Peripheral Test Block

Revision: r0p0

Technical Reference Manual

ARM®

ARM DDI 0364A

Peripheral Test Block
Technical Reference Manual

Release Information

The table below shows the release state and change history of this document.

Change history

Date	Issue	Change
01 February 2005	A	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
Peripheral Test Block Technical Reference Manual

List of Tables
Peripheral Test Block Technical Reference Manual

List of Figures
Peripheral Test Block Technical Reference Manual

Preface

This preface introduces the *Peripheral Test Block Revision r0p0 Technical Reference Manual* (TRM). It contains the following sections:

- *About this manual* on page x
- *Feedback* on page xiv.

About this manual

This is the TRM for the *Peripheral Test Block* (PTB).

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this manual, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This manual is written for system designers, system integrators, and verification engineers who are implementing a *System-on-Chip* (SoC) device based on the Peripheral Test Block.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for a description of the PrimeCell types, an overview of the PTB and validation environments.

Chapter 2 *Functional Overview*

Read this chapter for a description of the major functional blocks of the PTB and functional operation, including FIFO options for usage models.

Chapter 3 *Programmer's Model*

Read this chapter for a description of the PTB registers and programming details.

Glossary Read the Glossary for definitions of terms used in this manual.

Conventions

This section describes the conventions that this manual uses:

- *Typographical* on page xi
- *Timing diagrams* on page xi
- *Signal naming* on page xii
- *Numbering* on page xiii.

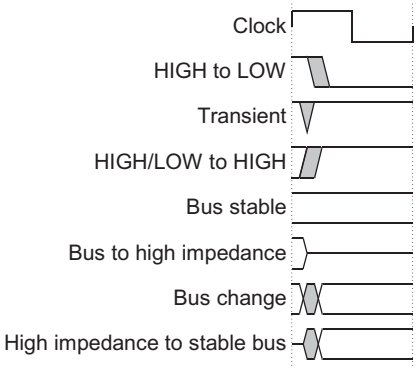
Typographical

This manual uses the following typographical conventions:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	denotes language keywords when used outside example code.
< and >	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example: • `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>` • The Opcode_2 value selects which register is accessed.

Timing diagrams

This manual contains one or more timing diagrams. The figure named *Key to timing diagram conventions* on page xii on page xii explains the components used in these diagrams. When variations occur they have clear labels. You must not assume any timing information that is not explicit in the diagrams.



Key to timing diagram conventions

Signal naming

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals:

Prefix A Denotes *Advanced eXtensible Interface* (AXI) global and address channel signals.

Prefix B Denotes AXI write response channel signals.

Prefix C Denotes AXI low-power interface signals.

Prefix H Denotes *Advanced High-performance Bus* (AHB) signals.

Prefix n Denotes active-LOW signals except in the case of AHB or *Advanced Peripheral Bus* (APB) reset signals. These are named **HRESETn** and **PRESETn** respectively.

Prefix P Denotes an APB signal.

Prefix R Denotes AXI read channel signals.

Prefix W Denotes AXI write channel signals.

Numbering

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

This section lists publications by ARM Limited, and by third parties.

ARM Limited periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the ARM Limited Frequently Asked Questions list.

ARM publications

This manual contains information that is specific to the Peripheral Test Block subsystem. See the following documents for other relevant information:

- *AMBA® Specification (Rev 2.0)* (ARM IHI 0011)
- *PrimeXsys® System Verification Software Framework User Guide* (ARM DUI 0296)
- *Verification Framework API Software User Guide (ARM* DUI 0297).

Feedback

ARM Limited welcomes feedback on the Peripheral Test Block subsystem and its documentation.

Feedback on the product

If you have any comments or suggestions about this product, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this manual

If you have any comments on this manual, send email to `errata@arm.com` giving:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM Limited also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the PTB. With many different validation methodologies available and the complexity of validating the PrimeCells, either stand alone or within a SoC, it is important to have a test block which is reusable across different validation environments. This reduces development time scales because a block developed for one environment can be re-used across the others.

This chapter contains the following sections:

1.1 PrimeCell types

Peripherals can be split into four groups related to the method they use to transfer data. These are:

1. *Sequential Data Peripherals* (SDP).

   These devices transfer the data sequentially, for example, a single word, byte, nibble at a time. The format of the sequential data is not defined for the transfer larger than the entity size. Peripherals of this type include UARTs and GPIO.

2. *Sequential Block Peripherals* (SBP).

   These devices transfer the data sequentially a block of data at a time, for example, multiple words, bytes, nibbles. The format of the data in the block is defined for the transfer and the order in which the blocks are transferred is fixed. Peripherals of this type include video controllers and network interfaces, for example Ethernet, and *Universal Serial Bus* (USB).

3. *Random Data Peripherals* (RDP).

   This group is used to describe memory-like peripherals, where data elements can be accessed in any order. For these devices, a memory model is normally used to give the required validation.

4. *Random Block Peripherals* (RBP).

   These devices transfer the data a block at a time. Unlike SBP the blocks can be accessed randomly, but the data within the block is sequentially read out, the format of the data in the block is defined. Peripherals of this type include *Serial Peripheral Interface* (SPI), Memory Stick, *Secure Digital* (SD)-Card and *MultiMedia Card* (MMC).

The PTB is aimed at the sequential data peripherals, sequential block peripherals and random block peripherals.

1.2 PTB overview

The PTB is part of a methodology for creating a standard test interface for all peripherals within a SoC. It enables the re-use of the block between different validation environments and gives a generic register structure for the verification/validation engineer. The block sits between the peripheral I/O signals, *Direct Memory Access* (DMA) interface, interrupt request signals and the validation environment.

The PTB is placed in the validation environment as shown in Figure 1-1.

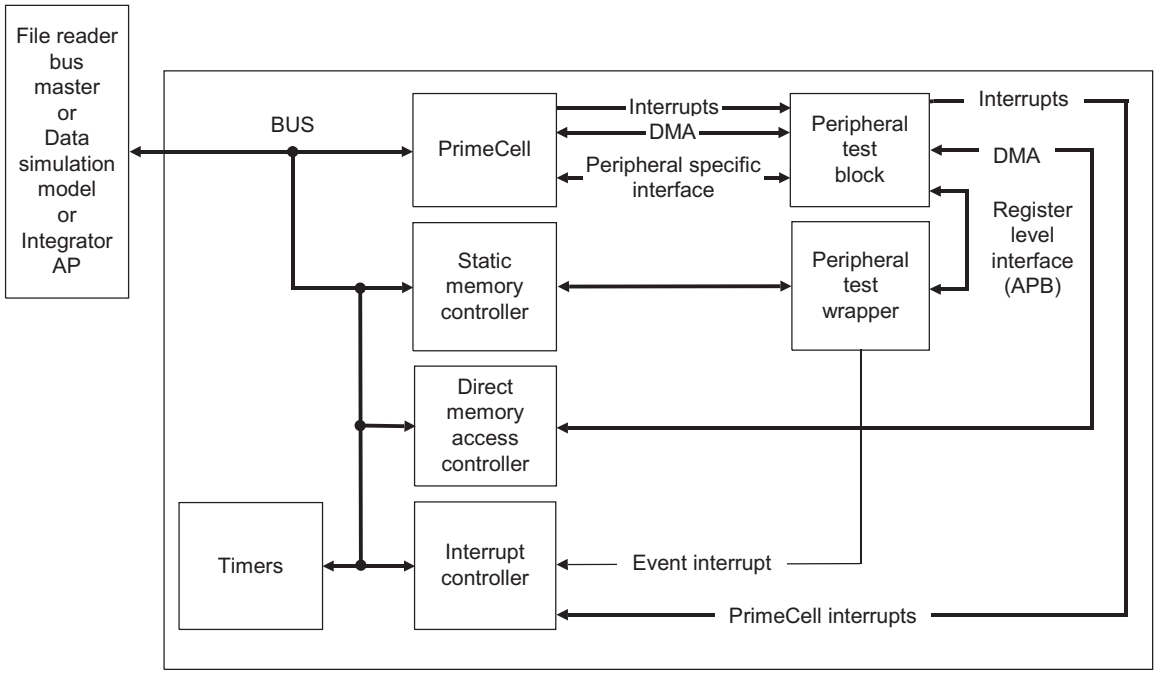


Figure 1-1 PTB integration overview

If the environment contains the *Direct Memory Access Controller* (DMAC) and *Interrupt Controller* (IC) then the PTB can act as a pass through for these signals and perform a monitoring only function. If these components are not present then the PTB can generate events on these signals, enabling emulation of the IC and DMAC for validating these interfaces on the PrimeCell.

The *Peripheral Test Interface Bus* (PTIB) contained within the *Peripheral Test Wrapper* (PTW) enables a common interface using the AMBA APB 2.0 interface specification to either:

- an asynchronous memory bus, through the Static Memory Controller (SMC), or

- an AHB/APB/AXI interface through the *AMBA Development Kit* (ADK) bridges.

The PTB has an address space of 4KB in the basic register level interface. Using the PTIB it occupies 4 words of memory space. This is useful if address space is an issue, especially in static memory mapped implementations.

The SMC asynchronous memory interface is mainly for platform level validation. It enables the PTB to be placed externally to the platform.

1.3 Validation environments

This section provides an overview of the validation environments that the PTB can be implemented in. It consists of:

- *Directed, integration and system environments*
- *XVC environment* on page 1-6
- *Platform* on page 1-8.

1.3.1 Directed, integration and system environments

The directed, integration and system environments are based on the ADK EASY_FRBM. They consist of a single layer bus structure with timers, DMAC and IC. Because this is a fixed architecture, the memory mapping has been altered to reflect the Integrator AP memory map. This enables the same environment to be simulated with an *File Reader Bus Master* (FRBM), DSM model or placed into an FPGA for testing in the Integrator AP environment.

Figure 1-2 shows the Directed, integration and system environments. All the components within the PTB box are common components between all three environments. This enables migration and debugging of tests to occur between environments with minimal overhead.

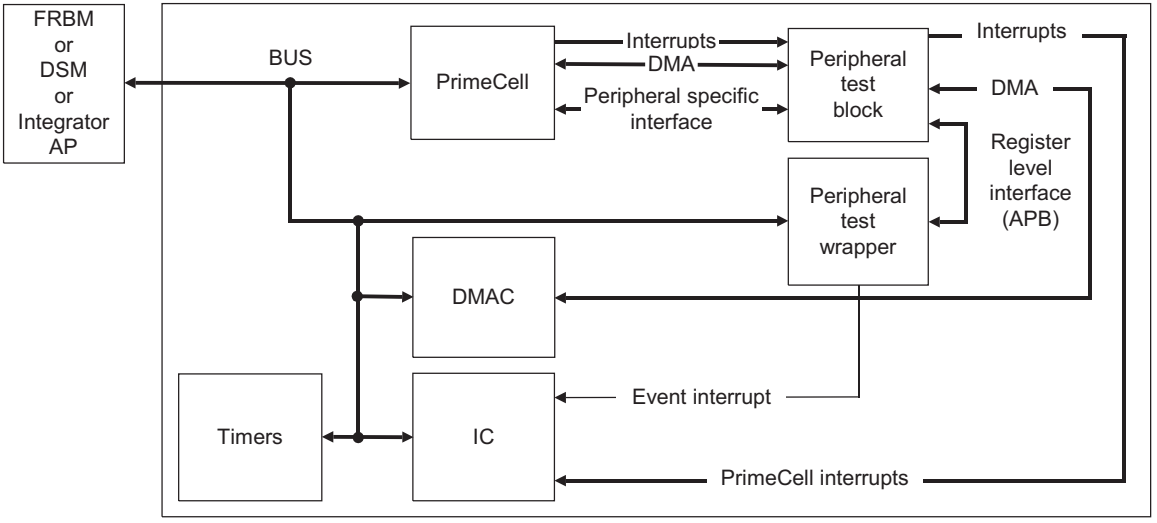


Figure 1-2 Directed, integration and system environments

The PTB contains the RTL code required to perform the basic tests. It also contains behavioral protocol checkers and property checking. These are both non synthesizable.

DMAC handling

If you do not require the DMAC, or if it is not present, the FRBM can poll the PTB event register waiting for the DMA request going active, through the **DMACSREQ**, **DMACBREQ**, **DMACLSREQ** and **DMACLBREQ** signals, and then mimic the DMAC operations by performing transfers to/from the PrimeCell. After completing the DMA transfers it then writes to the PTB to clear the DMAC request signals. This drives **DMACCLR** and **DMACTC**.

Interrupt handling

If you do not require the Interrupt Controller, or it is not present, then the FRBM can use the PTB for interrupt monitoring. The FRBM requires only to poll the PTB event register waiting for an interrupt to occur, it then continues clearing down the interrupt by accessing the PrimeCell and finally checks that the interrupt is cleared by polling the PTB event register.

1.3.2 XVC environment

Simulation performance degradation using Specman is related to the large number of system calls required to directly drive signals at the *Hardware Design Language* (HDL) level. To overcome this, and increase simulation performance, the *eXtensible Verification Component* (XVC) passes over transactions, and functions such as write word, using one system call. This enables the PTB to translate this into its operation cycle based signal driving.

Figure 1-3 shows the XVC environment.

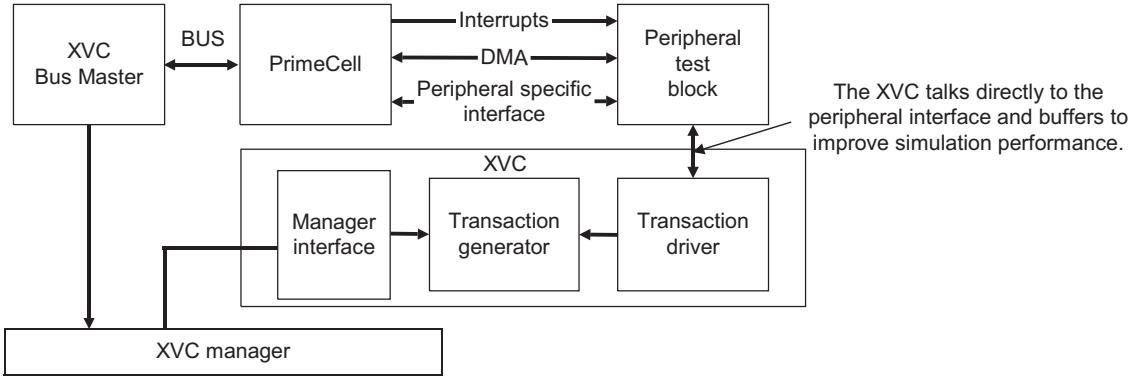


Figure 1-3 XVC environment

The XVC can contain the PTW, depending on requirements, but for performance reasons directly interfaces to the *Peripheral Test Interface* (PTI), within the PTB, with the translator. The XVC performs all high level data formatting and advanced test functions with the PTB generating the basic communications to the PrimeCell.

Although the XVC communicates directly to the PTI the actual PTB performs the same way as in the other environments. This enables the reuse of the PTB across environments, improving the debugging capability.

The XVC master is capable of recording FRBM transactions, that can then be replayed on the FRBM released environment.

The XVC environment is based on the internal development environment, using the same memory map and components. The Figure 1-3 on page 1-6 does not show this relationship, it concentrates on the way the XVC links to the PTB because this interconnect is very different from the other environments.

Specman e and SystemC specific implementations

Specman e and SystemC languages have the ability to probe signals at any point within the design, they also have the ability to directly manipulate signals, registers and arrays. Use this feature to optimize the performance. The overhead in the system calls required by these languages, means optimal performance is achieved if the translator communicates directly with the signals at the PI and buffers. This feature configures all the registers, and fills memories in a single system call.

Figure 1-4 on page 1-8 shows the Specman/SystemC environment.

Communicating directly to the PI and buffers, enables the Specman/SystemC environment to receive a single call from the simulator to update all registers, from the event bus. This procedure consumes zero simulation time and minimizes the call overhead. This also bypasses the PTI and register address mapping giving best performance for Specman e.

This scenario requires the use of consistent naming for the signals to and from the buffers and PI, together with a well defined timing interface that can accept both the PTI and translator interface. You cannot run the Specman/SystemC environments in any clock domain, but they can be run in an event/transaction driven environment. Only monitor event signals that are changing infrequently. This minimizes the system call overhead, and implies the monitoring of the event signals and the FIFO status. You must also ensure that the monitored signals are glitch free.

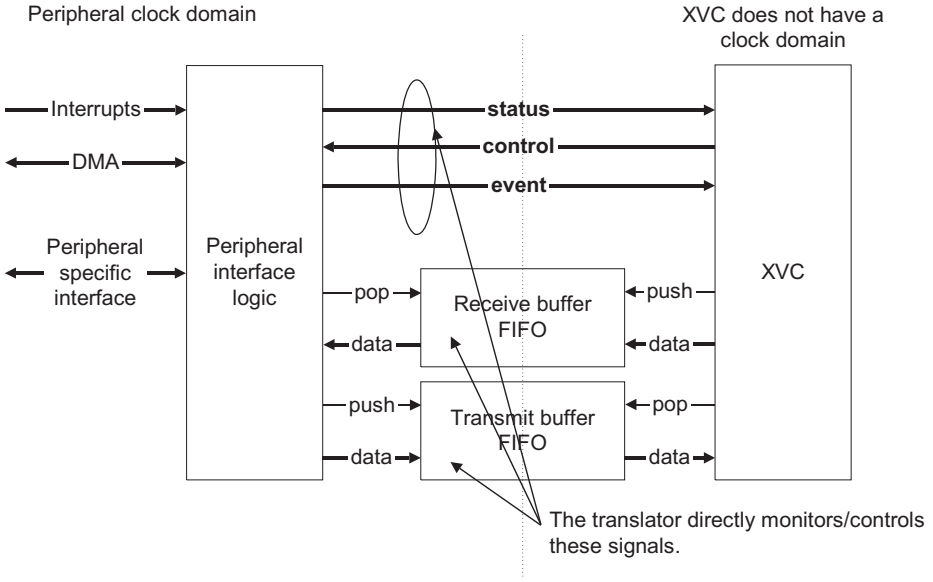


Figure 1-4 Specman/SystemC environment

The **status**, **control** and **event** signals are busses with the following naming:

- **status[n:0]**, where **n** is the width of the bus
- **control[m:0]**, where **m** is the width of the bus
- **event[31:0]**, for the event bus.

The assignment of signal functions in the busses is implementation specific.

The read and write buffers are implemented as arrays, these are PTB specific in width and depth, and are directly accessible by Specman and SystemC.

1.3.3 Platform

The PTB is designed to operate in this environment to prove the functionality of the PrimeCell at a system level using a real processor and system level components.

It is desirable for the PTB to provide loop-back functionality to assist with in-system testing. This enables testing to be performed without having to access the control interface on the PTB, and is more representative of real operation of the peripheral.There can be peripherals where the PTB functionality is replaced by an external peripheral, and autonomous operation of the PTB can mimic this.

Figure 1-5 on page 1-9 shows the platform environment.

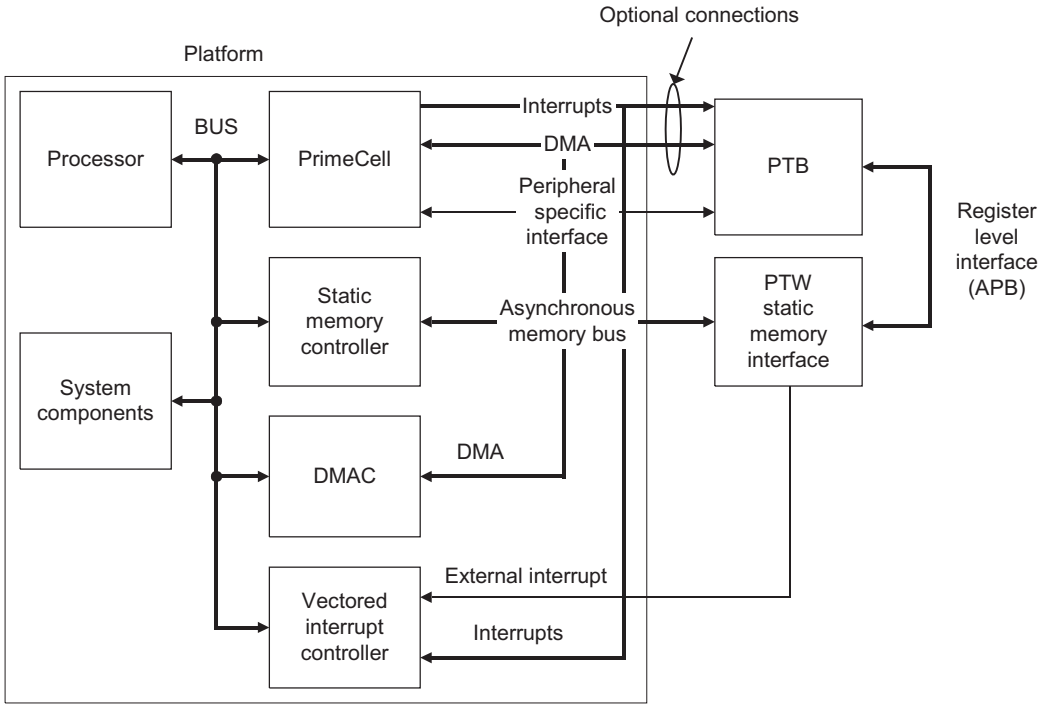


Figure 1-5 Platform environment

In a system environment, it is necessary to instantiate the PTB external to the system being tested. In this environment, there might be no means for direct connection from the processor running tests and the PTB. Therefore, you must re-use an existing interface on the system to provide control of the PTB.

By using the *Static Memory Controller* (SMC) as the interface through the PTW to the PTB, you can perform system level testing without modifications to the internal structure of the SoC platform. The SMC is the interface to the PTB for all data transfers. You can map the PTB externally by additional decode logic that can sit in unused SMC memory area. This interface models simple asynchronous SRAM.

The small memory footprint of the PTB interface is aimed specifically at this platform requirement, because many PTBs can sit in a single SMC area above the normal SRAM memory.

Any events generated by the PTB are signalled through the external interrupt input of the platform. If an external interrupt is not available then you can use the platform *General Purpose Input/Output* (GPIO) with the input pin configured to raise an interrupt on the event signal.

The peripheral DMA and interrupt signals are wired in the system, so if you require the PTB, use it in a monitoring only mode.

Chapter 2
Functional Overview

This chapter provides a functional description and operation of the PTB. It contains the following sections:

2.1 Functional description

The PTB methodology permits multiple PrimeCells, each with its own PTB within the SoC. Each PTB has its own internal structure, but share a common configuration and data interface to the validation environment.

Figure 2-1 shows the system level implementation. See also *Peripheral test wrapper* on page 2-10.

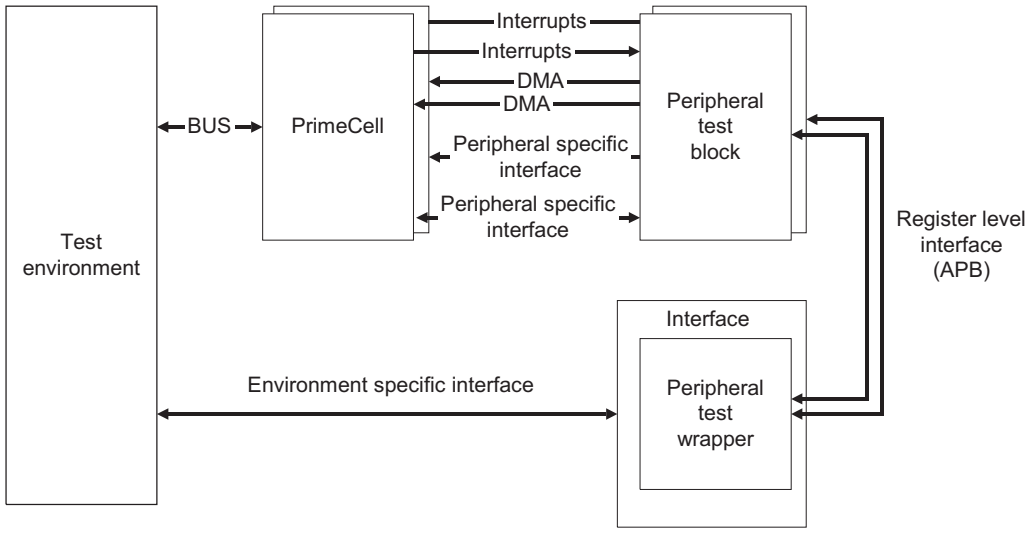


Figure 2-1 System level implementation

The PTB is a basic test block performing simple standalone functions such as loop back at initialization and protocol checking in simulation mode. You can select additional modes, for example, through the PTI data capture. If you require complex functions they can be implemented in the peripheral test wrapper or within other parts of the test environment. The complex tests can include error generation or model removal of the device, for example, if the peripheral is a removable card such as the *MultiMedia Card Interface* (MMCI).

2.2 Functional operation

This section describes:

- *PTB internal structure*
- *Transferring data between clock domains* on page 2-6
- *XVC data transfer* on page 2-8
- *Peripheral test wrapper* on page 2-10
- *Register level interface* on page 2-12
- *Peripheral test interface bus* on page 2-12
- *FIFO options for usage models* on page 2-15.

2.2.1 PTB internal structure

Figure 2-2 shows the PTB internal structure.

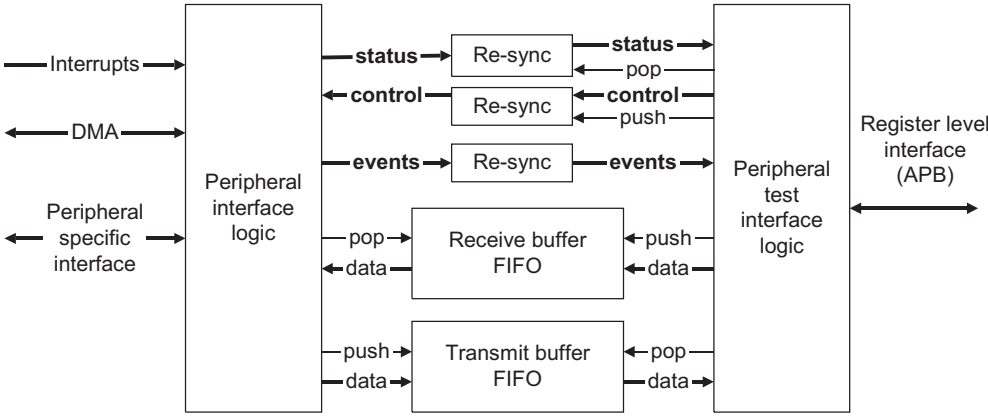


Figure 2-2 PTB internal structure

The PTB consists of four elements:

- The *Peripheral Interface Logic* (PI). This performs the data formatting and translation for a peripheral specific interface. It also performs protocol checking and monitoring/control of the interrupts/DMA signals, through the control/status/event signals.
- The buffers for holding the data transfers between the PI and the test environment, through the PTI.
- The PTI to present a consistent interface between different peripherals and to the test environment.

- The Re-sync blocks synchronize data transfers between the two clock domains.

This section describes:

- *Peripheral interface logic (PI)*
- *Buffers*
- *Peripheral test interface logic* on page 2-5
- *Internal signals* on page 2-5.

Peripheral interface logic (PI)

The PI logic works in the peripheral clock domain and performs data translation for the transfer of data at the peripheral specific interface. Protocol checking of this data and associated signals are performed by formal methods and HDL behavior. It also controls the transfer of data for the buffers.

Buffers

The buffers can be implemented as a FIFO or a dual ported RAM depending on the peripheral requirements. Each buffer is up to 8K bits wide, split into 32 bit registers, and the two interfaces of each buffer, to the PI or PTI, are in their respective clock domain.

The PTI interface has the ability to monitor the buffer status and reset the FIFO levels, or set the dual ported RAM address pointers.

When a dual ported RAM receive buffer implementation is used, you are required to select the option of filling the RAM with an image from the PTI. By setting the PI receive mode to compare, the image in the RAM is compared with the incoming data and an event is raised if a mismatch occurs or when the image has been read completely.

If you require buffers wider than 32 bits, then the most significant register is used to transfer the data into or alternatively to read the next value from the FIFO. For an example, a procedure for a read operation to a 96 bit register would be:

1. Read least significant register first then read incrementing registers to build up the data. The last register signals that the FIFO read is complete and the next FIFO value is ready for reading.
2. Write least significant register up to most significant register.

   Note

   The write to the most significant register transfers data from the holding registers into the FIFO/RAM.

Peripheral test interface logic

The PTI logic interfaces with the Register Level Interface APB bus to the internal registers enabling access to the buffers, **control** and **status** buses as word wide registers. It re-synchronizes signals from the PI logic, that is, the **status** and **even**t signals.

Internal signals

These consist of:

- *Status bus*
- *Control bus*
- *Event bus*.

Status bus

The **status** signals are outputs from the PI logic to show the internal status of the PI and the signal levels at the peripheral specific interface, for example UART signals DTR, RTS. The signals are generated in the peripheral clock domain and require re-synchronizing to the PTI clock domain. You can determine the width of the status bus by the number of status signals required. The width is peripheral specific, that is, up to 8K bits.

Control bus

The **control** signals are inputs to the PI logic. Their functionality is defined on a peripheral basis but can include setting of output signals, for example UART signals DSR, CTS. The control signals are in the PTI clock domain and require re-synchronizing to the peripheral clock domain. You can determine the width of the control bus by the number of control signals required. This is peripheral specific, and up to 8K bits.

Event bus

The **event** signals are outputs from the peripheral PI logic to signal that a condition has occurred requiring immediate attention. The number of events is limited to 32. The signals are generated in the peripheral clock domain and require re-synchronizing to the PTI clock domain.

2.2.2 Transferring data between clock domains

This section consists of:

- *Control, from PTI to PI*
- *Status, from PI to PTI*
- *Events, from PI to PTI*
- *Read buffer, from PI to PTI* on page 2-7
- *Write buffer, from PTI to PI* on page 2-7.

Note

To synchronize multiple control and status accesses, they are gated by writes to the Xfer Register. The same register is used to check that the information has been passed from one clock domain to the next.

Control, from PTI to PI

You can access the control registers as 32 bit wide registers from the PTI. You can write to any of these registers in any order. The contents of all the 32 bit wide control registers are transferred in a single cycle onto the control bus when the Xfer Register commit bit is set. Polling the commit bit for a 1 indicates when the transfer from the PTI to PI has completed and the PI has the new values in its registers.

Status, from PI to PTI

You can access the status registers as 32 bit wide registers from the PTI. You can read any of these registers in any order. The contents of all the 32 bit wide status registers are loaded in a single cycle from the status bus when the Xfer Register sample bit is set. Polling the sample bit for a 1 indicates when the transfer from PI to PTI has completed and registers have valid data.

Events, from PI to PTI

The event signals are re-synchronized into the PTI clock domain by double buffering logic. Each event is a single signal so no specific requirement is placed on making sure some signals do not change before others.

If you require transfers of information that are greater than a single bit, then place the information on the **status bus** and generate an event signal to indicate that information is available. When the event is processed the PTI can read the status bus, defined in *Status, from PI to PTI* to recover the required information.

Read buffer, from PI to PTI

The read buffer is able to support asynchronous transfers between the two clock domains, as either a dual ported RAM or FIFO.

In a FIFO implementation the status of the RdFifoEmpty flag indicates if data is ready for reading. You can use the RdFifoTide flag to indicate that a burst transfer of reads can occur. The tide level is set to the number of words before the FIFO is empty.

In a *Dual Ported RAM* (DPRAM) implementation it is the responsibility of the PTI to ensure that the data written is not going to overwrite the reading location. You can achieve this by stopping the PI from writing to the DPRAM when the PTI wants to read the data, although this could cause issues for streaming data. Alternatively, you can split the DPRAM into multiple blocks and generate an event when a block is filled. This procedure enables the PTI to read this block by monitoring the RdFifoTide flag, while the PI moves onto filling the next block. By reading the RdBufPIAddr Register, it is possible to see which block is being filled. The RdBufPTIAddr Register holds the present reading location, and therefore the reading block. From this, if an event has been missed, it is possible to determine by how many blocks the reading is lagging the writing. The RdBufControl Register bits RdBlockSize set the size of the block in words to generate the RdFifoTide flag event when that size is crossed. This size is a power of 2 and can be up to 64K Words.

Write buffer, from PTI to PI

The write buffer is able to support asynchronous transfers between the two clock domains as either a dual ported RAM or FIFO. In a FIFO implementation the status of the WrFifoFull flag indicates if space is available for writing. The WrFifoTide flag can be used to indicate that a burst transfer of data can occur. Here the tide level is set to the number of words before the FIFO is full.

In a *Dual Ported RAM* (DPRAM) implementation, it is the responsibility of the PTI to ensure that the data written is not going to overwrite the reading location. You can achieve this by stopping the PI from reading from the DPRAM when the PTI wants to write the data, although this could cause issues for streaming data. Alternatively, you can split the DPRAM into multiple blocks and generate an event when a block is fully read. This procedure enables the PTI to write to this block again by monitoring the WrFifoTide flag. By reading the WrBufPIAddr Register it is possible to see which block is being read. The WrBufPTIAddr Register holds the present writing location and therefore the writing block. From this, if an event has been missed, it is possible to determine by how many blocks the writing is leading the reading. The WrBufControl Register bits WrBlockSize set the size of the block in words. The PTI logic generates the WrFifoTide flag event when this size is crossed. This size is a power of two, and can be up to 64K Words.

2.2.3 XVC data transfer

Figure 2-3 shows the XVC data transfer.

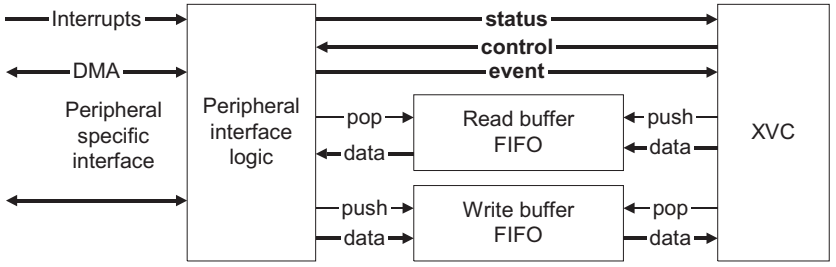


Figure 2-3 XVC data transfer

XVC transfers are:

- *Control, from XVC to PI*
- *Status, from PI to XVC*
- *Events, from PI to XVC* on page 2-9
- *Read buffer, from PI to XVC* on page 2-9
- *Write buffer, from XVC to PI* on page 2-9.

Control, from XVC to PI

The control bus is directly driven from the XVC. The XVC can modify all bits of the control bus in a single access. This means that control signalling is simpler than it is for the PTI interface. Because the signals are all set in zero time, the commit bit is not required in the Xfer Register.

The difference between the PTI and XVC in this situation is that the data is not written in the PTI clock domain, so the XVC must be aware of a time difference between the PTI and XVC values being placed on the control bus.

Status, from PI to XVC

The status bus is directly read by the XVC. It can read the whole bus width in zero time. This removes the requirement to sample the status bus into registers before reading and therefore the sample bit in the Xfer Register is not required.

The difference between the PTI and XVC in this situation is that the data is read in the PI clock domain, the PTI reads data in the PTI clock domain, so the XVC must be aware of a time difference between the PTI and XVC values.

Events, from PI to XVC

The XVC directly monitors these signals. The XVC must use these signals to monitor activity to determine when to perform transfers.

The XVC does not require re-synchronizing of the event signals into the PTI clock domain so the events can been seen at an earlier time compared to the PTI.

Read buffer, from PI to XVC

In a FIFO implementation the status of the RdFifoEmpty flag indicates if data is ready for reading, see also *Read Buffer Status Register* on page 3-11. You can use the RdFifoTide flag to indicate that a burst transfer of reads can occur, by setting the tide level to the number of words before the FIFO is empty. These two signals are defined events on the event bus. You can use them to enable the XVC to read the FIFO contents, either as a single read or reads from multiple FIFO registers. The XVC can directly read the contents of the registers, or RAMs, and then set the RdFifoRdPtr flag to the RdFifoWrPtr value, effectively setting the FIFO to empty, or increment the RdFifoRdPtr flag the number of reads performed.

In a DPRAM implementation then it is the responsibility of the XVC to ensure that the new data written is not going to overwrite the reading location. Because the XVC can access the memory in zero time this is not an issue. The RdFifoTide flag event is monitored by the XVC to determine when it can read data from the DPRAM. You can use the block size to generate an event when the block is full, and ready for reading. The RdBufControl register bits RdBlockSize sets the size of the block in words to generate the RdFifoTide event when that size is crossed, this size is a power of 2, and can be up to 64K Words.

Write buffer, from XVC to PI

In a FIFO implementation the status of the WrFifoFull flag indicates if space is available for writing, see also *Write Buffer Status Register* on page 3-14. You can use the WrFifoTide flag to indicate that a burst transfer write can occur, by setting tide level to the number of words before the FIFO is full. These two signals are defined events on the event bus and can be used to enable the XVC to write to the FIFO, either as a single write or writes to multiple FIFO registers. The XVC can directly write the contents of the registers, or RAMs it then sets the WrFifoWrPtr flag to the WrFifoRdPtr flag value, effectively setting the FIFO to full, or increments the WrFifoWrPtr flag by the number of writes performed.

In a DPRAM implementation then it is the responsibility of the XVC to ensure that the new data written is not going to overwrite the reading location. Because the XVC can access the memory in zero time this is not a problem. The WrFifoTide flag event is monitored by the XVC to determine when it can write data to the DPRAM. You can use

the block size to generate an event when the block is empty and ready for writing. The WrBufControl Register bits WrBlockSize sets the size of the block in words to generate the WrFifoTide event when that size is crossed. This size is a power of 2 and can be up to 64k Words.

2.2.4 Peripheral test wrapper

The PTW converts an arbitrary interface to an APB interface. It acts as an interface translator between the register level interface and the validation environment interface, that is, APB, AHB, AXI or memory. If you require the PTBs to occupy a very small memory area it also contains the *Peripheral Test Interface Bus* (PTIB).

For AXI and AHB interfaces you can use the *Advanced Development Kit* (ADK) bridges to generate the APB interface to directly drive the register level interface, or the PTIB. If you require an externally memory mapped option, then use a static memory to APB bridge. This is part of the PTB environment deliverables.

This section describes:

- *PTIB mapped*
- *Direct mapped* on page 2-11.

PTIB mapped

Figure 2-4 shows the implementation of the wrapper with the PTIB implementation. It translates the four word address space into the 1K word address space of the register level interface, see *Peripheral test interface bus* on page 2-12. A single **PSEL** signal is used for each PTB and directly maps to the **PSEL** at the register level interface.

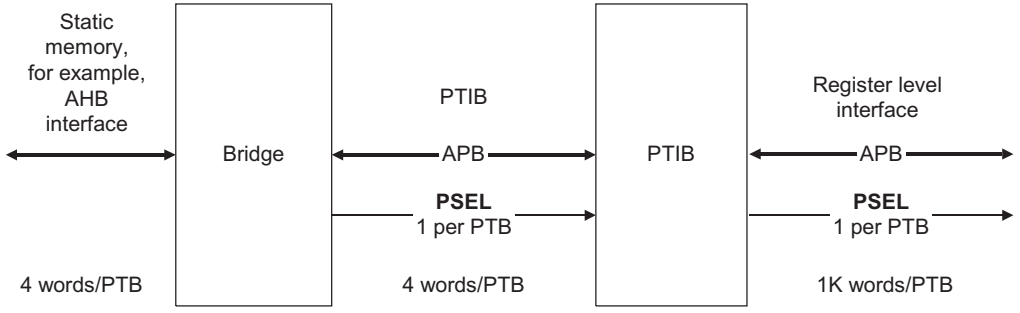


Figure 2-4 PTIB mapped peripheral test wrapper

Direct mapped

Figure 2-5 shows the implementation when the register level interface is mapped directly onto the validation environment interface bus. In this scenario the PTB occupies 1K words of the validation environment interface bus. You can remove the wrapper completely in an APB based validation environment, and connect the PTB to the system APB with one PSEL for each PTB.

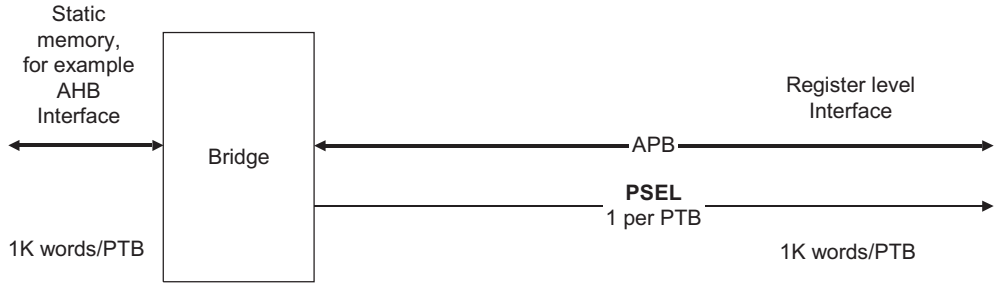


Figure 2-5 Direct mapped peripheral test wrapper

2.2.5 Register level interface

Figure 2-6 shows APB timing. The register level interface occupies 4K bits of address space per PTB. This bus uses the same timing as the APB, see the AMBA 2.0 specification, and can interface directly to the APB. Access to the registers is by 32 bit word transfers only, all other accesses are not permitted. A single 32 bit wide write bus and a 32 bit wide read bus are implemented. The required register is selected by the address input, 10 bits wide **PADDR[12:2]**. Figure 2-6 shows the basic timing for the APB 2.0 bus, see the AMBA 2.0 Specification section APB for more details of this timing.

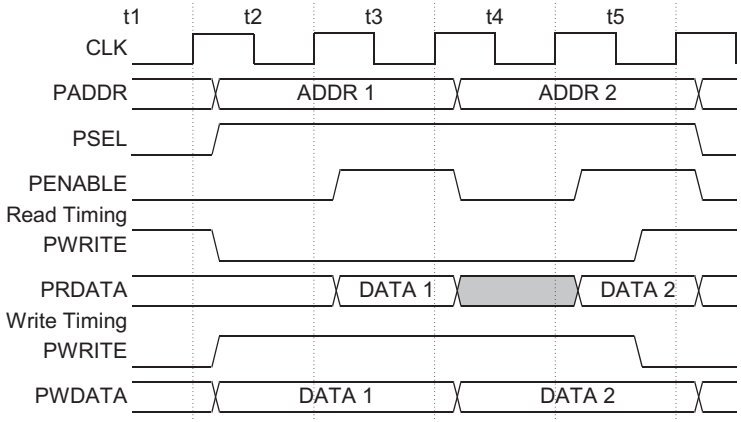


Figure 2-6 APB 2.0 timing

2.2.6 Peripheral test interface bus

The standard interface to the PTB requires 4K bytes, 1K words, of address space. When multiple PTBs exist in the system this could require an address space larger than available, for example, an external memory interface. In this situation you can use the PTIB to reduce the number of addressable registers to four for each PTB. This enables 256 PTBs to occupy the same address space as a single PTB, of size 4K bytes. This does add a level of indirection to the register accesses. You must handle this by a generic device driver software that can sit between the normal driver and the hardware for the PTB. In normal operation this drive is in pass through mode. In PTIB operation, it performs the address/data translation for the PTIB.

Each PTB occupies four words of memory. The device consists of four registers:

1. An event register, EVENT. You can read it to indicate if any trigger events have occurred.

2. An address register, ADDR which is written to select the required memory location for access, or read to determine which memory location is selected.

3. The non incrementing data register, SINGLE. It is read/write capable to enable access to the specified memory location referenced by ADDR. When you access this register, the ADDR Register is not modified.

4. The incrementing data register, INCR. It is read/write capable to enable access to the specified memory location referenced by ADDR. After the access the ADDR Register is incremented.

You can use the INCR Register to enable fast access for writing/reading large blocks of memory, such as the DPRAMS.

You can use the ADDR Register to enable the internal structure of the PTB to access the full 4K bytes of the PTB memory space while maintaining a small memory footprint to the outside.

The PTIB is an APB interface with a 32 bit data bus and associated command signals. It has a two cycle response:

- address and control set on first rising clock
- data written/read, on next rising clock

The device can perform single transfers or burst transfers to/from the PTB register interface. The registers are single transfer access only. Each PTB occupies four contiguous words of memory. All accesses are 32 bit wide only and unused addresses and bits within decoded addresses must be returned as zero.

PTIB timing

You can access any register within the PTB by writing the address required to offset `0x4`, the ADDR Register, and then reading or writing, at offset `0x8` for a non-incrementing access or at `0xC` for post incrementing accesses. Reading the ADDR Register gives the address being pointed to for the next data access.

Address `0x00` is a read only register returning the status of the masked events register.

Transfers must assert the **PSEL** signal active for the required PTB otherwise the transfer is ignored. This structure enables multiple PTBs to be placed alongside one another in a small area of memory, 256 PTBs use 4K of address space, therefore they can sit in a small area of the memory map of the static/dynamic memory controller.

Figure 2-7 on page 2-14 shows non-incrementing read/write timing. For a non-incrementing transfer the required register address is first written to offset `0x004` ADDR Register at time t2-t3, data for this register is then read or written at time t4-t5.

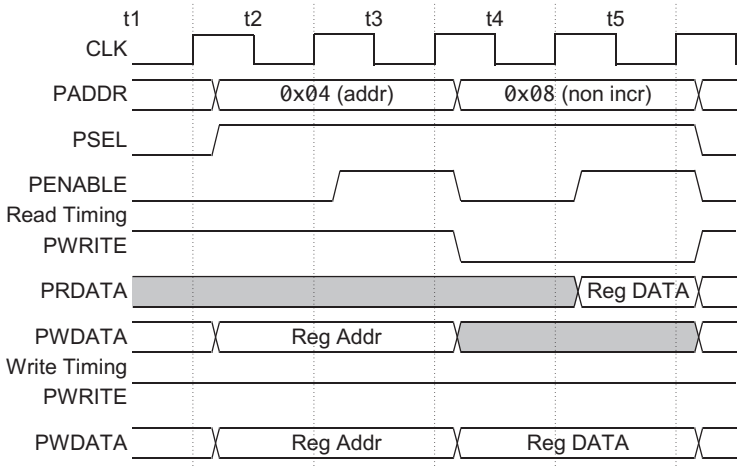


Figure 2-7 Non-incrementing read/write timing

Figure 2-8 shows incrementing read/write. For an incrementing transfer the required register address is first written to offset `0x004` ADDR Register, at time t2-t3. At time t4-t5 data for this register is then read or written and the ADDR Register is incremented. This enables a read or write at time t6-t7 to the next adjacent register.

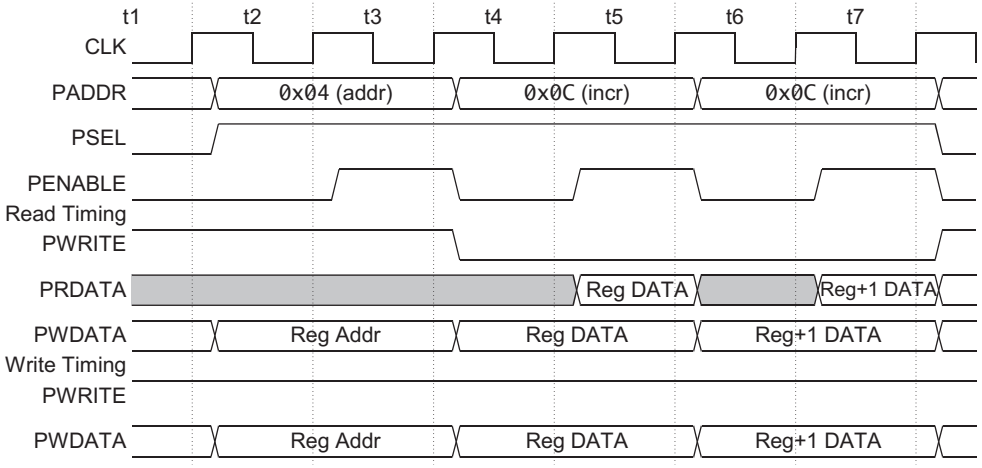


Figure 2-8 Incrementing read/write timing

Table 2-1 lists the PTIB address map.

Table 2-1 PTIB address map

Offset	Mode	Name	Description
0x0	R	EVENT	Output of the masked events register.
0x4	R/W	ADDR	Address of register to access, points to present address to be accessed in incrementing mode.
0x8	R/W	SINGLE	Single address access, non incrementing.
0xC	R/W	INCR	Post incrementing address access.

2.2.7 FIFO options for usage models

The width and depth of the FIFO is not fixed in the PTB. It can have any width from 1 to 2048 bits and any depth as long as it is a power of two. The choice of width and depth is application specific, however in general the width is defined by the amount of data required at the peripheral interface in a single clock cycle.

This section describes:

- *FIFO width*
- *Continuous data streams* on page 2-16
- *High data rate burst streams* on page 2-16
- *Example FIFO configurations* on page 2-16.

FIFO width

Because the accesses to the FIFO are limited to 32 bit words then it takes the same amount of time at the PTI to read or write a bit wide, byte wide or word wide FIFO. Therefore for optimum performance at the PTI interface a FIFO width of less than a word is in efficient and must only be used for very low data rate applications, but ideally not used at all.

FIFOs larger than a word have the same level of performance as a word wide FIFO, because the number of accesses to write the data is related to the number of word writes at the PTI. For example, a 66 bit wide packet with a 66 bit wide FIFO requires three word accesses for a packet transfer because the PTI interface is 32 bits wide. If the FIFO were 32 bits wide then it still requires three accesses to transfer the packet. The width of the FIFO in this case is determined by the rate the data requirements reading at the PI.

It is sometimes best for synchronization and tracing to keep the FIFO width equal to the packet width and enable the depth to determine the level of buffering required.

Continuous data streams

For a continuous data stream application, for example *Color Liquid Crystal Display* (CLCD), the latency in responding to a request to fill/empty the FIFO defines the depth of the FIFO. You must assume that the FIFO can be filled/emptied faster than it can be emptied/filled at the peripheral end.

High data rate burst streams

Because some applications have a large block of data with a high burst transfer rate, that is greater than the ability to fill/empty the FIFO from the PTI, but a low average data rate, then it would be beneficial for the size of the FIFO to match the size of the data packet. You can define the depth of the FIFO by the data packet size divided by the width of the FIFO. The minimum width is defined as the amount of data required at the PI interface in a single clock cycle.

Example FIFO configurations

If you require an application that supplies six packets of data with a data rate of 120 bits per clock cycle, then Figure 2-9 shows a typical example that holds six data packets in the FIFO.

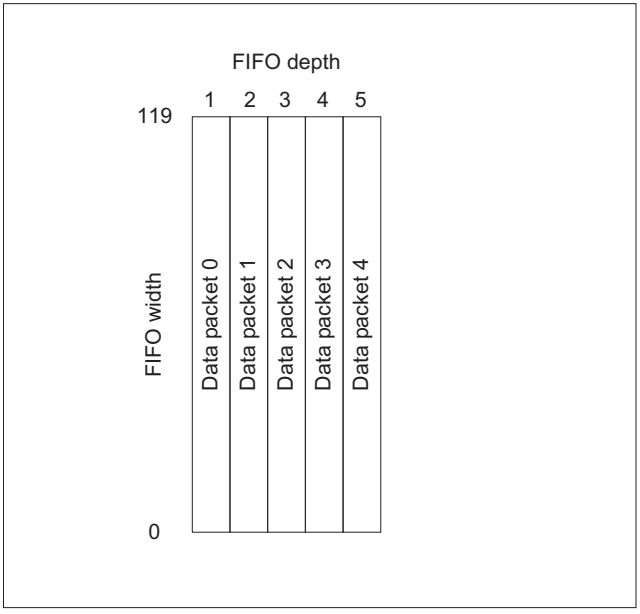


Figure 2-9 Example FIFO configuration 1

If the data rate is halved to 60 bits per clock cycle then the width of the FIFO can be reduced and the depth increased, with the packet spread across two FIFO registers, as shown in Figure 2-10.

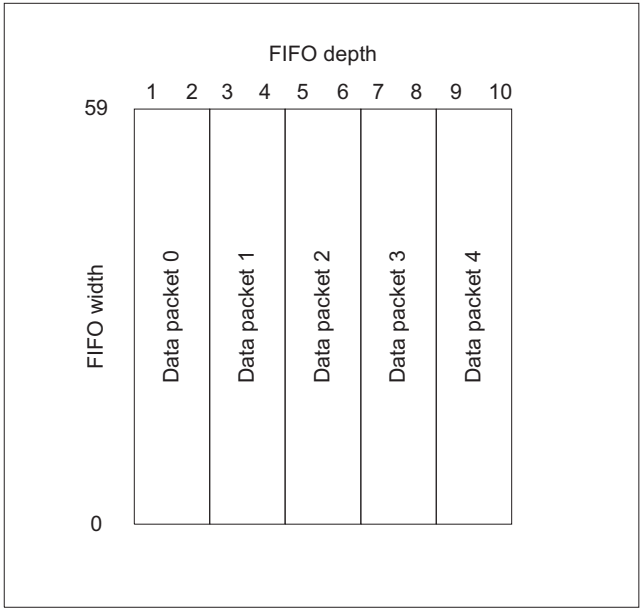


Figure 2-10 Example FIFO configuration 2

Chapter 3
Programmer's Model

This chapter describes the registers of the PTB and provides details for programming the device. It contains the following sections:

3.1 About the programmer's model

The following applies to the registers used in the PTB:

- The base address of the PTB is not fixed, and can be different for any particular system implementation. However, the offset of any particular register from the base address is fixed.
- Reserved or unused address locations must not be accessed because this can result in unpredictable behavior of the device.
- Reserved or unused bits of registers must be written as zero, and ignored on read unless otherwise stated in the relevant text.
- All register bits are reset to a logic 0 by a system or power-on reset unless otherwise stated in the relevant text.
- All registers support read and write accesses unless otherwise stated in the relevant text. A write updates the contents of a register and a read returns the contents of the register.

3.2 Summary of registers

All register addresses in the PTB are fixed relative to the base address. Table 3-1 lists the registers in base offset order:

Table 3-1 Summary of Peripheral Test Block registers

Name	Base offset	Type	Reset value	Description
EventMaskedStatus	0x000	RO	0x00000000	See *Event Masked Status Register* on page 3-6
EventMask	0x004	R/W	0x00000000	See *Event Mask Register* on page 3-6
EventRawStatus	0x008	RO	0x00000000	See *Raw Event Status Register* on page 3-7
EventClear	0x00C	R/W	0x00000000	See *Event Clear Register* on page 3-8
DmaclntrControl	0x010	R/W	0x00000000	See *Direct Memory Access Controller Register* on page 3-8
Xfer	0x014	R/W	0x00000000	See *Xfer Register* on page 3-9
ctr10	0x100	R/W	0x00000000	See *Control0 Register* on page 3-10
ctrl1	0x104	R/W	0x00000000	See *Control1 Register* on page 3-10
Sts0	0x200	RO	0x00000000	See *Status0 Register* on page 3-10
Sts1	0x204	RO	0x00000000	See *Status1 Register* on page 3-11
RdBufStatus	0x300	RO	0x00000000	See *Read Buffer Status Register* on page 3-11
RdBufControl	0x304	R/W	0x00000000	See *Read Buffer Control Register* on page 3-12
RdBufPTIStAddr	0x308	R/W	0x00000000	See *Read Buffer PTI Start Address Register* on page 3-13
RdBufPIStAddr	0x30C	R/W	0x00000000	See *Read Buffer PI Start Address Register* on page 3-13
RdBufPTIAddr	0x310	RO	0x00000000	See *Read Buffer PTI Present Address Register* on page 3-13
RdBufPIAddr	0x314	RO	0x00000000	See *Read Buffer PI Start Address Register* on page 3-14
WrBufStatus	0x400	RO	0x00000000	See *Write Buffer Status Register* on page 3-14

Table 3-1 Summary of Peripheral Test Block registers (continued)

Name	Base offset	Type	Reset value	Description
WrBufControl	0x404	R/W	0x00000000	See *Write Buffer Control Register* on page 3-15
WrBufPTIStAddr	0x408	R/W	0x00000000	See *Write Buffer PTI Start Address Register* on page 3-16
WrBufPIStAddr	0x40C	R/W	0x00000000	See *Write Buffer PI Start Address Register* on page 3-16
WrBufPTIAddr	0x410	RO	0x00000000	See *Write Buffer PTI Present Address Register* on page 3-16
WrBufPIAddr	0x414	RO	0x00000000	See *Write Buffer PI Start Address Register* on page 3-17
RdBufData0	0x500	RO	0x00000000	See *Read Buffer Data0 Register* on page 3-17
RdBufData1	0x504	RO	0x00000000	See *Read Buffer Data1 Register* on page 3-17
WrBufData0	0x600	WO	0x00000000	See *Write Buffer Data0 Register* on page 3-18
WrBufData1	0x604	WO	0x00000000	See *Write Buffer Data1 Register* on page 3-18
PeriphID4	0xFD0	RO	0x00000000	See *Peripheral Identification Register 4* on page 3-22
PeriphID5	0xFD4	RO	0x00000000	See *Peripheral Identification Register 5* on page 3-22
PeriphID6	0xFD8	RO	0x00000000	See *Peripheral Identification Register 6* on page 3-23
PeriphID7	0xFDC	RO	0x00000080	See *Peripheral Identification Register 7* on page 3-23
PeriphID0	0xFE0	RO	0x00000000	See *Peripheral Identification Register 0* on page 3-20
PeriphID1	0xFE4	RO	0x00000010	See *Peripheral Identification Register 1* on page 3-21
PeriphID2	0xFE8	RO	0x00000004	See *Peripheral Identification Register 3* on page 3-22
PeriphID3	0xFEC	RO	0x00000000	See *Peripheral Identification Register 3* on page 3-22
CompID0	0xFF0	RO	0x0000000D	See *Component Identification Register 0* on page 3-25

Table 3-1 Summary of Peripheral Test Block registers (continued)

Name	Base offset	Type	Reset value	Description
CompID1	0xFF4	RO	0x000000F0	See *Component Identification Register 1* on page 3-25
CompID2	0xFF8	RO	0x00000005	See *Peripheral Identification Register 2* on page 3-21
CompID3	0xFFC	RO	0x000000B1	See *Component Identification Register 3* on page 3-26

Note

If required, the address space from 0x0000 1000 to 0xFFFF is available for the mapping of a *Dual Ported RAM* (DPRAM). This can be accessed by using the single or incrementing mode.

3.3 Register descriptions

This section describes the Peripheral Test Block registers. Table 3-1 on page 3-3 provides cross references to individual registers.

3.3.1 Event Masked Status Register

The EventMaskedStatus Register is read-only. On a read, it returns the status from your user defined event from the peripheral interface logic, or the status of the corresponding current masked buffer event. Figure 3-1 shows the register bit assignments.

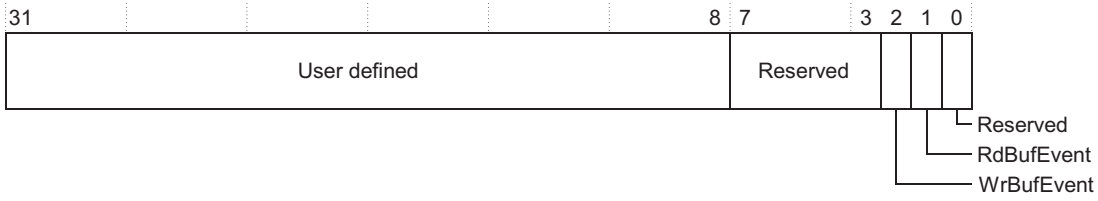


Figure 3-1 EventMaskedStatus Register bit assignments

Table 3-2 lists the register bit assignments.

Table 3-2 EventMaskedStatus Register bit assignments

Bit	Name	Function
[31:8]	-	User defined.
[7:3]	-	Reserved, read undefined.
[2]	WrBufEvent	Masked ORed events from write buffer status flags.
[1]	RdBufEvent	Masked ORed events from read buffer status flags.
[0]	-	Reserved, read undefined.

3.3.2 Event Mask Register

The EventMask Register is read and write. On a read, it returns the status from your user defined event from the peripheral interface logic, or the status of the mask events read buffer status flags. On a write it changes the status of user defined events in the peripheral logic, or the write buffer events mask status flags. Figure 3-2 on page 3-7 shows the register bit assignments.

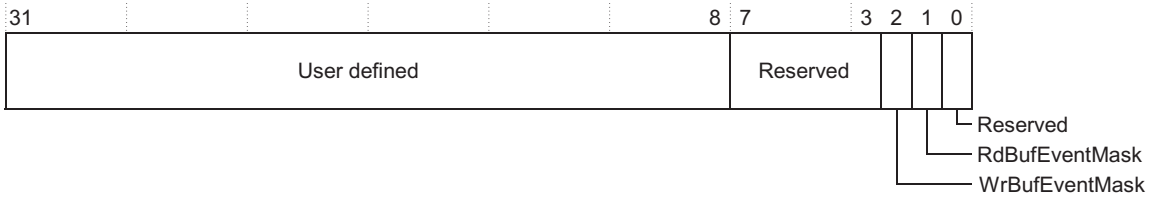


Figure 3-2 EventMask Register bit assignments

Table 3-3 lists the register bit assignments.

Table 3-3 EventMask Register bit assignments

Bit	Name	Function
[31:8]	-	User defined.
[7:3]	-	Reserved, read undefined, write as zero.
[2]	WrBufEventMask	Masked ORed events from write buffer status flags.
[1]	RdBufEventMask	Masked ORed events from read buffer status flags.
[0]	-	Reserved, read undefined, write as zero.

3.3.3 Raw Event Status Register

The EventRawStatus Register is read-only. On a read, it returns the status from your user defined event from the peripheral interface logic, or the unmasked events status from the corresponding read or write buffer. Figure 3-3 shows the register bit assignments.

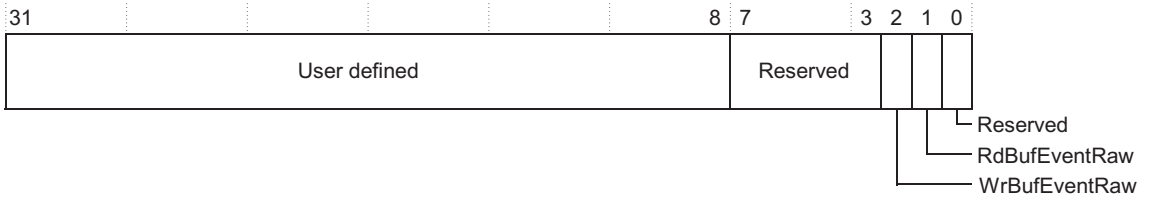


Figure 3-3 EventRawStatus Register bit assignments

Table 3-4 lists the register bit assignments.

Table 3-4 EventRawStatus Register bit assignments

Bit	Name	Function
[31:8]	-	User defined.
[7:3]	-	Reserved, read undefined.
[2]	WrBufEventRaw	ORed events from write buffer status flags.
[1]	RdBufEventRaw	ORed events from read buffer status flags.
[0]	-	Reserved, read undefined.

3.3.4 Event Clear Register

The EventClear Register is read and write. On a read, it returns the status from your user defined event from the peripheral interface logic. On a write all selected events in your user defined area are cleared. Figure 3-4 shows the register bit assignments.

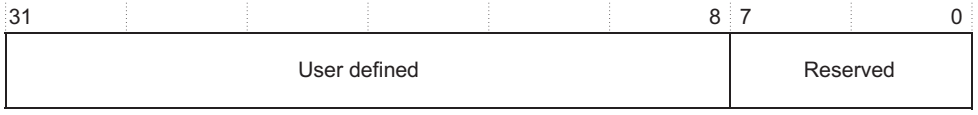


Figure 3-4 EventClear Register bit assignments

Table 3-5 lists the register bit assignments.

Table 3-5 EventClear Register bit assignments

Bit	Name	Function
[31:8]	-	User defined.
[7:0]	-	Reserved, read undefined, write as zero.

3.3.5 Direct Memory Access Controller Register

The DmaclntrControl Register is read and write. On a read, it returns the status from the corresponding direct memory flag. On a write the corresponding direct memory access flag is altered. Figure 3-5 on page 3-9 shows the register bit assignments.

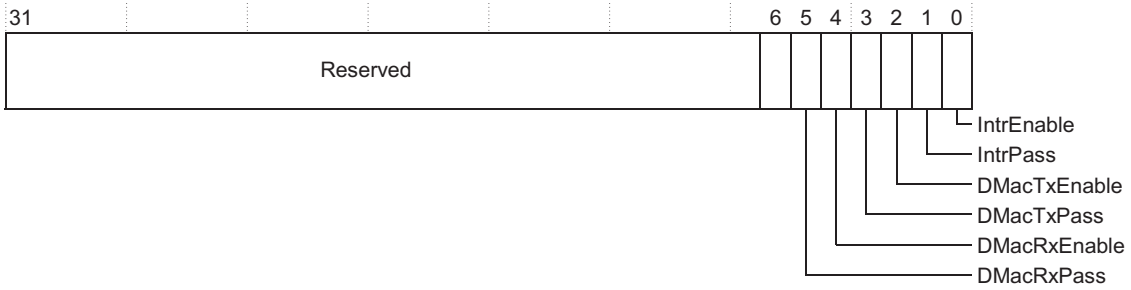


Figure 3-5 DmacIntrControl Register bit assignments

Table 3-6 lists the register bit assignments.

Table 3-6 DmacIntrControl Register bit assignments

Bit	Name	Function
[31:6]	-	Reserved, read undefined, write as zero.
[5]	DmacRxPass	Enable Dmac Rx pass through, route to external DMAC.
[4]	DmacRxEnable	Enable Dmac Rx emulation, emulate DMAC.
[3]	DmacTxPass	Enable Dmac Tx pass through, route to external DMAC.
[2]	DMacTxEnable	Enable Dmac Tx emulation, emulate DMAC.
[1]	IntrPass	Enable interrupt pass through, route to external interrupt controller.
[0]	IntrEnable	Enable interrupt emulation, emulate interrupt controller.

3.3.6 Xfer Register

The Xfer Register is read and write. On a read, the contents of the status bus are sampled into status registers. On a write, the contents of the control registers are emptied onto the control bus. Figure 3-6 shows the register bit assignments.

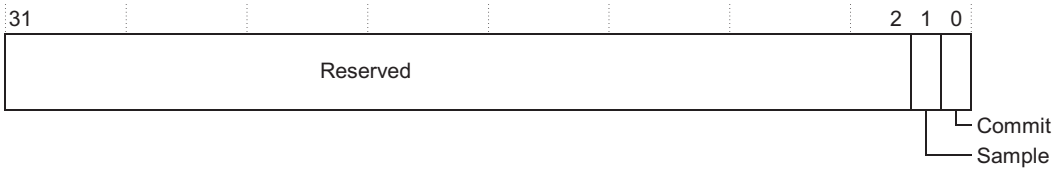


Figure 3-6 Xfer Register bit assignments

Table 3-7 lists the register bit assignments.

Table 3-7 Xfer Register bit assignments

Bit	Name	Function
[31:2]	-	Reserved, read undefined, write as zero.
[1]	Sample	Samples contents of status bus into status registers.
[0]	Commit	Commits contents of control registers onto control bus.

3.3.7 Control0 Register

The Ctrl0 Register is read and write. The contents are user defined.

Table 3-8 lists the register bit assignments.

Table 3-8 Ctrl0 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.8 Control1 Register

The Ctrl1 Register is read and write. The contents are user defined.

Table 3-9 lists the register bit assignments.

Table 3-9 Ctrl1 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined

3.3.9 Status0 Register

The Sts0 Register is read only. The contents are user defined.

Table 3-10 lists the register bit assignments.

Table 3-10 Sts0 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.10 Status1 Register

The Sts1 Register is read only. The contents are user defined.

Table 3-11 lists the register bit assignments.

Table 3-11 Sts1 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.11 Read Buffer Status Register

The RdBufStatus Register is read-only. On a read it returns the status of the corresponding buffer flag. Figure 3-7 shows the register bit assignments.

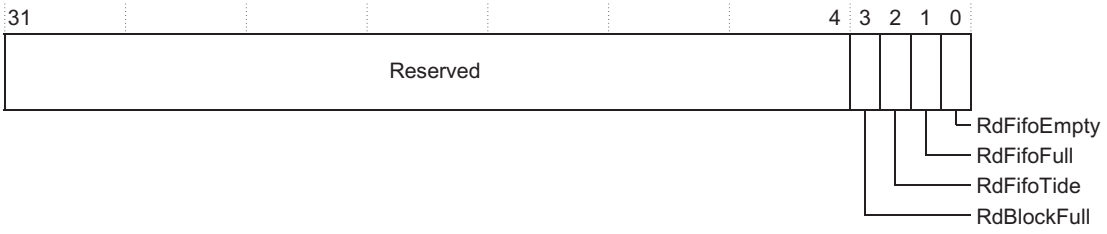


Figure 3-7 RdBufStatus Register bit assignments

Table 3-12 lists the register bit assignments.

Table 3-12 RdBufStatus Register bit assignments

Bit	Name	Function
[31:4]	-	Reserved, read undefined.
[3]	RdBlockFull	Read DPRAM block full flag.
[2]	RdFifoTide	Read FIFO over tide level flag.
[1]	RdFifoFull	Read FIFO full flag.
[0]	RdFifoEmpty	Read FIFO empty flag.

3.3.12 Read Buffer Control Register

The RdBufControl Register is read and write. On a read it returns the status of the corresponding read buffer control flags. On a write it changes the status of the corresponding read buffer control flags. Figure 3-8 shows the register bit assignments.

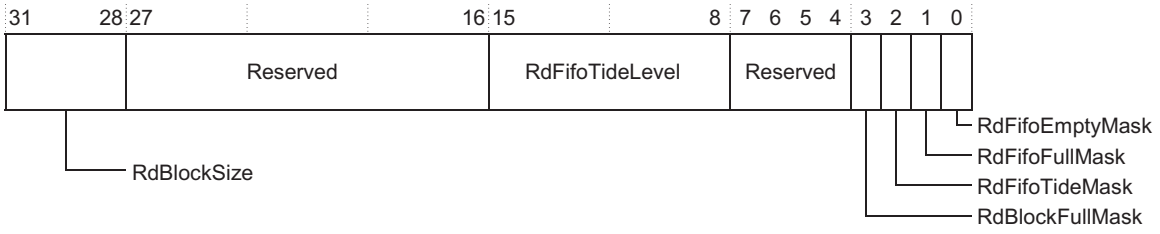


Figure 3-8 RdBufControl bit assignments

Table 3-13 lists the register bit assignments.

Table 3-13 RdBufControl Register bit assignments

Bit	Name	Function
[31:28]	RdBlockSize	Sets size of block in 32bit words, 2RdBlockSize 1 to 64K words.
[27:16]	-	Reserved, read undefined, write as zero.
[15:8]	RdFifoTideLevel	Sets tide level for read FIFO, depends on depth of FIFO.
[7:4]	-	Reserved, read undefined, write as zero.

Table 3-13 RdBufControl Register bit assignments (continued)

Bit	Name	Function
[3]	RdBlockFullMask	Read DPRAM block full flag mask to RdBufEvent bit.
[2]	RdFifoTideMask	Read FIFO over tide level flag mask to RdBufEvent bit.
[1]	RdFifoFullMask	Read FIFO full flag mask to RdBufEvent bit.
[0]	RdFifoEmptyMask	Read FIFO empty flag mask to RdBufEvent bit.

3.3.13 Read Buffer PTI Start Address Register

The RdBufPTIStAddr Register is read and write. This register is user defined.

Table 3-14 lists the register bit assignments.

Table 3-14 RdBufPTIStAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.14 Read Buffer PI Start Address Register

The RdBufPIStAddr Register is read and write. This register is user defined.

Table 3-15 lists the register bit assignments.

Table 3-15 RdBufPIStAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.15 Read Buffer PTI Present Address Register

The RdBufPTIAddr Register is read-only. This register is user defined.

Table 3-16 lists the register bit assignments.

Table 3-16 RdBufPTIAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.16 Read Buffer PI Start Address Register

The RdBufPIStAddr Register is read-only. This register is user defined.

Table 3-17 lists the register bit assignments.

Table 3-17 RdBufPIAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.17 Write Buffer Status Register

The WrBufStatus Register is read-only. On a read, it returns the current status of the corresponding write buffer FIFO flags. Figure 3-9 shows the register bit assignments.

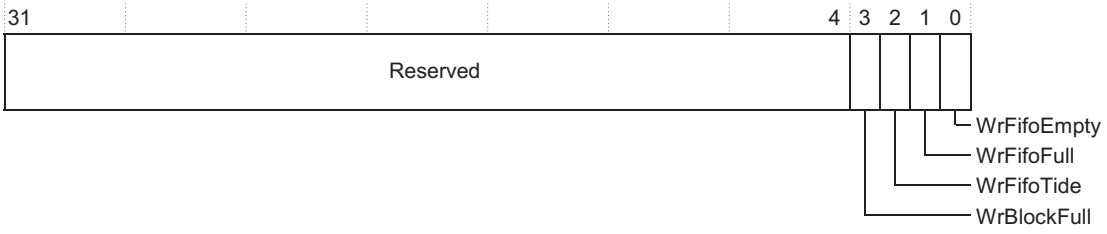


Figure 3-9 WrBufStatus Register bit assignments

Table 3-18 lists the register bit assignments.

Table 3-18 WrBufStatus Register bit assignments

Bit	Name	Function
[31:4]	-	Reserved, read undefined, write as zero.
[3]	WrBlockEmpty	Write DPRAM block empty.
[2]	WrFifoTide	Write FIFO below tide level flag.
[1]	WrFifoFull	Write FIFO full flag.
[0]	WrFifoEmpty	Write FIFO empty flag.

3.3.18 Write Buffer Control Register

The WrBufControl Register is read and write. On a read it returns the status of the corresponding write buffer control flags. On a write it changes the status of the corresponding write buffer control flags. Figure 3-10 shows the register bit assignments.

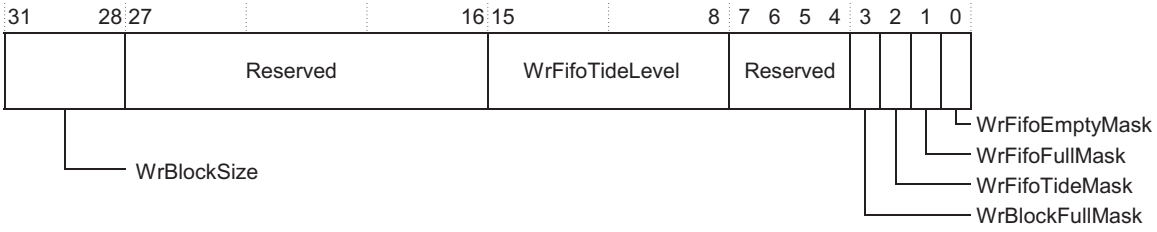


Figure 3-10 WrBufControl Register bit assignments

Table 3-19 lists the register bit assignments.

Table 3-19 WrBufControl Register bit assignments

Bit	Name	Function
[31:28]	WrBlockSize	Sets size of block in 32bit words, 2RdBlockSize 1 to 64K words.
[27:16]	-	Reserved, read undefined, write as zero.
[15:8]	WrFifoTideLevel	Sets tide level for read FIFO, depends on depth of FIFO.

Table 3-19 WrBufControl Register bit assignments (continued)

Bit	Name	Function
[7:4]	-	Reserved, read undefined, write as zero.
[3]	WrBlockFullMask	Write DPRAM block full flag mask to WrBufEvent bit.
[2]	WrFifoTideMask	Write FIFO over tide level flag mask to WrBufEvent bit.
[1]	WrFifoFullMask	Write FIFO full flag mask to WrBufEvent bit.
[0]	WrFifoEmptyMask	Write FIFO empty flag mask to WrBufEvent bit.

3.3.19 Write Buffer PTI Start Address Register

The WrBufPTIStAddr Register is read and write. This register is user defined.

Table 3-20 lists the register bit assignments.

Table 3-20 WrBufPTIStAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.20 Write Buffer PI Start Address Register

The WrBufPIStAddr Register is read and write. This register is user defined.

Table 3-15 on page 3-13 lists the register bit assignments.

Table 3-21 WrBufPIStAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.21 Write Buffer PTI Present Address Register

The WrBufPTIAddr Register is read-only. This register is user defined.

Table 3-16 on page 3-14 lists the register bit assignments.

Table 3-22 WrBufPTIAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.22 Write Buffer PI Start Address Register

The WrBufPIStAddr Register is read-only. This register is user defined.

Table 3-17 on page 3-14 lists the register bit assignments.

Table 3-23 WrBufPIAddr Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.23 Read Buffer Data0 Register

The RdBufData0 Register is read-only. This register is user defined.

Table 3-24 lists the register bit assignments.

Table 3-24 RdBufData0 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.24 Read Buffer Data1 Register

The RdBufData1 Register is read-only. This register is user defined.

Table 3-25 lists the register bit assignments.

Table 3-25 RdBufData1 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.25 Write Buffer Data0 Register

The WrBufData0 Register is write-only. This register is user defined.

Table 3-24 on page 3-17 lists the register bit assignments.

Table 3-26 WrBufData0 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.26 Write Buffer Data1 Register

The WrBufData1 Register is write-only. This register is user defined.

Table 3-25 on page 3-17 lists the register bit assignments.

Table 3-27 WrBufData1 Register bit assignments

Bit	Name	Function
[31:0]	-	User defined.

3.3.27 Peripheral Identification Registers

The Peripheral Identification Registers are eight, 8-bit read-only registers. They span two address locations:

- PeriphID0-3 Registers span address locations `0xFE0-0xFEC`
- PeriphID4-7 Registers span address locations `0xFD0-0xFDC`.

Each of these blocks of registers 0-3 and 4-7 can conceptually be treated as one 32-bit read-only register. The PeriphID0-3 Registers provide the peripheral options listed in Table 3-28.

Table 3-28 Peripheral Identification Register options, PeriphID0-3

Bits	Description
Configuration 1[31:24]	The configuration option of the peripheral.
Revision number[23:20]	The revision number of the peripheral. The revision number starts from 0.
Designer[19:12]	The designer identification. ARM Limited is `0x41` (ASCII A).
Part number[11:0]	The peripheral, using the three digit product code.

Note

When you design a systems memory map then you must remember that the register has a 4KB-memory footprint. All memory accesses to the Peripheral Identification Registers must be 32-bit, using the `LDR` and `STR` instructions.

Figure 3-11 shows the bit assignments for the IECPeriphID0-3 Registers.

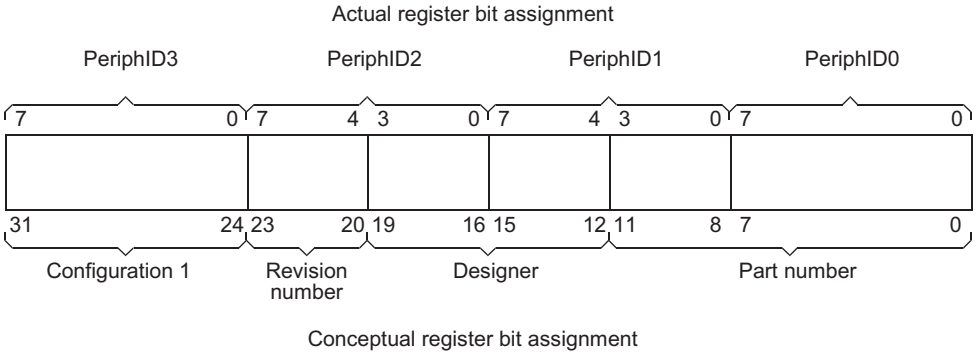


Figure 3-11 Peripheral Identification Register bit assignments, PeriphID0-3

The PeriphID4-7 Registers provide the peripheral options listed in Table 3-29.

Table 3-29 Peripheral Identification Register options, PeriphID4-7

Bits	Description
Configuration 5[31:24]	Reserved, read undefined.
Configuration 4[23:16]	Reserved, read undefined.
Configuration 3[15:8]	Reserved, read undefined.
Configuration 2[7:0]	Reserved, read undefined.

Figure 3-12 on page 3-20 shows the bit assignments for the IECPeriphID4-7 Registers.

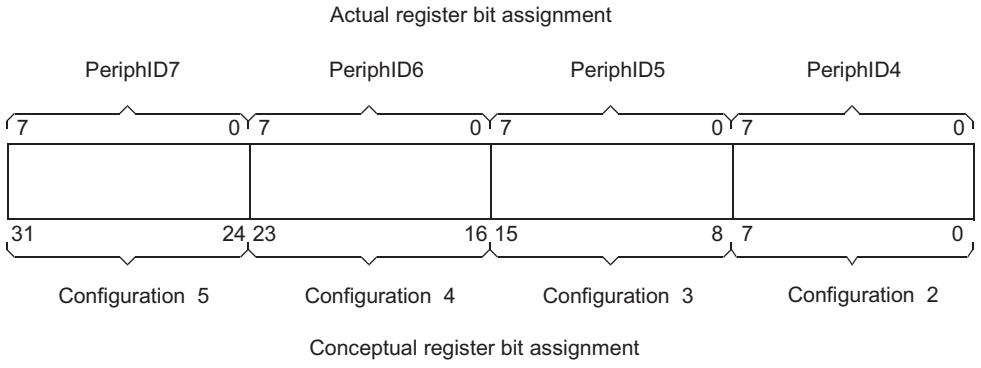


Figure 3-12 Peripheral Identification Register bit assignments, IECPeriphID4-7

The eight, 8-bit peripheral identification registers are described in the following subsections:

- *Peripheral Identification Register 0*
- *Peripheral Identification Register 1* on page 3-21
- *Peripheral Identification Register 2* on page 3-21
- *Peripheral Identification Register 3* on page 3-22
- *Peripheral Identification Register 4* on page 3-22
- *Peripheral Identification Register 5* on page 3-22
- *Peripheral Identification Register 6* on page 3-23
- *Peripheral Identification Register 7* on page 3-23.

Peripheral Identification Register 0

The PeriphID0 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-30 lists the register bit assignments.

Table 3-30 IECPeriphID0 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	Partnumber0	See note.

Note

The Partnumber0 value is equal to the same value contained within the peripheral ID registers of the peripheral which the PTB accompanies. A unique part number is assigned for PTBs that are not designed to accompany a specific peripheral, for example, a system-level clock controller.

Peripheral Identification Register 1

The PeriphID1 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-31 lists the register bit assignments.

Table 3-31 PeriphID1 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:4]	Designer0	These bits read back as `0x1`.
[3:0]	Partnumber1	See note.

Note

The Partnumber1 value is equal to the same value contained within the peripheral ID registers of the peripheral which the PTB accompanies. A unique part number is assigned for PTBs that are not designed to accompany a specific peripheral, for example, a system-level clock controller.

Peripheral Identification Register 2

The PeriphID2 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-32 lists the register bit assignments.

Table 3-32 PeripID2 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:4]	Revision	See note.
[3:0]	Designer1	These bits read back as `0x4`.

Note

The Revision value is equal to the same value contained within the peripheral ID registers of the peripheral which the PTB accompanies. The Revision is equal to `0x00` for PTBs that are not designed to accompany a specific peripheral, for example, a system-level clock controller.

Peripheral Identification Register 3

The PeriphID3 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-33 lists the register bit assignments.

Table 3-33 PeriphID3 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	Configuration1	These bits contain the revision number of the PTB.

Peripheral Identification Register 4

The PeriphID4 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-34 lists the register bit assignments.

Table 3-34 PeriphID4 Register bit assignments

Bit	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	Configuration 2	These bits are all reserved.

Peripheral Identification Register 5

The PeriphID5 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-35 lists the register bit assignments

Table 3-35 PeriphID5 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	Configuration 3	These bits are all reserved.

Peripheral Identification Register 6

The PeriphID6 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-36 lists the register bit assignments

Table 3-36 PeriphID6 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	Configuration 4	These bits are all reserved.

Peripheral Identification Register 7

The PeriphID7 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-37 lists the register bit assignments

Table 3-37 PeriphID7 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	Configuration 5	These bits are all reserved.

3.3.28 Component Identification Registers

The Component Identification Registers are four, 8-bit read-only registers. They span two address locations:

- CompID0-3 Registers span address locations `0xFF0-0xFFC`.

Each of these blocks of registers 0-3 can conceptually be treated as one 32-bit read-only register. The CompID0-3 Registers provide the values listed in Table 3-38.

Table 3-38 Component Identification Register options, PeriphID0-3

Bits	Description
Component1 [31:24]	Preamble.
Component2 [23:20]	Preamble.
Component3 [19:12]	Preamble.
Component4r[11:0]	Preamble.

Note

When you design a systems memory map then you must remember that the register has a 4KB-memory footprint. All memory accesses to the peripheral identification registers must be 32-bit, using the `LDR` and `STR` instructions.

Figure 3-11 on page 3-19 shows the bit assignments for the CompID0-3 Registers.

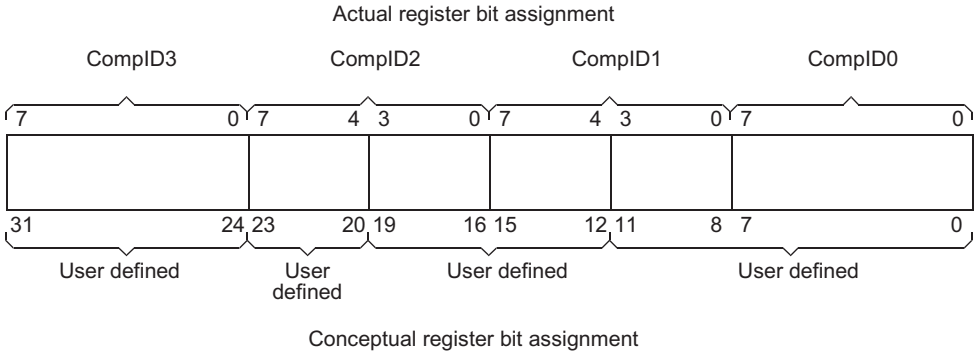


Figure 3-13 Peripheral Identification Register bit assignments, PeriphID0-3

Note

Each CompID register contains a single 8-bit value, actual register assignment[7:0]. The conceptual register bit assignment is a single 32-bit value referred to as the preamble which takes the value `0xB105B00D` for PTBs.

The eight, 8-bit peripheral identification registers are described in the following subsections:

- *Component Identification Register 0* on page 3-25
- *Component Identification Register 1* on page 3-25
- *Component Identification Register 2* on page 3-25
- *Component Identification Register 3* on page 3-26.

Component Identification Register 0

The CompID0 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-30 on page 3-20 lists the register bit assignments.

Table 3-39 CompID0 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	ComponentID0	These bits read back as `0x0D`.

Component Identification Register 1

The CompID1 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-31 on page 3-21 lists the register bit assignments.

Table 3-40 CompID1 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	ComponentID1	These bits read back as `0xB0`.

Component Identification Register 2

The CompID2 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-32 on page 3-21 lists the register bit assignments.

Table 3-41 PeripID2 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	ComponentID2	These bits read back as `0x05`.

Component Identification Register 3

The CompID3 Register is read-only. It is hard coded and the fields in the register determine the reset value. Table 3-33 on page 3-22 lists the register bit assignments.

Table 3-42 PeriphID3 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined.
[7:0]	ComponentId3	These bits read back as `0xB1`.

Glossary

This glossary describes some of the terms used in ARM manuals. Where terms can have several meanings, the meaning presented here is intended.

Advanced eXtensible Interface (AXI)

This is a bus protocol that supports separate address/control and data phases, unaligned data transfers using byte strobes, burst-based transactions with only start address issued, separate read and write data channels to enable low-cost DMA, ability to issue multiple outstanding addresses, out-of-order transaction completion, and easy addition of register stages to provide timing closure.The AXI protocol also includes optional extensions to cover signaling for low-power operation.

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

Advanced High-performance Bus (AHB)

The AMBA Advanced High-performance Bus system connects embedded processors such as an ARM core to high-performance peripherals, DMA controllers, on-chip memory, and interfaces. It is a high-speed, high-bandwidth bus that supports multi-master bus management to maximize system performance.

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
AMBA is the ARM open standard for multi-master on-chip buses, capable of running with multiple masters and slaves. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a System-on-Chip (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules. AHB conforms to this standard.

Advanced Peripheral Bus (APB)
The AMBA Advanced Peripheral Bus is a simpler bus protocol than AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

See also Advanced High-performance Bus.

AHB *See* Advanced High-performance Bus.

AHB Access Port (AHB-AP)
An optional component of the DAP that provides an AHB interface to an SoC.

AHB-AP *See* AHB Access Port.

AHB-Lite AHB-Lite is a subset of the full AHB specification. It is intended for use in designs where only a single AHB master is used. This can be a simple single AHB master system or a multi-layer AHB system where there is only one AHB master on a layer.

Aligned Aligned data items are stored so that their address is divisible by the highest power of two that divides their size. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

AMBA *See* Advanced Microcontroller Bus Architecture.

APB *See* Advanced Peripheral Bus.

Architecture The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

AXI *See* Advanced eXstensible Interface.

BST *See* Boundary scan trickbox.

Burst A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AHB buses are controlled using the **HBURST** signals to specify if transfers are single, four-beat, eight-beat, or 16-beat bursts, and to specify how the addresses are incremented.

Byte An 8-bit data item.

Communications channel
The hardware used for communicating between the software running on the processor, and an external host, using the debug interface. When this communication is for debug purposes, it is called the Debug Comms Channel. In an ARMv6 compliant core, the communications channel includes the Data Transfer Register, some bits of the Data Status and Control Register, and the external debug interface controller, such as the DBGTAP controller in the case of the JTAG interface.

Control bits The bottom eight bits of a Program Status Register (PSR). The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Core A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.

Core module In the context of an ARM Integrator, a core module is an add-on development board that contains an ARM processor and local memory. Core modules can run standalone, or can be stacked onto Integrator motherboards.

DBGTAP *See* Debug Test Access Port.

Direct Memory Access (DMA)
An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

Doubleword A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.

Doubleword-aligned
A data item having a memory address that is divisible by eight.

DSM *See* Design Simulation Model.

DVS *See* Device Validation Suite.

Event 1 (Simple) An observable condition that can be used by an ETM to control aspects of a trace.

2 (Complex) A boolean combination of simple events that is used by an ETM to control aspects of a trace.

Exception A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.

Exponent The component of a floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number.

Fraction The floating-point field that lies to the right of the implied binary point.

Half-rate clocking (ETM)
Dividing the trace clock by two so that the TPA can sample trace data signals on both the rising and falling edges of the trace clock. The primary purpose of half-rate clocking is to reduce the signal transition rate on the trace clock of an ASIC for very high-speed systems.

Halfword A 16-bit data item.

Macrocell A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as a processor, an ETM, and a memory block) plus application-specific logic.

Microprocessor *See* Processor.

Processor A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

Programming Language Interface (PLI)
For Verilog simulators, an interface by which so-called foreign code (code written in a different language) can be included in a simulation.

Read Reads are defined as memory operations that have the semantics of a load. That is, the ARM instructions LDM, LDRD, LDC, LDR, LDRT, LDRSH, LDRH, LDRSB, LDRB, LDRBT, LDREX, RFE, STREX, SWP, and SWPB, and the Thumb instructions LDM, LDR, LDRSH, LDRH, LDRSB, LDRB, and POP. Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

SBO *See* Should Be One.

SBZ *See* Should Be Zero.

SBZP *See* Should Be Zero or Preserved.

Should Be Zero or Preserved (SBZP)
Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

Significand The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of the implied binary point and a fraction field to the right.

Tightly coupled memory (TCM)
An area of low latency memory that provides predictable instruction execution or data load timing in cases where deterministic performance is required. TCMs are suited to holding: - critical routines (such as for interrupt handling) - scratchpad data - data types whose locality is not suited to caching - critical data structures (such as interrupt stacks).

Unpredictable For reads, the data returned from the location can have any value. For writes, writing to the location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.

Unsupported values
Specific data values that are not processed by the VFP coprocessor hardware but bounced to the support code for completion. These data can include infinities, NaNs, subnormal values, and zeros. An implementation is free to select which of these values is supported in hardware fully or partially, or requires assistance from support code to complete the operation. Any exception resulting from processing unsupported data is trapped to user code if the corresponding exception enable bit for the exception is set.

Word A 32-bit data item.

Word-invariant In a word-invariant system, the address of each byte of memory changes when switching between little-endian and big-endian operation, in such a way that the byte with address A in one endianness has address A EOR 3 in the other endianness. As a result, each aligned word of memory always consists of the same four bytes of memory in the same order, regardless of endianness. The change of endianness occurs because of the change to the byte addresses, not because the bytes are rearranged.The ARM architecture supports word-invariant systems in ARMv3 and later versions. When word-invariant support is selected, the behavior of load or store instructions that are given unaligned addresses is instruction-specific, and is in general not the expected behavior for an unaligned access. It is recommended that word-invariant systems should use the endianness that produces the desired byte addresses at all times, apart possibly from very early in their reset handlers before they have set up the endianness, and that this early part of the reset handler should use only aligned word memory accesses.

See also Byte-invariant.

Write Writes are defined as operations that have the semantics of a store. That is, the ARM instructions SRS, STM, STRD, STC, STRT, STRH, STRB, STRBT, STREX, SWP, and SWPB, and the Thumb instructions STM, STR, STRH, STRB, and PUSH. Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

Write-back (WB) In a write-back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. (Also known as copyback).

Write buffer A block of high-speed memory, arranged as a FIFO buffer, between the data cache and main memory, whose purpose is to optimize stores to main memory.

Write completion The memory system indicates to the processor that a write has been completed at a point in the transaction where the memory system is able to guarantee that the effect of the write is visible to all processors in the system. This is not the case if the write is associated with a memory synchronization primitive, or is to a Device or Strongly Ordered region. In these cases the memory system might only indicate completion of the write when the access has affected the state of the target, unless it is impossible to distinguish between having the effect of the write visible and having the state of target updated.

This stricter requirement for some types of memory ensures that any side-effects of the memory access can be guaranteed by the processor to have taken place. You can use this to prevent the starting of a subsequent operation in the program order until the side-effects are visible.

Write-through (WT) In a write-through cache, data is written to main memory at the same time as the cache is updated.

WT *See* Write-through.

XVC *See* eXternal Verification Component.

XVC Test Scenario Manager

This co-ordinates the operation of multiple XVCs.

See also eXternal Verification Component

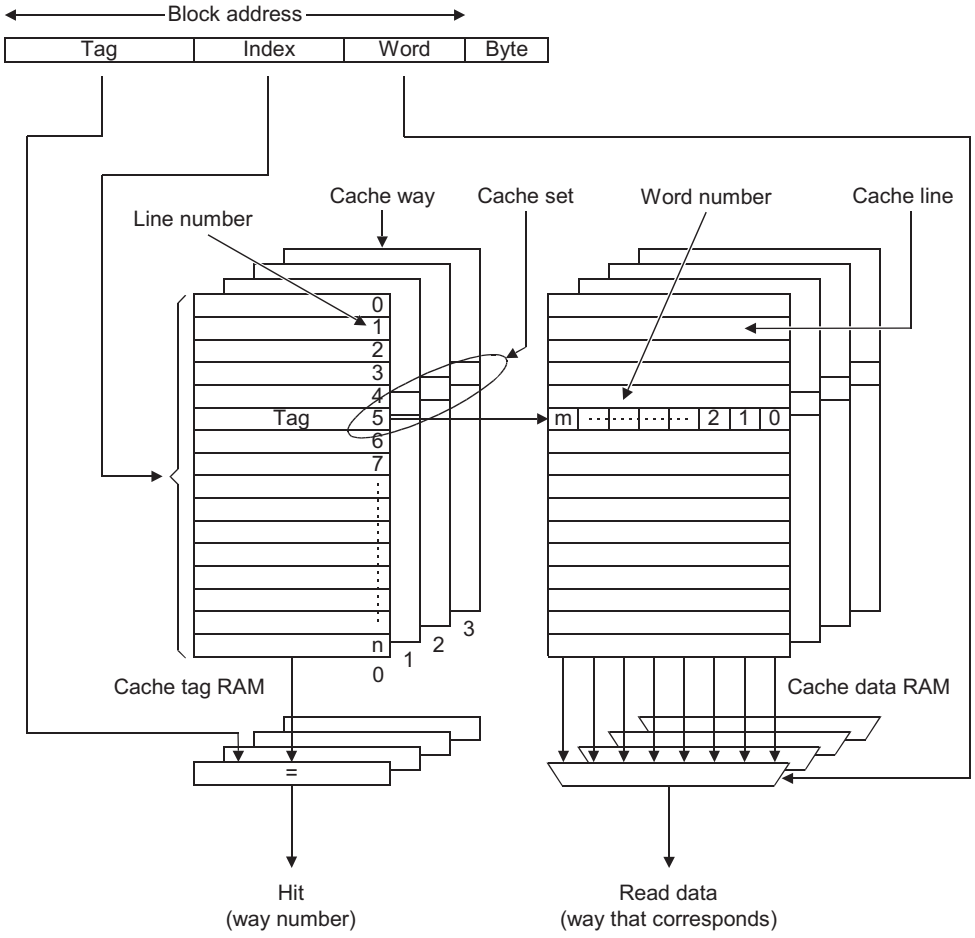
XTSM *See* XVC Test Scenario Manager.

Cache terminology diagram

The diagram below illustrates the following cache terminology:

- block address
- cache line
- cache set

- cache way
- index
- tag.



Index

M

N

P

R

S

T

U

V

W

X