Timer 5 Registers
0x64 to 0x74	Timer 6 Registers
0x78 to 0x88	Timer 7 Registers
0x8c to 0x9c	Timer 8 Registers

[Table 5-2](#) lists registers associated with Timer 1; use this table as an example for timers 2-8.

Table 5-2 Memory Map of Timer 1 Registers

Name	Address Offset	Width	R/W	Description
Timer1LoadCount	0x00, 0x01, 0x02, 0x03	See Description	R/W	Value to be loaded into Timer1. Width: <i>TIMER_WIDTH_1-1</i> Range: 0 to [$2^{TIMER_WIDTH_1} - 1$] Default value: <i>TIMER_WIDTH_1-1'b0</i>
Timer1CurrentValue	0x04, 0x05, 0x06, 0x07	See Description	R	Current Value of Timer1. Width: <i>TIMER_WIDTH_1-1</i> Range: 0 to [$2^{TIMER_WIDTH_1} - 1$] Default value: <i>TIMER_WIDTH_1-1'b0</i>
Timer1ControlReg	0x08	3 bits	R/W	Control Register for Timer1. Default value: 3'b0
Timer1EOI	0x0C	1 bit	R	Clears the interrupt from Timer1. Default value: 1'b0
Timer1IntStatus	0x10	1 bit	R	Contains the interrupt status for Timer1. Default value: 1'b0

DW_apb_timers also contain the following three system registers, listed in Table 5-3.

Table 5-3 DW_apb_timers System Registers

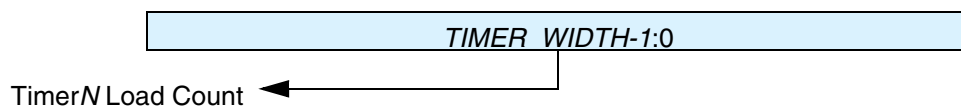
Name	Address Offset	Width	R/W	Description
TimersIntStatus	0xa0	See Description	R	Contains the interrupt status of all timers in the component. Width: <i>NUM_TIMERS</i> -1:0 Default value: <i>NUM_TIMERS</i> 'b0
TimersEOI	0xa4	See Description	R	Returns all zeroes (0) and clears all active interrupts. Width: <i>NUM_TIMERS</i> -1:0 Default value: <i>NUM_TIMERS</i> 'b0
TimersRawIntStatus	0xa8	See Description	R	Contains the unmasked interrupt status of all timers in the component. Width: <i>NUM_TIMERS</i> -1:0 Default value: <i>NUM_TIMERS</i> 'b0
TIMERS_COMP_VERSION	0xac	32 bits	R	Current revision number of the DW_apb_timers component.

5.3 Register and Field Descriptions

The following sections contain the memory diagrams and field descriptions for the individual registers.

5.3.1 TimerNLoadCount

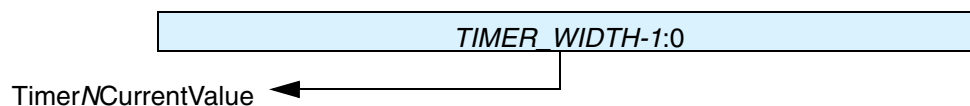
- ❖ **Name:** TimerN Load Count Register
- ❖ **Size:** *TIMER_WIDTH_N-1*, where *N* is 1...8
- ❖ **Address Offset:**
 - for *N* = 1, 0x00
 - for *N* = 2, 0x14
 - for *N* = 3, 0x28
 - for *N* = 4, 0x3C
 - for *N* = 5, 0x50
 - for *N* = 6, 0x64
 - for *N* = 7, 0x78
 - for *N* = 8, 0x8C
- ❖ **Read/write access:** read/write



Bits	Name	R/W	Description
<i>TIMER_WIDTH_N-1:0</i> , where <i>N</i> is 1...8	TimerNLoad Count Register	R/W	Value to be loaded into TimerN. This is the value from which counting commences. Any value written to this register is loaded into the associated timer.

5.3.2 TimerNCurrentValue

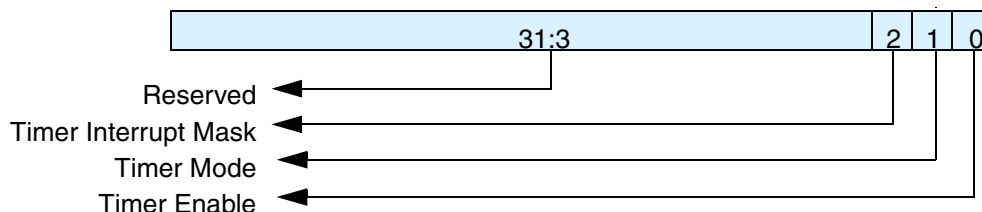
- ❖ **Name:** TimerN Current Value Register
- ❖ **Size:** $TIMER_WIDTH_N-1$, where N is 1...8
- ❖ **Address Offset:**
 - for $N = 1$, 0x04
 - for $N = 2$, 0x18
 - for $N = 3$, 0x2C
 - for $N = 4$, 0x40
 - for $N = 5$, 0x54
 - for $N = 6$, 0x68
 - for $N = 7$, 0x7C
 - for $N = 8$, 0x90
- ❖ **Read/write access:** read



Bits	Name	R/W	Description
$TIMER_WIDTH_N-1$: 0, where N is 1...8	Timer N Current Value	R	Current Value of Timer N . This register is supported only when timer_ N _clk is synchronous to pclk. Reading this register when using independent clocks results in an undefined value.

5.3.3 TimerNControlReg

- ❖ **Name:** TimerN Control Register
- ❖ **Size:** 3 bits
- ❖ **Address Offset:**
 - for $N = 1$, 0x08
 - for $N = 2$, 0x1C
 - for $N = 3$, 0x30
 - for $N = 4$, 0x44
 - for $N = 5$, 0x58
 - for $N = 6$, 0x6C
 - for $N = 7$, 0x80
 - for $N = 8$, 0x94
- ❖ **Read/write access:** read/write

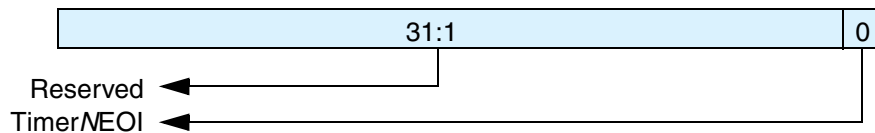


This register controls enabling, operating mode (free-running or defined-count), and interrupt mask of TimerN. You can program each TimerNControlReg to enable or disable a specific timer and to control its mode of operation.

Bits	Name	R/W	Description
31:3	Reserved, read as zero		
2	Timer Interrupt Mask	R/W	Timer interrupt mask for TimerN. 0: not masked 1: masked
1	Timer Mode	R/W	Timer mode for TimerN. 0: free-running mode 1: user-defined count mode For more information about these modes, see “Timer Operation” on page 27.
0	Timer Enable	R/W	Timer enable bit for TimerN. 0: disable 1: enable

5.3.4 TimerNEOI

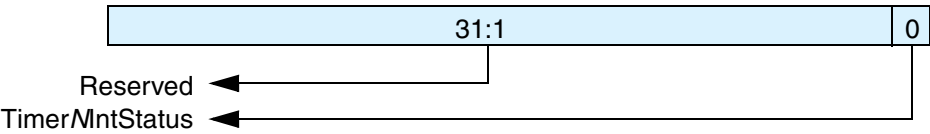
- ❖ **Name:** TimerN End-of-Interrupt Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:**
 - for N = 1, 0x0C
 - for N = 2, 0x20
 - for N = 3, 0x34
 - for N = 4, 0x48
 - for N = 5, 0x5C
 - for N = 6, 0x70
 - for N = 7, 0x84
 - for N = 8, 0x98
- ❖ **Read/write access:** read



Bits	Name	R/W	Description
31:1	Reserved, read as zero		
0	TimerN End-of-Interrupt Register	R	Reading from this register returns all zeroes (0) and clears the interrupt from TimerN.

5.3.5 TimerMntStatus

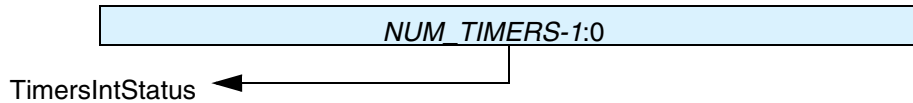
- ❖ **Name:** TimerN Interrupt Status Register
- ❖ **Size:** 1 bit
- ❖ **Address Offset:**
 - for N = 1, 0x10
 - for N = 2, 0x24
 - for N = 3, 0x38
 - for N = 4, 0x4C
 - for N = 5, 0x60
 - for N = 6, 0x74
 - for N = 7, 0x88
 - for N = 8, 0x9C
- ❖ **Read/write access:** read



Bits	Name	R/W	Description
31:1	Reserved, read as zero		
0	TimerN Interrupt Status Register	R	Contains the interrupt status for TimerN.

5.3.6 TimersIntStatus

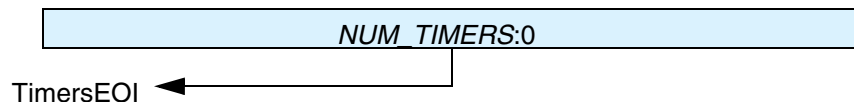
- ❖ **Name:** Timers Interrupt Status Register
- ❖ **Size:** *NUM_TIMERS*
- ❖ **Address Offset:** 0xa0
- ❖ **Read/write access:** read



Bits	Name	R/W	Description
<i>NUM_TIMERS</i> -1:0	Timers Interrupt Status Register	R	Contains the interrupt status of all timers in the component. If a bit of this register is 0, then the corresponding timer interrupt is not active – and the corresponding interrupt could be on either the timer_intr bus or the timer_intr_n bus, depending on the interrupt polarity you have chosen. Similarly, if a bit of this register is 1, then the corresponding interrupt bit has been set in the relevant interrupt bus. In both cases, the status reported is the status after the interrupt mask has been applied. Reading from this register does not clear any active interrupts: 0 = either timer_intr or timer_intr_n is not active after masking 1 = either timer_intr or timer_intr_n is active after masking

5.3.7 TimersEOI

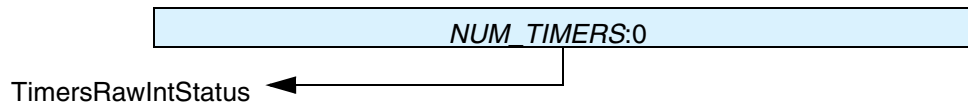
- ❖ **Name:** Timers End-of-Interrupt Register
- ❖ **Size:** *NUM_TIMERS*
- ❖ **Address Offset:** 0xa4
- ❖ **Read/write access:** read



Bits	Name	R/W	Description
<i>NUM_TIMERS</i> -1:0	Timers End-of-Interrupt Register	R	Reading this register returns all zeroes (0) and clears all active interrupts.

5.3.8 TimersRawIntStatus

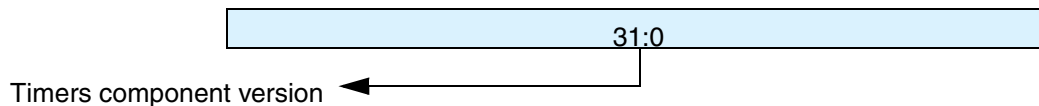
- ❖ **Name:** Timers Raw Interrupt Status Register
- ❖ **Size:** *NUM_TIMERS*
- ❖ **Address Offset:** 0xa8
- ❖ **Read/write access:** read



Bits	Name	R/W	Description
<i>NUM_TIMERS</i> -1:0	Timers Raw Interrupt Status Register	R	The register contains the unmasked interrupt status of all timers in the component. 0 = either timer_intr or timer_intr_n is not active prior to masking 1 = either timer_intr or timer_intr_n is active prior to masking

5.3.9 TIMERS_COMP_VERSION

- ❖ **Name:** Timers Component Version
- ❖ **Size:** 32 bits
- ❖ **Address Offset:** 0xac
- ❖ **Read/write access:** read



Bits	Name	R/W	Description
31:0	Timers Component Version	R	Current revision number of the DW_apb_timers component. Reset Value: For the value, see the releases table in the AMBA 2 release notes

6

Programming Considerations

This chapter describes the programmable features of the DW_apb_timers.

In order to avoid potential synchronization problems when initializing, loading, and enabling a timer, you should follow the basic procedure outline in “DW_apb_timers Usage Flow” on page 27.

The DW_apb_timers module is little-endian. All timers are disabled on reset and are enabled by writing “1” to the timer enable bit of the [TimerNControlReg](#). DW_apb_timers contains both timer-specific and system registers. [Table 5-1](#) on page 46 shows the address range of the registers of each timer, which are aligned to 32-bit boundaries.

If a timer is wider than the read data bus to which the slave is attached, more than one access must be performed to read the [TimerNCurrentValue](#) register. If more than one access is performed to read a timer value, the coherency of the value read cannot be guaranteed unless you configure read/write coherency for the specific timer. Read/write coherency is meaningful only if the `TIMER_WIDTH` is greater than the `APB_DATA_WIDTH`, under which circumstances the coherency registers are never instantiated in the design.

If there is no coherency set for a specific timer, software should read the registers more than once. For example, the software should read least-significant bits (LSBs), then most-significant bits (MSBs), and then LSBs again.

**Note**

The coherency circuitry incorporates an upper byte method that requires you to program the load register in LSB-to-MSB order when the peripheral width is smaller than the register width. Additionally, you must read LSB-to-MSB for the coherency circuitry solution to operate correctly.

When the upper byte is programmed, the value can be transferred and loaded into the load register. When the lower bytes are programmed, they need to be stored in shadow registers so that the previous load register is available to the timer 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.

**Note**

Reading the [TimerNCurrentValue](#) register is not supported if `timer_N_clk` is asynchronous to `pclk`. Any attempt to read this register when the clocks are independent may result in an undefined value.

7

Verification

This chapter provides an overview of the testbench available for DW_apb_timers verification.

**Note**

The DW_apb_timers verification testbench is built with DesignWare Verification IP (VIP).

8.1 Overview of Vera Tests

The DW_apb_timers verification testbench performs tests that have been written to verify three types of functionalities:

- ❖ test_readwrite_regs – Tests read/write functionalities of each timer register.
- ❖ test_reset – Tests functions related to resetting all timers.
- ❖ test_timer – Tests general functions of each timer.

The tests are performed on the following:

- ❖ APB Slave Interface – DW_apb_timers consists of an APB slave interface and a separate timer block for each timer instantiated. These tests verify that the APB Slave interface implements the memory map for DW_apb_timers and also contain metastability flip-flops to synchronize interrupt flags coming from the timer clock domains to the bus system clock domain. The tests are run for an 8-bit, 16-bit, and 32-bit APB system.
- ❖ Timer blocks – Each timer instantiated in the DW_apb_timers has a block clocked by its own timer_N_clk. These tests verify that the block flags interrupts to the APB slave interface and carries out pulse extension of signals going to the slave interface block to handle scenarios where the timer_N_clk runs at a higher frequency than pclk.

The tests perform the following tasks:

- ◆ Disables a timer, programs the load value, and re-enables it.
- ◆ Verifies that in free-running mode the timer counts from the load value down to zero before wrapping to its maximum value and proceeding with its count.
- ◆ Verifies that in user-defined mode, the timer wraps back to the load value after passing 0.

- ◆ Verifies that pulse extension, when configured, behaves so that the interrupt and current value are correctly extended for one, two, or three timer clock cycles as required.

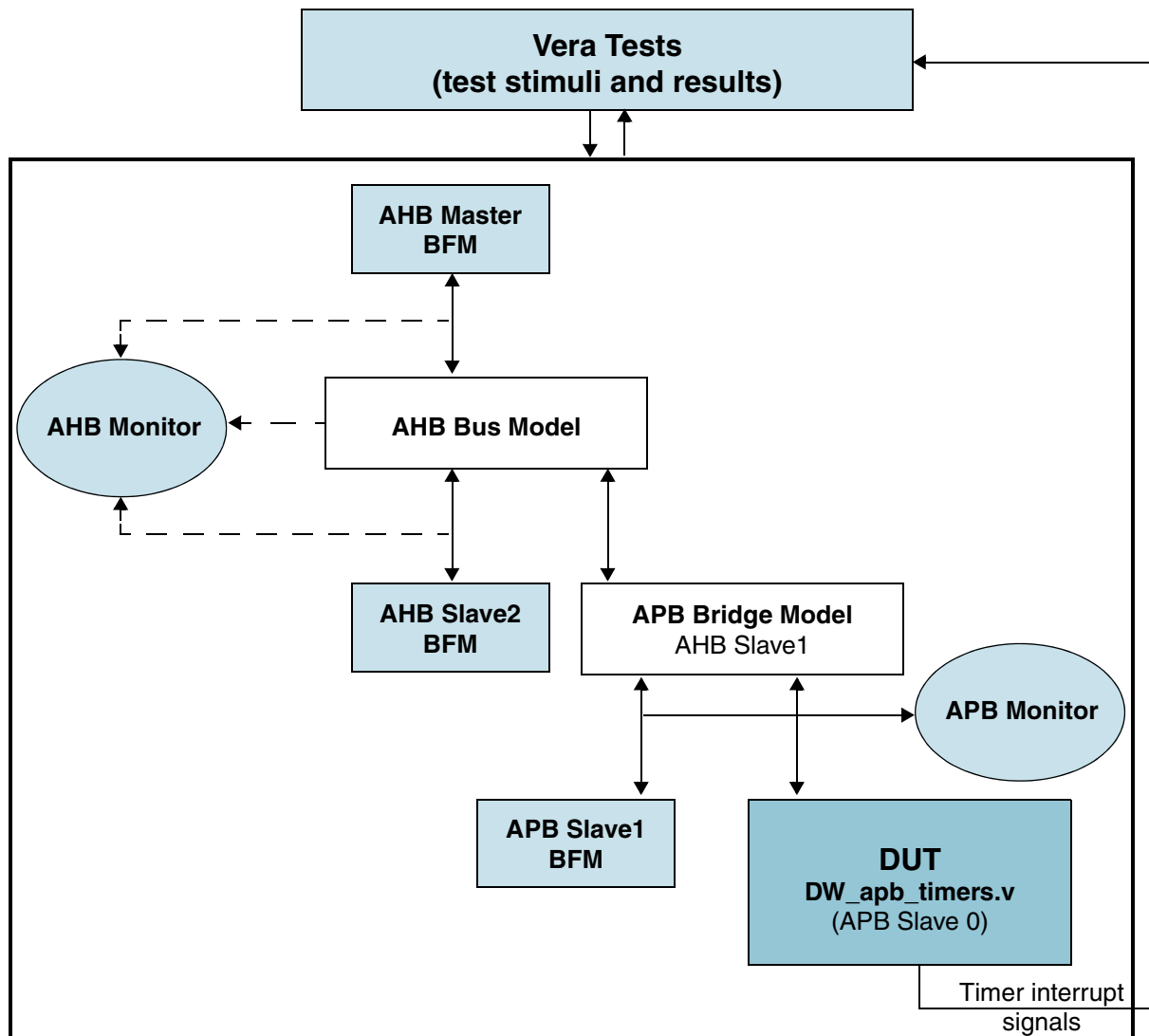
**Note**

All tests have achieved maximum RTL code coverage and use the APB Interface to dynamically program memory-mapped registers during tests.

7.2 Overview of DW_apb_timers Testbench

As illustrated in [Figure 7-1](#), the DW_apb_timers Verilog testbench includes an instantiation of the design under test (DUT), AHB and APB Bridge bus models, and a Vera shell.

Figure 7-1 DW_apb_timers Testbench



The Vera shell consists of an AHB master bus functional model (BFM), two AHB slave BFM, an AHB monitor, APB slave BFM, an APB monitor, test stimuli, BFM configuration, and test results. The AHB monitor tracks activity from the AHB master and slave BFM; the APB monitor oversees activity from the APB slave BFM.

8

Integration Considerations

8.1 Reading and Writing from an APB Slave

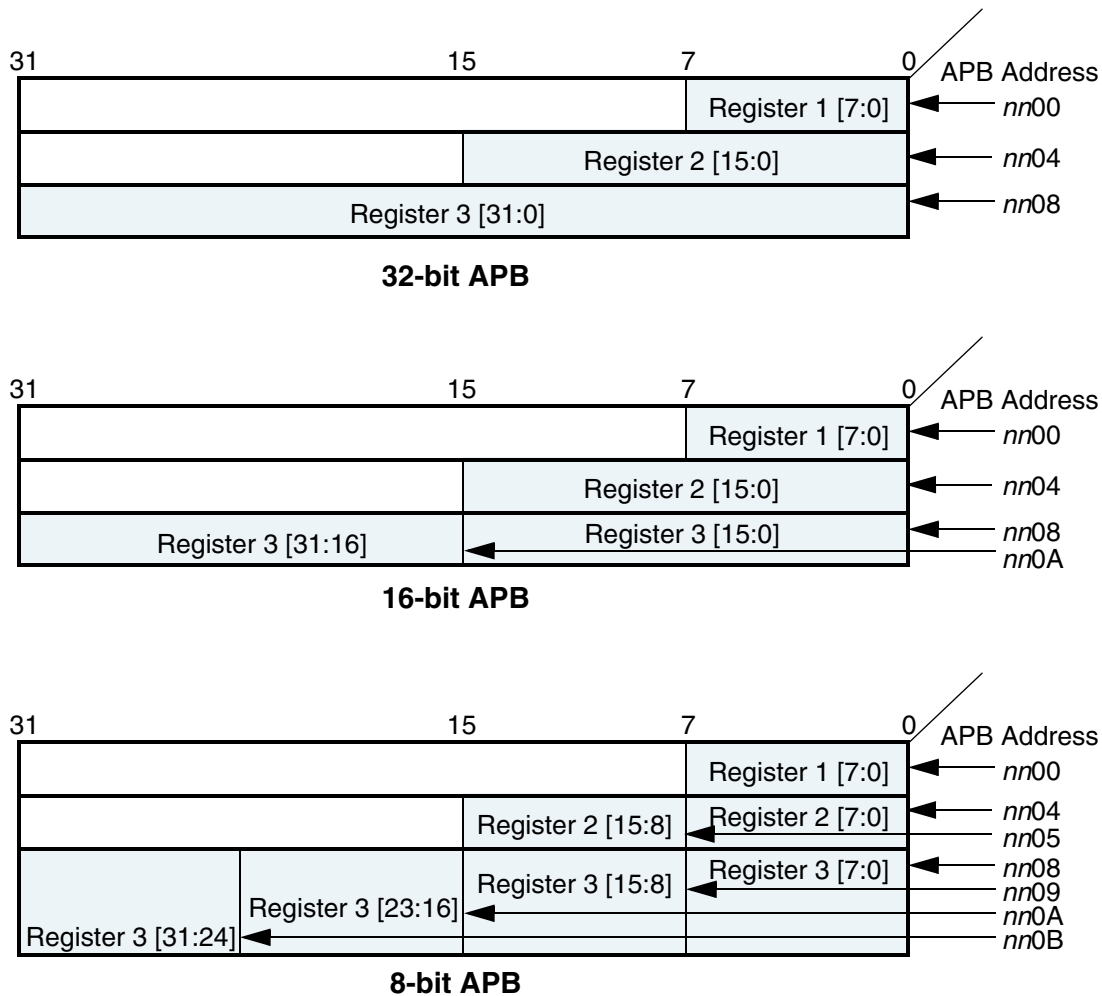
When writing to and reading from DesignWare APB slaves, you should consider the following:

- ❖ 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 ERROR, SPLIT, or RETRY responses; 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.

8.1.1 Reading From Unused Locations

Reading from an unused location or unused bits in a particular register always returns zeros. Unlike an AHB slave interface, which would return an error, there is no error mechanism in an APB slave and, therefore, in the DW_apb.

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 8-1 Read/Write Locations for Different APB Bus Data Widths

8.1.2 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.

9.1.3 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.



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

8.1.4 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.

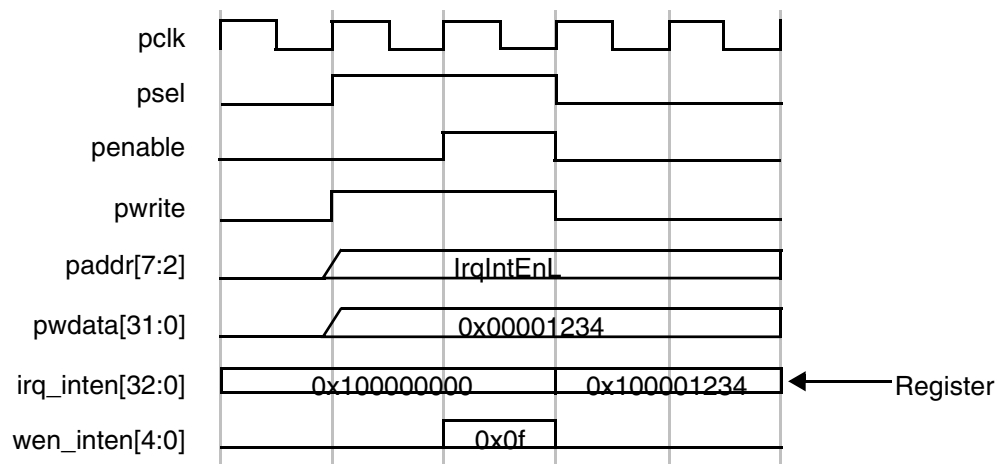
3. The register to be written to or read from is >16 and ≤ 32 bits

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.

8.2 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.

Figure 8-2 APB Write Transaction

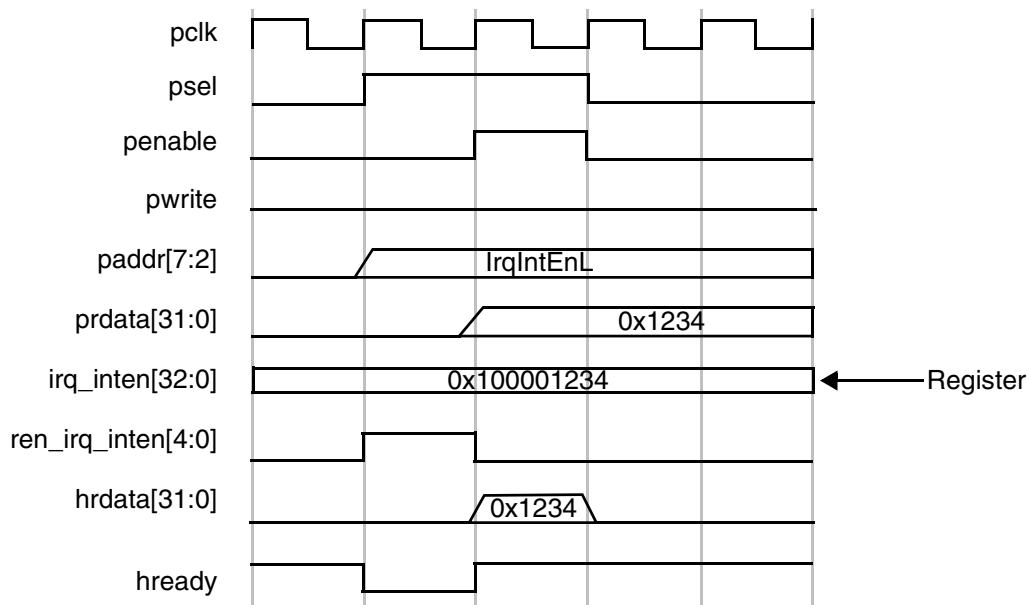
A write can occur after the first phase with `penable` low, or after the second phase when `penable` is high. The second phase is preferred and is used in all APB slave components. The timing diagram is shown with the write occurring after the second phase. Whenever the address on `paddr` matches a corresponding 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.

8.3 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.

Figure 8-3 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 is started, it is completed and the AHB bus is held until the data is returned from the slave

**Note**

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

8.4 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.

8.4.1 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 gives 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 8-1 Upper Byte Generation

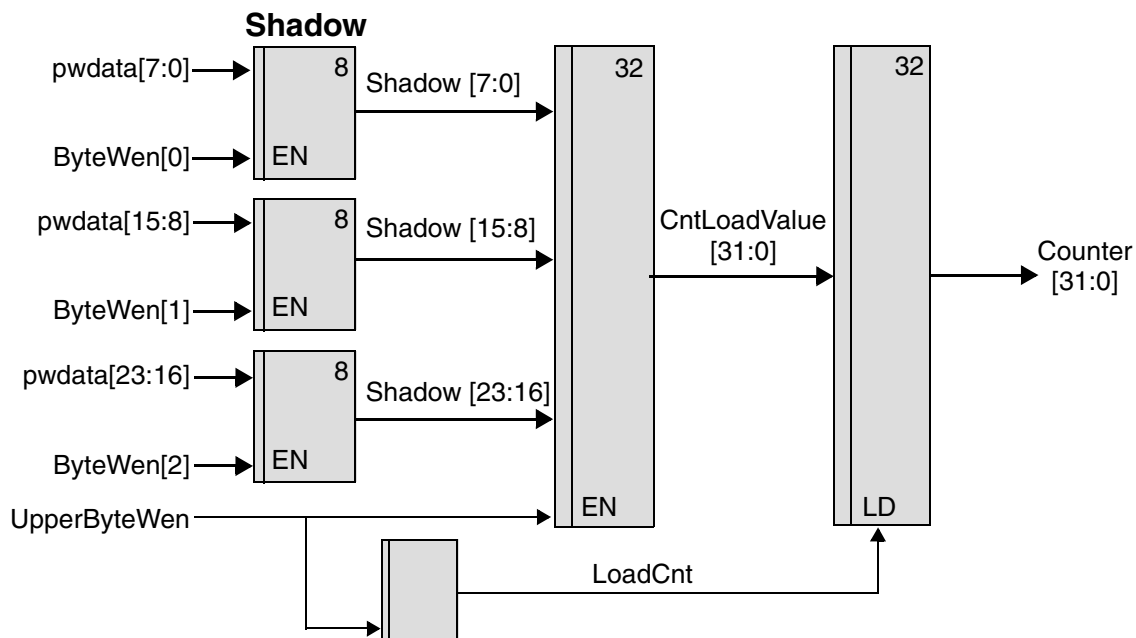
Load Register Width	Upper Byte Bus Width		
	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	1	NCR	NCR
17 - 24	2	2	NCR
25 - 32	3	2 (or 3)	NCR

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

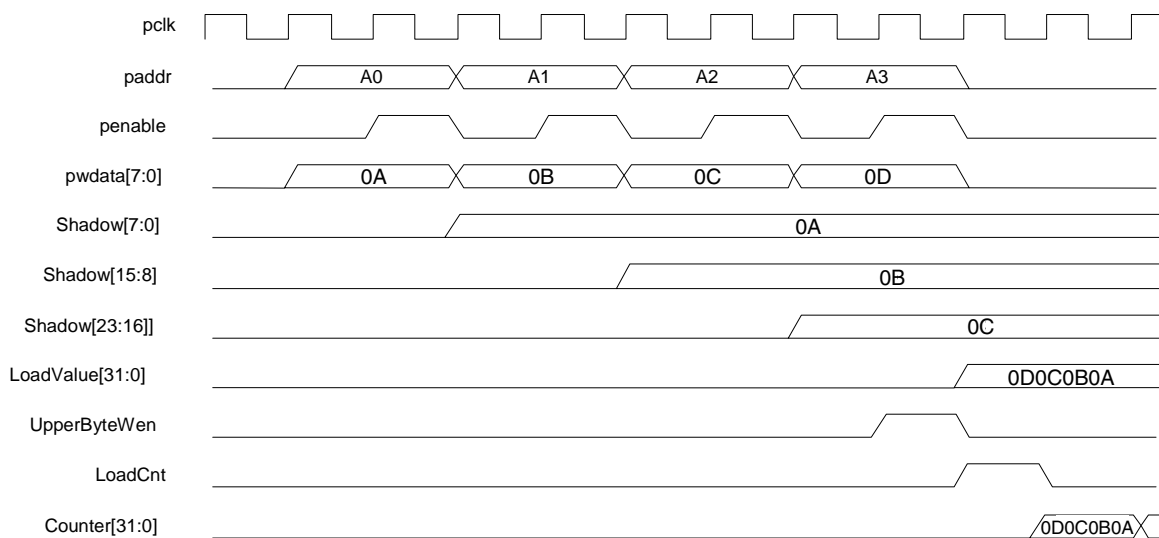
- ❖ Identical
- ❖ Synchronous (phase coherent but of an integer fraction)
- ❖ Asynchronous

8.4.1.1 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 8-4 Coherent Loading – Identical Synchronous Clocks

The following timing diagram shows 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 8-5 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.

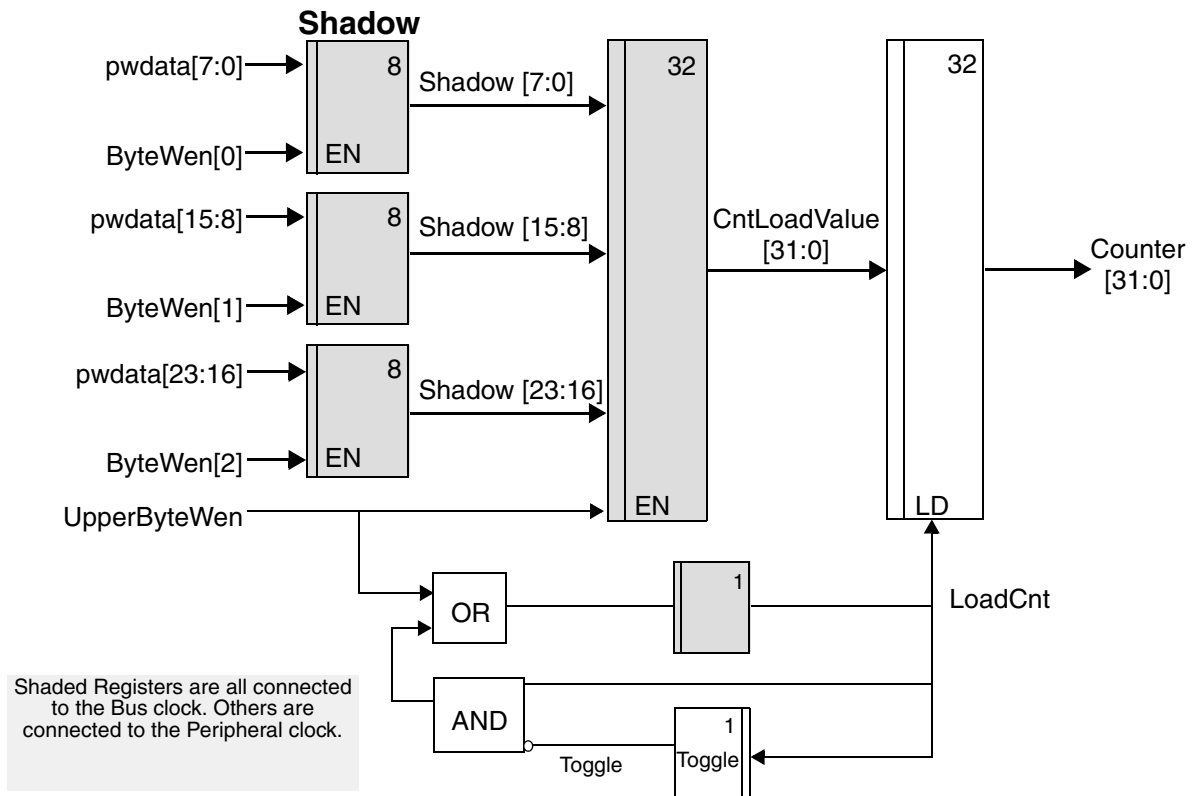
By using the shadow registers, the CntLoadValue is kept stable until it can be changed in one cycle. This allows the counter to be loaded in one access and the state of the counter is not affected by the latency in programming it. When there is a new value to be loaded into the counter initially, this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

8.4.1.2 Synchronous Clocks

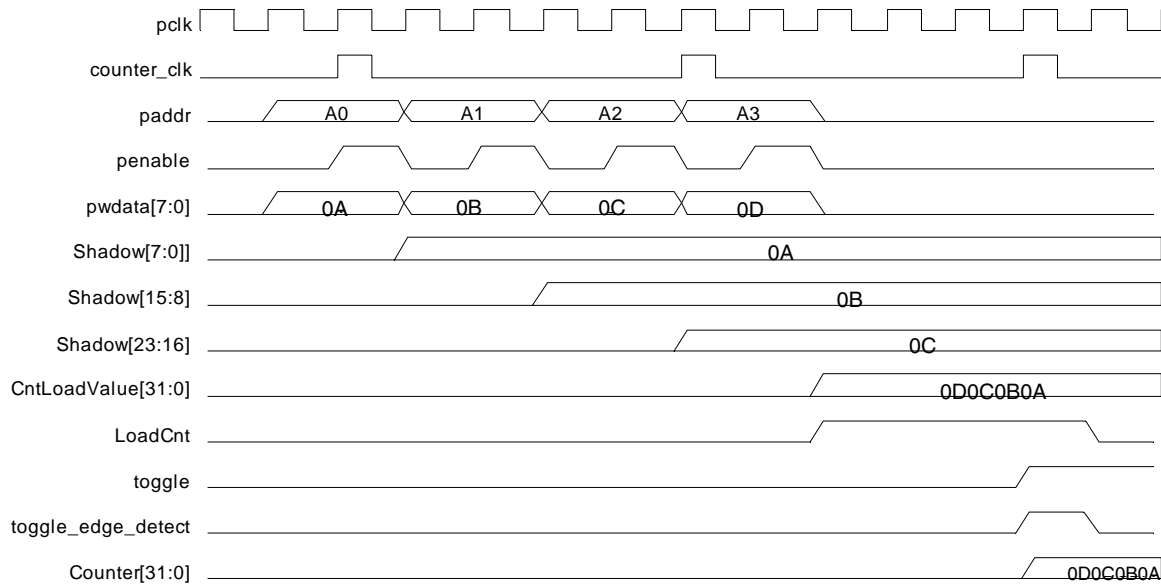
When the clocks are synchronous but do not have identical periods, the circuitry needs to be extended so that the LoadCnt signal is kept high until a rising edge of the counter clock occurs. This extension is necessary so that the value can be loaded, using LoadCnt, into the counter on the first counter 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.

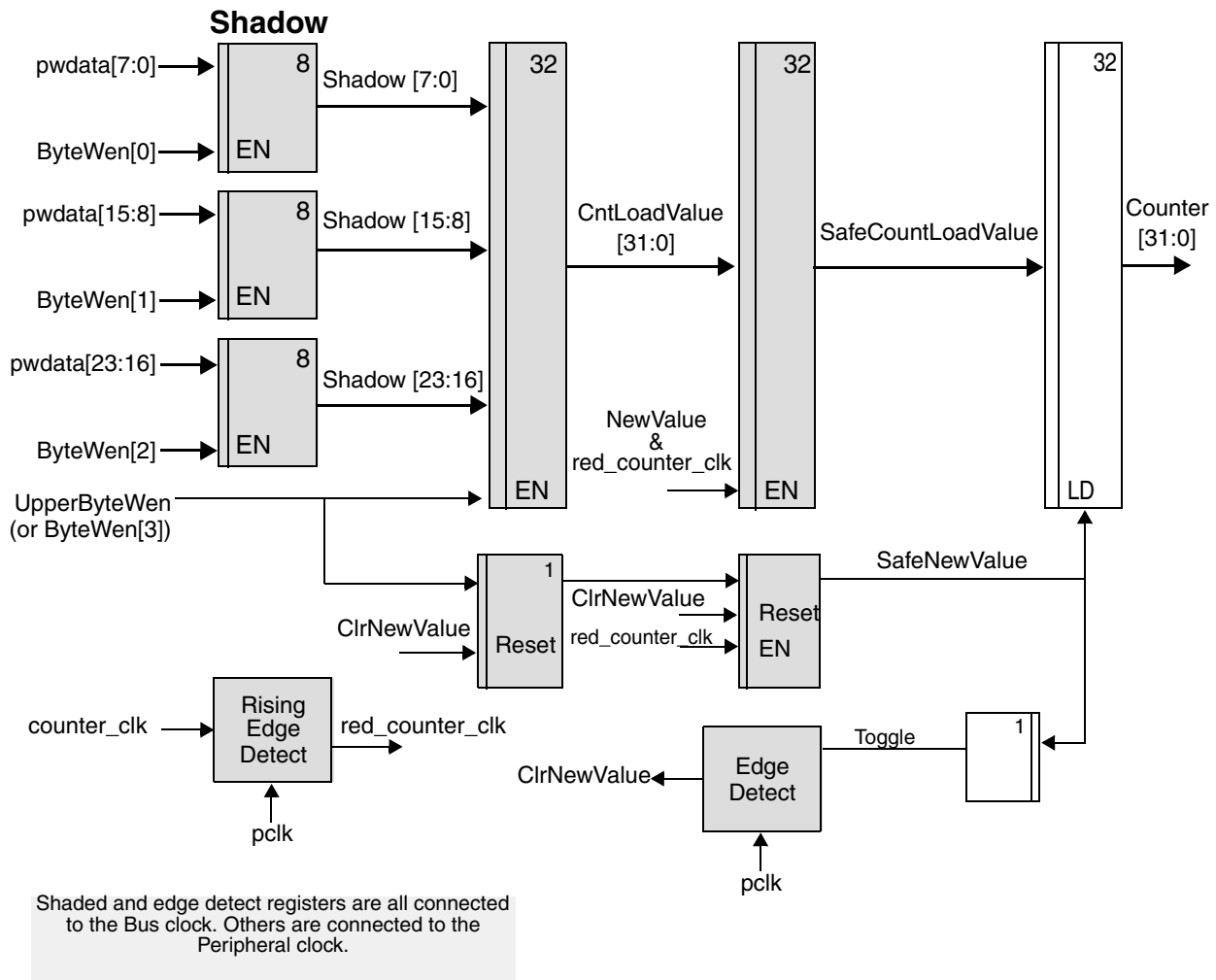
Figure 8-6 Coherent Loading – Synchronous Clocks



The following timing diagram shows 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 8-7 Coherent Loading – Synchronous Clocks**8.4.1.3 Asynchronous Clocks**

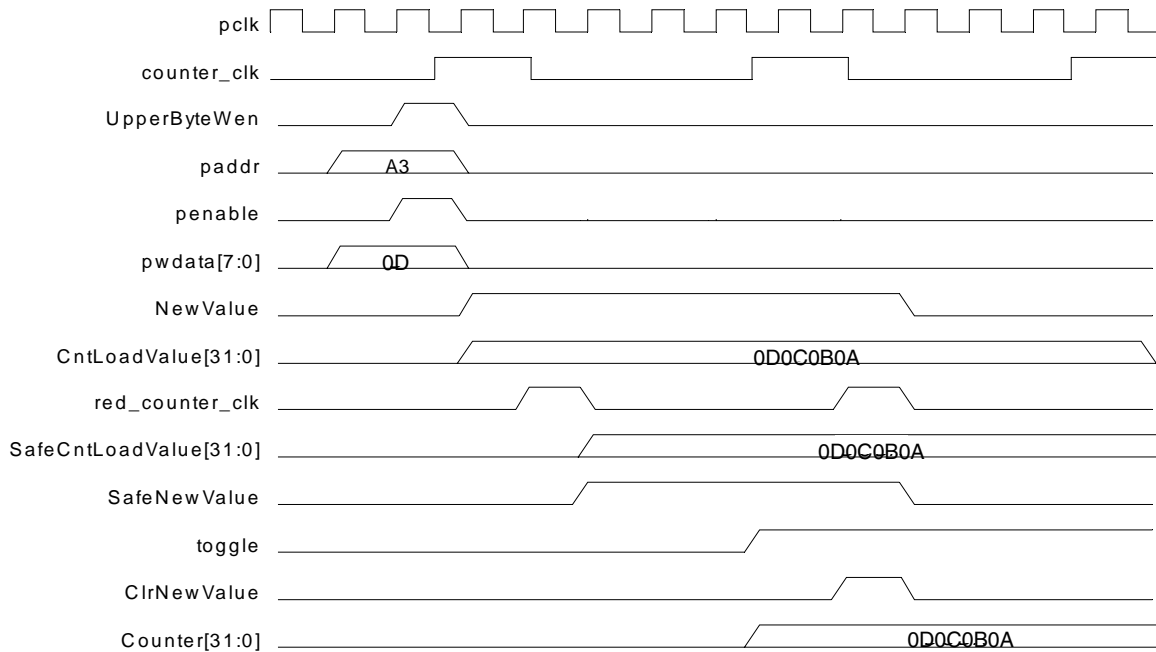
When the clocks are asynchronous, the processor clock needs to be three-times the speed of the peripheral clock for the re-timing to operate correctly. The high pulse time of the peripheral clock needs to be greater than the period of the processor clock. The following figure shows an RTL diagram for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

Figure 8-8 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 following timing diagram does not show the shadow registers being loaded. This is identical to the loading for the other clock modes. 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*.

Figure 8-9 Coherent Loading – Asynchronous Clocks

8.4.2 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. 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.

Table 8-2 Lower Byte Generation

Counter Register Width	Lower Byte Bus Width		
	8	16	32
1 - 8	NCR	NCR	NCR
9 - 16	0	NCR	NCR
17 - 24	0	0	NCR
25 - 32	0	0	NCR

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

- ❖ Identical and/or synchronous
- ❖ Asynchronous

8.4.2.1 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 timing diagram.

Figure 8-10 Coherent Registering – Synchronous Clocks

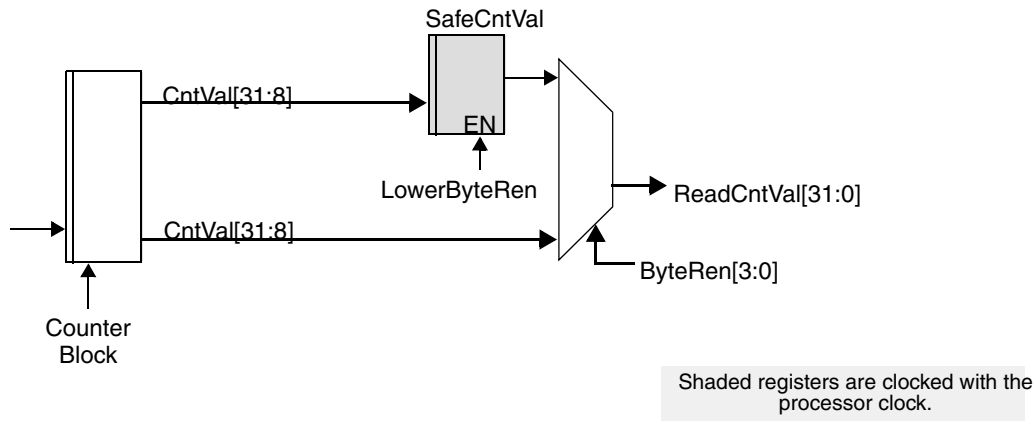
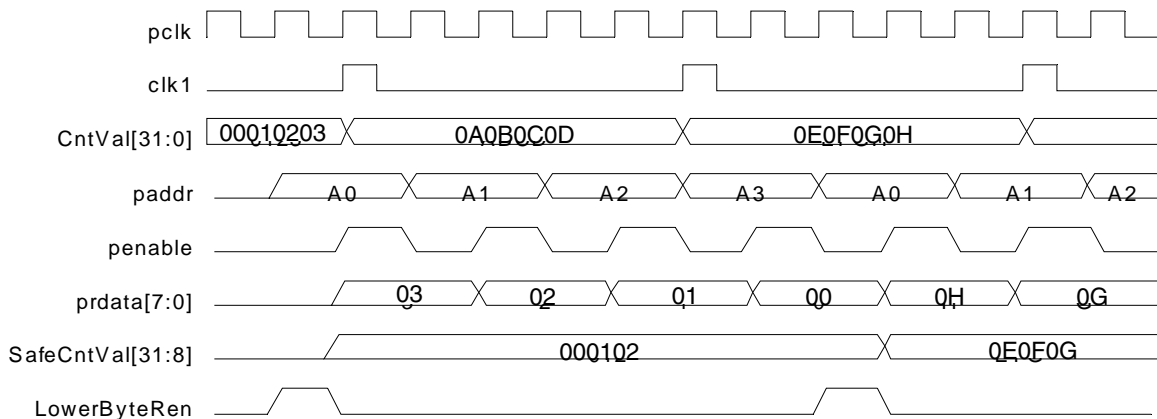


Figure 8-11 Coherent Registering – Synchronous Clocks



8.4.2.2 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.



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 and timing diagram illustrate the synchronization of the counter clock and the update of the shadow register.

Figure 8-12 Coherency and Shadow Registering – Asynchronous Clocks

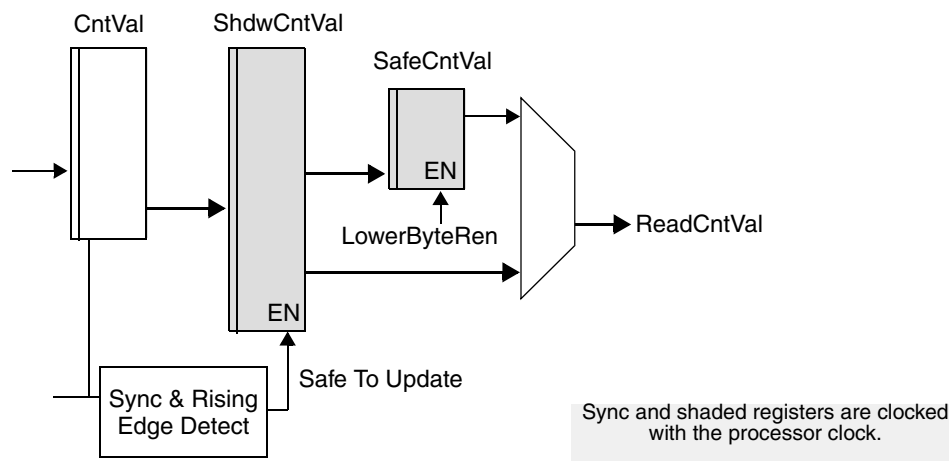


Figure 8-13 Transfer to Shadowing Registers– Asynchronous Clocks

