AMBA Design Kit

Revision: r3p0

Technical Reference Manual

ARM®

ARM DDI 0243C

AMBA Design Kit
Technical Reference Manual

Release Information

The following changes have been made to this document:

Change history

Date	Issue	Confidentiality	Change
February 2003	A	Confidential	First release
October 2003	B	Non-Confidential	Updated for r3p0 release
August 2007	C	Non-Confidential	C for r3p0 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
AMBA Design Kit Technical Reference Manual

List of Tables
AMBA Design Kit Technical Reference Manual

List of Figures
AMBA Design Kit Technical Reference Manual

Preface

This preface introduces the *AMBA Design Kit r3p0 Technical Reference Manual*. It contains the following sections:

About this manual

This is the technical reference manual for the *AMBA Design Kit* (ADK).

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 document is written for *System-on-Chip* (SoC) designers and system architects, and provides a description of components within the ADK architecture.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ADK.

Chapter 2 *Functional Overview*

Read this chapter for an overview of the top-level structure of the ADK, and examples of how the ADK can be used.

Chapter 3 *AHB Components*

Read this chapter for a description of the AHB components used in the ADK.

Chapter 4 *APB Components*

Read this chapter for a description of the APB components used in the ADK.

Chapter 5 *Behavioral Models*

Read this chapter for a description of the behavioral models in the ADK.

Chapter 6 *PrimeCell GPIO*

Read this chapter for a description of how the PrimeCell *General Purpose Input/Output* (GPIO) is integrated within the ADK.

Appendix A *AHB-Lite Overview*

Read this appendix for an overview of the AHB-Lite. AHB-Lite is a subset of the full AHB specification.

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

Conventions

This section describes the conventions that this manual uses:

- *Typographical*
- *Timing diagrams* on page xvi
- *Signals* on page xvi
- *Numbering* on page xvii.

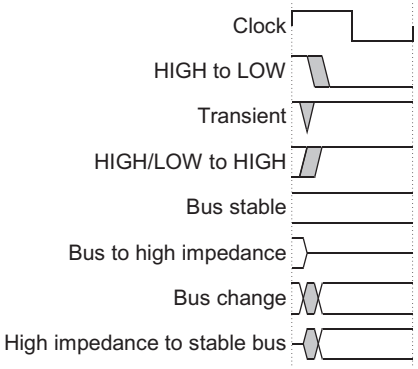
Typographical

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 the register that is accessed.

Timing diagrams

This manual contains one or more timing diagrams. The figure named *Key to timing diagram conventions* 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

Signals

The signal conventions are:

Signal level 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.

Lower-case n Denotes an active-LOW signal.

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

Prefix P Denotes *Advanced Peripheral Bus* (APB) signals.

Numbering

The numbering convention is:

<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.

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 ADK. Refer to the following documents for other relevant information:

- *AMBA Specification (Rev 2.0)* (ARM IHI 0011)
- *AMBA 3 AHB-Lite Protocol v1.0 Specification* (ARM IHI 0033)
- *Multi-layer AHB Overview* (ARM DVI 0045)
- *AMBA 3 APB Protocol Specification* (ARM IHI 0024)
- *ARM PrimeCell General Purpose Input/Output (PL061) Technical Reference Manual* (ARM DDI 0190)
- *AMBA Design Kit User Guide* (ARM DUI 0183)
- *AMBA Designer (FD001) User Guide* (ARM DUI 0333)
- *ARM PrimeCell High-Performance Matrix (PL301) Technical Reference Manual* (ARM DDI 0397).

Feedback

ARM Limited welcomes feedback on the ADK 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 book

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

- the document title
- the document number
- the 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 describes the *AMBA Design Kit* (ADK). It contains the following sections:

1.1 About the ADK

The ADK comprises the building blocks required to create an example system based on the low-power, generic design methodology of the *Advanced Microcontroller Bus Architecture* (AMBA). Three preconfigured and validated examples enable you to develop custom devices in very short design cycles. Therefore, you can easily reuse the resulting subcomponents in future designs.

Note

ADK EASY environments use AHB and APB protocols.

1.2 AMBA signals

This section describes the following signals:

- *AMBA AHB signals*
- *AMBA APB signals* on page 1-5.

Note

For a description of the non-AMBA signals used in the ADK, see Chapter 3 *AHB Components*, Chapter 4 *APB Components*, and Chapter 5 *Behavioral Models*.

1.2.1 AMBA AHB signals

Table 1-1 lists the AMBA AHB signals used in the ADK.

Table 1-1 AMBA AHB signals

Signal	Direction				Description
	Slave	Master	Arbiter	Decoder	
HADDR[31:0]	Input	Output	-	Input	The 32-bit system address bus.
HBURST[2:0]	Input	Output	Input	-	These signals indicate if the transfer forms part of a burst. Four, eight, and sixteen beat bursts are supported and the burst can be either incrementing or wrapping.
HBUSREQx	-	Output	Input	-	A signal from bus master **x** to the bus arbiter to indicate that the bus master requires the bus. There is an **HBUSREQ** signal for each bus master in the system, up to a maximum of 16 bus masters.
HCLK	Input	Input	Input	-	This clock times all bus transfers.
HGRANTx	-	Input	Output	-	This signal indicates that the bus master is currently the highest priority master. Ownership of the address/control signals changes at the end of a transfer when **HREADY** is HIGH, so the master gets access to the bus when both **HREADY** and **HGRANT** are HIGH.
HLOCK	-	Output	Input	-	When HIGH, this signal indicates that the master requires locked access to the bus and no other master must be granted the bus until this signal is LOW.

Table 1-1 AMBA AHB signals (continued)

Signal	Direction				Description
	Slave	Master	Arbiter	Decoder	
HMASTER[3:0]	Input	-	Output	-	These signals from the arbiter indicate the bus master that is currently performing a transfer and is used by the slaves that support SPLIT transfers to determine the master that is attempting an access. The timing of **HMASTER** is aligned with the timing of the address and control signals.
HMASTLOCK	Input	-	Output	-	Indicates that the current master is performing a locked sequence of transfers. This signal has the same timing as the **HMASTER** signals.
HPROT[3:0]	Input	Output	-	-	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that requires some level of protection. The signals indicate if the transfer is an opcode fetch or data access, as well as if the transfer is a privileged mode access or User mode access. For bus masters with a memory management unit these signals also indicate whether the current access is cacheable or bufferable.
HRDATA[31:0] or **HRDATA[63:0]**	Output	Input	-	-	The read data bus transfers data from bus slaves to the bus master during read operations. ARM recommends a minimum data bus width of 32 bits. However, you can easily extend this to enable higher bandwidth operation.
HREADY **HREADYOUT**	Input Output	Input -	Input -	- -	When HIGH, the **HREADY** signal indicates that a transfer has finished on the bus. You can drive this signal LOW to extend a transfer.
HRESETn	Input	Input	Input	Input	The bus reset signal is active LOW. It resets the system and the bus.
HRESP[1:0]	Output	Input	Input	-	The transfer response provides additional information on the status of a transfer. Four different responses are provided, OKAY, ERROR, RETRY, and SPLIT.

Table 1-1 AMBA AHB signals (continued)

Signal	Direction				Description
	Slave	Master	Arbiter	Decoder	
HSEL	Input	-	-	Output	Each AHB slave has its own slave select signal and this signal indicates that the current transfer is intended for the selected slave. This signal is simply a combinatorial decode of the address bus.
HSIZE[2:0]	Input	Output	-	-	These signals indicate the size of the transfer, typically byte (8-bit), halfword (16-bit), or word (32-bit). The protocol permits larger transfer sizes up to a maximum of 1024 bits.
HSPLIT[15:0]	Output	-	Input	-	A split-capable slave uses the 16-bit split bus to indicate to the arbiter the bus masters that can reattempt a split transaction. Each bit of this split bus corresponds to a single bus master.
HTRANS[1:0]	Input	Output	Input	-	This indicates the type of the current transfer. This can be NONSEQUENTIAL, SEQUENTIAL, IDLE, or BUSY.
HWDATA[31:0] or **HWDATA[63:0]**	Input	Output	-	-	The write data bus transfers data from the master to the bus slaves during write operations. ARM recommends a minimum data bus width of 32 bits. However, you can easily extend this to enable higher bandwidth operation.
HWRITE	Input	Output	-	-	When HIGH, this signal indicates a write transfer, and when LOW, a read transfer.

1.2.2 AMBA APB signals

Table 1-2 on page 1-6 lists the AMBA APB signals used in the ADK.

Table 1-2 AMBA APB signals

Signal	Type	Direction from bridge	Description
PADDR[31:0]	Peripheral address bus	Output	This is the APB address bus, and can be up to 32 bits wide. Individual peripherals use this bus for decoding register accesses to the peripheral. The address becomes valid after the first rising edge of the clock at the start of the transfer. If there is a following APB transfer, the address changes to the new value. Otherwise it holds its current value until the start of the next APB transfer.
PCLK	Peripheral clock	Input	This clock times all bus transfers. All events occur on rising edges of this signal.
PENABLE	Peripheral enable	Output	This enable signal times all accesses on the peripheral bus. **PENABLE** goes HIGH on the second clock rising edge of the transfer, and LOW on the third (last) rising clock edge of the transfer.
PRDATA[31:0]	Peripheral read data bus	Input	The peripheral read data bus is driven by the selected peripheral bus slave during read cycles, when **PWRITE** is LOW.
PRESETn	Peripheral reset	Input	The bus reset signal is active LOW and resets the system.
PSELx	Peripheral slave select	Output	There is one of these signals for each APB peripheral present in the system. The signal indicates that the slave device is selected, and that a data transfer is required. It has the same timing as the peripheral address bus. It becomes HIGH at the same time as **PADDR**, but is set LOW at the end of the transfer.
PWDATA[31:0]	Peripheral write data bus	Output	The peripheral write data bus is continuously driven by this module, changing during write cycles, when **PWRITE** is HIGH.
PWRITE	Peripheral transfer direction	Output	This signal indicates a write to a peripheral when HIGH, and a read from a peripheral when LOW. It has the same timing as the peripheral address bus.

1.3 Product revisions

This section describes differences in functionality between product revisions of the ADK:

r3p0 Contains the following additions to functionality:

- AMBA Designer integration of Bus Matrix
- Configurable address map for AHB matrix
- Targeted at TSMC CL013G library.

Chapter 2
Functional Overview

This chapter describes the top-level structure of the *AMBA Design Kit* (ADK), and gives examples of how you can use the ADK. It contains the following sections:

2.1 About the ADK toolkit

The ADK toolkit enables designers to create AMBA-based components and *System-on-Chip* (SoC) designs using the following capabilities:

- a fully working simulation environment to enable you to become familiar with the AMBA protocol
- a development environment for AMBA modules
- a number of different starting preconfigured arrangements from which complex SoC designs can be built
- Verilog language support
- example synthesis scripts for the synthesizable blocks.

2.2 ADK components

The ADK is made up of components that are summarized in the following sections:

- *AHB components*
- *APB components* on page 2-5
- *Behavioral models* on page 2-6.

2.2.1 AHB components

This section summarizes the AHB components. For more details see Chapter 3 *AHB Components*.

Reset controller

- generates a system-wide reset from power-on reset and watchdog reset signals.

Arbiter

- uses a simple priority algorithm to control the master that has access to the AHB bus
- provides three masters plus dummy
- arbitration scheme defined in a separate sub-block for easy modification and reuse.

Retry slave, 32 or 64-bit

- a starting point for a typical AHB slave design
- simple four-word register bank plus read-only registers providing arithmetic combinations of the writable registers
- higher order address bits control the response
- supports retry response and wait states.

Static memory interface

- AHB access to the external RAM and ROM
- intended as a simple example
- replaceable with PrimeCell *Static Memory Controller* (SMC) (PL241).

Test interface controller

- now included within the static memory interface
- converts test vectors applied on the external pins of the device into valid AHB transfers on an internal bus
- to be used to test relevant ARM processor cores.

Bus matrix

- enables parallel access paths between multiple masters and slaves
- improves overall systems bandwidth
- gives increased system design flexibility.

AHB to APB bridge

- provides a 16-slot interface between the high-performance pipelined AHB and the lower performance peripheral bus
- supports AMBA 3 APB protocol.

Example bus master (32 or 64 bit)

- a simple bus master that is intended to act as a reference framework for bus master development
- demonstrates bus master activity by read-pause-write bursts
- interfaces through the AHB-Lite to AHB wrapper
- configurable address and pause length.

AHB to AHB bridges

- provide an interface between two separate AHB buses, supporting various clocking schemes.

Interrupt controller

- generates the two ARM interrupt signals from the multiple interrupt sources that can exist in a system
- individual bit-level control of the masking of interrupts
- interrupt driven or polled method of operation

- pin compatible with the PrimeCell *Vectored Interrupt Controller* (VIC) (PL190)
- can be daisy-chained with the PrimeCell VIC.

Downsizer

- AHB Slave gasket enables connection of 32-bit slave to a 64-bit bus.

Funnel

- interfaces 32-bit slaves to a 64-bit bus where accesses are word size or smaller.

2.2.2 APB components

This section summarizes the APB components. For more details see Chapter 4 *APB Components*.

Timers

- two 32-bit counters with an optional prescaler up to 8 bits
- three modes of operation:
  - free running
  - periodic interrupt
  - one shot.

Watchdog

- generates a regular, programmed interrupt
- asserts WDOGRES reset signal if device remains unserviced.

Remap/pause controller

- basic glue logic functions that are required to implement correct boot up behavior and a basic low-power mode of operation.

Example APB slave

- a reference framework for a typical APB slave design
- simple four-word register bank plus read-only registers providing arithmetic combinations of the writable registers.

PrimeCell *General Purpose Input/Output* (GPIO) (PL061)

- sixteen individually-programmable input/output pins, eight input, eight output, with enables for usage as eight input/output pins when combined with external tristate drivers
- control word read-back capability
- additional test registers and modes implemented for functional verification and manufacturing test.

2.2.3 Behavioral models

This section summarizes the behavioral models. For more details see Chapter 5 *Behavioral Models*.

Internal memory

- 32 or 64-bit wide on-chip SRAM model
- memory can be initialized from an ASCII file
- variable size (default 1KB).

Test interface driver

- reads test input file
- outputs vectors to the test interface controller
- checks read data from the test interface controller against expected values

File reader master (32 or 64-bit)

- enables generation of legal AHB transactions from an external stimulus file
- interfaces through the AHB-Lite to AHB wrapper.

2.3 Example systems

The following example systems are provided with the ADK:

- *FRM-based AMBA system, EASY_FRBM*
- *ARM7TDMI-based example AMBA system, EASY_ARM7* on page 2-8
- *ARM922T-based example AMBA system, EASY_ML* on page 2-10.

2.3.1 FRM-based AMBA system, EASY_FRBM

This example provides an AMBA system based around the AHB *File Reader Master* (FRM). You can use it to generate any AMBA transaction to exercise the bus and peripherals. The design is supplied with example files to drive the bus peripherals and you can use it as the basis of a testbench for your peripherals. Figure 2-1 shows the structure of the EASY_FRBM example AMBA system.

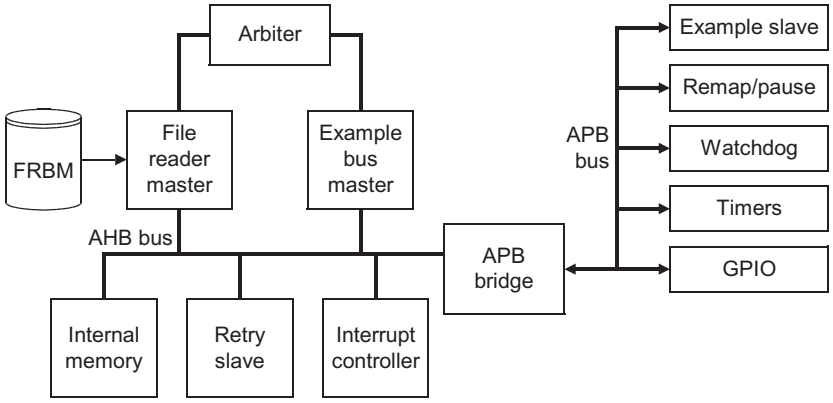


Figure 2-1 EASY_FRBM example AMBA system

The EASY_FRBM example AMBA system contains the following components:

- `FileRdMaster32`
- `EgMaster32`
- `IntMem32`
- `Ahb2Apb`
- `RetrySlave32`
- `Interrupt`
- `Arbiter3`
- `Decoder`
- `DefaultSlave`
- `MuxM2S`

- MuxS2M
- ResetCntl
- MuxP2B
- Watchdog
- Timers
- GPIO
- RemapPause
- EgAPBSlave.

The testbench associated with this example system is TBEasy_FRBM and contains:

- EASY_FRBM
- Tube (GPIO connected)
- Clock generation
- Reset generation.

2.3.2 ARM7TDMI-based example AMBA system, EASY_ARM7

This example provides a complete working example of an ARM7TDMI-based ASIC design. The design is supplied with:

- example software to run in the system
- synthesis scripts to generate a netlist version of the design
- examples of the use of a test interface that enables the application of functional test vectors to the ARM core.

Figure 2-2 on page 2-9 shows the EASY_ARM7 example AMBA system.

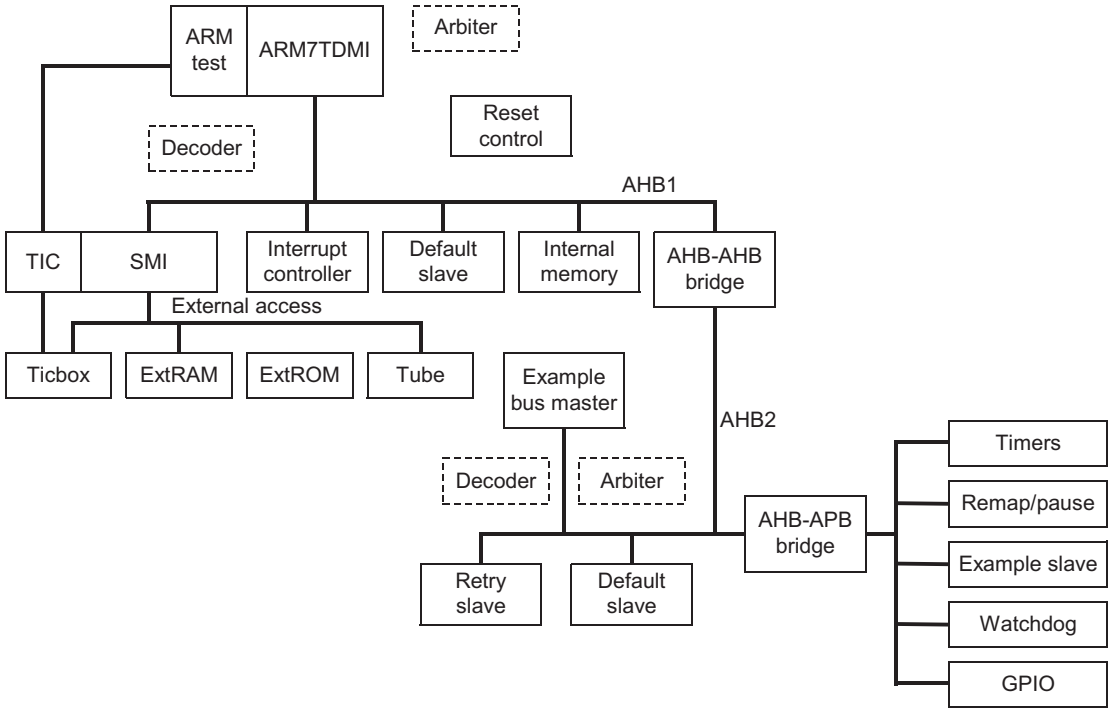


Figure 2-2 EASY_ARM7 example AMBA system

There are two AHB subsystems in the EASY_ARM7 example AMBA system, consisting of the following components:

- Common:
  - ResetCntl
  - Ahb2Ahb.
- AHB1:
  - A7TDMI - ARM Test
  - SMI - TIC
  - IntMem32
  - Interrupt
  - DefaultSlave
  - Arbiter3
  - Decoder
  - MuxM2S
  - MuxS2M.

- AHB2:
  - EgMaster32
  - Ahb2Apb
  - RetrySlave32
  - DefaultSlave
  - Arbiter3
  - Decoder
  - MuxM2S
  - MuxS2M.
- APB (connected on AHB2):
  - MuxP2B
  - Watchdog
  - Timers
  - GPIO
  - RemapPause
  - EgApbSlave.

The testbenches associated with this example system include:

- TBEasy_ARM7, that contains:
  - EASY_ARM7
  - Tube (SMI connected)
  - GPIO loop-back test logic
  - Memory (External RAM and ROM)
  - Clock generation
  - Reset generation.
- TBEasy_ARM7_TIC, used exclusively for TIF CPU test vectors, that contains:
  - EASY_ARM7
  - GPIO loop-back test logic
  - Ticbox
  - Clock generation
  - Reset generation
  - test clock generation for ARM7TDMI.

2.3.3 ARM922T-based example AMBA system, EASY_ML

This example provides a complete working example of an ARM922T-based ASIC design using the multi-layer AMBA architecture. The design is supplied with:

- example software to run in the system

- synthesis scripts to generate a netlist version of the design
- examples of the use of a test interface that enables the application of functional test vectors to the ARM core.

The system is configured with two layers (one with two masters on one AHB), a 2-to-3 bus matrix and three slave ports (one with three slaves sharing one port). Figure 2-3 shows the EASY_ML example AMBA system.

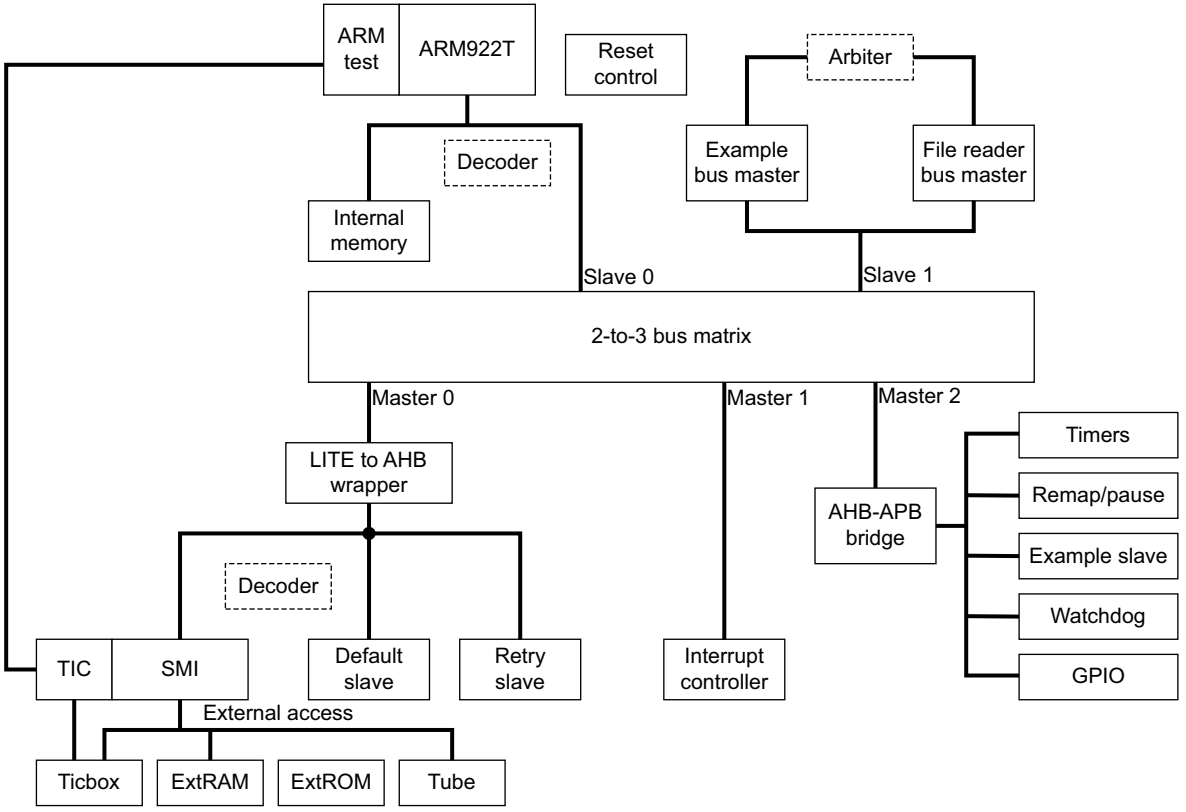


Figure 2-3 EASY_ML example AMBA system

The EASY_ML example AMBA system contains the following components:

- ResetCntl
- Bus Matrix (2-input by 3-output port variant)
  - Inport0
- A922T
- ARM Test
- IntMem32

- Decoder
- MuxS2M
  - Inport1
- EgMaster
- FileRdMaster32
- MuxM2S
- Arbiter3
  - Outport0
- Lite2Ahb
- SMI
- TIC
- RetrySlave32
- MuxS2M
- Decoder
- DefaultSlave
  - Outport1
- Interrupt
  - Outport2
- Ahb2Apb
- MuxP2B
- Watchdog
- Timers
- GPIO
- RemapPause
- EgApbSlave.

The testbenches associated with this example system include:

- TBEasy_ML, that contains:
  - EASY_ML
  - Tube, SMI connected
  - GPIO loop-back test logic
  - Memory, external RAM and ROM
  - Clock generation
  - ARM922T Fast Cache clock generation
  - Reset generation.
- TBEasy_ML_TIC, used exclusively for TIF CPU test vectors, that contains:
  - EASY_ML
  - GPIO loop-back test logic

- Ticbox
- Clock generation
- ARM922T Fast Cache clock generation
- Reset generation
- test clock generation for ARM922T.

2.3.4 Address map

Figure 2-4 on page 2-14 shows a block diagram of the ADK address map.

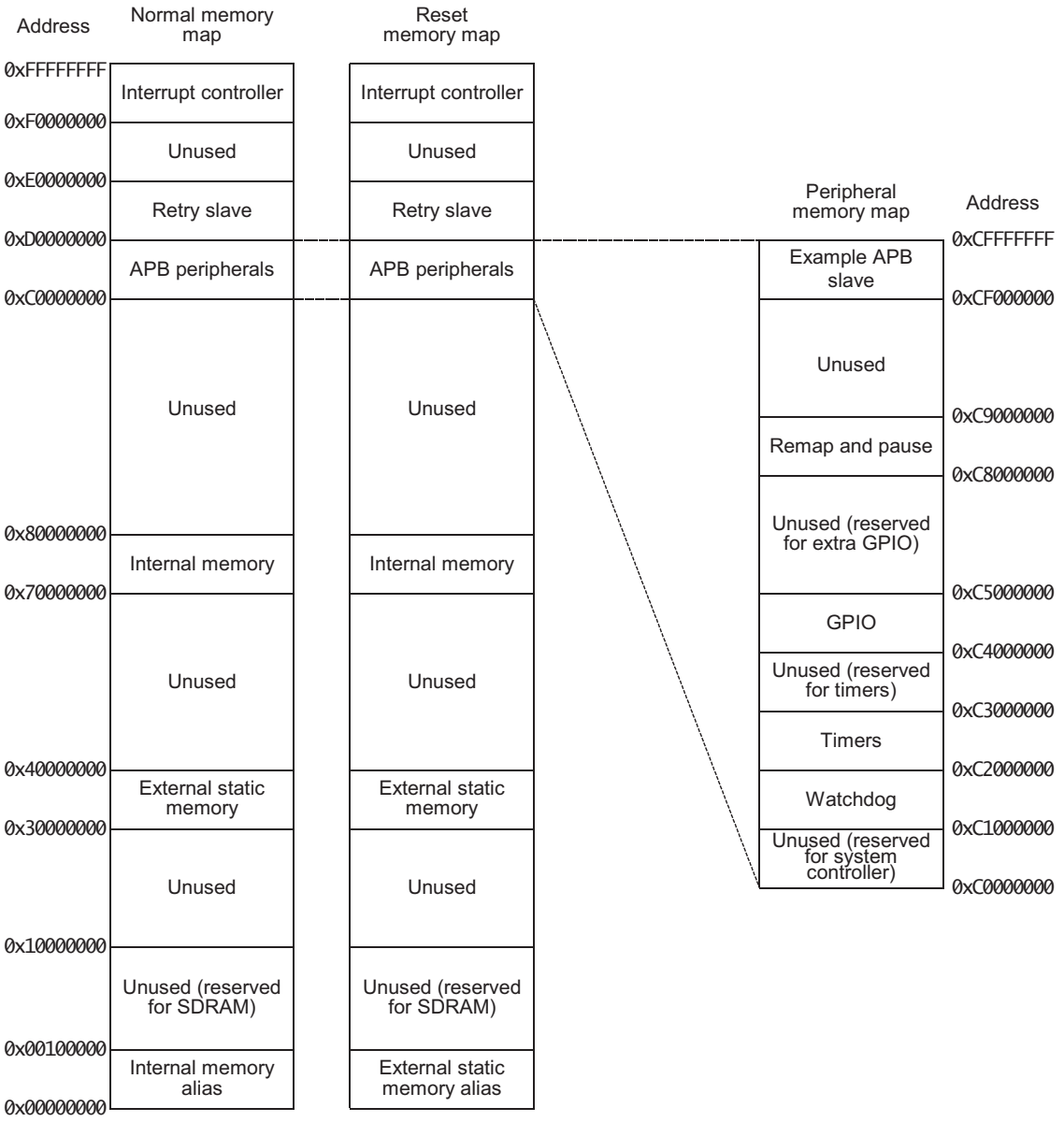


Figure 2-4 ADK address map

Chapter 3
AHB Components

This chapter describes the AHB components used in the *AMBA Design Kit* (ADK). It contains the following sections:

- *64-bit to 32-bit funnel* on page 3-93.

3.1 Reset controller

The reset controller, `ResetCntl`, generates the system reset signal from an external reset input. This module is based on a state machine that is used to detect the external reset being asserted, and is used to generate the system reset output. Figure 3-1 shows the reset controller module block diagram.

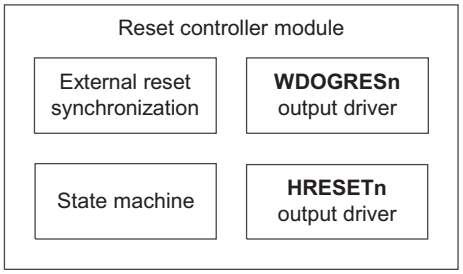


Figure 3-1 Reset controller module components

3.1.1 Signal descriptions

Table 3-1 lists the non-AMBA signals used by the reset controller.

Table 3-1 Reset controller signals

Signal	Type	Direction	Description
HRESETn	System reset	Output	The system reset output.
nPOReset	Power-on reset	Input	Power-on reset input. This active LOW signal causes a cold reset when LOW. Can be asserted asynchronously to **HCLK**. The source of the **nPOReset** signal is implementation-dependent.
WDOGRES	Watchdog reset	Input	The watchdog clock domain reset input.
WDOGRESn	Watchdog reset	Output	The watchdog clock domain reset output.

3.2 Arbiter

The AMBA bus specification is a multi-master bus standard. As a result, a bus arbiter is required to ensure that only one bus master has access to the bus at any particular time. The arbiter, `Arbiter3`, can support up to three bus masters. Figure 3-2 shows the arbiter module block diagram.

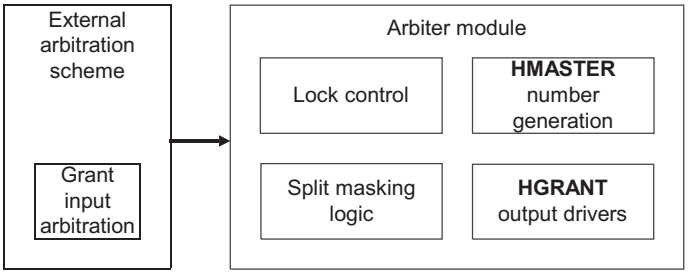


Figure 3-2 Arbiter module components

3.2.1 Operation

Operation of the arbiter is described in the following sections:

- *Arbitration scheme*
- *Dummy master* on page 3-5
- *SPLIT and LOCK* on page 3-5.

Arbitration scheme

The arbiter contains a fixed arbitration scheme that supports connection of three AHB bus masters, plus the dummy master. The priority scheme is as follows:

- **HBUSREQ3** is the highest priority.
- **HBUSREQ0** is the second highest priority. This must only be connected to a Pause input because it requests that the dummy master is granted the bus.
- **HBUSREQ2** is the third highest priority.
- **HBUSREQ1** is the lowest priority and default bus master. This input is usually used for an uncached ARM core, for example ARM7TDMI.

The arbiter re-arbitrates on every rising edge of **HCLK**, and so can potentially degrant a master part way through a burst. However, if a master has already commenced a fixed-length burst (that is, **HBURST** is not SINGLE or INCR), the arbiter does not re-arbitrate until the fixed-length burst has completed.

Because the **HBUSREQ/HGRANT** logic is pipelined, if a higher priority master requests the bus in the cycle before the current master begins a fixed-length burst, the current master performs the first access in the fixed length burst and is then degranted. You can change this behavior, but only by introducing a combinatorial path from **HTRANS** or **HBURST** to **HGRANT** within the arbiter. This would fail to meet the required timing budget.

Dummy master

Bus master 0 is reserved for the dummy bus master, that never performs real transfers. This master is granted in the following conditions:

- when the previously granted master is performing a locked transfer that has received a SPLIT response
- when the default master receives a SPLIT response and no other master is requesting the bus
- when all masters have received SPLIT responses.

Note

See the AMBA FAQ on the ARM Limited website for more details of the dummy master.

SPLIT and LOCK

When a master declares a locked sequence of transfers (through **HLOCK**), the arbiter ensures that no other master is granted access to the bus until the first master completes the locked sequence. During the locked sequence, the arbiter asserts **HMASTLOCK**. If the master receives a SPLIT response to a locked transfer, the arbiter grants the dummy master until the first master is unsplit, indicated by the slave asserting the relevant bit of the **HSPLIT** bus. During this time, the arbiter deasserts the **HMASTLOCK** signal.

Note

Further arbitration schemes within the bus hierarchy, such as within a multi-layer bus matrix, or within a multi-port slave, must also track the SPLIT/LOCK combination if this behavior is to be fully supported by a system.

To avoid system issues, ARM Limited recommends that masters never perform locked sequences to slave regions that can give a SPLIT response. SPLIT-capable slaves must respond with wait states to a locked transfer, rather than responding with SPLIT.

3.2.2 Signal descriptions

Table 3-2 lists the non-AMBA signals used by the arbiter.

Table 3-2 Arbiter signals descriptions

Signal	Type	Direction	Description
HBUSREQMx	Bus request	Input	This signal indicates that the bus master is currently the highest priority master. Ownership of the address/control signals changes at the end of a transfer when **HREADY** is HIGH, so the master gets access to the bus when both **HREADY** and **HGRANTMx** are HIGH.
HGRANTMx	Bus grant	Output	Lock signal from the bus master.
HLOCKMx	Locked transfers	Input	Indicates the bus master that owns the current data phase. Used by MuxM2S to remove the requirement for sequential logic within that block.
HMASTER[3:0]	Bus master	Output	**HMASTER[3:0]** indicates the master that controls the current address phase.
HMASTERD[3:0]	Bus master	Output	**HMASTERD[3:0]** indicates the master that controls the current data phase.

Note

For a description of the AMBA signals used by the arbiter, see *AMBA signals* on page 1-3.

3.3 Default slave

The default slave, `DefaultSlave`, responds to transfers that are made to undefined regions of memory, where no AHB system slaves are mapped. A zero wait OKAY response is made to IDLE or BUSY transfers, with an ERROR response being generated if a NONSEQUENTIAL or SEQUENTIAL transfer is performed. Figure 3-3 shows the default slave module block diagram.

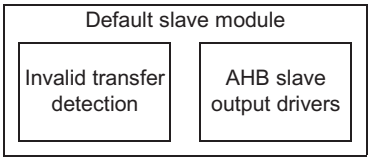


Figure 3-3 Default slave module components

3.3.1 Signal descriptions

The default slave uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

3.4 Master-to-slave multiplexor

The master-to-slave multiplexor, `MuxM2S`, connects the outputs of each AHB master to all of the AHB slaves on the bus segment. It uses the values on **HMASTER** and **HMASTERD** to select the appropriate bus master outputs, and also generates the default master outputs when no other masters are selected. The default configuration is three masters plus one dummy master. Figure 3-4 shows the master-to-slave multiplexor module block diagram.

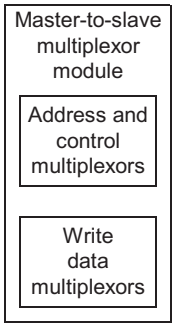


Figure 3-4 Master-to-slave multiplexor module components

3.4.1 Signal descriptions

The master-to-slave multiplexor uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

3.5 Slave-to-master multiplexor

The slave-to-master multiplexor, `MuxS2M`, connects the read data and response signals of the system bus slaves to the bus masters. It uses the current decoder **HSELx** outputs to select the bus slave outputs to use. The default configuration is for seven slaves. Figure 3-5 shows the slave-to-master module block diagram.

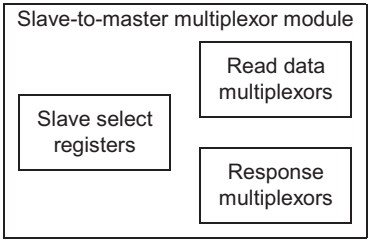


Figure 3-5 Slave-to-master multiplexor module components

3.5.1 Signal descriptions

The slave-to-master multiplexor uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

3.6 Example retry slave

The retry slave, `RetrySlave32` and `RetrySlave64`, is a rudimentary module that demonstrates how to build an AHB slave. 32-bit and 64-bit versions of the retry slave are available. The slave generates various logic functions of these registers, that can be read from different locations. Figure 3-6 shows a basic block diagram of the retry slave module system.

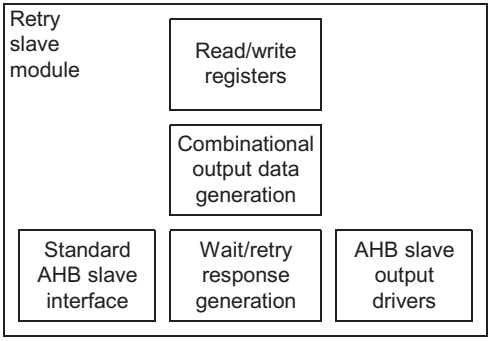


Figure 3-6 Retry slave module components

3.6.1 Signal descriptions

The retry slave uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

3.7 Example static memory interface

The *Static Memory Interface* (SMI), `SMI`, is an AMBA-compliant example design that shows the basic requirements of an *External Bus Interface* (EBI). It is not intended to be a ready-made EBI for a real system because such an EBI design has to take process, package, and varying external delays into account.

The SMI is an AMBA slave module, and connects to the AHB. It contains an AHB *Test Interface Controller* (TIC) AMBA master block that you can use to test the processor core using externally applied TIF vectors.

You can use the SMI as an interface between an AMBA AHB system bus and external, off-chip, memory devices. The SMI provides support for up to eight independently configurable memory banks simultaneously. Each memory bank is capable of supporting:

- SRAM
- ROM.

Figure 3-7 shows a block diagram of the SMI.

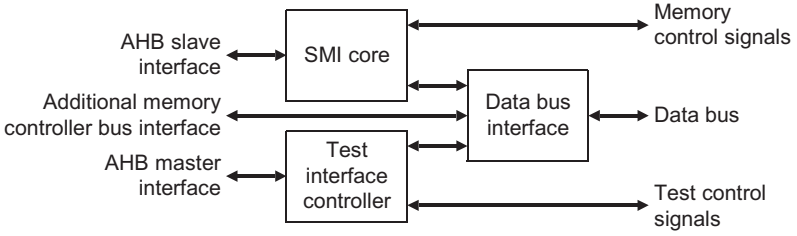


Figure 3-7 SMI components

3.7.1 SMI programmer's model

Eight memory banks are supported, with a separate chip select output for each bank. The chip select lines **SMCS[7:0]** for all banks are configurable to be either active HIGH or active LOW (default). The memory bank selection is controlled by the AMBA AHB address lines **HADDR[28:26]**, as shown in Table 3-3, where all **SMCS** are shown as active HIGH.

Table 3-3 Static memory bank select coding, Remap = 1

HADDR[28:26]	SMCS[7:0]	Memory bank (Remap 1)	Memory bank (Remap 0)
000	00000001	Bank 0	Bank 7
001	00000010	Bank 1	Bank 1
010	00000100	Bank 2	Bank 2
011	00001000	Bank 3	Bank 3
100	00010000	Bank 4	Bank 4
101	00100000	Bank 5	Bank 5
110	01000000	Bank 6	Bank 6
111	10000000	Bank 7	Bank 7

The base address of the external memory banks and the base address of the SMI memory bank registers are defined in the AMBA AHB address decoder that generates the AHB slave select signal **HSELSMC**.

3.7.2 Test interface controller

The *Test Interface Controller* (TIC) is used during testing to read external test vectors and apply them to the system through the AMBA AHB master interface. The TIC is a state machine that provides an AMBA AHB bus master for system test. It reads test write and address data from the external data bus **SMDATA**, and drives the external bus with test read data, enabling the use of only one set of output tristate buffers onto **SMDATA**.

The TIC converts externally applied test vectors into internal transfers on the AHB bus. It uses a three-wire external handshake protocol, with two inputs controlling the type of vector that is applied and a single output that indicates when the next vector can be applied.

Typically, the TIC is the highest priority AMBA bus master, and ensures test access under all conditions.

The TIC model supports address incrementing and control vectors. This means that the TIC can automatically generate the address for burst transfers.

3.7.3 TIC programmer's model

The TIC operates as a standard AHB bus master during system test when the external test pins show that the system is required to enter test mode. In this mode, the TIC requests control of the AHB and, when granted, uses the AHB to perform system tests.

Table 3-4 shows the operation of the external test pins to change the TIC mode from normal operation into test mode.

Table 3-4 Test control signals during normal operation

TESTREQA	TESTREQB	TESTACK	Description
0	-	0	Normal operation
1	-	0	Enter test mode request
-	-	1	Test mode entered

During system test the external test pins control the operation of the TIC. Table 3-5 shows the operation of these pins.

Table 3-5 Test control signals during test operation

TESTREQA	TESTREQB	TESTACK	Description
-	-	0	Current access incomplete
1	1	1	Address vector or Control vector or Turnaround vector
1	0	1	Write vector
0	1	1	Read vector
0	0	1	Exit test mode

On entry into test mode the TIC indicates that it has switched to the test clock input by asserting the **TESTACK** signal.

Test vector types

The following types of test vector are associated with the test interface:

Address vector The address for all subsequent read and write transfers is sampled by the TIC.

Write vector The TIC performs an AHB write cycle, using the write data currently driven onto the external data bus.

Read vector The TIC performs an AHB read cycle, driving the read data onto the external data bus when it becomes valid.

Control vector Internal TIC registers are set, that control the types of read and write transfers that are performed.

Turnaround vector Used between a read cycle and a write cycle to avoid clashes on the external data bus.

The address, control, and turnaround vectors are all indicated by the same value on the **TESTREQA** and **TESTREQB** signals. You can use the following rules to determine the type of vector that is being applied:

- a read vector, or burst of read vectors, is followed by two turnaround vectors
- when a single address or control vector is applied it is an address vector
- when multiple address or control vectors are applied, they are all address vectors apart from the last that is a control vector.

Control vectors

The control vector determines the types of transfer the TIC can perform, by setting the values of the **HSIZETIC**, **HPROTTIC,** and **HLOCKTIC** AHB master outputs.

The default TIC bus master transfer type is:

32-bit transfer width

HSIZETIC[2:0] signifies word transfer.

Privileged system access HPROTTIC[3:0] signifies supervisor data access, uncacheable and unbufferable.

Bit 0 of the control vector indicates if the control vector is valid. Therefore, if a control vector is applied with bit 0 LOW, the vector is ignored and does not update the control information. This mechanism enables address vectors that have bit 0 LOW to be applied for many cycles without updating the control information.

Although the default settings are sufficient for testing many systems, you can use the control vectors to change the control signals of the transfer, and also to select whether the TIC must generate fixed addresses or incrementing addresses.

Table 3-6 defines the bit positions of the control vector. The control vector bit definitions are designed to be backwards compatible with earlier versions of the TIC and therefore not all of the control bits are in obvious positions.

Table 3-6 Control vector bit definitions

Bit position	Description
0	Control vector valid
1	Reserved
2	**HSIZETIC[0]**
3	**HSIZETIC[1]**
4	**HLOCKTIC**
5	**HPROTTIC[0]**
6	**HPROTTIC[1]**
7	Address increment enable
8	Reserved
9	**HPROTTIC[2]**
10	**HPROTTIC[3]**

There is no mechanism to control the types of burst that the TIC can perform and only incrementing bursts of an undefined length are supported. The TIC only supports 8-bit, 16-bit, and 32-bit transfers. Therefore, you cannot alter **HSIZETIC[2]** and it is always LOW.

To support burst accesses using the test interface, the TIC can support incrementing of the bus address. The TIC increments eight address bits and the address range that can be covered by this incrementer depends on the size of the transfers being performed.

The control vector can enable and disable the address incrementer within the TIC. This enables burst accesses to incremental addresses, as used for testing internal RAM. Alternatively, you can disable the address increment so that successive accesses of a burst occur to the same address, as required to continually read from a single peripheral register.

The address incrementer is disabled by default and you must enable it using a control vector before use.

Note

The control vector primarily changes signals that have the same timing as the address bus. However, it also enables you to change the lock signal, that is actually required before the locked transfer commences. If the **HLOCKTIC** signal is used during testing it must be set one cycle before the transfer in which it is required. This difference in timing on the **HLOCKTIC** signal can, in some cases, cause an additional transfer to be locked both before and after the sequence that must in fact be locked.

3.7.4 Signal descriptions

Table 3-7 lists non-AMBA signals used by the SMI. A number of pins, although present, are not used on the SMI, but are reserved for backward compatibility.

Table 3-7 Signal descriptions

Signal	Type	Direction	Description
BIGENDIAN	Input	System	Reserved.
CANCELSMWAIT	Input	Input pad	Reserved.
EXTBUSMUX	Input	System	Reserved.
HADDRTIC[31:0]	Output	AMBA AHB slave	The 32-bit system address bus.
HBURSTTIC[2:0]	Output	AMBA AHB slave	Indicates if the transfer forms part of a burst. The TIC always performs incrementing bursts of unspecified length.
HBUSREQTIC	Output	AMBA AHB arbiter	A signal from the TIC to the bus arbiter to indicate that it requires the bus.
HGRANTTIC	Input	AMBA AHB arbiter	This signal indicates that the TIC is currently the highest priority master. Ownership of the address or control signals changes at the end of the transfer when **HREADYIN** is HIGH.
HLOCKTIC	Output	AMBA AHB arbiter	When HIGH, this signal indicates that the TIC requires locked access to the bus and no other master must be granted the bus until this signal is LOW.
HPROTTIC[3:0]	Output	AMBA AHB slave	The protection control signals indicate if the transfer is an opcode fetch or data access, as well as if the transfer is a Supervisor mode access or User mode access. These signals can also indicate whether the current access is cacheable or unbufferable.

Table 3-7 Signal descriptions (continued)

Signal	Type	Direction	Description
HRDATATIC[31:0]	Input	AMBA AHB slave	The read data bus transfers data from bus slaves to the bus master during test read operations.
HREADYINTIC	Input	Other AHB TIC slaves	Transfer completed input. Multiplexed **HREADY** input from all slaves on TIC AHB bus.
HRESPTIC[1:0]	Input	AMBA AHB slave	The transfer response provides additional information on the status of a transfer. The TIC supports both SPLIT and RETRY responses.
HSELREG	Input	AMBA AHB decoder	Reserved.
HSELSMC	Input	AMBA AHB decoder	Slave select signal for PrimeCell AHB SMI memory banks.
HSIZETIC[2:0]	Output	AMBA AHB slave	Transfer size signal. This signal indicates the size of the current transfer, and can be byte (8-bit), halfword (16-bit), or word (32-bit). The TIC does not support larger transfer sizes.
HTRANSTIC[1:0]	Output	AMBA AHB slave	Indicates the type of the current transfer, and can be NONSEQUENTIAL, SEQUENTIAL, or IDLE. The TIC does not use the BUSY transfer type.
HWDATATIC[31:0]	Output	AMBA AHB slave	The write data bus transfers data from the master to bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, you can easily extend this to enable higher bandwidth operation.
HWRITETIC	Output	AMBA AHB slave	Transfer direction signal. When HIGH, this signal indicates a write to the SMI and when LOW, a read from the SMI.
MCADDR[31:0]	Input	Additional memory controller	Reserved.
MCBUSGNT	Output	Additional memory controller	Reserved.
MCBUSREQ	Input	Additional memory controller	Reserved.
MCDATAEN[3:0]	Input	Additional memory controller	Reserved.

Table 3-7 Signal descriptions (continued)

Signal	Type	Direction	Description
MCDATAOUT[31:0]	Input	Additional memory controller	Reserved.
nHCLK	Input	Clock control	Reserved.
nSMBLS[3:0]	Output	Output pad	Byte lane select signals, active LOW. The signals **nSMBLS[3:0]** select byte lanes [31:24], [23:16], [15:8], and [7:0] on the data bus.
nSMDATAEN[3:0]	Output	Output pad	Tristate input/output pad enable for the byte lanes of the external memory data bus **SMDATA[31:0]**, active LOW. Enables the byte lanes [31:24], [23:16], [15:8], and [7:0] of the data bus independently.
nSMOEN	Output	Output pad	Output enable for external memory banks, active LOW.
nSMWEN	Output	Output pad	Reserved.
REMAP	Input	System	Indicates the state of the memory map: 0 = reset memory map (SMCS7 mapped to SMCS0) 1 = normal memory map.
SCANENABLE	Input	System	Dummy pin that is used as a dedicated scan enable input.
SCANINHCLK	Input	System	Dummy pin that is used as a dedicated **HCLK** scan chain input.
SCANINnHCLK	Input	System	Reserved.
SCANOUTHCLK	Output	System	Dummy pin that is used as a dedicated **HCLK** scan chain input.
SCANOUTnHCLK	Output	System	Reserved.
SMADDR[25:0]	Output	Output pad	External memory address bus, to external memory banks.
SMBUSGNTEBI	Input	External bus multiplexor	Reserved.
SMBUSREQEBI	Output	External bus multiplexor	Reserved.
SMCS[7:0]	Output	Output pad	Chip select for external memory banks 7 to 0. The default is active LOW.
SMDATAIN[31:0]	Input	Input pad	External input data bus used to read data from memory bank.
SMDATAOUT[31:0]	Input	Input pad	External output data used to write data from **SMI** to memory bank.

Table 3-7 Signal descriptions (continued)

Signal	Type	Direction	Description
SMMWCS7[1:0]	Input	Input pad	Reserved.
SMWAIT	Input	Input pad	Reserved.
TBUSOUTEBI[31:0]	Output	External bus multiplexor	Reserved.
TESTACK	Output	Output pad	The test bus acknowledge signal gives external indication that the TIC is granted and also indicates when a test access is complete. When **TESTACK** is LOW, the current test vector must be extended until **TESTACK** becomes HIGH.
TESTREQA	Input	Input pad	This is the Test Bus Request A input signal and is required as a dedicated device pin. During normal system operation, the **TESTREQA** signal requests entry into the test mode. During test **TESTREQA**, in combination with **TESTREQB**, indicates the type of test vector that is applied in the following cycle.
TESTREQB	Input	Input pad	During test this signal, in combination with **TESTREQA**, indicates the type of test vector that is to be applied in the following cycle.
TICBUSGNTEBI	Input	External bus multiplexor	Reserved.
TICBUSREQEBI	Output	External bus multiplexor	Reserved.
TICREADEBI	Output	External bus multiplexor	Reserved.

Note

For a description of the AMBA signals used by the SMI, see *AMBA signals* on page 1-3.

3.8 Bus matrix

This section describes the Bus Matrix configurable features and operation. The section contains the following sections:

- *Key features*
- *Bus Matrix configurability* on page 3-21
- *Relationship between the AMBA Designer and Bus Matrix* on page 3-21
- *BusMatrix module* on page 3-22
- *Operation* on page 3-25
- *Programmer's model* on page 3-26
- *Block functionality* on page 3-27
- *Arbitration and locked transfers* on page 3-29
- *Address map* on page 3-30
- *Signal descriptions* on page 3-34.

3.8.1 Key features

The Bus Matrix has the following key features:

- Number of slave ports between 1 to 16.
- Number of master ports between 1 to 16.
- Routing data width, a choice of 32 or 64 bits.
- Routing address width - between 32 to 64 bits.
- Architecture type, AHB and ARM11 extensions:
  - AHB2, support an AHB 2.0 interface, the default
  - V6, support all ARM11 AHB extensions
  - Excl(usive), support the ARM11 exclusive access extensions only
  - Unalign - support the ARM11 unaligned access extensions only.
- Arbiter type, choice of round robin, fixed and burst.
- Default slave included with each slave port.
- Optional **xUSER** signals, between 0 to 32 bits, with zero meaning excluded.
- Sparse connectivity:
  - The sparse connectivity feature removes any un-necessary connections, thereby reducing area and multiplexer delays.

  - Separate instances of output stage and output arbiter will be generated for each master port.
  - For input-output stages with only one sparse connection, the choice of arbiter is overridden with single arbiter and output stage modules. These single modules also permit 1xn interconnects.
- Design entry by command line
  - Calculated address map, but excluding REMAP support.
- Design entry by AMBA Designer
  - Provides a *Graphical User Interface* (GUI) method to configure the BusMatrix
  - Enables user specified address map, including REMAP support
- User specified module names or automatically derived top-level name
- User specified source and target directories
- Optional `timescale Verilog directives.

3.8.2 Bus Matrix configurability

The Bus Matrix is a configurable component that enables multiple AHB masters to be connected to multiple AHB slaves. The Bus Matrix RTL is generated automatically through the use of the `BuildBusMatrix.pl` script. The script takes in different configuration parameters, for example, the number of masters, number of slaves, and data-width, and then generates the corresponding Verilog RTL.

AMBA Designer provides a flexible method of design entry. The AMBA Designer configuration method enables you to specify all design parameters in addition to fully configurable address map information. The various design parameters are explained in more details in the *AMBA Design Kit User Guide.*

3.8.3 Relationship between the AMBA Designer and Bus Matrix

AMBA Designer is a configuration tool that generates a specific implementation of a Bus Matrix. AMBA Designer drives the Bus Matrix generation engine to provide the RTL for a set of configuration parameters.

The generated files come from two sources:

- the AMBA Designer tool
- the Bus Matrix.

The two products are designed to work together in the following manner:

Configuration file

The AMBA Designer tool is responsible for setting the user entered design parameters. AMBA Designer then automatically launches the BusMatrix build engine, which is responsible for generating the Verilog files of the configuration.

Note

There is no Verilog testbench available for the Bus Matrix.

There are no test vectors available for the Bus Matrix.

Bus Matrix and AMBA Designer documentation

The Bus Matrix and AMBA Designer documentation suites are designed to be used together to describe the principles of the Bus Matrix and the actual configuration options. There is no duplication between the two sets of documentation.

ADK

The AMBA Design Kit Bus Matrix documentation, that is, this section and the *AMBA Design Kit User Guide* describe:

- the functionality of the component
- example use of the `BuildBusMatrix.pl` script in the *AMBA Design Kit User Guide*
- the effect of the configuration options set in AMBA Designer.

AMBA Designer

The *AMBA Designer (FD001) User Guide* describes:

- how to install AMBA Designer
- how to produce an example interconnect
- how to generate the RTL
- the address map, configuration options, ranges, and default values.

3.8.4 BusMatrix module

The BusMatrix module, `BusMatrix`, enables multiple AHB masters from different AHB buses to be connected to multiple AHB slaves on multiple AHB slave buses. It enables parallel access to a number of shared AHB slaves from a number of different AHB

masters. The Bus Matrix determines the master that gains access to each slave, and routes the control signals and data signals between them. This block is required in multi-layer AHB systems.

Figure 3-8 on page 3-24 shows a block diagram of the BusMatrix module.

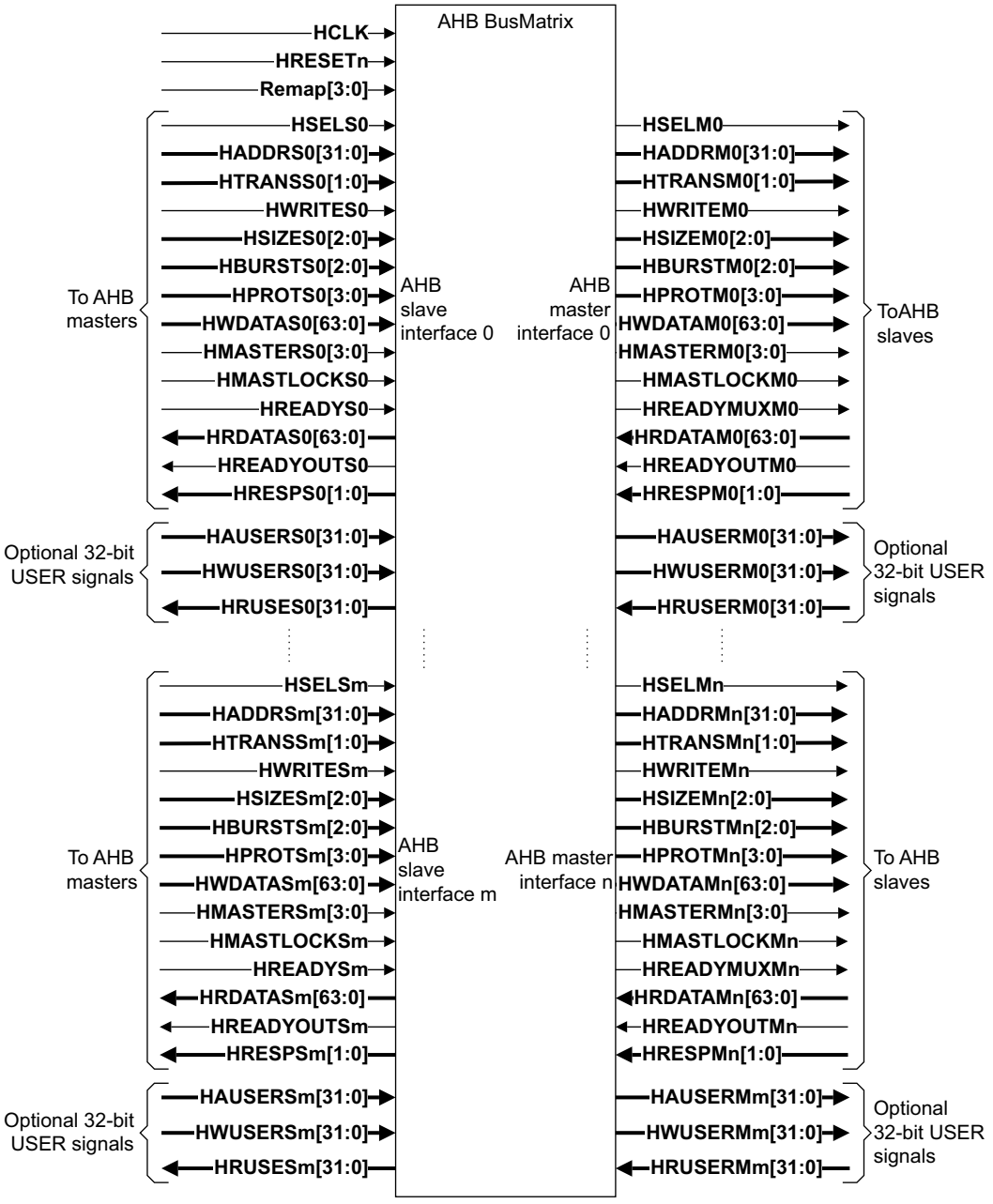


Figure 3-8 BusMatrix module components

Figure 3-8 on page 3-24 shows a BusMatrix module components block diagram for (m+1) input ports, (n+1) output ports, 64-bit routing data width, 32-bit routing address width and 32-bit user signals width.

The Bus Matrix signal names have suffixes for port and pin naming:

- signals on the AHB slave interface coming from AHB masters have the suffix S
- signals on the AHB master interface going to AHB slaves have the suffix **M**.

The Bus Matrix connects to the masters and slaves using this naming scheme, with an additional integer to identify the correct master and slave. For example, connect **HWDATAS0[63:0]** to the 64-bit AHB Master 0 write data port, and **HWDATAM0[63:0]** to the AHB Slave 0 write data port.

Note

If the Bus Matrix is configured using AMBA Designer, the signal names get appended with their associated interface names as a suffix. So for consistency, it is recommended to name the slave interfaces with the letter S plus an identifying integer and the master interfaces with the letter M plus an identifying integer.

3.8.5 Operation

The following sections describe the operation of the Bus Matrix:

- *Integrating the Bus Matrix*
- *Locked sequences* on page 3-26
- *Full AHB and AHB-Lite* on page 3-26.

Integrating the Bus Matrix

When integrating the Bus Matrix component:

- The input ports, with signal suffix **S**, are AHB slave ports, and must be connected accordingly.
- The output ports, with signal suffix **M**, are AHB *slave gasket* ports. That is, they are designed to be attached directly to an AHB slave port, that they mirror, and must not be treated as full AHB master ports.
- If the output from a Bus Matrix must be used as a bus master on a further AHB layer, it is recommended that a component such as an AHB bridge, for example, the ADK Ahb2AhbPass component, is used.

Note

When connecting to an output port on the Bus Matrix, the **HSEL** pin must be connected to the attached slave even if there is only one slave present. If this is not done, the slave might see spurious transfers under certain circumstances.

Locked sequences

The Bus Matrix is only designed to support locked sequences that target a single output port. Because of this, a snooping bus across all input ports is not required. This provides arbitration for locked transfers on all layers simultaneously. In addition, the Bus Matrix is not designed to cope with a SPLIT response to a locked transfer. If this occurs, the Bus Matrix correctly passes the SPLIT response back to the initiating master, but it might then enable another master, connected to a different input port, to access the output port targeted by the first master.

Full AHB and AHB-Lite

The Bus Matrix inherently supports both full AHB and AHB-Lite systems. However, you must take care with SPLIT responses. The Bus Matrix correctly passes back a SPLIT response, but then relies on an arbiter on the AHB layer connected to the relevant input port to ensure that the initiating master is degranted until unsplit by the slave.

3.8.6 Programmer's model

The design of the Bus Matrix can be divided into input stage, decode stage, and output stage as described in *Block functionality* on page 3-27. Figure 3-9 on page 3-27 shows an Bus Matrix design with:

- four slave ports, for connection to bus masters
- three master ports, for connection to slaves.

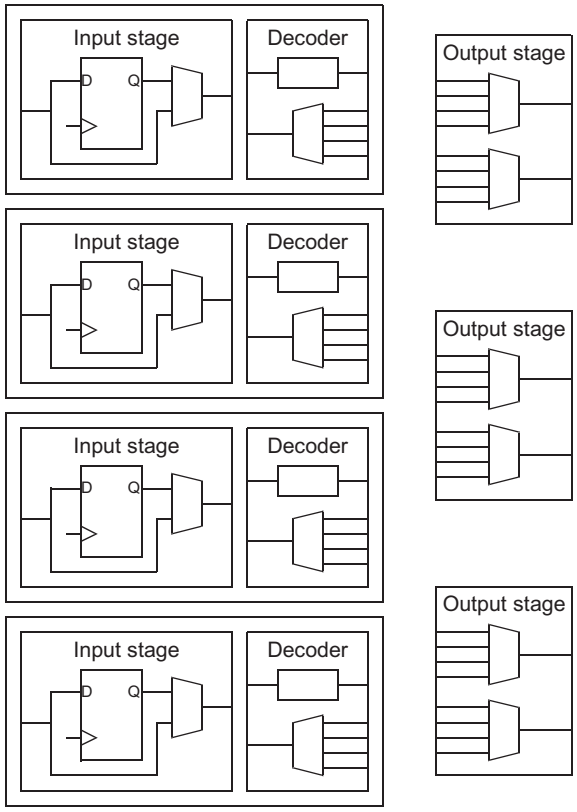


Figure 3-9 Example Bus Matrix design configuration

3.8.7 Block functionality

Functionality of the BusMatrix module is described in the following sections:

- *Input stage*
- *Decode stage* on page 3-28
- *Output stage* on page 3-28.

Input stage

The input stage provides the following functions:

- It registers and holds an incoming transfer if the receiving slave is not able to accept the transfer immediately.

- If the Bus Matrix switches between input ports while in the middle of an undefined length burst, the input stage modifies the **HTRANS** and **HBURST** signals for the interrupted input port, so that when it is reinstated, the remaining transfers in the burst meet the AHB specification.

Decode stage

The decode-stage generates the select signal for individual slaves. It also handles the multiplexing of response signals and read data. During the address phase of a transfer, the decoder asserts the slave-select signal for the appropriate output stage corresponding to the address of the transfer. In addition the decoder routes an Active signal from the output stage back to the input stage. This signal indicates to the input that its address is currently being driven onto the chosen slave. During the data phase of a transfer the decoder routes the response signals and Read data back to the input port.

Each slave port, connected to an AHB master, is associated with a separate decoder. This enables the AHB masters to have independent address maps, that is a shared slave does not require to appear in the same address location for all masters. This is typically useful for multi-processor systems.

Any gaps in the memory map are redirected to a default slave, which returns an OKAY or ERROR response depending upon the type of access. There is an instance of a default slave associated with each decoder.

The decoder stage also supports the system address Remap function. A 4-bit Remap control signal connects to the decoder. Remapping might be used to change the address of physical memory or a device after the application has started executing. This is typically done to permit RAM to replace ROM when the initialization has been completed.

In multi-layer AHB systems that have local slaves on some of the AHB layers, the address decoding is performed in two stages. The first address decoder selects between local slaves and the shared slaves available through the AHB BusMatrix module. To support this, the decode-stage within the BusMatrix includes an **HSEL** input that indicates if the address from an input port is destined for a shared slave.

Output stage

The output stage has the following functions:

- selects the address and control signals from the input stages
- selects the corresponding write data from the input stage
- determines when to switch between input ports in the input stage.

The output stage only selects an input source when that input has a transfer in the holding register.

The output stage generates an active signal for each input port when the address from that input port is being driven onto the slave. This signal enables the input stage to determine when transfers from other masters must be held up because the slave is not currently available.

When a sequence of transfers to a shared slave has finished and there are no more transfers to the slave required by any of the input ports, the output stage switches the address and control signals to an idle state.

3.8.8 Arbitration and locked transfers

This section describes arbitration and locked transfers.

Arbitration

The arbitration within the BusMatrix module determines the input port that has access to the shared slave and each shared slave has its own arbitration. Different arbitration schemes provide different system characteristics in terms of access latency and overall system performance.

The slave switch supports the following arbitration schemes:

Fixed arbitration

One port always has highest priority and the order of priority for all other ports is fixed.

A burst transfer can be broken up if a higher-priority master requests the same slave, except where the burst transfer is a locked transfer.

Fixed (burst) arbitration

This is similar to fixed arbitration but it does not break defined length burst transfers and it is default arbitration for the Bus Matrix.

Round robin arbitration

Arbitration is performed during every active clock cycle, indicated by **HREADYM**. Priority initially goes to the lowest-numbered requestor, that is input port 0. When multiple requests are active, priority goes to the next lowest-numbered requestor compared to the currently active one. Fixed-length bursts are not broken. The arbitration waits for the end of the burst before passing control to the next requestor, if there is one.

INCR bursts are treated as four-beat bursts, to optimize memory accesses, with guard logic to ensure that a sequence of short **INCR** bursts does not freeze the arbitration scheme.

Locked transfers

Using a multi-layer AHB system requires certain restrictions to be placed on the use of locked transfers to prevent a system deadlock. It is required that a sequence of locked transfers is performed to the same slave within the system. Because the minimum address space that can be allocated to a single slave is 1KB, a bus master can ensure this restriction is met by ensuring that it does not perform a locked sequence of transfers over a 1KB boundary, ensuring that it never crosses an address decode boundary.

Therefore, if a bus master is to perform two locked transfer sequences to different address regions, the bus master must not start the second locked transfer sequence until the final data phase of the first locked transfer sequence has completed.

3.8.9 Address map

If the AMBA Designer configuration method is not used and command line parameters are used instead, then the address map gets calculated automatically as follows:

- Figure 3-10 shows the equations that enable the address map to be divided into a number of regions depending on the number of master ports:

$$\text{regions} = \text{round\_to\_highest\_2toN}(\text{total\_master\_ports})$$

$$\text{region\_size} = \frac{2^{\text{routing\_address\_width}}}{\text{regions}}$$

$$\text{region\_base} = \text{region\_size} \times \text{master\_port\_instance}$$

$$\text{region\_top} = \text{region\_base} + \text{region\_size} - 1$$

Figure 3-10 Region equations

- Each slave port has the same decoder instance.
- No Remap support.

The decode-stage can have a fully customized address map when using the AMBA Designer configuration method and each slave port can have and independent view of the address space. Example 3-1 on page 3-31 shows a slave port address map description. See the *AMBA Design Kit User Guide* for parameter descriptions.

Example 3-1

```
<slave_interface name="SI1">
    <address_region interface="MI0" mem_lo="40000000" mem_hi="4fffffff" remapping="move"/>
    <address_region interface="MI0" mem_lo="70000000" mem_hi="7fffffff" remapping="alias"/>
    <address_region interface="MI1" mem_lo="80000000" mem_hi="9fffffff" remapping="none"/>
    <address_region interface="MI2" mem_lo="a0000000" mem_hi="bfffffff" remapping="move"/>
    <address_region interface="MI3" mem_lo="00000000" mem_hi="1fffffff" remapping="move"/>

    <remap_region   interface="MI0" mem_lo="00000000" mem_hi="1fffffff" bit="0"/>
    <remap_region   interface="MI1" mem_lo="50000000" mem_hi="5fffffff" bit="0"/>
    <remap_region   interface="MI2" mem_lo="60000000" mem_hi="6fffffff" bit="1"/>
    <remap_region   interface="MI3" mem_lo="c0000000" mem_hi="dfffffff" bit="0"/>

</slave_interface>
```

Address region

The address region parameters determine the routing of transactions to the master interfaces. Each master interface can have multiple non-contiguous address regions, when multiple sets of address region parameters are defined. However, the address regions of different master interfaces must not overlap. The `mem_lo` parameter defines the lower bound address and the `mem_hi` parameter defines the upper bound address for the master interface.

The `remapping` configuration parameter defines the behavior of master interfaces that support address remapping. It becomes active when the relevant REMAP bit is set. The following types of address remapping behavior exist:

- If the `remapping` parameter is set to alias or none, the remapping creates an alias of the defined region in the new address space.
- If the `remapping` parameter is set to move, the address region gets removed from the original address space and master interface appears at the location defined by the remap region in the new address space

Remap region

These regions get activated when using the remap facility and each remap region is associated with a bit of the **REMAP** signal.

When the relevant remap bit is set, the remap regions take higher priority than normal address regions for the same master interface. Also any normal regions that have the remapping parameter set to 'move', get removed from the address space. If more than one bit is asserted for the same master interface, the least significant bit takes priority.

Figure 3-11 on page 3-33 shows the address map of the slave interface, defined in the above example, at different remap states.

The address map is explained at the remap state `REMAP = 0001`:

- In normal address map MI0 appears at two non-contiguous regions `0x40000000` and `0x70000000`. When remap bit 0 is set the `0x40000000` region was removed because the remapping parameter is set to 'move' and MI0 appears at the new remap region `0x00000000` to `0x1FFFFFFF`. The MI0 region at `0x70000000` is not removed because its remapping parameter is declared as alias.
- When remap bit 0 is set, MI1 appears at the new remap region `0x50000000` to `0x5FFFFFFF` and at the region at `0x80000000` did not get changed because its remapping is set to 'none'.
- MI2 did not change even though its remapping is declared as move, because it is associated with remap bit 1 which is not set at the current remap state
- MI3 is moved to a new base address `0xC0000000`.

Note

MI0 can be considered as ROM and MI3 can be considered as RAM. At boot time, the **REMAP** signal is set to 0001 and ROM can be seen at base address `0x00000000`. After booting the **REMAP** signal is set to 0000, the RAM now appears at `0x00000000` and the ROM is moved up in the address space to `0x40000000`.

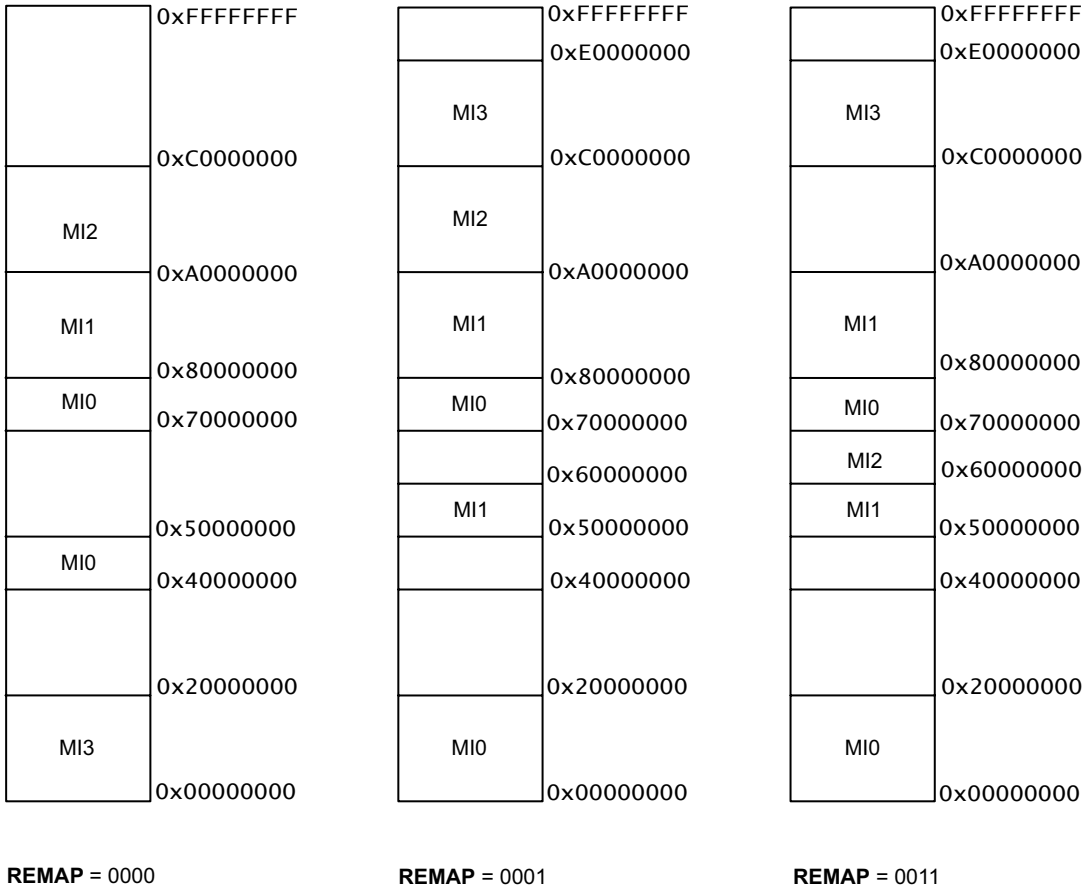


Figure 3-11 Address map at different remap states

3.8.10 Signal descriptions

Table 3-8 lists the signal list for the Bus Matrix.

Table 3-8 Bus Matrix signals

Signal	Direction	Description
HCLK	Input	System bus clock. Logic is triggered on clock rising edge.
HRESETn	Input	Activate low asynchronous reset.
System address control		
REMAP[3:0]	Input	System address remap control.
Interface to masters (AHB slave)		
HADDRSx[N]	Input	N-bit address bus from AHB master. N can be in the range [31 to 63].
HBURSTSx[2:0]	Input	Burst size information.
HMASTERSx[3:0]	Input	Current active master.
HMASTLOCKSx	Input	Indicate the transfer on the master AHB is a locked transfer.
HPROTSx[3:0]	Input	Protection information.
HRDATASx[63:0 or **31:0]**	Output	Read data to bus master. Width configurable to be either 64-bit or 32-bit wide.
HREADYOUTSx	Output	**HREADY** signal feedback to the master bus, indicating if the AHB BusMatrix module is ready for next operation.
HREADYSx	Input	**HREADY** signal on the master AHB bus, indicating start/ending of transfer.
HRESPSx[1:0]	Output	Response from AHB BusMatrix module to AHB master. Width depends on architecture choice.
HSELSx	Input	Active HIGH select signal to indicate shared slave connected to the AHB BusMatrix module is selected.
HSIZESx[2:0]	Input	Size of the data.
HWDATASx[63:0 or **31:0]**	Input	Write data from AHB masters. Width configurable to be either 64-bit or 32-bit wide.
HWRITESx	Input	Indication of WRITE/READ operation.
Interface to slaves (AHB master)		

Table 3-8 Bus Matrix signals (continued)

Signal	Direction	Description
HADDRMx[N]	Output	N-bit address bus for AHB slave. N can be in the range [31 to 63].
HBURSTMx[2:0]	Output	Burst size information.
HMASTERMx[3:0]	Output	Current active master.
HMASTLOCKMx	Output	Indicates the transfer on the slave AHB is a locked transfer.
HPROTMx[3:0]	Output	Protection information.
HRDATAMx[63:0 or **31:0]**	Input	Data read back from AHB slave(s). Width configurable to be either 64-bit or 32-bit wide.
HREADYOUTMx	Input	**HREADY** from AHB slave or slave multiplexor.
HREADYMUXMx	Output	**HREADY** feed back to all slaves on slave AHB.
HRESPMx[2:0]	Input	**HRESP** from AHB slave or slave multiplexor. Width depends on architecture choice.
HSELMx	Output	Active HIGH select signal to indicate slave bus is accessed. You can use this signal to drive a single AHB slave directly, or drive a secondary AHB decoder if multiple AHB slaves are used.
HSIZEMx[2:0]	Output	Size of the data.
HWDATAMx[63:0 or **31:0]**	Output	Write data to AHB slave(s). Width configurable to be either 64-bit or 32-bit wide.
HWRITEMx	Output	Indicates write/read operation.
User signals		
HAUSERSx	Input	Additional sideband bus that has same the timing as the slave interface address payload signals.
HWUSERSx	Input	Additional sideband bus that has the same timing as the slave interface write data payload signals.
HRUSERSx	Output	Additional sideband bus that has the same timing as the slave interface read data payload signals.

Table 3-8 Bus Matrix signals (continued)

Signal	Direction	Description
HAUSERMx	Output	Additional sideband bus that has the same timing as the master interface address payload signals.
HWUSERMx	Output	Additional sideband bus that has the same timing as the master interface write data payload signals.
HRUSERMx	Input	Additional sideband bus that has the same timing as the master interface read data payload signals.

User signals

The Bus Matrix supports USER signals on master and slave interfaces. These signals are optional, and a value of zero on the `user_signal_width` parameter removes them from the generated Verilog. If the `user_signal_width` parameter or the `--userwidth` command line switch is set to a non-zero value, that value defines the width of those USER signals. The USER signals have the same timing as the payload signals for that channel. For example, the **HAUSER** signals have the same timing as the address payload signals.

The USER signals are:

- **HAUSER**
- **HRUSER**
- **HWUSER**.

N-bit addressing

The Bus Matrix supports N-bit addressing, that is, you can configure the address bus to be in the range of 32 bits up to 64 bits. By default the address width is set to 32 bits, but you can change this by supplying the `--addrwidth` command line switch or by changing the `routing_address_width` global parameter in AMBA Designer. Setting the address width affects both the slave ports and master ports address buses.

Note

The presence of USER signals and the support of N-bit addressing enables the Bus Matrix to fully connect to the AXI High Performance Matrix. The bus matrix user signals are fully mapped to their AXI counterparts. This is typically useful in mixed protocol designs. See the *PrimeCell High-Performance Matrix PL301 Technical Reference Manual* for more information.

3.9 System decoder

The system decoder, `Decoder`, decodes the address bus and generates select lines to each of the system bus slaves, indicating that a read or write access to that slave is required. The default configuration is 16 slots. Figure 3-12 shows the decoder module block diagram.

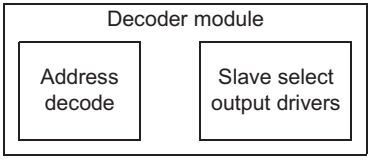


Figure 3-12 Decoder module components

3.9.1 Programmer's model

This section of code defines the memory map for the whole system. If modules are added, removed, or moved to new locations, you must modify the code to match these system changes, ensuring that the correct slave is selected for each address used.

The decoder controls the memory map of the system, and generates a slave select signal for each memory region. The **REMAP** signal provides a different memory map at reset, when ROM is required at address `0x00000000`, and during normal operation, when internal RAM can be used at address `0x00000000`.

The **REMAP** signal is typically provided by a remap and pause peripheral, that drives **REMAP** LOW at reset. The signal is driven HIGH only after a particular address in the remap and pause peripheral is accessed.

Figure 3-13 on page 3-38 shows both the normal and reset memory maps.

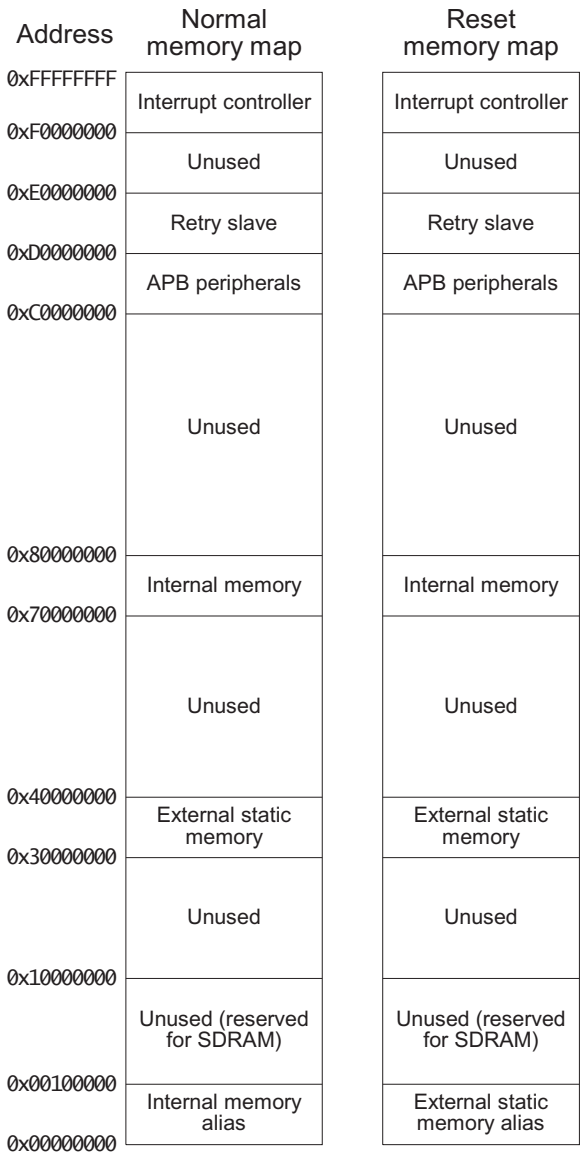


Figure 3-13 System memory map

3.9.2 Signal descriptions

Table 3-9 lists non-AMBA signals used by the decoder module.

Table 3-9 Decoder module signals

Signal	Type	Direction	Description
REMAP	Reset memory map	Input	When LOW, the internal memory is not part of the system memory map, and external memory is mapped from address `0x00000000` that contains the system startup code. In normal operation, this signal is HIGH, permitting use of the internal memory.

Note

For a description of the AMBA signals used by the decoder, see *AMBA signals* on page 1-3.

3.10 APB bridge

The AHB-APB bridge is an AHB slave, providing an interface between the high-speed AHB domain and the low-power APB domain. Read and write transfers on the AHB are converted into corresponding transfers on the APB through a master interface. Because the APB is not pipelined, wait states are added during transfers to and from the APB when the AHB is required to wait for the APB protocol. The default configuration is 16 slots.

In this release of ADK there are two versions of the APB bridge.

3.10.1 Ahb2Apb bridge

The Ahb2Apb bridge consists of the following:

- It is the original ADK bridge.
- Instanced within EASY systems, and is used in those examples.
- Supports APB 2.0 only.
- Always returns an OKAY response.
- Peripheral slot size decoded from **HADDR[27:24]**, and is 16MB.
- Uses `MuxP2B` peripheral to bridge multiplexor.

Figure 3-14 shows the `Ahb2Apb` bridge module diagram.

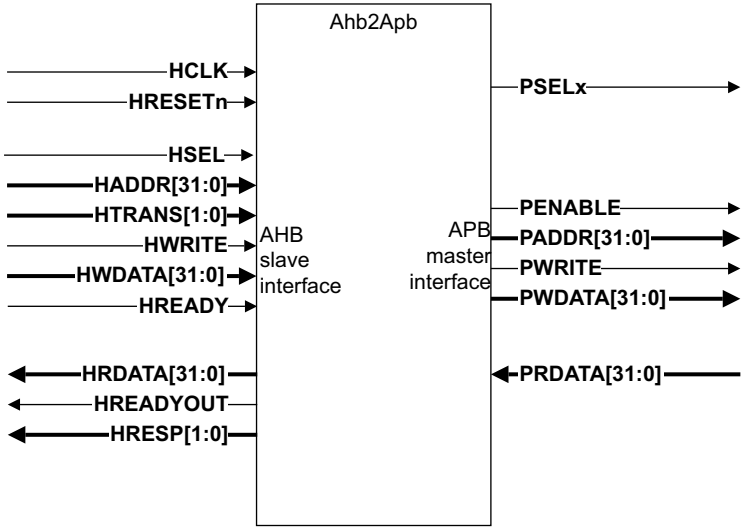


Figure 3-14 Ahb2Apb bridge module

This subsection describes:

- *Programmer's model*
- *Signal descriptions*.

Programmer's model

The Aph2Apb bridge controls the memory map for the peripherals, and generates a select signal for each peripheral. Figure 3-15 shows the default system memory map used within the EASY example systems.

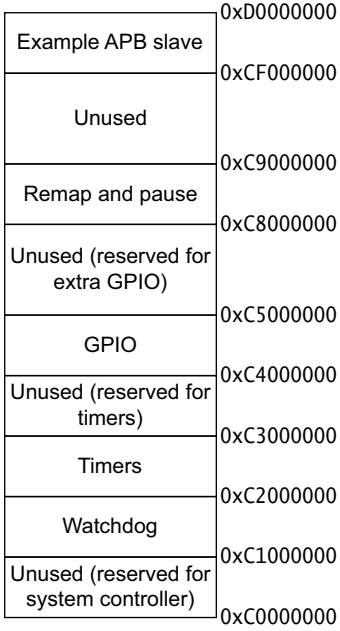


Figure 3-15 Allocation of APB memory slots within EASY systems

Signal descriptions

The Aph2Apb bridge uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

Note

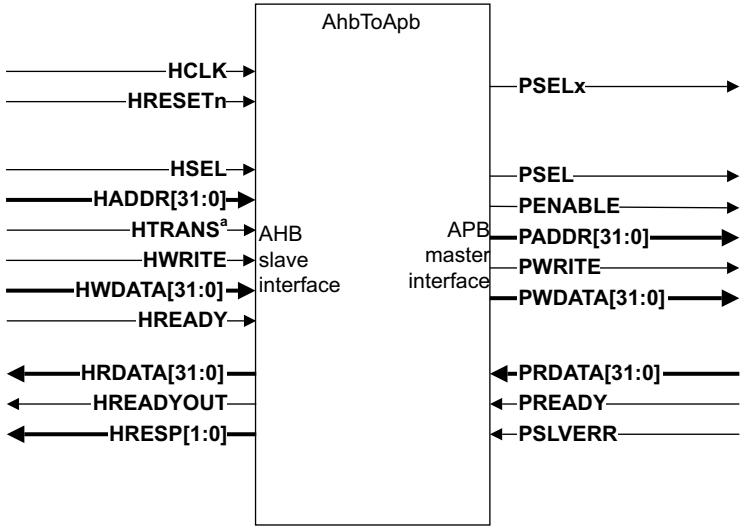
Timing diagrams showing the relationship between AHB and APB transfers are in the *APB Specification*.

3.10.2 AhbToAPB bridge

The AhbToApb bridge consists of the following:

- New for ADK r3p0.
- Not instanced anywhere else within the ADK in this edition.
- Supports both APB 2.0 and 3.0.
- Has new signals **PSEL**, **PREADY** and **PSLVERR**.
- Can return OKAY and ERROR responses.
- Peripheral slot size decoded from **HADDR[15:12]**, and is 4KB.
- Uses the new `MuxPToB` peripheral to bridge multiplexor.

Figure 3-16 shows the `AhbToApb` bridge module block diagram.



a. You must connect signal **HTRANS** to **HTRANS[1]** on the AHB slave interface. **HTRANS[0]** on the AHB slave interface is not required as input.

Figure 3-16 AhbToApb bridge module

Signal descriptions

The AhbToApb bridge uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

Table 3-10 describes the extra signals declared in the AhbToApb bridge including the APB 3.0 signals.

Table 3-10 AhbToAPB Bridge signals

Signal	Direction	Description
PSEL	Output	Select. It indicates that an APB slave is selected and a data transfer is required. **PSEL** is a combined select output, when HIGH indicates that an APB slave is selected and that a data transfer is required. You can use this signal as an input to an external address decoder.
PREADY	Input	Ready. The APB slave uses this signal to extend an APB transfer.
PSLVERR	Input	This signal indicates a transfer failure and is driven by the APB peripheral.

Note

Timing diagrams showing the relationship between AHB and APB 3.0 transfers are in the *AMBA 3 APB Protocol Specification*.

3.11 Example bus master

The *Example Bus Master* (EBM), `EgMaster32` and `EgMaster64`, consists of:

- an AHB-Lite core, that generates a fixed set of transfers in a continuous way
- an AHB-Lite to AHB wrapper, that enables its connection to a standard AHB bus.

This provides an example of both AHB-Lite bus master design and the use of the wrapper to interface to an AHB system. You can also use this in the ADK system, in place of an ARM processor model, to enable rapid simulation of AHB transactions.

The block is supplied as both 32-bit and 64-bit versions. Figure 3-17 shows the EBM module.

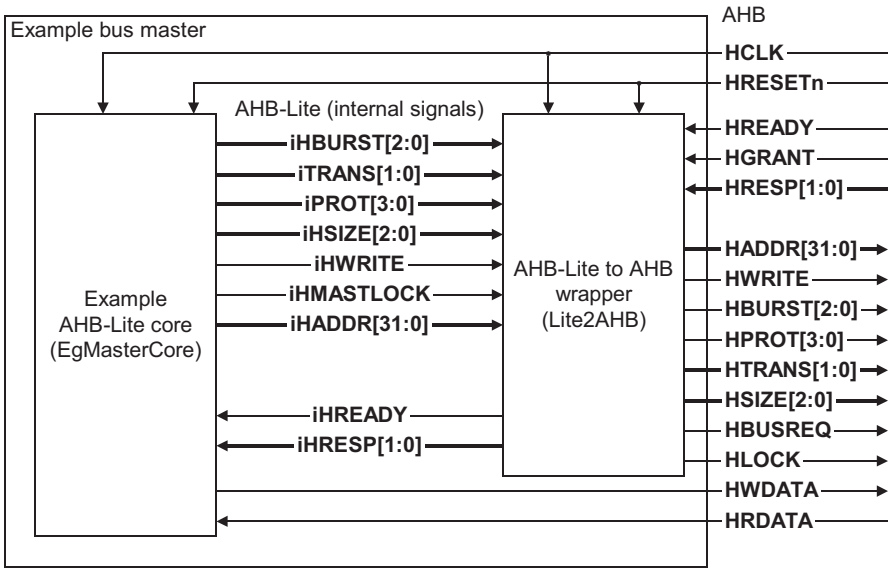


Figure 3-17 EBM module components

3.11.1 Programmer's model

Programming details for the EBM are described in the following sections:

- *Example AHB-Lite core* on page 3-45
- *Configurable options* on page 3-46
- *Endianness* on page 3-46.

Example AHB-Lite core

Figure 3-18 shows the example AHB-Lite core.

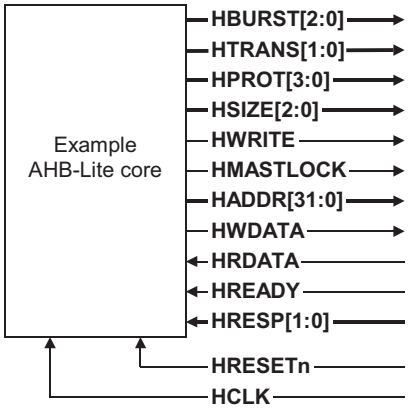


Figure 3-18 Example AHB-Lite core

The core is completely synchronous with the AHB bus clock signal, **HCLK**, and is reset by the AHB reset signal, **HRESETn**. Read transfers are not locked, but **HMASTLOCK** is asserted HIGH to lock write transfers. The **HREADY** signal stalls the core.

In AHB-Lite, SPLIT and RETRY responses are not supported, so only **HRESP[0]** must be decoded and used in the core. However, it is safer to decode the full **HRESP[1:0]** signal. When this signal indicates an ERROR response during a read operation, the corresponding data is ignored.

The core consists of a counter, burst generation logic, and a 4x32 or 64-bit register bank. From a power-up or reset state, it uses the counter to wait for a predefined number of clock cycles. The core then performs a 4-beat, incrementing, read burst from a parameterized base source address into the register bank. When the burst is complete, the bus master waits for further predefined time, before writing data from the registers as 32 single byte transfers. This data is written, least significant byte first, to a single non-incrementing destination address. This process continues, using the same source and destination addresses, until a power-down or reset condition is reached.

Configurable options

The EBM has a fixed functionality and is not programmable. There are, however, compilation-time or simulation-time options, using Verilog parameters. Table 3-11 lists the configurable options.

Table 3-11 Configurable options

Parameter	Type	Default value	Description
EBMenable	Boolean	'1' (True)	When set to 0 (false), this parameter prevents the master from generating any transfers
EBMreadAddr	8-bit vector	`0xD0`	Bits [31:24] of the base address for the read burst, dword aligned
EBMwriteAddr	8-bit vector	`0xC4`	Bits [31:24] of the base address for the write transfers, dword aligned
EBMinitCount	10-bit vector	`0x004`	The number of IDLE transactions between bursts

Endianness

The EBM supports little-endian mode only.

3.11.2 Signal descriptions

The EBM uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

3.12 Synchronous AHB to AHB bridge

The AHB-AHB bridges (`Ahb2Ahb32`, `Ahb2Ahb64`, `Ahb2AhbPass32`, `Ahb2AhbPass64`, `Ahb2AhbSyncDn32`, `Ahb2AhbSyncDn64`, `Ahb2AhbSyncUp32`, and `Ahb2AhbSyncUp64`) provide a unidirectional link between two AHB domains. They enable a master to access a slave on another bus, with the transfer initiated from one side only. These bridges enable various synchronous clocking schemes to be implemented between the AHB buses. Each bridge is implemented with an AHB slave interface and an AHB-Lite master interface. These are packaged with an AHB-Lite to AHB master gasket if full AHB master support is required. An asynchronous bridge is also available (see *Asynchronous AHB-AHB bridge* on page 3-60).

The synchronous bridges have the following features:

- 32 or 64-bit data bus
- bursts are preserved across the bridge, although subject to override if the bridge master is degranted
- transfer sequences can be locked across the bridge
- SPLIT and RETRY responses are serviced by the bridge, so remain local to the issuing slave
- fully registered designs, except for Ahb2AhbPass
- in-burst pre-emptive address generation, to reduce latency on read transfers
- read transfers incur a minimum of one wait state
- buffered writes are zero wait-state minimum, nonbufferable writes incur a minimum of two wait-states
- modular design to facilitate the removal of logic that is not essential for a particular application.

3.12.1 Bridge designations

The following synchronous AHB-AHB bridges are described in this section:

- *Ahb2Ahb (1:1)* on page 3-48
- *Ahb2AhbPass (1:1)* on page 3-49
- *Ahb2AhbSyncDn (N:1)* on page 3-48
- *Ahb2AhbSyncUp (1:N)* on page 3-48.

Ahb2Ahb (1:1)

This is a fully registered bridge for connecting AHB buses that share a common clock. This bridge includes one level of write buffering to enable zero wait-state write transfers across it. Figure 3-19 shows the Ahb2Ahb bridge.

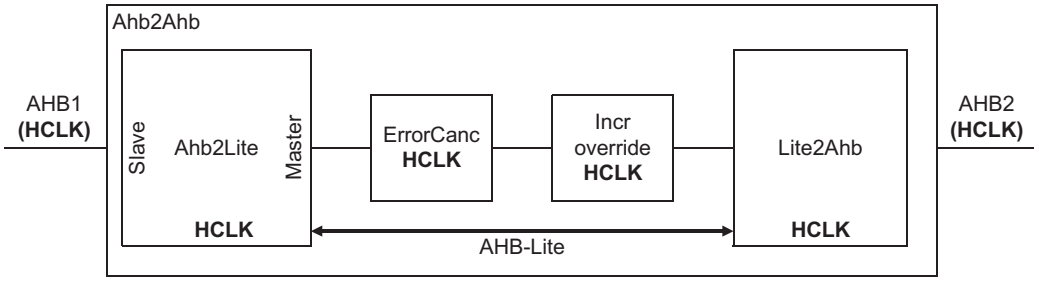


Figure 3-19 Ahb2Ahb bridge

Ahb2AhbSyncDn (N:1)

This bridge connects buses running at different, synchronous frequencies, where clocks share concurrent edges, and where the master is clocked at the same or a higher frequency than the slave.

Figure 3-20 shows the Ahb2AhbSyncDn bridge.

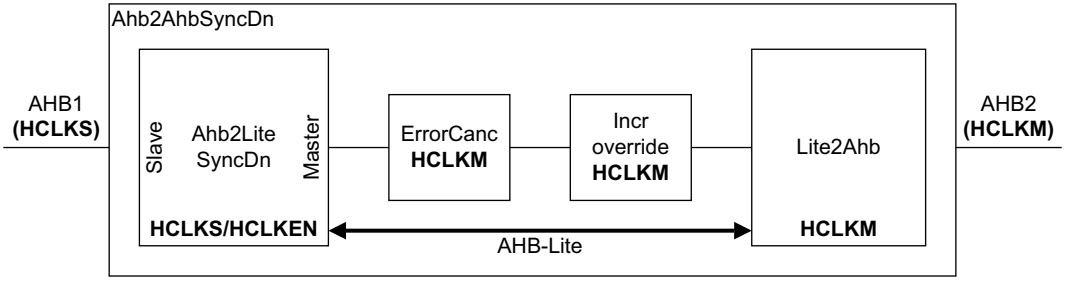


Figure 3-20 Ahb2AhbSyncDn bridge

Ahb2AhbSyncUp (1:N)

This bridge connects buses running at different, synchronous frequencies, where clocks share concurrent edges, and where the master is clocked at the same or a lower frequency than the slave. Figure 3-21 on page 3-49 shows the Ahb2AhbSyncUp bridge.

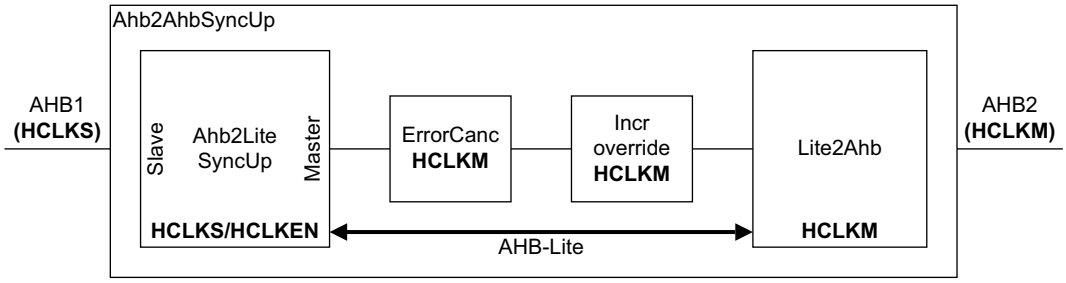


Figure 3-21 Ahb2AhbSyncUp bridge

Trial synthesis of the `Ahb2Ahb`, `Ahb2AhbSyncUp`, and `Ahb2AhbSyncDn` bridges is targeted at a clock period of 6ns, implying a frequency of ~166MHz. The required input and output port constraints are set as follows:

Inputs Maximum setup time of 30% of clock cycle, implying 1.8ns.

Outputs Maximum output valid delay of 40% of clock cycle, implying 2.4ns.

Note

These are the preferred synthesis targets, that are not always achievable depending on the technology library used. Trial synthesis using the TSMC 0.13 library has shown that all internal, register-to-register, paths meet the 166MHz target, but that constraints on some ports might have to be relaxed.

Ahb2AhbPass (1:1)

This is a simple combinatorial bridge that connects AHB buses without incurring a latency penalty. You can use this bridge as a latency-free pin-compatible alternative to the other bridges or where a slave gasket, such as the downsizer, is required to connect to a multi-master bus. Figure 3-22 shows the Ahb2AhbPass bridge.

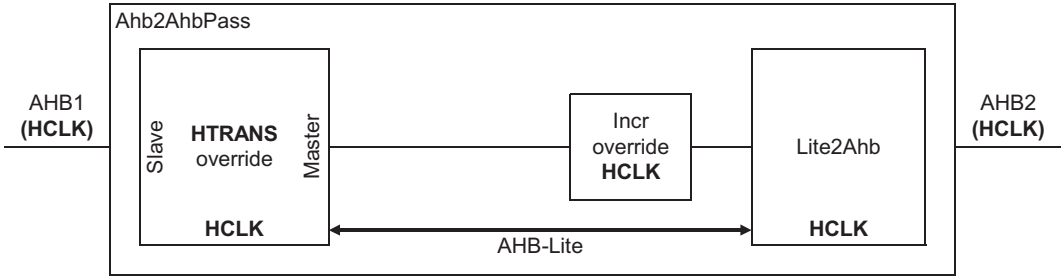


Figure 3-22 Ahb2AhbPass bridge

Trial synthesis of the Ahb2AhbPass bridge is targeted at a clock period of 6ns, implying a frequency of approximately 166MHz. Because of the presence of combinatorial paths through the design, the timing constraints are set as follows:

Input to register 30% of clock cycle, implying 1.8ns.

Register to output 30% of clock cycle, implying 1.8ns.

Input to output 15% of clock cycle, implying 0.9ns.

These are the preferred synthesis targets, that are not always achievable depending on the technology library used. Trial synthesis using the TSMC 0.13 library has shown that all internal, register-to-register, paths meet the 166MHz target, but that constraints on some ports might have to be relaxed.

3.12.2 Typical applications

The primary use of the bridges is to connect AHB domains, enabling masters to access slaves in a different timing or clock domain. This is commonly used in multiprocessor systems or where an off-chip interface is required. Other less obvious applications of the bridges are to help with timing closure when a bus is heavily loaded or when interfacing to a slave with particularly poor timing characteristics, excluding the passthrough bridge.

Note

In general, AHB-AHB bridges do not provide a high bandwidth link so must not be used to access performance-critical slaves. The performance of the bridges also degrades with frequency mismatch between buses.

Figure 3-23 shows applications of synchronous AHB-AHB bridges.

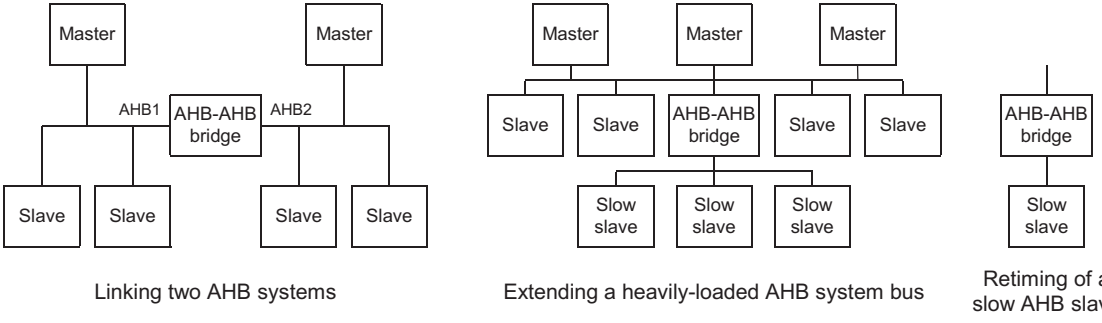


Figure 3-23 Applications of synchronous AHB-AHB bridges

3.12.3 Programmer's model

The general concept of the bridge is described in the following sections:

- *Preserved address map*
- *Aliased or piecewise address map*
- *Functionality* on page 3-53
- *Bidirectional bridging* on page 3-54.

Note

The scope of memory-map visibility is user-defined.

Preserved address map

Figure 3-24 shows the memory maps for a system with one-to-one address mapping. It is assumed that full visibility of the slaves in system 2 is required.

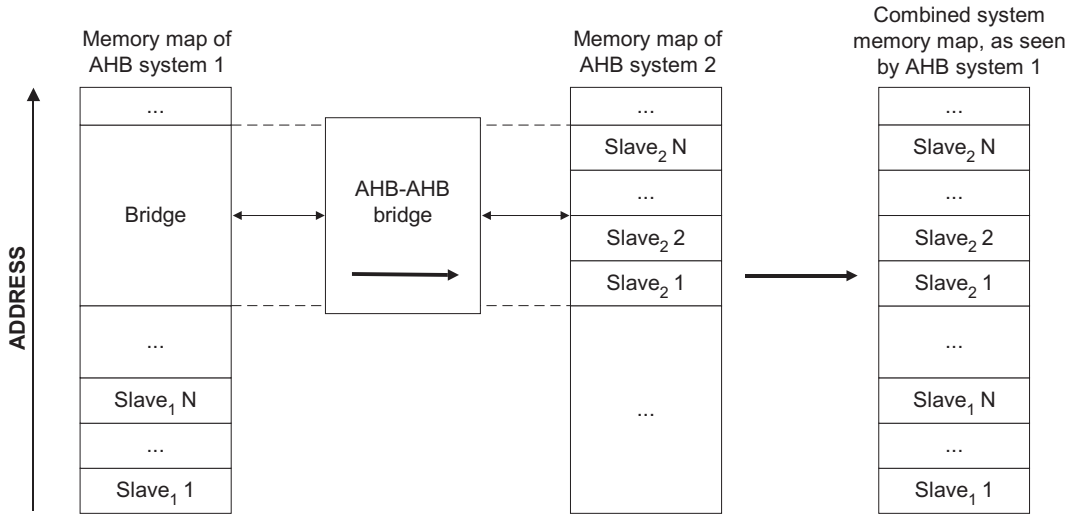


Figure 3-24 System memory maps without address aliasing

Aliased or piecewise address map

Address-aliasing is a function of the address decoder and of masking-off the high-order address bits in hardware. It is recommended that this is implemented in AHB system 1. Figure 3-25 on page 3-52 shows a generalized scheme.

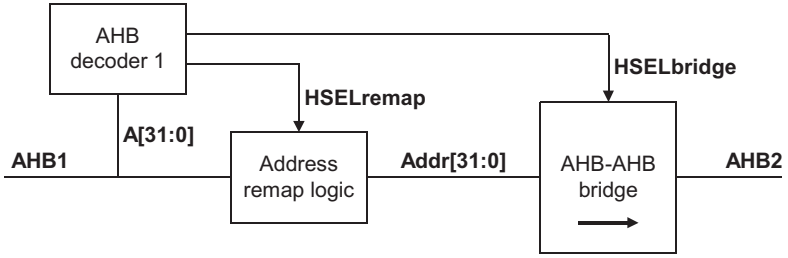


Figure 3-25 Address-aliasing hardware

Figure 3-26 and Figure 3-27 on page 3-53 show the use of address-aliasing to make memory in system 2 appear both at address 0 and also at a higher address, so that it can be accessed by system 1 through the bridge. In this design, only the area of ROM is visible through the bridge.

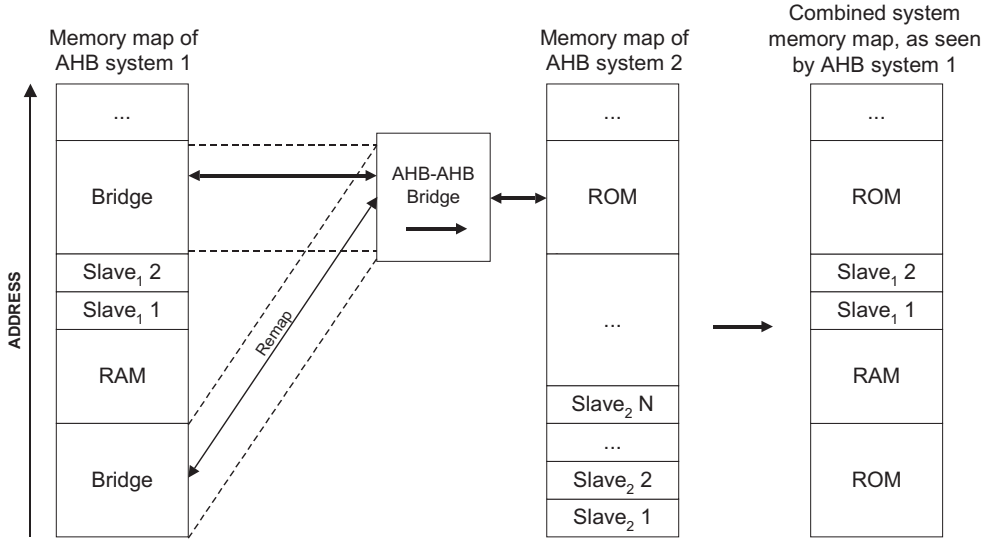


Figure 3-26 System memory maps with aliased addressing

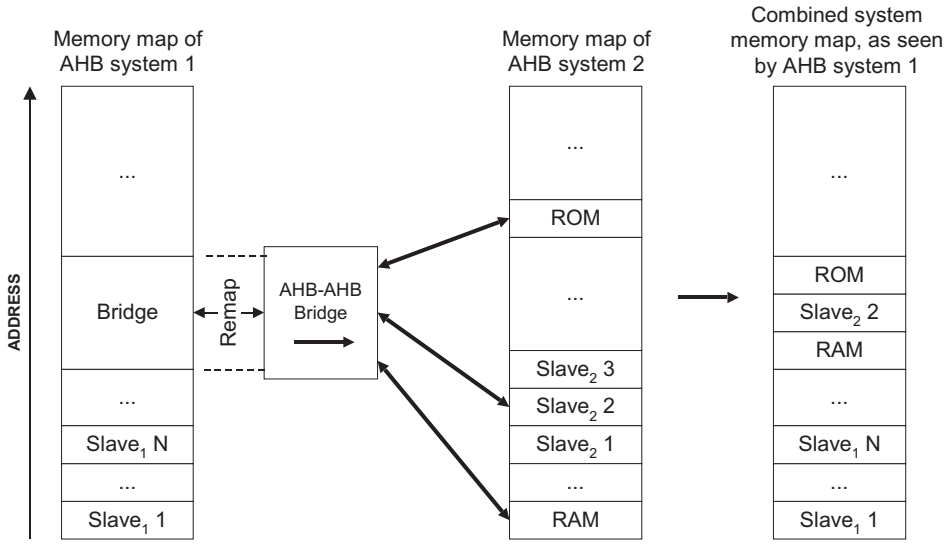


Figure 3-27 System memory maps with piecewise addressing

Functionality

This section describes the functionality of the synchronous AHB-AHB bridges.

Reset It is recommended that the buses on either side of the bridge are reset together, for at least three cycles of each **HCLK**. However, AHB2 can be held in reset while AHB1 is free-running, if no transaction is directed at the bridge. You can use this to reduce power consumption on AHB2 while it is unused.

Slave responses

SPLIT and RETRY responses from remote slaves are supported, but they are not propagated back to the master. The bridge inserts wait states on bus 1 using **HREADY** to hold up the master until the remote slave is ready to complete the outstanding transfer. The bridge slave interface does not generate SPLIT or RETRY responses.

Error ERROR responses from remote slaves are normally propagated back to the master to determine further action for that transfer. In the Ahb2Ahb bridge (1:1), this response is suppressed for buffered writes because of a transfer correlation issue. For buffered transfers, the master cannot determine exactly when the transfer completes at the slave. If a

confirmation of completion is required, a single read can be directed at a remote slave that only completes when any previously buffered writes are also complete.

Wait states It is an AHB recommendation that slaves must not generate more than 16 wait-states. This is included to help with system latency predictions. However, when using an AHB-AHB bridge, the effects of registered paths, crossing clock domains, slave wait-states, RETRY/SPLIT responses, and bus arbitration have a cumulative effect on the number of wait-states generated by the bridge slave. This implies that the bridge can easily generate wait periods that are greater than 16 cycles in length.

Locked transfers

The bridges are designed so that **HMASTLOCK** is asserted on AHB2 when the first locked transfer is directed at the bridge. The lock on AHB2 is then held until the entire locked sequence completes on AHB1, even if it contains transfers not directed at the bridge. This ensures coherency of both buses during the locked sequence.

Bidirectional bridging

All AHB-AHB bridges are unidirectional in nature, but you can use them as a pair to form a bidirectional bridge between buses if the following potential deadlock situation is avoided:

- If both bridges are active simultaneously, that is, a master is accessing a slave across the bridge, then deadlock can occur while both bridges are waiting to be granted the remote bus.

Figure 3-28 on page 3-55 shows bidirectional bridging. If M0 accesses S3 through bridge 1 while M3 accesses S0 through bridge 2, the bridge masters, M1, M2, are not granted the bus because the masters M0 and M3 have bus control until their transfers complete.

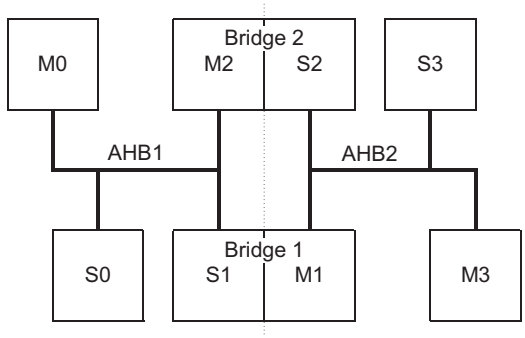


Figure 3-28 Bidirectional bridging

Deadlock can also happen if one bridge master happens to direct a transfer at the other bridge slave, for example:

M0 → S1/M1 → S2/M2 → S0.

One way to avoid deadlock is to use multi-layer bus matrix on one or both of the AHB buses. The decode must be chosen so that a bridge loop is not possible. Using multi-layer bus matrix on AHB1 for example, enables the transfer M2 → S0 to complete before the transfer M0 → S1, and the deadlock situation is avoided.

3.12.4 Optional additional blocks

The following blocks are described in this section:

- *Error cancel*
- *IncrOverride* on page 3-57.

Error cancel

When an AHB master receives an ERROR response from a slave, it can optionally cancel any pending transfer, while **HREADY** is LOW, by driving **HTRANS** to IDLE during the second cycle of the ERROR response. Figure 3-29 on page 3-56 shows this.

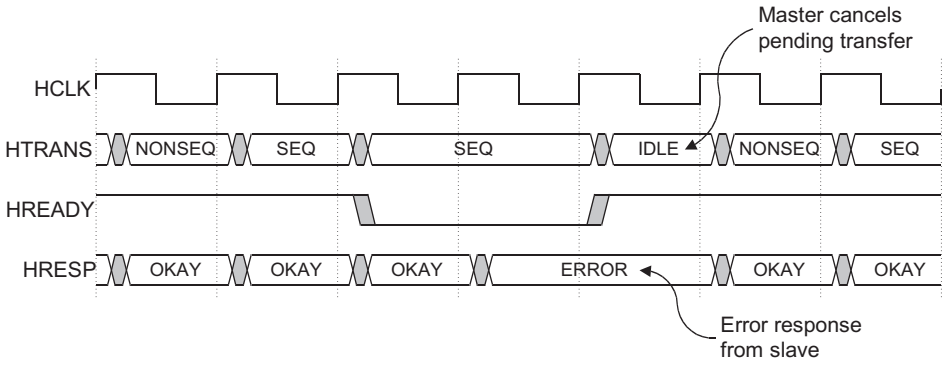


Figure 3-29 Error cancel timing

Because a registered bridge, except Ahb2AhbPass, cannot predict what the master will do, it must always exhibit either a cancel or continue behavior. Because most AHB masters continue with a pending transfer, the bridge also does this by default. Therefore, if a master that can potentially cancel a transfer on ERROR response is permitted to use the bridge, the ErrorCanc block is required to change the bridge behavior.

When an ERROR response is received by the bridge master, the ErrorCanc logic immediately cancels any pending transfer and the ERROR response is passed back to the originating master. If the master continues with a pending transfer and completes the burst, the ErrorCanc block suppresses the transfer and responds with an ERROR response to each transfer remaining in the burst. Figure 3-30 shows this.

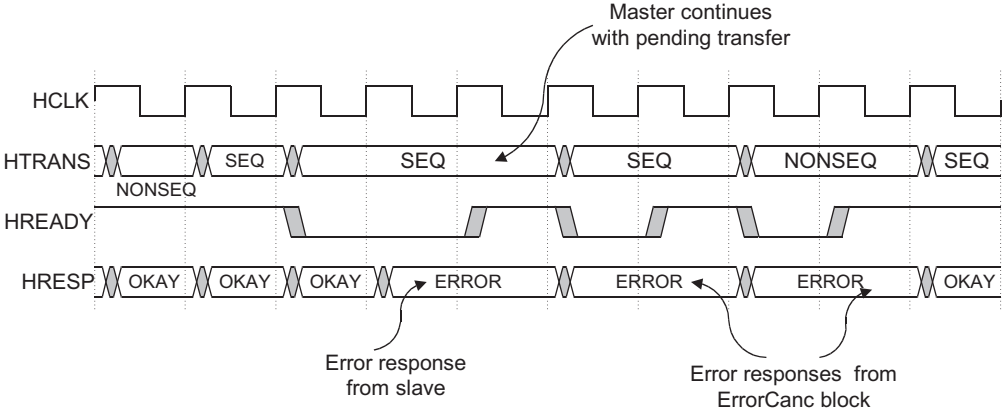


Figure 3-30 Error cancel using ErrorCanc timing

Note

For buffered write transfers, Ahb2Ahb bridge only, slave responses cannot be passed back to the master. The error-cancelling functionality is therefore disabled for buffered transfers in the Ahb2Ahb bridge. To facilitate this, the ErrorCanc logic is integrated into the top level of the Ahb2Ahb bridge.

IncrOverride

An AHB master can only abort a burst if it receives an ERROR response or is degranted before the burst is complete. If a master on AHB1, accessing a peripheral through the bridge is degranted before the burst is complete, the bridge master on AHB2 appears to have illegally aborted the burst. Therefore, if the arbiter on AHB1 has the potential to rearbitrate before a burst is complete, to avoid breaking protocol, the bridge must not generate fixed-length bursts. To avoid this, the IncrOverride block must be used. This block overrides **HBURSTM** to always be INCR.

Note

Under certain circumstances, the ADK arbiter and ADK v1 revisions of the Bus Matrix arbiter might rearbitrate in mid-burst. If either of these are used, the IncrOverride block must be used with the bridges.

3.12.5 Signal descriptions

Table 3-12 lists the interface signals for the synchronous AHB-AHB bridge.

Table 3-12 Synchronous AHB-AHB bridge interface signals

Signal	AHB bus	Direction	Description
HADDRM[31:0]	2	Output	The 32-bit system address bus.
HADDRS[31:0]	1	Input	The 32-bit system address bus.
HBURSTM[2:0]	2	Output	Indicates if the transfer forms part of a burst. The bridge supports all types of transfer, that is single, incrementing, or wrapping.
HBURSTS[2:0]	1	Input	Indicates if the transfer forms part of a burst. The bridge supports all types of transfer, that is single, incrementing, or wrapping.
HBUSREQM	2	Output	A signal from the bridge to the arbiter, that indicates that the master interface requires bus 2. There is an **HBUSREQ** signal for each bus master in the system.

Table 3-12 Synchronous AHB-AHB bridge interface signals (continued)

Signal	AHB bus	Direction	Description
HCLKEN	-	Input	This signal describes the relationship between **HCLKS** and **HCLKM**. This is HIGH for coincident edges between clocks.
HCLKM	2	Input	This clock times all bus transfers on AHB2. All signal timings on AHB2 are related to the rising edge of **HCLKM**.
HCLKS	1	Input	This clock times all bus transfers on AHB1. All signal timings on AHB1 are related to the rising edge of **HCLKS**.
HGRANTM	2	Input	This signal indicates that the bridge is currently the highest priority master. Ownership of the address and control signals changes at the end of a transfer when **HREADYM** is HIGH, so the master gets access to the bus when both **HREADYM** and **HGRANTM** are HIGH.
HLOCKM	2	Output	When HIGH, this signal indicates that the master requires locked access on bus 2 and no other master must be granted that bus until this signal is LOW.
HMASTLOCKS	1	Input	When HIGH, this signal indicates that the master on bus 1 requires locked access through the bridge and no other master must be granted the bus until this signal is LOW.
HPROTM[3:0]	2	Output	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection.
HPROTS[3:0]	1	Input	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wants to implement some level of protection.
HRDATAM	2	Input	The read data bus, 32 or 64-bit, transfers data from the slave(s) on bus 2, to the bridge, during read operations.
HRDATAS	1	Output	The read data bus, 32 or 64-bit, transfers data from the bridge to the bus master during a read operation.
HREADYM	2	Input	When HIGH, the **HREADYM** signal indicates that a transfer has finished on bus 2. This signal can be driven LOW by a slave to extend a transfer.
HREADYOUTS	1	Output	When HIGH, the **HREADYOUTS** signal indicates that a transfer has finished on bus 1. This signal can be driven LOW by the bridge to extend a transfer.
HREADYS	1	Input	Input version of **HREADYOUTS**, required by the slave interface.

Table 3-12 Synchronous AHB-AHB bridge interface signals (continued)

Signal	AHB bus	Direction	Description
HRESETn	-	Input	This signal is active LOW and resets the system and the bus.
HRESPM[1:0]	2	Input	The transfer response provides additional information on the status of a transfer. Four different responses are supported, OKAY, ERROR, RETRY, and SPLIT.
HRESPS[1:0]	1	Output	The transfer response provides additional information on the status of a transfer. Only two responses are supported on the slave interface, OKAY and ERROR.
HSELS	1	Input	The bridge slave interface uses the **HSELS** signal to determine when it must respond to a bus transfer.
HSIZEM[2:0]	2	Output	Indicates the size of the transfer. The bridge uses 32-bit data for read and write transfers.
HSIZES[2:0]	1	Input	Indicates the size of the transfer. The bridge uses 32-bit data for read and write transfers.
HTRANSM[1:0]	2	Output	Indicates the type of the current transfer, and can be NONSEQUENTIAL, SEQUENTIAL, IDLE, or BUSY.
HTRANSS[1:0]	1	Input	Indicates the type of the current transfer, and can be NONSEQUENTIAL, SEQUENTIAL, IDLE, or BUSY.
HWDATAM	2	Output	The write data bus, 32 or 64-bit, transfers data from the bridge to the slave(s) on bus 2, during write operations.
HWDATAS	1	Input	The write data bus, 32 or 64-bit, transfers data from the bus master to the bridge during a write operation.
HWRITEM	2	Output	When HIGH, this signal indicates a write transfer, and when LOW, a read transfer.
HWRITES	1	Input	When HIGH, this signal indicates a write transfer, and when LOW, a read transfer.

3.13 Asynchronous AHB-AHB bridge

The asynchronous AHB-AHB bridge, `Ahb2AhbAsync32` and `Ahb2AhbAsync64`, provides a unidirectional link between two AHB domains with asynchronous clocks. Its function is to enable a master to access a slave on another bus, with the transfer initiated from one side only. The asynchronous bridge is not intended for performance applications, because the synchronization logic increases the overhead of a registered bridge design.

The asynchronous AHB-AHB bridge has the following features:

- 32 or 64-bit data bus
- transfer sequences can be locked across the bridge
- bursts are split into single transfers to prevent excessive latency from BUSY transfers
- SPLIT and RETRY responses are serviced by the bridge, so remain local to the issuing slave.

Figure 3-31 shows the asynchronous AHB-AHB bridge module.

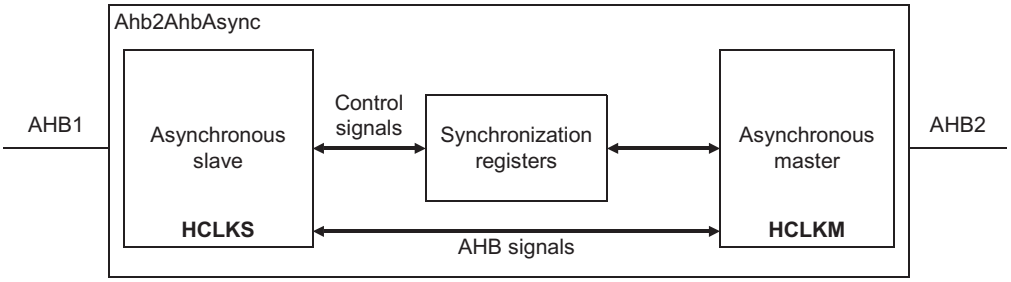


Figure 3-31 Asynchronous AHB-AHB bridge module components

3.13.1 Programmer's model

The following sections describe the programming and operation of the asynchronous AHB-AHB bridge:

- *Reset* on page 3-61
- *Low power operation* on page 3-61
- *Slave responses* on page 3-61
- *Locked transfers* on page 3-61.

Reset

The bridge has two asynchronous reset domains, **HRESETSn** and **HRESETMn**. Each reset must be asserted for at least one cycle of the relevant clock. No transfer must be attempted through the bridge until both sides have exited from reset.

Low power operation

For low power operation, you can stop the clock on AHB2 or hold it in reset while keeping AHB1 active, if any current bridge transfer has completed and the bridge is not addressed while AHB2 is inactive.

Slave responses

Slave responses of SPLIT or RETRY are serviced locally in the appropriate way. The transfer is driven back onto the bus while AHB1 is stalled. The behavior of the master to an ERROR response is undefined, so the error is passed across the bridge to the originating master, and can either continue or abort the current burst.

Locked transfers

If the bridge is accessed with a locked transfer, that is, **HMASTLOCKS** is HIGH, it locks the transfer onto AHB2, using **HLOCKM**.

Note

For a succession of locked transfers on AHB1, the bridge keeps AHB2 locked by keeping **HLOCKM** asserted for the IDLE transfers between NONSEQ transactions, even if not directed at the bridge. HLOCKM remains asserted after a locked transfer, until the next unlocked transfer has propagated across the bridge.

3.13.2 Signal descriptions

Table 3-13 on page 3-62 lists the interface signals for the asynchronous AHB-AHB bridge.

Table 3-13 Asynchronous AHB-AHB bridge interface signals

Signal	Block	Clock domain	Direction	Description
HADDRM[31:0]	Master interface	Master	Output	The 32-bit system address bus.
HADDRS[31:0]	Slave interface	Slave	Input	The 32-bit system address bus.
HBURSTM[2:0]	Master interface	Master	Output	Indicates if the transfer forms part of a burst. The bridge only generates transfers of type SINGLE.
HBURSTS[2:0]	Slave interface	Slave	Input	Indicates if the transfer forms part of a burst. The bridge supports all types of transfer, that is single, incrementing, or wrapping.
HBUSREQM	Master interface	Master	Output	A signal from the bridge to the arbiter, that indicates that the master interface requires bus 2.
HCLKM	Master interface	-	Input	This clock times all bus transfers. All signal timings on the AHB2 are related to the rising edge of **HCLKM**.
HCLKS	Slave interface	-	Input	This clock times all bus transfers. All signal timings on AHB1 are related to the rising edge of **HCLKS**.
HGRANTM	Master interface	Master	Input	This signal indicates that the bridge is currently the highest priority master. Ownership of the address and control signals changes at the end of a transfer when **HREADYM** is HIGH, so the master gets access to the bus when both **HREADYM** and **HGRANTM** are HIGH.
HLOCKM	Master interface	Master	Output	When HIGH, this signal indicates that the master requires locked access on bus 2.
HMASTLOCKS	Slave interface	Slave	Input	When HIGH, this signal indicates that the master on bus 1 requires locked access through the bridge.
HPROTM[3:0]	Master interface	Master	Output	The protection control signals provide additional information about a bus access and are passed across the bridge.
HPROTS[3:0]	Slave interface	Slave	Input	The protection control signals provide additional information about a bus.
HRDATAM[31:0]	Master interface	Master	Input	The 32-bit read data bus transfers data from the slave(s) on bus 2, to the bridge, during read operations.

Table 3-13 Asynchronous AHB-AHB bridge interface signals (continued)

Signal	Block	Clock domain	Direction	Description
HRDATAS[31:0]	Slave interface	Slave	Output	The 32-bit read data bus transfers data from the bridge to the bus master during a read operation.
HREADYM	Master interface	Master	Input	When HIGH, the **HREADYM** signal indicates that a transfer has finished on bus 2. This signal can be driven LOW by a slave to extend a transfer.
HREADYOUTS	Slave interface	Slave	Output	When HIGH, the **HREADYOUTS** signal indicates that a transfer has finished on bus 1. This signal can be driven LOW by the bridge to extend a transfer.
HREADYS	Slave interface	Slave	Input	Multiplexed **HREADYOUT** signals to indicate when transfer is complete on AHB1.
HRESETMn	Master interface	Master	Input	This signal is active LOW and resets AHB2.
HRESETSn	Slave interface	Slave	Input	This signal is active LOW and resets AHB1.
HRESPM[1:0]	Master interface	Master	Input	The transfer response provides additional information on the status of a transfer. Four different responses are supported, OKAY, ERROR, RETRY, and SPLIT.
HRESPS[1:0]	Slave interface	Slave	Output	The transfer response provides additional information on the status of a transfer.
HSELS	Slave interface	Slave	Input	The bridge slave interface uses the **HSELS** signal to determine when it must respond to a bus transfer.
HSIZEM[M:0]	Master interface	Master	Output	Indicates the size of the transfer. The bridge supports sizes up to 32-bit.
HSIZES[2:0]	Slave interface	Slave	Input	Indicates the size of the transfer. The bridge supports sizes up to 32-bit.
HTRANSM[1:0]	Master interface	Master	Output	Indicates the type of the current transfer. The bridge supports transfers of type IDLE or NONSEQ.
HTRANSS[1:0]	Slave interface	Slave	Input	Indicates the type of the current transfer, and can be NONSEQUENTIAL, SEQUENTIAL, IDLE, or BUSY.
HWDATAM[31:0]	Master interface	Master	Output	The 32-bit write data bus transfers data from the bridge to the slave(s) on bus 2, during write operations.

Table 3-13 Asynchronous AHB-AHB bridge interface signals (continued)

Signal	Block	Clock domain	Direction	Description
HWDATAS[31:0]	Slave interface	Slave	Input	The 32-bit write data bus transfers data from the bus master to the bridge during a write operation.
HWRITEM	Master interface	Master	Output	When HIGH, this signal indicates a write transfer, and when LOW, a read transfer.
HWRITES	Slave interface	Slave	Input	When HIGH, this signal indicates a write transfer, and when LOW, a read transfer.
SCANENABLE	Scan	Slave	Input	Scan test mode enable.
SCANINHCLKM	Scan	Master	Input	Scan chain input for **HCLKM** registers.
SCANINHCLKS	Scan	Slave	Input	Scan chain input for **HCLKS** registers.
SCANOUTHCLKM	Scan	Master	Output	Scan chain output for **HCLKM** registers.
SCANOUTHCLKS	Scan	Slave	Output	Scan chain output for **HCLKS** registers.

3.14 AHB-Lite to AHB wrapper

AHB-Lite is a subset of the full AHB specification, and is intended for designs that only use a single bus master. This can either be a simple single-master system or a multi-layer AHB system where there is only one AHB master per layer.

Figure 3-32 shows the AHB-Lite to AHB wrapper, `Lite2Ahb`.

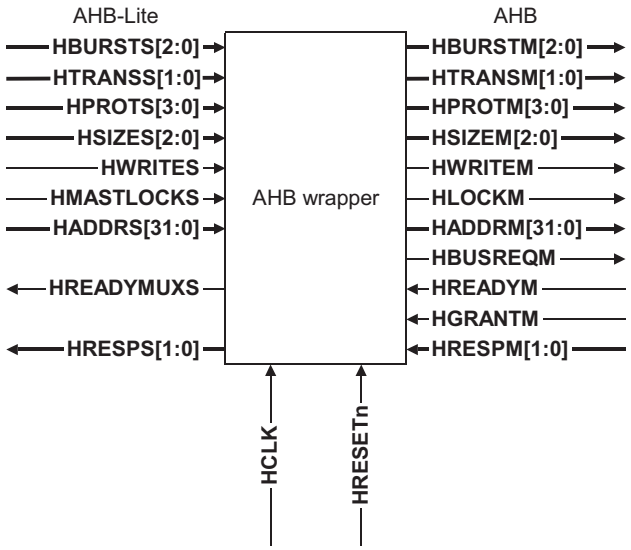


Figure 3-32 AHB-Lite to AHB wrapper

3.14.1 Programmer's model

The wrapper is designed to enable any AHB-Lite master to connect to a standard AHB bus. This requires the wrapper to add support for the following features:

Bus ownership When a transfer is initiated by the master, **HTRANSS[1:0]** is NONSEQ, the wrapper generates a request using **HBUSREQM**. The bus is also requested again if ownership is lost mid-burst.

Early terminated bursts

If a burst is terminated prematurely through loss of the bus, the wrapper holds the master using **HREADYMUXS** and rebuilds the burst when bus control is regained.

Slave responses The slave responses of SPLIT and RETRY are handled by the wrapper. Bursts are also regenerated in these cases. ERROR responses are passed to the master through the **HRESPS[1:0]** signal and the master must determine an appropriate action.

Locked transfers The **HMASTLOCK** signal is retimed into the arbitration phase as **HLOCK**. Additionally, unlocked IDLE transfers are inserted between locked bursts, as recommended in the *AMBA Specification (Rev 2.0)*.

3.15 Interrupt controller

The interrupt controller, `Interrupt`, provides a software interface to the interrupt system. In an ARM system, two levels of interrupt are available:

- *Fast Interrupt Request* (FIQ) for fast, low latency interrupt handling
- *Interrupt Request* (IRQ) for more general interrupts.

Only a single FIQ source at a time is generally used in a system, to provide a true low-latency interrupt. This has the following benefits:

- You can execute the interrupt service routine directly without determining the source of the interrupt.
- Interrupt latency is reduced. You can use the banked registers available for FIQ interrupts more efficiently, because a context save is not required.

There are 32 interrupt lines. The interrupt controller uses a bit position for each different interrupt source. The software can control each request line to generate software interrupts.

Note

Unused interrupt lines must be tied LOW to disable them.

Figure 3-33 shows a block diagram of the interrupt controller.

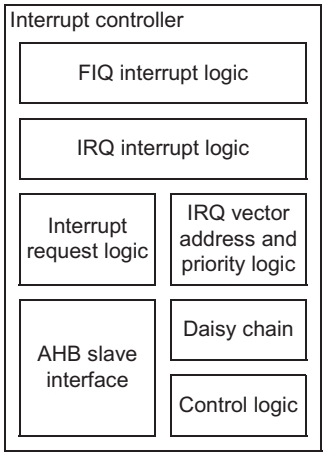


Figure 3-33 Interrupt controller components

The nonvectored and daisy-chained IRQ interrupts provide an address for an *Interrupt Service Routine* (ISR). Reading from the vector interrupt address register, ICVectAddr, provides the address of the ISR, and updates the interrupt priority hardware that masks out the current and any lower priority interrupt requests. Writing to the ICVectAddr register indicates to the interrupt priority hardware that the current interrupt is serviced, enabling lower priority interrupts to go active.

The FIQ interrupt has the highest priority, followed by nonvectored IRQ interrupts. Daisy-chained interrupts have the lowest priority. A programmed interrupt request enables you to generate an interrupt under software control. This register is typically used to downgrade an FIQ interrupt to an IRQ interrupt.

Note

The priority of the FIQ over IRQ is set by the ARM core. The interrupt controller can raise both an FIQ and an IRQ at the same time.

The IRQ and FIQ request logic has an asynchronous path. This enables interrupts to be asserted when the clock is disabled.

3.15.1 Programmer's model

Table 3-14 lists how, by convention, the IRQ interrupt bits [5:1] must be used. Bit 0 and bit 6 upwards are available for use as required. For the FIQ interrupt, the bits can be used as required.

Table 3-14 Interrupt standard configuration

Bit	Interrupt source
1	Software interrupt
2	Comms Rx
3	Comms Tx
4	Timer 1
5	Timer 2

The software can control the source interrupt lines to generate software interrupts. These interrupts are generated before interrupt masking, in the same way as external source interrupts. Software interrupts are cleared by writing to the software interrupt clear register, ICSoftIntClear. See *Software Interrupt Clear Register* on page 3-74. This is normally done at the end of the interrupt service routine.

Interrupt flow sequence

The following procedure shows the sequence for the vectored interrupt flow:

1. An interrupt occurs.
2. The ARM processor branches to either the IRQ or FIQ exception vector.
3. If the interrupt is an IRQ, read the ICVectAddr register and branch to the interrupt service routine. This can be done using an LDR PC instruction. Reading the ICVectorAddr register updates the interrupt controllers hardware priority register.
4. Stack the workspace so that IRQ interrupts can be re-enabled.
5. Enable the IRQ interrupts so that a higher priority can be serviced.
6. Execute the *Interrupt Service Routine* (ISR).
7. Clear the requesting interrupt in the peripheral, or write to the ICSoftIntClear register if the request was generated by a software interrupt.
8. Disable the interrupts and restore the workspace.
9. Write to the ICVectAddr register. This clears the respective interrupt in the internal interrupt priority hardware.
10. Return from the interrupt. This re-enables the interrupts.

Simple interrupt flow

The following procedure shows how you can use the interrupt controller without using vectored interrupts or the interrupt priority hardware. For example, you can use it for debugging.

1. An interrupt occurs.
2. Branch to IRQ or FIQ exception vector.
3. Branch to the interrupt handler.
4. Interrogate the ICIRQStatus register to determine the source that generated the interrupt, and prioritize the interrupts if there are multiple active interrupt sources. This takes a number of instructions to compute.
5. Branch to the correct ISR.
6. Execute the ISR.

7. Clear the interrupt. If the request was generated by a software interrupt, the ICSoftIntClear register must be written to.
8. Check the ICIRQStatus register to ensure that no other interrupt is active. If there is an active request go to Step 4.
9. Return from the interrupt.

Note

If the above flow is used, you must not read or write to the ICVectorAddr register.

To ensure that the vector address register can be read in a single instruction, the IC base address must be `0xFFFFF000`, the upper 4K of memory. See *Vector Address Register* on page 3-75. Placing the IC anywhere else in memory increases interrupt latency as the ARM processor is unable to access the ICVectorAddr register using a single instruction. The offset of any particular register from the base address is fixed.

Table 3-15 lists the registers in base offset order.

Table 3-15 Interrupt controller registers

Name	Base offset	Type	Width	Reset value	Description
ICIRQSTATUS	`0x000`	Read	32	`0x00000000`	See *IRQ Status Register* on page 3-72
ICFIQSTATUS	`0x004`	Read	32	`0x00000000`	See *FIQ Status Register* on page 3-72
ICRAWINTR	`0x008`	Read	32	-	See *Raw Interrupt Status Register* on page 3-72
ICINTSELECT	`0x00C`	Read/ write	32	`0x00000000`	See *Interrupt Select Register* on page 3-73
ICINTENABLE	`0x010`	Read/ write	32	`0x00000000`	See *Interrupt Enable Register* on page 3-73
ICINTENCLEAR	`0x014`	Write	32	-	See *Interrupt Enable Clear Register* on page 3-73
ICSOFTINT	`0x018`	Read/ write	32	`0x00000000`	See *Software Interrupt Register* on page 3-74
ICSOFTINTCLEAR	`0x01C`	Write	32	-	See *Software Interrupt Clear Register* on page 3-74
ICPROTECTION	`0x020`	Read/ write	1	`0x0`	See *Protection Enable Register* on page 3-74

Table 3-15 Interrupt controller registers (continued)

Name	Base offset	Type	Width	Reset value	Description
ICVECTADDR	0x030	Read/ write	32	0x00000000	See *Vector Address Register* on page 3-75
ICDEFVECTADDR	I0x034	Read/ write	32	0x00000000	See *Default Vector Address Register* on page 3-76
ICITCR	0x300	Read/ write	1	-	See *Test Control Register* on page 3-76
ICITIP1	I0x304	Read	2	-	See *Test Input Register 1* on page 3-76
ICITIP2	0x308	Read	32	-	See *Test Input Register 2* on page 3-77
ICITOP1	0x30C	Read	2	0x0	See *Test Output Register 1* on page 3-77
ICITOP2	0x310	Read	32	0x00000000	See *Test Output Register 2* on page 3-78
ICPERIPHID0	0xFE0	Read	8	0x08	See *Peripheral Identification Registers* on page 3-78
ICPERIPHID1	0xFE4	Read	8	0x18	See *Peripheral Identification Registers* on page 3-78
ICPERIPHID2	0xFE8	Read	8	0x04	See *Peripheral Identification Registers* on page 3-78
ICPERIPHID3	0xFEC	Read	8	0x00	See *Peripheral Identification Registers* on page 3-78
ICPCELLID0	0xFF0	Read	8	0x0D	See *PrimeCell Identification Registers* on page 3-81
ICPCELLID1	0xFF4	Read	8	0xF0	See *PrimeCell Identification Registers* on page 3-81
ICPCELLID2	0xFF8	Read	8	0x05	See *PrimeCell Identification Registers* on page 3-81
ICPCELLID3	0xFFC	Read	8	0xB1	See *PrimeCell Identification Registers* on page 3-81

IRQ Status Register

The ICIRQSTATUS register is read-only. It provides the status of interrupts [31:0] after IRQ masking. Table 3-16 lists the register bit assignments.

Table 3-16 ICIRQSTATUS Register bit assignments

Bits	Name	Function
[31:0]	IRQStatus	Shows the status of the interrupts after masking by the ICIntEnable and ICIntSelect registers. A HIGH bit indicates that the interrupt is active, and generates an interrupt to the processor.

FIQ Status Register

The ICFIQSTATUS register is read-only. It provides the status of the interrupts after FIQ masking. Table 3-17 lists the register bit assignments.

Table 3-17 ICFIQSTATUS Register bit assignments

Bits	Name	Function
[31:0]	FIQStatus	Shows the status of the interrupts after masking by the ICIntEnable and ICIntSelect registers. A HIGH bit indicates that the interrupt is active, and generates an interrupt to the processor.

Raw Interrupt Status Register

The ICRAWINTR register is read-only. It provides the status of the source interrupts, and software interrupts, to the interrupt controller. Table 3-18 lists the register bit assignments.

Table 3-18 ICRAWINTR Register bit assignments

Bits	Name	Function
[31:0]	RawInterrupt	Shows the status of the interrupts before masking by the enable registers. A HIGH bit indicates that the appropriate interrupt request is active before masking.

Interrupt Select Register

The ICINTSELECT register is read/write. It selects whether the corresponding interrupt source generates an FIQ or an IRQ interrupt. Table 3-19 lists the register bit assignments.

Table 3-19 ICINTSELECT Register bit assignments

Bits	Name	Function
[31:0]	IntSelect	Selects type of interrupt for interrupt request: 0 = IRQ interrupt 1 = FIQ interrupt.

Interrupt Enable Register

The ICINTENABLE register is read/write. It enables the interrupt request lines, by masking the interrupt sources for the IRQ interrupt. Table 3-20 lists the register bit assignments.

Table 3-20 ICINTENABLE Register bit assignments

Bits	Name	Function
[31:0]	IntEnable	Enables the interrupt request lines: 0 = Interrupt disabled. 1 = Interrupt enabled. Enables interrupt request to processor. On reset, all interrupts are disabled. A HIGH bit sets the corresponding bit in the ICIntEnable register. A LOW bit has no effect.

Interrupt Enable Clear Register

The ICINTENCLEAR register is write-only. It clears bits in the ICIntEnable register. Table 3-21 lists the register bit assignments.

Table 3-21 ICINTENCLEAR Register bit assignments

Bits	Name	Function
[31:0]	IntEnable Clear	Clears bits in the ICIntEnable register. A HIGH bit clears the corresponding bit in the ICIntEnable register. A LOW bit has no effect.

Software Interrupt Register

The ICSOFTINT register is read/write. It generates software interrupts. Table 3-22 lists the register bit assignments.

Table 3-22 ICSOFTINT Register bit assignments

Bits	Name	Function
[31:0]	SoftInt	Setting a bit generates a software interrupt for the specific source interrupt before interrupt masking. A HIGH bit sets the corresponding bit in the ICSoftInt register. A LOW bit has no effect.

Software Interrupt Clear Register

The ICSOFTINTCLEAR register is write-only. It clears bits in the ICSoftInt register. Table 3-23 lists the register bit assignments.

Table 3-23 ICSOFTINTCLEAR Register bit assignments

Bits	Name	Function
[31:0]	SoftIntClear	Clears bits in the ICSoftInt register. A HIGH bit clears the corresponding bit in the ICSoftInt register. A LOW bit has no effect.

Protection Enable Register

The ICPROTECTION register is read/write. It enables or disables protected register access. Figure 3-34 shows the register bit assignments.

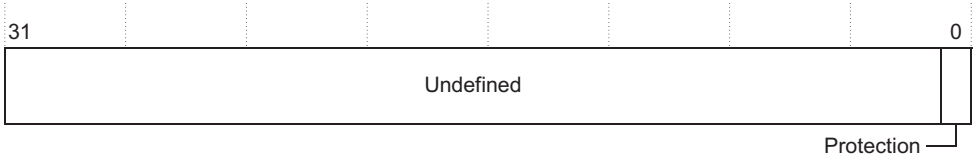


Figure 3-34 ICPROTECTION Register bit assignments

Table 3-24 lists the register bit assignments.

Table 3-24 ICPROTECTION Register bit assignments

Bits	Name	Function
[31:1]	Reserved	-
[0]	Protection	Enables or disables protected register access. When enabled, only privileged mode accesses, reads and writes, can access the interrupt controller registers. When disabled, both User mode and privileged mode can access the registers. This register is cleared on reset, and can only be accessed in privileged mode.

Note

If the bus master cannot generate accurate protection information, leave this register in its reset state to enable User mode access.

Vector Address Register

The ICVECTADDR register is read/write. It contains the *Interrupt Service Routine* (ISR) address of the currently active interrupt. Table 3-25 lists the register bit assignments.

Table 3-25 ICVECTADDR Register bit assignments

Bits	Name	Function
[31:0]	VectorAddr	Contains the address of the currently active ISR. Any writes to this register clear the interrupt.

Reading from this register provides the address of the ISR, and indicates to the priority hardware that the interrupt is being serviced. Writing to this register indicates to the priority hardware that the interrupt has been serviced. You must use the register as follows:

- the ISR reads the ICVectAddr register when an IRQ interrupt is generated
- at the end of the ISR, the ICVectAddr register is written to, to update the priority hardware.

Reading or writing to the register at other times can cause incorrect operation.

Default Vector Address Register

The ICDEFVECTADDR register is read/write. It contains the default ISR address. Table 3-26 lists the register bit assignments.

Table 3-26 ICDEFVECTADDR Register bit assignments

Bits	Name	Function
[31:0]	Default VectorAddr	Contains the address of the default ISR handler

Test Control Register

The ICITCR register is read/write. It selects test mode, and is cleared on reset. Figure 3-35 shows the register bit assignments.

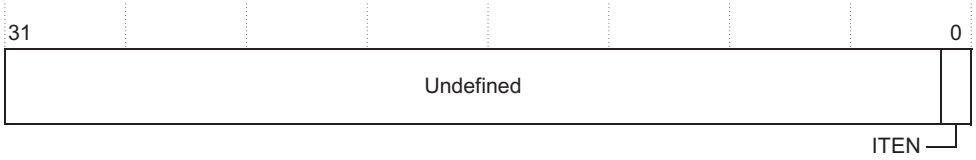


Figure 3-35 ICITCR Register bit assignments

Table 3-27 lists the register bit assignments.

Table 3-27 ICITCR Register bit assignments

Bits	Name	Function
[31:1]	Reserved	-
[0]	ITEN	Selects test mode, to use ICITIP test registers in place of input signals

Test Input Register 1

The ICITIP1 register is read-only. It indicates the status of the **nICIRQIN** and **nICFIQIN** daisy chain input lines. Figure 3-36 on page 3-77 shows the register bit assignments.

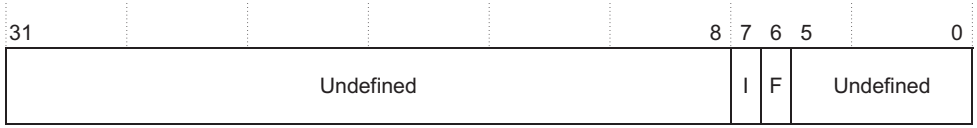


Figure 3-36 ICITIP1 Register bit assignments

Table 3-28 lists the register bit assignments.

Table 3-28 ICITIP1 Register bit assignments

Bits	Name	Function
[31:8]	Reserved	-
[7]	I	Indicates status of **nICIRQIN** when ICITCR register is LOW
[6]	F	Indicates status of **nICFIQIN** when ICITCR register is LOW
[5:0]	Reserved	-

Test Input Register 2

The ICITIP2 register is read-only. It indicates the status of the **ICVECTADDRIN** daisy chain input lines. Table 3-29 lists the register bit assignments.

Table 3-29 ICITIP2 Register bit assignments

Bits	Name	Function
[31:0]	VectorAddrIn	Indicates status of **ICVECTADDRIN** when ICITCR register is LOW

Test Output Register 1

The ICITOP1 register is read-only. It indicates the status of the **nICIRQ** and **nICFIQ** interrupt request lines to the processor. Figure 3-37 shows the register bit assignments.

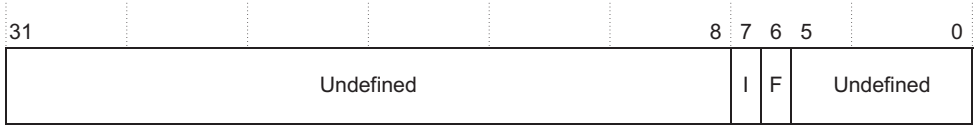


Figure 3-37 ICITOP1 Register bit assignments

Table 3-30 lists the bit assignments for the ICITOP1 register.

Table 3-30 ICITOP1 Register bit assignments

Bits	Name	Function
[31:8]	Reserved	-
[7]	I	Status of **nICIRQ** interrupt line. If set HIGH, the interrupt request is active
[6]	F	Status of **nICFIQ** interrupt line. If set HIGH, the interrupt request is active
[5:0]	Reserved	-

Test Output Register 2

The ICITOP2 register is read-only. It indicates the status of the **ICVECTADDROUT** lines from the interrupt controller. Table 3-31 lists the register bit assignments.

Table 3-31 ICITOP2 Register bit assignments

Bits	Name	Function
[31:0]	VectorAddrOut	Indicates status of **ICVECTADDROUT** from interrupt controller

Peripheral Identification Registers

The ICPERIPHID0-3 registers are four 8-bit registers, that span address locations `0xFE0-0xFEC`. The registers can conceptually be treated as a single 32-bit register. The read-only registers provide the following options of the peripheral:

Part number [11:0]

This identifies the peripheral. The three digit product code `0x90` is used for the interrupt controller.

Designer [19:12]

This is the identification of the designer. ARM Ltd is `0x41` (ASCII A).

Revision number [23:20]

This is the revision number of the peripheral. The revision number starts from 0.

Configuration [31:24]

This is the configuration option of the peripheral. The configuration value is 0.

Figure 3-38 shows the register bit assignments.

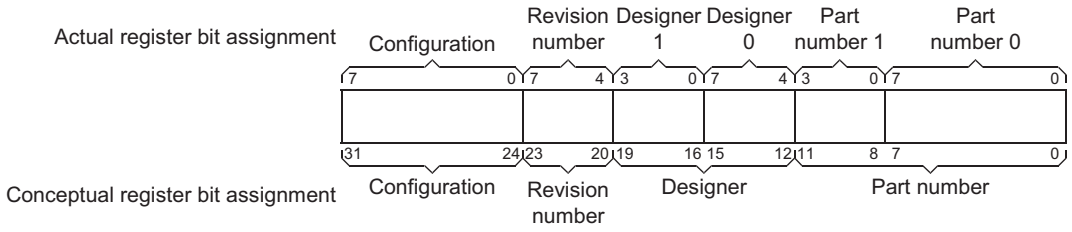


Figure 3-38 ICPERIPHID0-3 Register bit assignments

The four 8-bit peripheral identification registers are described in the following sections:

- *Peripheral Identification Register 0*
- *Peripheral Identification Register 1* on page 3-80
- *Peripheral Identification Register 2* on page 3-80
- *Peripheral Identification Register 3* on page 3-80.

Peripheral Identification Register 0

The ICPERIPHID0 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-32 lists the register bit assignments.

Table 3-32 ICPERIPHID0 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:0]	Partnumber0	These bits read back as `0x90`

Peripheral Identification Register 1

The ICPERIPHID1 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-33 lists the register bit assignments.

Table 3-33 ICPERIPHID1 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:4]	Designer0	These bits read back as `0x1`
[3:0]	Partnumber1	These bits read back as `0x1`

Peripheral Identification Register 2

The ICPERIPHID2 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-34 lists the register bit assignments.

Table 3-34 ICPERIPHID2 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:4]	Revision	These bits read back as `0x0`
[3:0]	Designer1	These bits read back as `0x4`

Peripheral Identification Register 3

The ICPERIPHID3 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-35 lists the register bit assignments.

Table 3-35 ICPERIPHID3 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:0]	Configuration	These bits read back as `0x0`

PrimeCell Identification Registers

The ICPCELLID0-3 registers are four 8-bit registers, that span address locations 0xFF0-0xFFC. The read-only register can conceptually be treated as a single 32-bit register. The register is used as a standard cross-peripheral identification system. Figure 3-39 shows the register bit assignments.

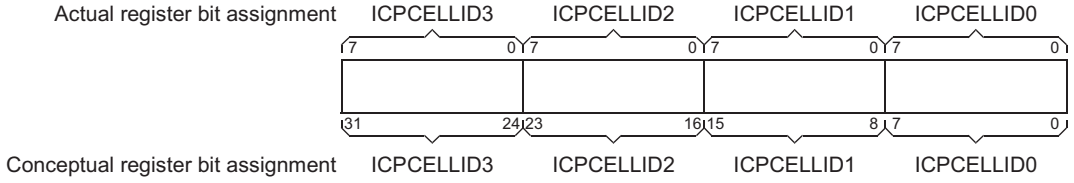


Figure 3-39 ICPCELLID0-3 Register bit assignments

The four 8-bit registers are described in the following subsections:

- *PrimeCell Identification Register 0*
- *PrimeCell Identification Register 1* on page 3-82
- *PrimeCell Identification Register 2* on page 3-82
- *PrimeCell Identification Register 3* on page 3-82.

PrimeCell Identification Register 0

The ICPCELLID0 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-36 lists the register bit assignments.

Table 3-36 ICPCELLID0 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:0]	ICPCellID0	These bits read back as 0x0D

PrimeCell Identification Register 1

The ICPCELLID1 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-37 lists the register bit assignments.

Table 3-37 ICPCELLID1 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:0]	ICPCellID1	These bits read back as `0xF0`

PrimeCell Identification Register 2

The ICPCELLID2 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-38 lists the register bit assignments.

Table 3-38 ICPCELLID2 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:0]	ICPCellID2	These bits read back as `0x05`

PrimeCell Identification Register 3

The ICPCELLID3 register is read-only. It is hard-coded and the fields within the register determine the reset value. Table 3-39 lists the register bit assignments.

Table 3-39 ICPCELLID3 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must be written as zeros
[7:0]	ICPCellID3	These bits read back as `0xB1`

3.15.2 Signal descriptions

Table 3-40 lists non-AMBA signals that the interrupt controller uses.

Table 3-40 Interrupt controller signals

Signal	Type	Direction	Description
HSELIC	Input	Decoder	Slave select signal. This is a combinatorial decode of the address bus. It indicates that the current transfer is intended for the selected slave.
ICINTSOURCE[31:0]	Input	Peripheral interrupt request	Interrupt source input. Unused interrupt lines must be tied LOW to disable them.
ICVECTADDRIN[31:0]	Input	External interrupt controller	Connects to the **ICVECTADDROUT[31:0]** signal of the previous interrupt controller if daisy chaining is used. Connects to logic 0 if the interrupt controller is not daisy-chained.
ICVECTADDROUT[31:0]	Output	Interrupt controller	Connects to the **ICVECTADDRIN[31:0]** signal of the next interrupt controller if daisy chaining is used. Leave unconnected if the interrupt controller is not daisy-chained.
nICFIQ	Output	Interrupt controller	Fast interrupt request to processor.
nICFIQIN	Input	External interrupt controller	Connects to the **nICFIQ** signal of the previous interrupt controller if daisy chaining is used. Connects to logic 1 if the interrupt controller is the last in the daisy chain, or if interrupt controller is not daisy-chained.
nICIRQ	Output	Interrupt controller	Interrupt request to processor.
nICIRQIN	Input	External interrupt controller	Connects to the **nICIRQ** signal of the previous interrupt controller if daisy chaining is used. Connects to logic 1 if the interrupt controller is the last in the daisy chain, or if interrupt controller is not daisy-chained.

Table 3-40 Interrupt controller signals (continued)

Signal	Type	Direction	Description
SCANENABLE	Input	Scan controller	Scan enable.
SCANINHCLK	Input	Scan controller	Scan data input for **HCLK** domain.
SCANOUTHCLK	Output	Scan controller	Scan data output for **HCLK** domain.

Note

For a description of the AMBA signals used by the interrupt controller, see *AMBA signals* on page 1-3.

3.16 64-bit to 32-bit downsizer

The AHB downsizer module, `Downsizer64`, converts 64-bit wide AHB master data buses to narrower 32-bit slave AHB data buses. The downsizer module reduces the width of the data bus by half from an AHB master to an AHB slave. You can use full-width master transfers. This process involves modification of the transfer type, burst, and size, and latching half of the master read data. In addition, multiple slave writes or reads might be required to transfer data to and from the narrow slave.

Figure 3-40 shows the signal interface of the downsizer module.

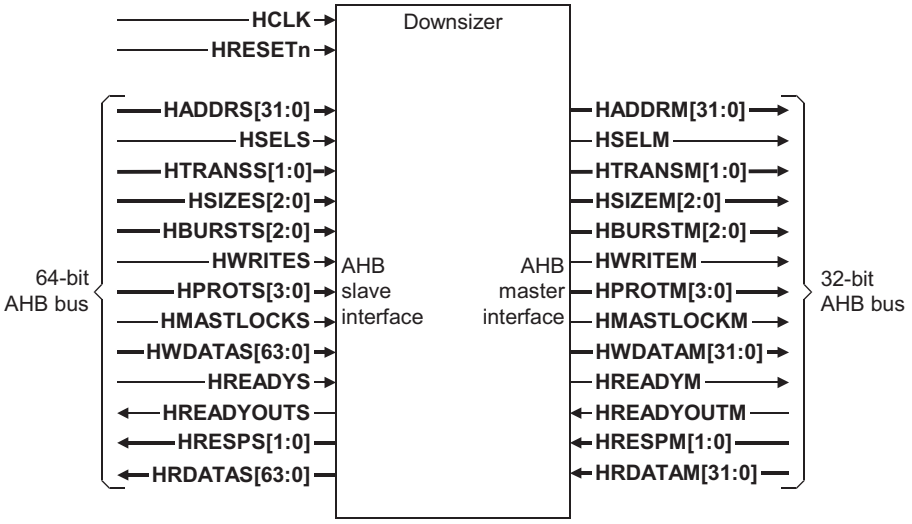


Figure 3-40 Downsizer module

3.16.1 Programmer's model

Programming details for the downsizer are described in the following sections:

- *Downsizer transfers* on page 3-86
- *Unsupported transfers* on page 3-88
- *Burst blocking after error* on page 3-88
- *Slave responses* on page 3-89
- *Modification of control signals* on page 3-89.

Downsizer transfers

The following are options for downsizer transfers:

Downsizer not selected

When the **HSELS** signal of the downsizer module is LOW, the transfer is passed to the 32-bit AHB and **HSELM** is driven LOW. The 32-bit slaves must ignore the transfers by monitoring **HSELM** and **HADDRM**. **HREADYS** from the 64-bit bus is output to **HREADYM**. All 32-bit devices connected to the 32-bit AHB must monitor this **HREADYM** signal to determine the end of the current transfer and the start of the next.

Narrow transfers, downsizer selected

If the downsizer module is selected, and the transfer is 32 bits or less, the downsizer module passes the transfer through. All of the control signals and responses from the slave are left unmodified. In this case, the only function of the downsizer is to route the appropriate half of the wide master write data bus on to the narrow slave data bus for write transfers.

Read transfers require even less control and the narrow slave read data is replicated across the wide master bus.

Table 3-41 shows the handling of narrow transfers.

Table 3-41 Narrow transfer handling

Transfer on 64-bit AHB	Transfer on 32-bit AHB	Address
32, 16, or 8-bit transfer	32, 16, or 8-bit transfer	**HADDR** pass through. If **HADDR[2]** = 0 then **HWDATAS[31:0]** pass through else **HWDATAS[63:32]** pass through. **HRDATAS** = **HRDATAM**, **HRDATAM**.

For 32, 16, and 8-bit transfers, **HWDATA** is selected by bit [2] of the transfer address. If this bit is set to 0, **HWDATA[31:0]** is routed to the 32-bit AHB. If this bit is set to 1, bits [63:32] are routed.

If an ERROR, SPLIT, or RETRY response is received from the 32-bit slave, the downsizer module automatically terminates the current transfer by passing the response to the 64-bit bus. If the current transfer request on the 64-bit bus is a valid transfer (NON_SEQ or SEQ), it is captured by the registers in the

downsizer module and is applied to the 32-bit AHB one cycle later. The downsizer module inserts a wait state on the 64-bit bus to ensure the next transfer is not missed.

If the transfer is a burst and an ERROR response is received from a 32-bit slave, the rest of the burst is blocked. This behavior is generated by the error-blocking logic and can be removed from the code if necessary.

Wide transfers, downsizer selected

The role of the downsizer module is more involved for 64-bit transfers. For both read and write transfers, the wide master transfers are broken down into two narrow slave cycles. The address going to the slave is modified, to ensure that the two slave accesses go to different address locations. Table 3-42 shows the address line modification and data routing.

Table 3-42 Address line modification and data routing

Transfer on 64-bit AHB	Transfer on 32-bit AHB	Address
64-bit transfer	Cycle 1	**HADDR** pass through. **HWDATAS[31:0]** pass through. **HRDATAM** stored in downsizer module. **HADDRS[2:0]** must equal 000.
	Cycle 2	**HADDRM[2]** set to 1. **HWDATAS[63:32]** pass through. **HRDATAM** pass through to **HRDATAS[63:32]**. Previous stored data output to **HRDATAS[31:0]**.

64-bit write transfers are split into two 32-bit transfers on two successive addresses. Table 3-42 lists the generation of **HADDRM[2]** and the routing of data write. Because **HWDATAS** is stable during the two AHB transfers on the 32-bit AHB, no register is required to hold **HWDATA**.

During 64-bit read accesses, the construction of a full-width word for the master to read two slave accesses is required. The data from the first read is latched, and the data from the second read flows straight through the block. Bits [31:0] are always transferred in the first cycle, and bits [63:32] are transferred in the second cycle using the next word address. This transfer characteristic occurs independently of target system endianness.

If an ERROR, SPLIT, or RETRY response is received from the 32-bit slave, the response is passed to the 64-bit bus without delay. If this happens on the first half of the 64-bit transfer, the second half of the transfer is not carried out.

If a two-cycle response is received, the downsizer module automatically aborts the current transfer by inserting an IDLE cycle on the 32-bit bus. If the current transfer request on the 64-bit bus is a valid transfer, NON_SEQ or SEQ, it is captured by the registers in the downsizer module and is applied to the 32-bit AHB one cycle later. The downsizer module inserts a wait state on the 64-bit bus to ensure the next transfer is not missed.

If the transfer is a burst and an ERROR response is received from 32-bit slave, the rest of the burst is blocked.

Unsupported transfers

The following transfer types are not supported by the downsizer module:

Wide transfers

If the downsizer module receives a transfer request greater than 64 bits wide, with **HSELS** = 1, the response is undefined.

Unaligned transfers

Unaligned transfers are not supported.

Burst blocking after error

If an ERROR response is received from a 32-bit slave during a 64-bit burst, and if the 64-bit master continues the burst, the rest of the burst is blocked. During blocking, the ERROR response is fed back to the 64-bit AHB and an IDLE transfer is issued to the 32-bit AHB. The blocking ends when a nonsequential transfer request is detected, or if **HSELS** on the downsizer module is LOW. This feature ensures that there is no discontinuity in **HADDR** and **HTRANS**.

The blocking does not apply to 32,16, or 8-bit transfers. In these cases, the rest of the transfer requests pass through as normal. If a busy cycle is detected during burst blocking, the downsizer module replies with an OKAY response. However, the subsequent sequential transfers are still blocked.

If the ERROR response occurs in the last cycle of the burst, no blocking is generated because the next transfer is an IDLE or nonsequential access. In this case, if the next access is nonsequential, the downsizer module issues an IDLE cycle on the 32-bit AHB in the second cycle of the ERROR response, stores the transfer control information, and applies it to the 32-bit AHB in the following cycle.

A wait state is inserted on the 64-bit bus to enable the 32-bit bus to catch up with the transfer.

Slave responses

When a RETRY or SPLIT response is received, an IDLE cycle is issued to the 32-bit AHB in the second cycle of the RETRY or SPLIT response. If the response occurs during the first half of a 64-bit transfer, the second half is not completed. If the 64-bit master continues to output a valid transfer while the downsizer module is still selected, the transfer is stored and applied to the 32-bit AHB a cycle later. A wait state is output to the 64-bit bus to enable the 32-bit AHB to catch up.

In the case of SPLIT or RETRY responses during 64-bit transfers, the **HRDATA** value received is unpredictable and must be ignored.

Modification of control signals

Table 3-43 lists that, for both read and write transfers, the control signals are modified in the same way.

Table 3-43 Signal mapping when downsizer module is activated

Control signals	Master cycle type		Replaced by slave cycles	Comments
HTRANS	IDLE	to	IDLE	-
	BUSY	to	BUSY	-
	NONSEQ	to	NONSEQ, followed by a SEQ	No change if transfer is 8, 16, or 32-bit.
	SEQ	to	SEQ, followed by a SEQ	No change if transfer is 8, 16, or 32-bit. Exception for WRAP16 boundary, WRAP16 is mapped to INCR and NONSEQ is issued at 32-word boundary.
HADDR[2]	= 0	to	0 then 1	No change if transfer is 8, 16, or 32-bit.
	= 1	-	-	Not permitted.

Table 3-43 Signal mapping when downsizer module is activated (continued)

Control signals	Master cycle type		Replaced by slave cycles	Comments
HSIZE	8/16/32 bit	to	8/16/32 bit	No conversion required.
	64 bit	to	32 bit	Conversion process activated.
	128/256 bit	to	32 bit	Not supported.
HBURST	SINGLE	to	INCR	No change if transfer is 8, 16, or 32-bit.
	INCR	to	INCR	No change if transfer is 8, 16, or 32-bit.
	INCR4	to	INCR8	No change if transfer is 8, 16, or 32-bit.
	WRAP4	to	WRAP8	No change if transfer is 8, 16, or 32-bit.
	INCR8	to	INCR16	No change if transfer is 8, 16, or 32-bit.
	WRAP8	to	WRAP16	No change if transfer is 8, 16, or 32-bit.
	INCR16	to	INCR	No change if transfer is 8, 16, or 32-bit.
	WRAP16	to	INCR	No change if transfer is 8, 16, or 32-bit. NONSEQ broadcast if WRAP boundary is reached.

3.16.2 Signal descriptions

Table 3-44 lists the signal connections for the downsizer module.

Table 3-44 Downsizer interface signals

Signal	Direction	Description
HCLK	Input	System bus clock. Logic is triggered on the clock rising edge.
HRESETn	Input	Activate low asynchronous reset.
64-bit AHB interface signals, AHB slave		
HADDRS[31:0]	Input	Address from the 64-bit AHB.
HBURSTS[2:0]	Input	Burst size information on the 64-bit AHB.
HMASTLOCKS	Input	Indicates that the transfer on the 64-bit AHB is locked.
HPROTS[3:0]	Input	Protection information on the 64-bit AHB.

Table 3-44 Downsizer interface signals (continued)

Signal	Direction	Description
HRDATAS[63:0]	Output	Read data to the 64-bit bus.
HREADYOUTS	Output	**HREADY** signal feedback to the 64-bit bus, indicating that the downsizer is ready for next operation.
HREADYS	Input	**HREADY** signal on the 64-bit AHB bus, indicating start and end of transfer on the 64-bit bus.
HRESPS[2:0]	Output	Response from downsizer module to 64-bit bus.
HSELS	Input	Active HIGH select signal to indicate 32-bit memory range is accessed on the 64-bit AHB.
HSIZES[2:0]	Input	Size of the data on the 64-bit AHB.
HWDATAS[63:0]	Input	Write data from the 64-bit bus.
HWRITES	Input	Indication of write/read operation on the 64-bit AHB.
32-bit AHB interface signals, AHB master		
HADDRM[31:0]	Output	Address for the 32-bit AHB.
HBURSTM[2:0]	Output	Burst size information on the 32-bit AHB.
HMASTLOCKM	Output	Indicates that the transfer on the 32-bit AHB is locked.
HPROTM[3:0]	Output	Protection information on the 32-bit AHB.
HRDATAM[31:0]	Input	Data read back from AHB slaves.
HREADYM	Output	**HREADY** feedback to all slaves on the 32-bit AHB.
HREADYOUTM	Input	**HREADY** from 32-bit AHB slaves or slave multiplexor.
HRESPM[2:0]	Input	**HRESP** from 32-bit AHB slaves or slave multiplexor.
HSELM	Output	Active HIGH select signal to indicate that a 32-bit bus is accessed. This signal can be used to drive a single AHB slave directly, or drive a secondary AHB decoder if multiple 32-bit AHB slaves are used.
HSIZEM[2:0]	Output	Size of the data on the 32-bit AHB.
HWDATAM[31:0]	Output	Write data to 32-bit AHB slaves.
HWRITEM	Output	Indication of write/read operation on the 32-bit AHB.

Instead of reading **HREADY** from the 64-bit bus or **HREADYOUT** from the 32-bit slave multiplexor, the AHB slaves on the 32-bit bus must read the **HREADYM** generated from the downsizer module. This signal is multiplexed between **HREADYOUTM**, when a slave attached to the M port of the downsizer is selected, including during 64-bit to 32-bit conversion, and **HREADYS**, when the downsizer is not selected.

During a conversion, the 64-bit transfer is split into two 32-bit transfers and all the AHB slaves on the 32-bit AHB bus are able to read the **HREADY** signal generated by the activated 32-bit slave. However, this **HREADY** signal must not be passed onto **HREADY** in the 64-bit bus system because this requires the insertion of wait states for the second 32-bit AHB transfer. Because of this, an additional **HREADYM** signal enables the AHB slave to determine when the end of an AHB transfer has occurred.

3.17 64-bit to 32-bit funnel

The AHB funnel, `Funnel`, is a data path multiplexor block used to convert a 64-bit data bus to 32-bit data bus in AMBA systems. This module is used where the 32-bit slaves are to be accessed by a 64-bit master using transfers of word size or smaller. Figure 3-41 shows the signal interface of the funnel module.

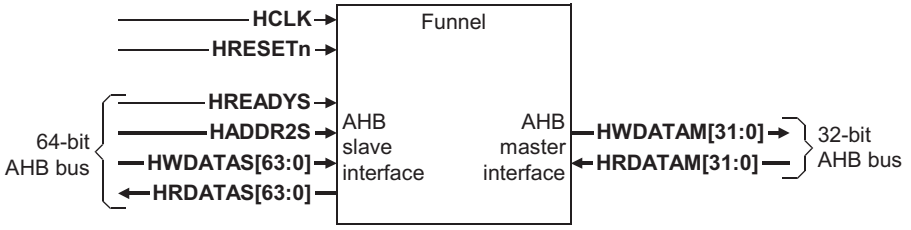


Figure 3-41 Funnel module

3.17.1 Programmer's model

The funnel has a simple interface design to work with AHB devices. Only a small subset of the AHB signals is implemented on the interface, because the other signals can be connected directly between AHB masters and slaves.

The funnel has two interfaces:

- one for connection to AHB masters or an 64-bit AHB bus
- a 32-bit AHB data bus interface for connection to a 32-bit AHB slave or 32-bit AHB slave bus.

Figure 3-42 shows an example application of the funnel.

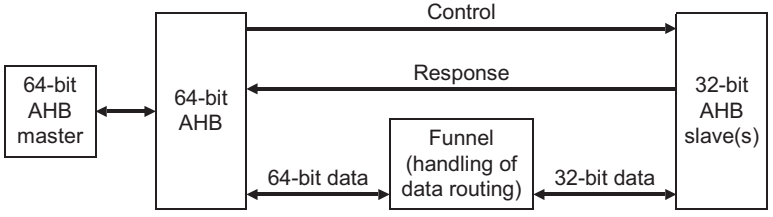


Figure 3-42 Typical funnel connection

The funnel can only function correctly if the 64-bit AHB master does not issue transfers of 64-bit to the 32-bit slave. For 32-bit accesses or accesses at smaller width, the correct word of the data bus is routed to the 32-bit slave. You must use the downsizer module if the transfers cannot be guaranteed to be 32-bit or less. See *64-bit to 32-bit downsizer* on page 3-85.

Funnel selected and accessing even word address

In the address phase of a transfer, the funnel stores bit [2] of the address bus, **HADDR**. During data phase, the stored **HADDR2S**, **Haddrs2Delayed**, determines the side of the **HWDATA** that must be routed to the 32-bit bus. **HWDATA[31:0]** is selected for even word addresses and **HWDATA[63:32]** for even word addresses.

For read transfers, the **HRDATAM** signal is routed to both **HRDATAS[31:0]** and **HRDATA[63:32]** and read by the 64-bit bus master.

Endianness

The funnel supports little-endian and word-invariant big-endian systems. It can be used in full big-endian mode by reversing the polarity of the **HWDATA** multiplexor.

3.17.2 Signal descriptions

The funnel has two AHB ports. The 64-bit AHB port is a slave interface and uses an **S** suffix. The 32-bit AHB port has an M suffix. Table 3-45 lists the signal connections for the funnel module.

Table 3-45 Funnel interface signals

Signal	Direction	Description
HCLK	Input	System bus clock. Logic is triggered on the clock rising edge.
HRESETn	Input	Activate low asynchronous reset.
Signals connected to 64-bit AHB		
HADDR2S	Input	Address bit 2 from 64-bit AHB.
HRDATAS[63:0]	Output	Read data to 64-bit bus.
HREADYS	Input	**HREADY** signal on the 64-bit AHB bus, indicating start and end of transfer on the 64-bit bus.
HWDATAS[63:0]	Input	Write data from 64-bit bus.

Table 3-45 Funnel interface signals (continued)

Signal	Direction	Description
Signals connected to 32-bit AHB		
HRDATAM[31:0]	Input	Read data from 32-bit slave.
HWDATAM[31:0]	Output	Write data from 32-bit slave.

Chapter 4
APB Components

This chapter describes the APB components used in the *AMBA Design Kit* (ADK). It contains the following sections:

4.1 Remap and pause controller

The remap and pause controller, RemapPause, is an APB slave, providing control of the system boot behavior and low-power wait for interrupt mode. Figure 4-1 shows a basic block diagram of the remap and pause controller module.

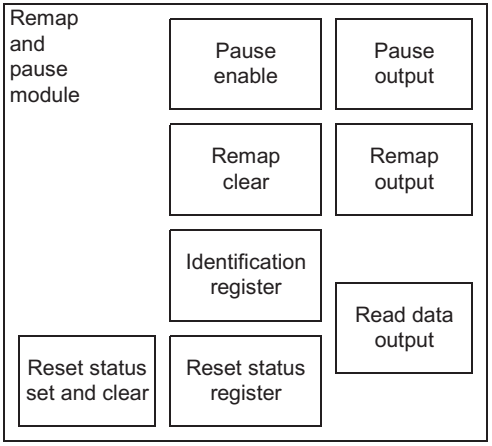


Figure 4-1 Remap and pause module components

4.1.1 Programmer's model

The base address of the remap and pause controller memory 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. Table 4-1 lists the remap and pause controller registers in base offset order.

Table 4-1 Remap and pause register summary

Name	Base offset	Type	Width	Reset value	Description
Pause	0x00	WO	-	-	See *Pause Register* on page 4-3
Remap	0x04	R/W	1	0x0	See *Remap Register* on page 4-3
ResetStatus	0x08	R/W	8	0x01	See *Reset Status Register* on page 4-3
ResetStatusClr	0x0C	WO	8	-	See *Reset Status Clear Register* on page 4-4
RpcPeriphID0	0xFE0	RO	8	0x09	See *Peripheral Identification Registers* on page 4-4
RpcPeriphID1	0xFE4	RO	8	0x18	See *Peripheral Identification Registers* on page 4-4

Table 4-1 Remap and pause register summary (continued)

Name	Base offset	Type	Width	Reset value	Description
RpcPeriphID2	0xFE8	RO	8	0x04	See *Peripheral Identification Registers* on page 4-4
RpcPeriphID3	0xFEC	RO	8	0x00	See *Peripheral Identification Registers* on page 4-4
RpcPCellID0	0xFF0	RO	8	0x0D	See *PrimeCell Identification Registers* on page 4-6
RpcPCellID1	0xFF4	RO	8	0xF0	See *PrimeCell Identification Registers* on page 4-6
RpcPCellID2	0xFF8	RO	8	0x05	See *PrimeCell Identification Registers* on page 4-6
RpcPCellID3	0xFFC	RO	8	0xB1	See *PrimeCell Identification Registers* on page 4-6

Pause Register

Any write to this location sets the **PAUSE** Register output HIGH. The exact effect of writing to this location is not defined, but typically this prevents the processor from fetching further instructions until the receipt of an interrupt of a power-on reset.

Remap Register

Bit 0 of the REMAP Register drives the **REMAP** output. This is typically used to change the memory map from that required during boot-up to that for normal operation. This bit is cleared at reset and, when set, can only be cleared again by resetting the block.

Reset Status Register

The RESETSTATUS Register provides the reset status of the system. Only bit 0 is defined in this specification, and provides the **PRESETn** status:

- set HIGH at reset
- set LOW through the ResetStatusClr register.

Further bits in the RESETSTATUS register can be implemented to provide more detailed reset information. The RESETSTATUS register has a dual mechanism for setting and clearing bits, enabling independent bits to be altered with no knowledge of the other bits in the register. A write to the RESETSTATUS register has the effect of setting all the bits, except bit 0, that have a corresponding HIGH in the write data.

Reset Status Clear Register

The RESETSTATUSCLR Register location clears bits in the RESETSTATUS register. Each HIGH bit in the write data to this location causes the corresponding RESETSTATUS bit to be cleared.

Peripheral Identification Registers

The RPCPERIPHID0-3 registers are four 8-bit registers, that span address locations `0xFE0-0xFEC`. The registers can conceptually be treated as a single 32-bit register. The read-only registers provide the following options for the peripheral:

Part number [11:0] This identifies the peripheral. The three-digit product code `0x809` is used for the remap and pause controller.

Designer [19:12] This is the identification of the designer. ARM Limited is `0x41` (ASCII A).

Revision number [23:20]

This is the revision number of the peripheral. The revision number starts from 0.

Configuration [31:24]

This is the configuration option of the peripheral. The configuration value is 0.

Figure 4-2 shows the register bit assignments.

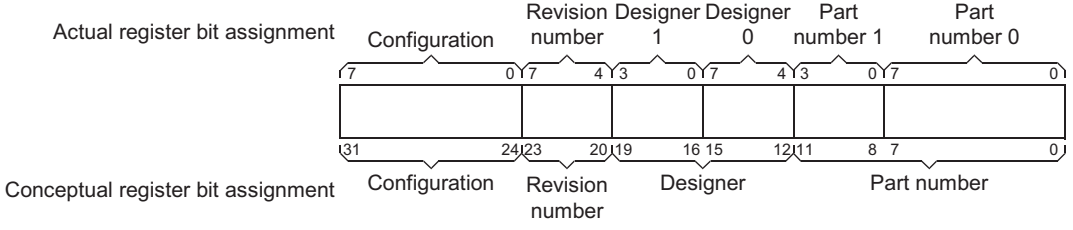


Figure 4-2 RPCPERIPHID0-3 Register bit assignment s

Note

When you design a systems memory map 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.

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

- *Peripheral Identification Register 0*
- *Peripheral Identification Register 1*
- *Peripheral Identification Register 2* on page 4-6
- *Peripheral Identification Register 3* on page 4-6.

Peripheral Identification Register 0

The RPCPERIPHID0 register is hard-coded and the fields within the register determine the reset value. Table 4-2 lists the register bit assignments.

Table 4-2 RPCPERIPHID0 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined must read as zeros
[7:0]	PartNumber0	These bits read back as `0x09`

Peripheral Identification Register 1

The RPCPERIPHID1 register is hard-coded and the fields within the register determine the reset value. Table 4-3 lists the register bit assignments.

Table 4-3 RPCPERIPHID1 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:4]	Designer0	These bits read back as `0x1`
[3:0]	PartNumber1	These bits read back as `0x8`

Peripheral Identification Register 2

The RPCPERIPHID2 register is hard-coded and the fields within the register determine the reset value. Table 4-4 lists the register bit assignments.

Table 4-4 RPCPERIPHID2 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:4]	Revision	These bits return the peripheral revision
[3:0]	Designer1	These bits read back as `0x4`

Peripheral Identification Register 3

The RPCPERIPHID3 register is hard-coded and the fields within the register determine the reset value. Table 4-5 lists the register bit assignments.

Table 4-5 RPCPERIPHID3 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	Configuration	These bits read back as `0x00`

PrimeCell Identification Registers

The RPCPCELLID0-3 registers are four 8-bit wide registers, that span address locations `0xFF0-0xFFC`. The registers can conceptually be treated as a 32-bit register. The register is used as a standard cross-peripheral identification system. The RPCPCELLID register is set to `0xB105F00D`. Figure 4-3 shows the register bit assignments.

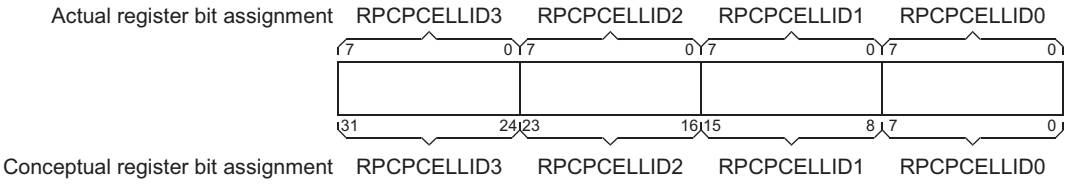


Figure 4-3 RPCPCELLID0-3 Register bit assignments

The four, 8-bit PrimeCell identification registers are described in the following subsections:

- *PrimeCell Identification Register 0*
- *PrimeCell Identification Register 1*
- *PrimeCell Identification Register 2*
- *PrimeCell Identification Register 3* on page 4-8.

PrimeCell Identification Register 0

The RPCPCELLID0 register is hard-coded and the fields within the register determine the reset value. Table 4-6 lists the register bit assignments.

Table 4-6 RPCPCELLID0 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	RpcPCellID0	These bits read back as `0x0D`

PrimeCell Identification Register 1

The RPCPCELLID1 register is hard-coded and the fields within the register determine the reset value. Table 4-7 lists the register bit assignments.

Table 4-7 RPCPCELLID1 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	RpcPCellID1	These bits read back as `0xF0`

PrimeCell Identification Register 2

The RPCPCELLID2 register is hard-coded and the fields within the register determine the reset value. Table 4-8 lists the register bit assignments.

Table 4-8 RPCPCELLID2 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	RpcPCellID2	These bits read back as `0x05`

PrimeCell Identification Register 3

The RPCPCELLID3 register is hard-coded and the fields within the register determine the reset value. Table 4-9 lists the register bit assignments.

Table 4-9 RPCPCELLID3 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	RpcPCellID3	These bits read back as `0xB1`

4.1.2 Signal descriptions

Table 4-10 lists the non-AMBA signals used by the remap and pause controller.

Table 4-10 Remap and pause controller signals

Signal	Type	Direction	Description
nFIQ	FIQ output	Input	FIQ interrupt input from the interrupt controller
nIRQ	IRQ output	Input	IRQ interrupt input from the interrupt controller
PAUSE	Pause mode	Output	HIGH when in the *wait for interrupt* pause mode, and LOW at all other times
REMAP	Reset memory map	Output	LOW when the reset memory map is in use, and HIGH when the normal memory map is in use

Note

For a description of the AMBA signals used by the remap and pause controller, see *AMBA signals* on page 1-3.

4.2 Example APB slave

The example APB slave, `EgApbSlave`, provides sample HDL code, that can be used as a basis for further enhancement, to produce a slave on the APB. Figure 4-4 shows that the example APB slave consists of five functional blocks contained within one top-level entity.

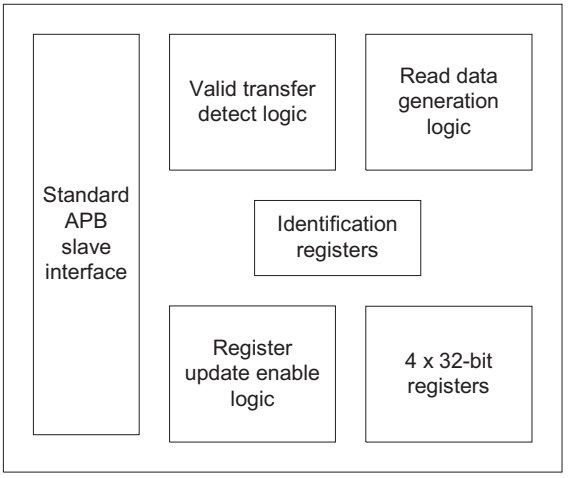


Figure 4-4 Example APB slave components

The example APB slave includes the following features:

- A simple APB slave interface.
- System bus clock and reset, that are synchronous between AHB and APB domains.
- 32-bit data bus, endian-independent, but data handling is 32-bits only.
- Data transfers require two clock cycles.
- 32-bit address bus.
- Four 32-bit registers to hold the write data. Registers can also be read.
- Seven read-only locations that return logical functions of the registered write data.

4.2.1 Programmer's model

The slave only responds to a transfer when not in reset, **PRESETn** HIGH, and when **PSEL** is HIGH and **PENABLE** is LOW at the time of a rising edge on **PCLK**. The timing of the slave corresponds to the APB specification.

The functionality of the slave depends on the address location being accessed and the direction of the data. Only bits [11:2] of the address are decoded, and this supports accesses with word-aligned addresses. Because of the partial address decode, the same location is accessible at several addresses. For example, Register 0 is accessible at `0x0000`, `0x1000`, `0x2000`, ..., until the decode range of **PSEL** from the bridge is exceeded.

Table 4-11 lists the example APB slave memory map.

Table 4-11 Example APB slave memory map

Name	Base offset	Type	Width	Reset value	Description
R0	`0x00`	R/W	32	`0x00000000`	Read/write data into R0.
R1	`0x04`	R/W	32	`0x00000000`	Read/write data into R1.
R2	`0x08`	R/W	32	`0x00000000`	Read/write data into R2.
R3	`0x0C`	R/W	32	`0x00000000`	Read/write data into R3.
READ10	`0x10`	RO	32	`0xFFFFFFFF`	Not R0.
READ14	`0x14`	RO	32	`0x00000000`	R0 and R1.
READ18	`0x18`	RO	32	`0x00000000`	R1 or R2.
READ1C	`0x1C`	RO	32	`0x00000000`	R2 xor R3.
READ20	`0x20`	RO	32	`0x00000000`	R0 and R1 and R2 and R3.
READ24	`0x24`	RO	32	`0x00000000`	R0 or R1 or R2 or R3.
READ28	`0x28`	RO	32	`0x00000000`	R0 xor R1 xor R2 xor R3.
Reserved	`0x2C-0xFDC`	R/W	32	`0x00000000`	Read as zero. Write has no effect.
PERIPHERALID0	`0xFE0`	RO	8	`0x06`	Peripheral ID register 0.
PERIPHERALID1	`0xFE4`	RO	8	`0x18`	Peripheral ID register 1.
PERIPHERALID2	`0xFE8`	RO	8	`0x04`	Peripheral ID register 2.
PERIPHERALID3	`0xFEC`	RO	8	`0x00`	Peripheral ID register 3.
PRIMECELLID0	`0xFF0`	RO	8	`0x0D`	PrimeCell ID register 0.
PRIMECELLID1	`0xFF4`	RO	8	`0xF0`	PrimeCell ID register 1.
PRIMECELLID2	`0xFF8`	RO	8	`0x05`	PrimeCell ID register 2.
PRIMECELLID3	Base + `0xFFC`	RO	8	`0xB1`	PrimeCell ID register 3.

The functionality in Table 4-11 on page 4-10 is the same as the retry slave described in *Example retry slave* on page 3-10, except there is no requirement to generate wait states and responses, and all data is a fixed size (32-bit).

Note

Address locations `0x2C-0xFDC` read as zero and perform no operation when written to.

The peripheral ID information is arranged as four 8-bit registers that span address locations `0xFE0-0xFEC`. The registers can conceptually be treated as a 32-bit read-only register. Table 4-12 lists the format of this information and the values for the peripheral.

Table 4-12 Peripheral ID format

Register field	Description
Part number [11:0]	This identifies the peripheral. The three digit product code `0x806` is used.
Designer ID [19:12]	This is the identification of the designer. ARM Limited is `0x41`, ASCII A.
Revision [23:20]	This is the revision number of the peripheral. The revision number starts from 0.
Configuration [31:24]	This is the configuration option of the peripheral. The configuration value is 0.

4.2.2 Signal descriptions

The example APB slave uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

4.3 Peripheral to bridge multiplexor

The peripheral to bridge multiplexor, MuxP2B, connects the read data outputs of each APB slave to the APB bridge module, using the **PSELx** signals to select the required data source. The default configuration is sixteen slots. The read data is switched for the duration of an APB transfer, when the **PSELx** signal is valid. A default value of zero is used when no slaves are selected. Figure 4-5 shows the peripheral to bridge multiplexor block diagram.

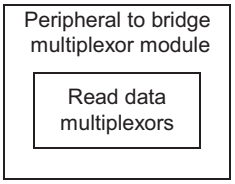


Figure 4-5 Peripheral to bridge multiplexor module components

Note

The MuxP2B is used with the Ahb2Apb which is AMBA 2.0 version compatible. A newer version, the MuxPToB is available to support APB 3.0. It is used with the AhbToApb, which is AMBA version 3.0 compatible.

4.3.1 Signal descriptions

The peripheral to bridge multiplexor uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

4.4 Watchdog unit

The Watchdog module is an Advanced Microcontroller Bus Architecture (AMBA) compliant *System-on-Chip* (SoC) peripheral developed, tested and licensed by ARM Limited. The Watchdog module is an AMBA slave module and connects to the *Advanced Peripheral Bus* (APB).

The Watchdog module is based around a 32-bit down counter that is initialized from the Reload Register, WdogLoad. The watchdog clock generates a regular interrupt, **WDOGINT**, depending on a programmed value. The counter decrements by one on each positive clock edge of **WDOGCLK** when the clock enable **WDOGCLKEN** is HIGH. The watchdog monitors the interrupt and asserts a reset WDOGRES signal, when the counter reaches zero, and the counter is stopped. On the next enabled **WDOGCLK** clock edge the counter is reloaded from the WdogLoad Register and the count down sequence continues. If the interrupt is not cleared by the time that the counter next reaches zero then the Watchdog module reasserts the reset signal. The Watchdog module is intended to be used to apply a reset to a system in the event of a software failure, providing a way of recovering from software crashes. You can enable or disable the watchdog unit as required. Figure 4-6 shows the watchdog block diagram.

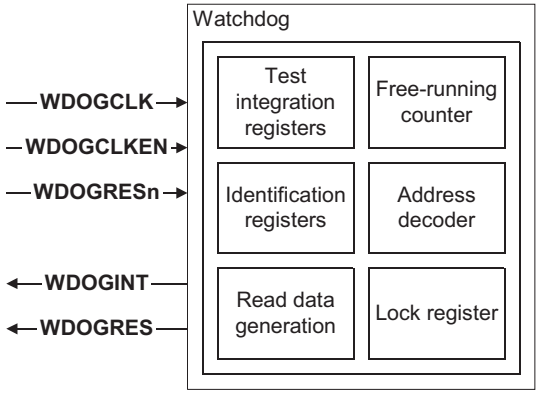


Figure 4-6 Watchdog components

4.4.1 Programmer's model

Table 4-13 lists the watchdog registers.

Table 4-13 Watchdog unit register summary

Name	Base offset	Type	Width	Reset value	Description
WDOGLOAD	0x00	R/W	32	0xFFFFFFFF	See *Watchdog Load Register* on page 4-15
WDOGVALUE	0x04	RO	32	0xFFFFFFFF	See *Watchdog Value Register* on page 4-15
WDOGCONTROL	0x08	R/W	2	0x0	See *Watchdog Control Register* on page 4-15
WDOGINTCLR	0x0C	WO	-	-	See *Watchdog Clear Interrupt Register* on page 4-15
WDOGRIS	0x10	RO	1	0x0	See *Watchdog Raw Interrupt Status Register* on page 4-16
WDOGMIS	0x14	RO	1	0x0	See *Watchdog Interrupt Status Register* on page 4-16
WDOGLOCK	0xC00	R/W	32	0x0	See *Watchdog Lock Register* on page 4-17
WDOGITCR	0xF00	R/W	1	0x0	See *Watchdog Integration Test Control Register* on page 4-18
WDOGITOP	0xF04	WO	2	0x0	See *Watchdog Integration Test Output Set Register* on page 4-18
WDOGPERIPHID0	0xFE0	RO	8	0x05	See *Peripheral Identification Register 0* on page 4-20
WDOGPERIPHID1	0xFE4	RO	8	0x18	See *Peripheral Identification Register 1* on page 4-20
WDOGPERIPHID2	0xFE8	RO	8	0x04	See *Peripheral Identification Register 2* on page 4-21
WDOGPERIPHID3	0xFEC	RO	8	0x00	See *Peripheral Identification Register 3* on page 4-21
WDOGPCELLID0	0xFF0	RO	8	0x0D	See *PrimeCell Identification Register 0* on page 4-22
WDOGPCELLID1	0xFF4	RO	8	0xF0	See *PrimeCell Identification Register 1* on page 4-22
WDOGPCELLID2	0xFF8	RO	8	0x05	See *PrimeCell Identification Register 2* on page 4-22
WDOGPCELLID3	0xFFC	RO	8	0xB1	See *PrimeCell Identification Register 3* on page 4-23

The following registers are described in this section:

Watchdog Load Register

The WDOGLOAD Register is a 32-bit register containing the value from which the counter is to decrement. When this register is written to, the count is immediately restarted from the new value. The minimum valid value for WDOGLOAD is 1.

Watchdog Value Register

The WDOGVALUE Register gives the current value of the decrementing counter.

Watchdog Control Register

The WDOGCONTROL Register is a read/write register that enables the software to control the watchdog unit. Figure 4-7 shows the register bit assignments.

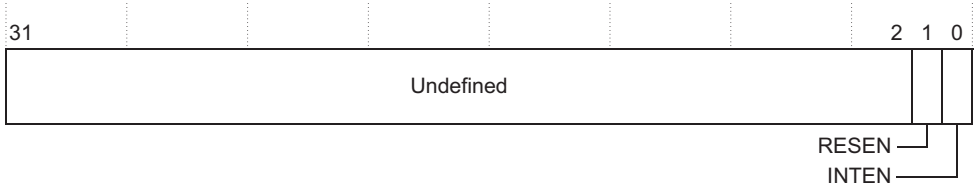


Figure 4-7 WDOGCONTROL Register bit assignments

Table 4-14 lists the register bit assignments.

Table 4-14 WDOGCONTROL Register bit assignments

Bits	Name	Function
[31:2]	-	Reserved, read undefined, must read as zeros.
[1]	RESEN	Enable Watchdog reset output, **WDOGRES**. Acts as a mask for the reset output. Set HIGH to enable the reset, and LOW to disable the reset.
[0]	INTEN	Enable the interrupt event, **WDOGINT**. Set HIGH to enable the counter and the interrupt, and set LOW to disable the counter and interrupt. Reloads the counter from the value in WDOGLOAD when the interrupt is enabled, and was previously disabled.

Watchdog Clear Interrupt Register

A write of any value to the WDOGINTCLR Register clears the watchdog interrupt, and reloads the counter from the value in WDOGLOAD.

Watchdog Raw Interrupt Status Register

The WDOGRIS Register is read-only. It indicates the raw interrupt status from the counter. This value is ANDed with the interrupt enable bit from the control register to create the masked interrupt, that is passed to the interrupt output pin. Figure 4-8 shows the register bit assignments.

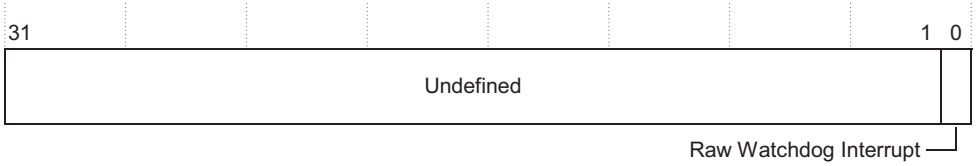


Figure 4-8 WDOGRIS Register bit assignments

Table 4-15 lists the register bit assignments.

Table 4-15 WDOGRIS Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined, must read as zeros
[0]	Raw Watchdog Interrupt	Raw interrupt status from the counter

Watchdog Interrupt Status Register

The WDOGMIS Register is read-only. It indicates the masked interrupt status from the counter. This value is the logical AND of the raw interrupt status with the INTEN bit from the control register, and is the same value that is passed to the interrupt output pin. Figure 4-9 shows the register bit assignments.

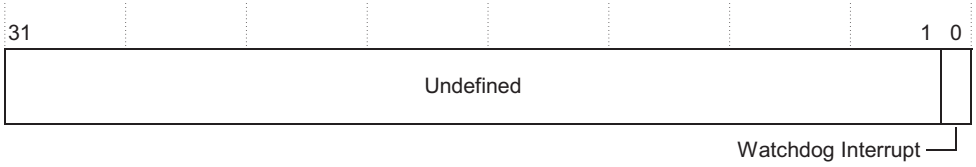


Figure 4-9 WDOGMIS Register bit assignments

Table 4-16 lists the register bit assignments.

Table 4-16 WDOGMIS Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined, must read as zeros
[0]	Watchdog Interrupt	Enabled interrupt status from the counter

Watchdog Lock Register

The WDOGLOCK Register is write-only. Use of this register causes write-access to all other registers to be disabled. This is to prevent rogue software from disabling the watchdog functionality. Writing a value of `0x1ACCE551` enables write access to all other registers. Writing any other value disables write accesses. A read from this register returns only the bottom bit:

- 0 indicates that write access is enabled, not locked
- 1 indicates that write access is disabled, locked.

Figure 4-10 shows the register bit assignments.

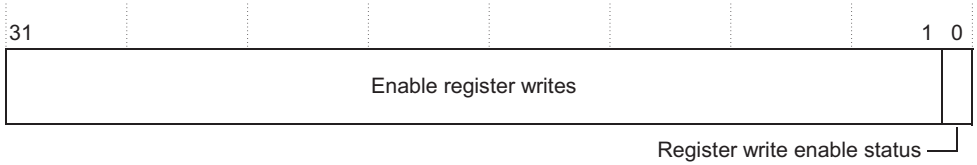


Figure 4-10 WDOGLOCK Register bit assignments

Table 4-17 lists the bit assignments for the WDOGLOCK register.

Table 4-17 WDOGLOCK Register bit assignments

Bits	Name	Function
[31:1]	Enable register writes	Enable write access to all other registers by writing `0x1ACCE551`. Disable write access by writing any other value.
[0]	Register write enable status	0 = write access to all other registers is enabled, default 1 = write access to all other registers is disabled

Watchdog Integration Test Control Register

The WDOGITCR Register is read/write. It is a single-bit register that enables integration test mode. When in this mode, the masked interrupt output, **WDOGINT**, and reset output, **WDOGRES**, are directly controlled by the test output set register.

Figure 4-11 shows the register bit assignments.

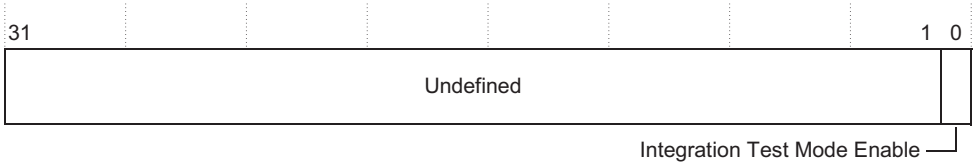


Figure 4-11 WDOGITCR Register bit assignments

Table 4-18 lists the register bit assignments.

Table 4-18 WDOGITCR Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined, must read as zeros
[0]	Integration Test Mode Enable	When set HIGH, places the Watchdog into integration test mode

Watchdog Integration Test Output Set Register

The WDOGITOP Register is write-only. When in integration test mode, the enabled interrupt output and reset output are driven directly from the values in this register. Figure 4-12 shows the register bit assignments.

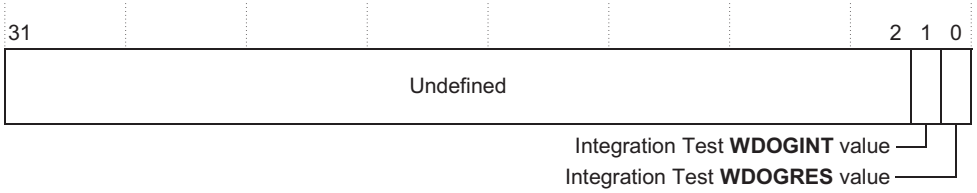


Figure 4-12 WDOGITOP Register bit assignments

Table 4-19 lists the register bit assignments.

Table 4-19 WDOGITOP Register bit assignments

Bits	Name	Function
[31:2]	-	Reserved, read undefined, must read as zeros
[1]	Integration Test **WDOGINT** value	Value output on **WDOGINT** when in Integration Test Mode
[0]	Integration Test **WDOGRES** value	Value output on **WDOGRES** when in Integration Test Mode

Peripheral Identification Registers

The WDOGPERIPHID0-3 registers are four 8-bit registers, that span address locations `0xFE0-0xFEC`. The registers can conceptually be treated as a single 32-bit register. The read-only registers provide the following options for the peripheral:

Part number [11:0] This identifies the peripheral. The three digit product code `0x805` is used for the watchdog unit.

Designer [19:12] This is the identification of the designer. ARM Limited is `0x41`, ASCII A.

Revision number [23:20]

This is the revision number of the peripheral. The revision number starts from 0.

Configuration [31:24]

This is the configuration option of the peripheral. The configuration value is 0.

Figure 4-13 shows the register bit assignments.

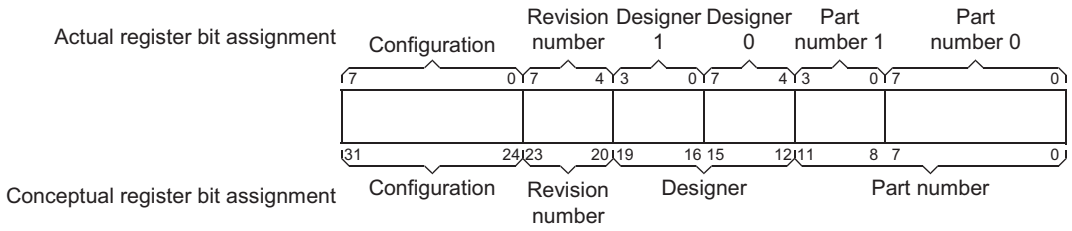


Figure 4-13 WDOGPERIPHID0-3 Register bit assignments

Note

When you design a system memory map 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.

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

- *Peripheral Identification Register 0*
- *Peripheral Identification Register 1*
- *Peripheral Identification Register 2* on page 4-21
- *Peripheral Identification Register 3* on page 4-21.

Peripheral Identification Register 0

The WDOGPERIPHID0 Register is hard-coded and the fields within the register determine the reset value. Table 4-20 lists the register bit assignments.

Table 4-20 WDOGPERIPHID0 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	PartNumber0	These bits read back as `0x05`

Peripheral Identification Register 1

The WDOGPERIPHID1 Register is hard-coded and the fields within the register determine the reset value. Table 4-21 lists the register bit assignments.

Table 4-21 WDOGPERIPHID1 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:4]	Designer0	These bits read back as `0x1`
[3:0]	PartNumber1	These bits read back as `0x08`

Peripheral Identification Register 2

The WDOGPERIPHID2 Register is hard-coded and the fields within the register determine the reset value. Table 4-22 lists the register bit assignments.

Table 4-22 WDOGPERIPHID2 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:4]	Revision	These bits read back as `0x0`
[3:0]	Designer1	These bits read back as `0x4`

Peripheral Identification Register 3

The WDOGPERIPHID3 Register is hard-coded and the fields within the register determine the reset value. Table 4-23 lists the register bit assignments.

Table 4-23 WDOGPERIPHID3 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	Configuration	These bits read back as `0x00`

PrimeCell Identification Registers

The WDOGPCELLID0-3 Registers are four 8-bit wide registers, that span address locations `0xFF0-0xFFC`. The registers can conceptually be treated as a 32-bit register. The register is used as a standard cross-peripheral identification system. The WDOGPCELLID Register is set to `0xB105F00D`. Figure 4-14 shows the register bit assignments.

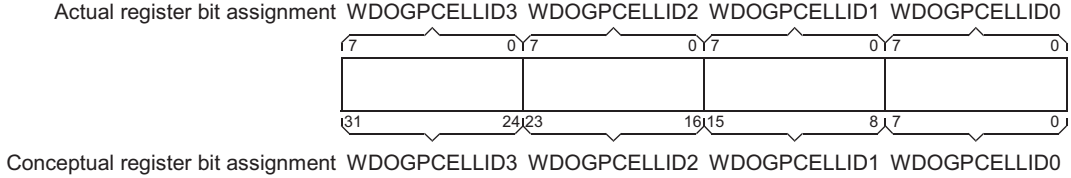


Figure 4-14 WDOGPCELLID0-3 Register bit assignments

The four, 8-bit PrimeCell identification registers are described in the following subsections:

- *PrimeCell Identification Register 0*
- *PrimeCell Identification Register 1*
- *PrimeCell Identification Register 2*
- *PrimeCell Identification Register 3* on page 4-23.

PrimeCell Identification Register 0

The WDOGPCELLID0 Register is hard-coded and the fields within the register determine the reset value. Table 4-24 lists the register bit assignments.

Table 4-24 WDOGPCELLID0 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	WdogPCellID0	These bits read back as `0x0D`

PrimeCell Identification Register 1

The WDOGPCELLID1 Register is hard-coded and the fields within the register determine the reset value. Table 4-25 lists the register bit assignments.

Table 4-25 WDOGPCELLID1 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	WdogPCellID1	These bits read back as `0xF0`

PrimeCell Identification Register 2

The WDOGPCELLID2 Register is hard-coded and the fields within the register determine the reset value. Table 4-26 lists the register bit assignments.

Table 4-26 WDOGPCELLID2 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	WdogPCellID2	These bits read back as `0x05`

PrimeCell Identification Register 3

The WDOGPCELLID3 register is hard-coded and the fields within the register determine the reset value. Table 4-27 lists the register bit assignments.

Table 4-27 WDOGPCELLID3 Register bit assignments

Bits	Name	Description
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	WdogPCellID3	These bits read back as `0xB1`

4.4.2 Signal descriptions

Table 4-28 lists the non-AMBA signals used by the watchdog unit.

Table 4-28 Watchdog unit signals

Signal	Type	Direction	Description
WDOGCLK	Watchdog clock	Input	The watchdog clock input.
WDOGCLKEN	Watchdog clock enable	Input	The enable for the watchdog clock input. The counters only decrement on a rising edge of **WDOGCLK** when **WDOGCLKEN** is HIGH.
WDOGRESn	Watchdog reset	Input	The watchdog clock domain reset input.
WDOGINT	Watchdog interrupt	Output	The watchdog interrupt.
WDOGRES	Watchdog reset	Output	The watchdog timeout reset.

Note

For a description of the AMBA signals used by the watchdog unit, see *AMBA signals* on page 1-3.

4.5 Dual input timer

The ARM Dual-Timer module is an AMBA compliant SoC peripheral developed, tested and licensed by ARM Limited. For more information, see the AMBA Specification (Rev 2.0).

The Dual Input Timers module, `Timers` is an AMBA slave module and connects to the APB. The Dual-Timer module consists of two programmable 32/16-bit down counters that can generate interrupts on reaching zero. A Timer module can be programmed for a 32-bit or 16-bit counter size and one of three timer modes using the Control Register. The operation of each Timer module is identical. It has one of three timer modes:

- free-running
- periodic
- one-shot.

The Dual Input Timers module, `Timers`, provides access to two interrupt-generating, programmable 32-bit *Free-Running decrementing Counters* (FRCs). The FRCs operate from a common timer clock, **TIMCLK** with each FRC having its own clock enable input, **TIMCLKEN1** and **TIMCLKEN2**. Each FRC also has a prescaler that can divide down the enabled **TIMCLK** rate by 1, 16, or 256. This enables the count rate for each FRC to be controlled independently using their individual clock enables and prescalers.

The system clock, **PCLK**, controls the programmable registers, and the second clock input drives the counter, enabling the counters to run from a much slower clock than the system clock. The two clocks must be synchronous while register accesses are performed. **TIMCLK** can be equal to or be a submultiple of the **PCLK** frequency. However, the positive edges of **TIMCLK** and **PCLK** must be synchronous and balanced.

Figure 4-15 on page 4-25 shows a top-level block diagram of the timers.

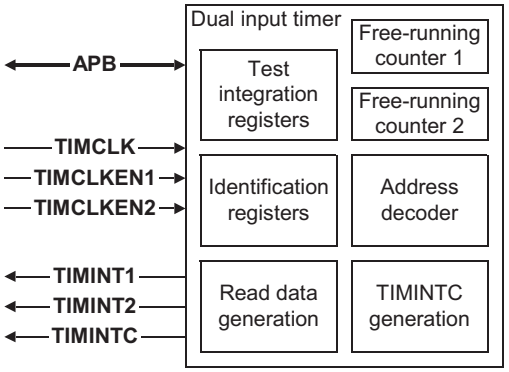


Figure 4-15 Dual input timer components

4.5.1 Functional description

Two timers are defined as the default provided, although you can easily expand this through extra instantiations of the FRC block. The same principle of simple expansion has been applied to the register configuration, enabling more complex counters to be used. For each timer, the following modes of operation are available:

Free-running mode

The counter wraps after reaching its zero value, and continues to count down from the maximum value. This is the default mode.

Periodic timer mode

The counter generates an interrupt at a constant interval, reloading the original value after wrapping past zero.

One-shot timer mode

The counter generates an interrupt once. When the counter reaches zero, it halts until reprogrammed by the user. This can be achieved by either clearing the One Shot Count bit in the control register, in which case the count proceeds according to the selection of Free-running or Periodic mode, or by writing a new value to the Load Value register.

4.5.2 Operation

Each timer has an identical set of registers as shown in Table 4-29 on page 4-28. The operation of each timer is identical. The timer is loaded by writing to the load register and, if enabled, counts down to zero. When a counter is already running, writing to the load register causes the counter to immediately restart at the new value. Writing to the

background load value has no effect on the current count. The counter continues to decrement to zero, and then recommences from the new load value, if in periodic mode, and one shot mode is not selected.

When zero is reached, an interrupt is generated. The interrupt can be cleared by writing to the clear register. If One Shot Mode is selected, the counter halts on reaching zero until the you deselect One Shot Mode, or write a new Load value. Otherwise, after reaching a zero count, if the timer is operating in free-running mode it continues to decrement from its maximum value. If periodic timer mode is selected, the timer reloads the count value from the Load Register and continues to decrement. In this mode the counter effectively generates a periodic interrupt. The mode is selected by a bit in the Timer Control Register. See Table 4-30 on page 4-30. At any point, the current counter value can be read from the Current Value Register. The counter is enabled by a bit in the Control Register. At reset, the counter is disabled, the interrupt is cleared, and the load register is set to zero. The mode and prescale values are set to free-running, and clock divide of 1 respectively. Figure 4-16 shows a block diagram of the free-running timer module.

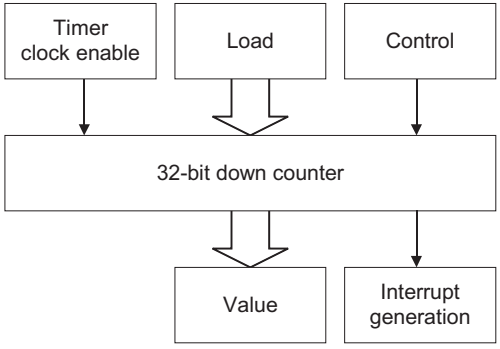


Figure 4-16 Free-running timer block

The timer clock enable is generated by a prescale unit. The enable is then used by the counter to create a clock with a timing of one of the following:

- the system clock
- the system clock divided by 16, generated by 4 bits of prescale
- the system clock divided by 256, generated by a total of 8 bits of prescale.

Figure 4-17 on page 4-27 shows how the timer clock frequency is selected in the prescale unit. This enables you to clock the timer at different frequencies.

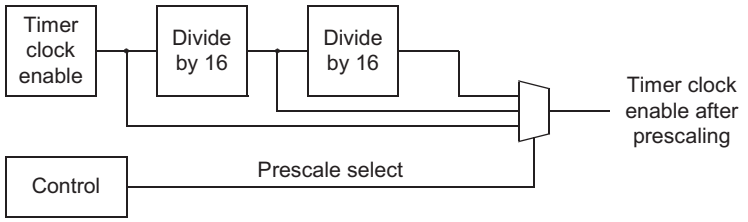


Figure 4-17 Prescale clock enable generation

Note

This selection is in addition to any similar facility already provided as part of any clock generation logic external to the Timers.

Interrupt generation

An interrupt is generated when the full 32-bit counter reaches zero, and is only cleared when the TimerXClear location is written to. A register holds the value until the interrupt is cleared. The most significant carry bit of the counter detects the counter reaching zero.

You can mask interrupts by writing 0 to the Interrupt Enable bit in the Control register. Both the raw interrupt status, prior to masking, and the final interrupt status, after masking, can be read from status registers.

The interrupts from the individual counters, after masking, are logically ORed into a combined interrupt, **TIMINTC**, provides an additional output from the Timer peripheral.

4.5.3 Clocking

The timers have two clock inputs, **PCLK** and **TIMCLK**. **PCLK** is the main APB system clock, and is used by the register interface. **TIMCLK** is the input to the prescale units and the decrementing counters. A pulse on **TIMCLK** must be qualified by the appropriate **TIMCLKENx** being HIGH.

The design of the timers assumes that **PCLK** and **TIMCLK** are synchronous. To enable the counter to operate from a lower effective frequency than that at which **PCLK** is running, you can do either of the following:

- both **PCLK** and **TIMCLK** inputs are connected to the APB **PCLK** signal, and **TIMCLKENx** is pulsed HIGH at the required frequency, synchronized to **PCLK**

- **TIMCLKENx** is tied HIGH and an enabled version of **PCLK** is fed into the TIMCLK input, giving sparse clock pulses synchronous to **PCLK**.

This provision of two clock inputs enables the counters to continue to run while the APB system is in a sleep state whereby **PCLK** is disabled. The changeover periods when **PCLK** is disabled and enabled must be handled by external system control logic, to ensure that the **PCLK** and **TIMCLK** inputs are fed with synchronous signals when any register access is to occur.

4.5.4 Programmer's model

Table 4-29 lists the timer registers.

Table 4-29 Timer register summary

Name	Base offset	Type	Width	Reset value	Description
TIMER1LOAD	0x00	R/W	32	0x00000000	See *Load Register* on page 4-29
TIMER1VALUE	0x04	RO	32	0xFFFFFFFF	See *Current Value Register* on page 4-30
TIMER1CONTROL	0x08	R/W	8	0x20	See *Timer Control Register* on page 4-30
TIMER1INTCLR	0x0C	WO	-	-	See *Interrupt Clear Register* on page 4-31
TIMER1RIS	0x10	RO	1	0x0	See *Raw Interrupt Status Register* on page 4-31
TIMER1MIS	0x14	RO	1	0x0	See *Interrupt Status Register* on page 4-32
TIMER1BGLOAD	0x18	R/W	32	0x00000000	See *Background Load Register* on page 4-32
TIMER2LOAD	0x20	R/W	32	0x00000000	See *Load Register* on page 4-29
TIMER2VALUE	0x24	RO	32	0xFFFFFFFF	See *Current Value Register* on page 4-30
TIMER2CONTROL	0x28	R/W	8	0x20	See *Timer Control Register* on page 4-30
TIMER2INTCLR	0x2C	WO	-	-	See *Interrupt Clear Register* on page 4-31
TIMER2RIS	0x30	RO	1	0x0	See *Raw Interrupt Status Register* on page 4-31
TIMER2MIS	0x34	RO	1	0x0	See *Interrupt Status Register* on page 4-32
TIMER2BGLOAD	0x38	R/W	32	0x00000000	See *Background Load Register* on page 4-32
TIMERITCR	0xF00	R/W	1	0x0	See *Integration Test Control Register* on page 4-33
TIMERITOP	0xF04	WO	2	0x0	See *Integration Test Output Set Register* on page 4-33

Table 4-29 Timer register summary (continued)

Name	Base offset	Type	Width	Reset value	Description
TIMERPERIPHID0	0xFE0	RO	8	0x04	See *Peripheral Identification Register 0* on page 4-35
TIMERPERIPHID1	0xFE4	RO	8	0x18	See *Peripheral Identification Register 1* on page 4-36
TIMERPERIPHID2	0xFE8	RO	8	0x04	See *Peripheral Identification Register 2* on page 4-36
TIMERPERIPHID3	0xFEC	RO	8	0x00	See *Peripheral Identification Register 3* on page 4-36
TIMERPCELLID0	0xFF0	RO	8	0x0D	See *PrimeCell Identification Register 0* on page 4-37
TIMERPCELLID1	0xFF4	RO	8	0xF0	See *PrimeCell Identification Register 1* on page 4-38
TIMERPCELLID2	0xFF8	RO	8	0x05	See *PrimeCell Identification Register 2* on page 4-38
TIMERPCELLID3	0xFFC	RO	8	0xB1	See *PrimeCell Identification Register 3* on page 4-38

Load Register

The TIMERXLOAD Register is a 32-bit register containing the value from which the counter is to decrement. This is the value used to reload the counter when Periodic mode is enabled, and the current count reaches zero.

When this register is written to directly, the current count is immediately reset to the new value at the next rising edge of **TIMCLK** that is enabled by **TIMCLKEN**.

The value in this register is also overwritten if the TIMERXBGLOAD Register is written to, but the current count is not immediately affected.

If values are written to both the TIMERXLOAD and TIMERXBGLOAD Registers before an enabled rising edge on **TIMCLK**, the following occurs:

1. On the next enabled **TIMCLK** edge, the value written to the TIMERXLOAD value replaces the current count value.
2. Then, each time the counter reaches zero, the current count value is reset to the value written to TIMERXBGLOAD.

Reading from the TIMERXLOAD Register at any time after the two writes have occurred retrieves the value written to TIMERXBGLOAD. That is, the value read from TIMERXLOAD is always the value that takes effect for Periodic mode after the next time the counter reaches zero.

Current Value Register

The TIMERXVALUE Register gives the current value of the decrementing counter.

Timer Control Register

The TIMERXCONTROL Register is a read/write register. Figure 4-18 shows the register bit assignments.

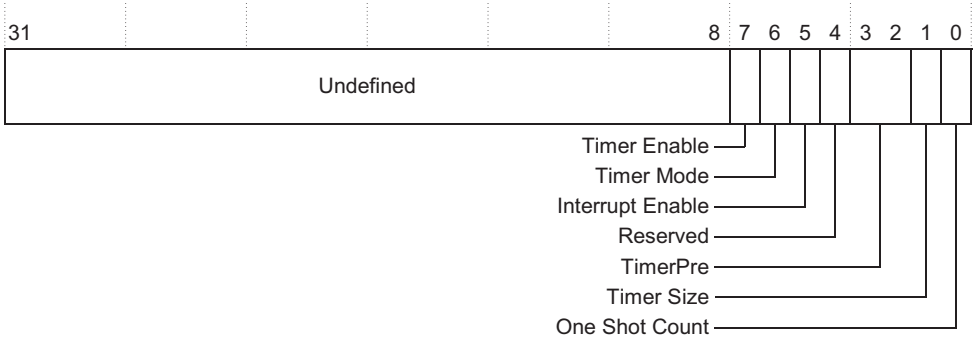


Figure 4-18 TIMERXCONTROL Register bit assignments

Table 4-30 lists the register bit assignments.

Table 4-30 TIMERXCONTROL Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must read as zeros
[7]	Timer Enable	Enable bit: 0 = Timer disabled, default 1 = Timer enabled.
[6]	Timer Mode	Mode bit: 0 = Timer is in free-running mode, default 1 = Timer is in periodic mode.
[5]	Interrupt Enable	Interrupt Enable bit: 0 = Timer Interrupt disabled 1 = Timer Interrupt enabled, default.
[4]	RESERVED	Reserved bit, do not modify, and ignore on read

Table 4-30 TIMERXCONTROL Register bit assignments (continued)

Bits	Name	Function
[3:2]	TimerPre	Prescale bits: 00 = 0 stages of prescale, clock is divided by 1, default 01 = 4 stages of prescale, clock is divided by 16 10 = 8 stages of prescale, clock is divided by 256 11 = Undefined, do not use.
[1]	Timer Size	Selects 16/32 bit counter operation: 0 = 16-bit counter, default 1 = 32-bit counter.
[0]	One Shot Count	Selects one-shot or wrapping counter mode: 0 = wrapping mode, default 1 = one-shot mode.

Interrupt Clear Register

Any write to the TIMERXINTCLR Register clears the interrupt output from the counter.

Raw Interrupt Status Register

This register is read-only. It indicates the raw interrupt status from the counter. This value is ANDed with the timer interrupt enable bit from the Timer Control Register to create the masked interrupt, that is passed to the interrupt output pin. Figure 4-19 shows the register bit assignments.

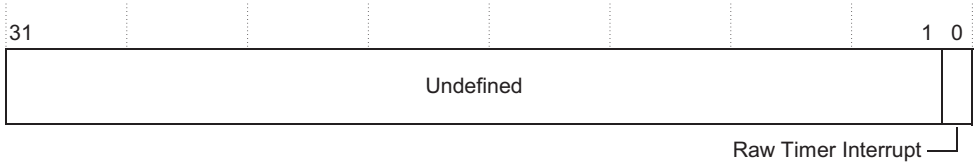


Figure 4-19 TIMERXRIS Register bit assignments

Table 4-31 lists the register bit assignments.

Table 4-31 TIMERXRIS Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined, must read as zeros
[0]	Raw Timer Interrupt	Raw interrupt status from the counter

Interrupt Status Register

The TIMERXMIS Register is read-only. It indicates the masked interrupt status from the counter. This value is the logical AND of the raw interrupt status with the timer interrupt enable bit from the Timer Control Register, and is the same value that is passed to the interrupt output pin. Figure 4-20 shows the register bit assignments.

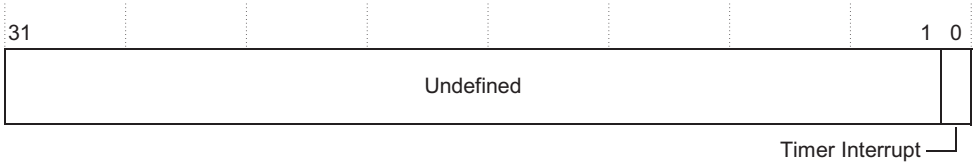


Figure 4-20 TIMERXMIS Register bit assignments

Table 4-32 lists the register bit assignments.

Table 4-32 TIMERXMIS Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined, must read as zeros
[0]	Timer Interrupt	Enabled interrupt status from the counter

Background Load Register

The TIMERXBGLOAC Register is 32-bits and contains the value from which the counter is to decrement. This is the value used to reload the counter when Periodic mode is enabled, and the current count reaches zero.

This register provides an alternative method of accessing the TIMERXLOAD Register. The difference is that writes to TIMERXBGLOAD do not cause the counter to immediately restart from the new value.

Reading from this register returns the same value returned from TIMERXLOAD. See *Load Register* on page 4-29 for more details.

Integration Test Control Register

The TIMERITCR Register is read/write. It is a single-bit register that enables integration test mode. When in this mode, the masked interrupt outputs are directly controlled by the Integration Test Output Set Register. The combined interrupt output TIMINTC then becomes the logical OR of the bits set in the Integration Test Output Set Register. Figure 4-21 shows the register bit assignments.

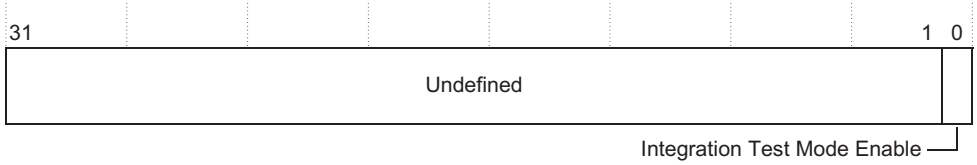


Figure 4-21 TIMERITCR Register bit assignments

Table 4-33 lists the register bit assignments.

Table 4-33 TIMERITCR Register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, read undefined, must read as zeros
[0]	Integration Test Mode Enable	When set HIGH, places the Timers into integration test mode

Integration Test Output Set Register

When in integration test mode, the enabled interrupt outputs are driven directly from the values in this write-only register, TIMERITOP. Figure 4-22 shows the register bit assignments.

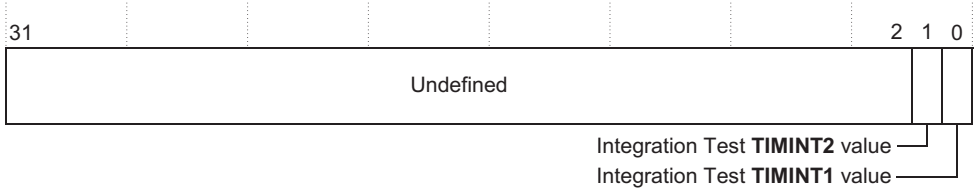


Figure 4-22 TIMERITOP Register bit assignments

Table 4-34 lists the register bit assignments.

Table 4-34 TIMERITOP Register bit assignments

Bits	Name	Function
[31:2]	-	Reserved, read undefined, must read as zeros
[1]	Integration Test TIMINT2 value	Value output on TIMINT2 when in Integration Test Mode
[0]	Integration Test TIMINT1 value	Value output on TIMINT1 when in Integration Test Mode

Peripheral Identification Registers

The TIMERPERIPHID0-3 Registers are four 8-bit registers, that span address locations `0xFE0-0xFEC`. The registers can conceptually be treated as a single 32-bit register. The read-only registers provide the following options of the peripheral:

Part number [11:0]

This identifies the peripheral. The three digit product code `0x804` is used for the timer.

Designer [19:12] This is the identification of the designer. ARM Limited is `0x41`, ASCII A.

Revision number [23:20]

This is the revision number of the peripheral. The revision number starts from 0.

Configuration [31:24]

This is the configuration option of the peripheral. The configuration value is 0.

Figure 4-23 shows the register bit assignments.

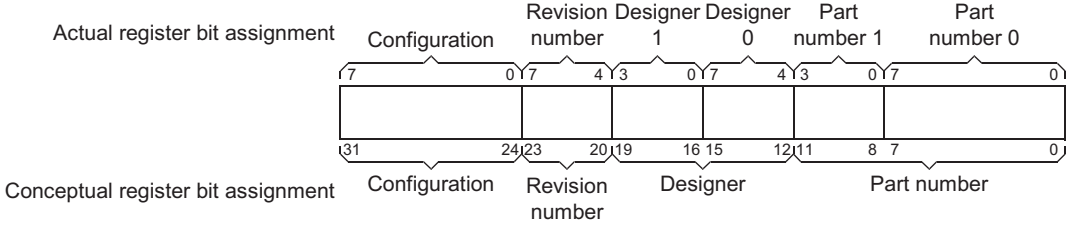


Figure 4-23 Peripheral identification register bit assignments

Note

When you design a system memory map, you must remember that the register has a 4KB-memory footprint.

The 4-bit revision number is implemented by instantiating a component called *RevisionAnd* four times with its inputs tied off as appropriate, and the output sent to the read multiplexor.

All memory accesses to the peripheral identification registers must be 32-bit, using the LDR and STR instructions.

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

- *Peripheral Identification Register 0*
- *Peripheral Identification Register 1* on page 4-36
- *Peripheral Identification Register 2* on page 4-36
- *Peripheral Identification Register 3* on page 4-36.

Peripheral Identification Register 0

The TIMERPERIPHID0 Register is hard-coded and the fields within the register determine the reset value. Table 4-35 lists the register bit assignments.

Table 4-35 TIMERPERIPHID0 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	Partnumber0	These bits read back as `0x04`

Peripheral Identification Register 1

The TIMERPERIPHID1 Register is hard-coded and the fields within the register determine the reset value. Table 4-36 lists the register bit assignments.

Table 4-36 TIMERPERIPHID1 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must read as zeros
[7:4]	Designer0	These bits read back as `0x1`
[3:0]	Partnumber1	These bits read back as `0x08`

Peripheral Identification Register 2

The TIMERPERIPHID2 Register is hard-coded and the fields within the register determine the reset value. Table 4-37 lists the register bit assignments.

Table 4-37 TIMERPERIPHID2 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must read as zeros
[7:4]	Revision	These bits read back as `0x0`
[3:0]	Designer1	These bits read back as `0x4`

Peripheral Identification Register 3

The TIMERPERIPHID3 Register is hard-coded and the fields within the register determine the reset value. Table 4-38 lists the register bit assignments.

Table 4-38 TIMERPERIPHID3 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	Configuration	These bits read back as `0x0`

PrimeCell Identification Registers

The TIMERPCELLID0-3 Registers are four 8-bit registers, that span address locations 0xFF0-0xFFC. The read-only registers can conceptually be treated as a single 32-bit register. The register is used as a standard cross-peripheral identification system. Figure 4-24 shows the register bit assignments.

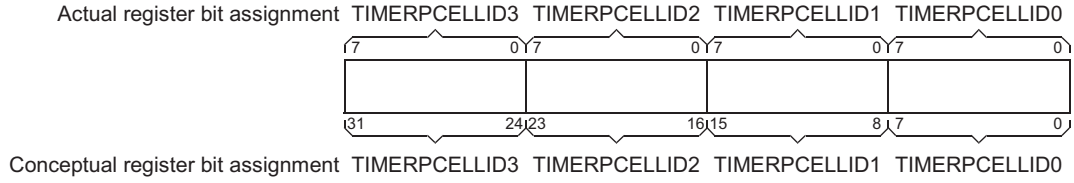


Figure 4-24 PrimeCell identification register bit assignments

The four, 8-bit registers are described in the following subsections:

- *PrimeCell Identification Register 0*
- *PrimeCell Identification Register 1* on page 4-38
- *PrimeCell Identification Register 2* on page 4-38
- *PrimeCell Identification Register 3* on page 4-38.

PrimeCell Identification Register 0

The TIMERPCELLID0 Register is hard-coded and the fields within the register determine the reset value. Table 4-39 lists the register bit assignments.

Table 4-39 TIMERPCELLID0 Register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, read undefined, must read as zeros
[7:0]	TimerPCellID0	These bits read back as 0x0D

PrimeCell Identification Register 1

The TIMERPCELLID1 Register is hard-coded and the fields within the register determine the reset value. Table 4-40 lists the register bit assignments.

Table 4-40 TIMERPCELLID1 Register bit assignments

Bits	Name	Type	Function
[31:8]	-	-	Reserved, read undefined, must read as zeros
[7:0]	TimerPCellID1	Read	These bits read back as `0xF0`

PrimeCell Identification Register 2

The TIMERPCELLID2 Register is hard-coded and the fields within the register determine the reset value. Table 4-41 lists the register bit assignments.

Table 4-41 TIMERPCELLID2 Register bit assignments

Bits	Name	Type	Function
[31:8]	-	-	Reserved, read undefined, must read as zeros
[7:0]	TimerPCellID2	Read	These bits read back as `0x05`

PrimeCell Identification Register 3

The TIMERPCELLID3 Register is hard-coded and the fields within the register determine the reset value. Table 4-42 lists the register bit assignments.

Table 4-42 TIMERPCELLID3 Register bit assignments

Bits	Name	Type	Function
[31:8]	-	-	Reserved, read undefined, must read as zeros
[7:0]	TimerPCellID3	Read	These bits read back as `0xB1`

4.5.5 Signal descriptions

Table 4-43 lists the non-AMBA signals used by the timer.

Table 4-43 Timer signals

Signal	Type	Direction	Description
TIMCLK	Timer clock	Input	The timer clock input. This must be synchronous to **PCLK** for normal operation.
TIMCLKEN1	Timer 1 clock enable	Input	The enable for the timer1 clock input. The counter will only decrement on a rising edge of **TIMCLK** when **TIMCLKEN1** is HIGH.
TIMCLKEN2	Timer 2 clock enable	Input	The enable for the timer clock input. The counter will only decrement on a rising edge of **TIMCLK** when **TIMCLKEN2** is HIGH.
TIMINT1	Counter 1 interrupt	Output	Active HIGH interrupt signal to the interrupt controller module. This signal indicates an interrupt has been generated by counter 1 having being decremented to zero.
TIMINT2	Counter 2 interrupt	Output	Active HIGH interrupt signal to the interrupt controller module. This signal indicates an interrupt has been generated by counter 2 having being decremented to zero.
TIMINTC	Combined Counter Interrupt	Output	Active HIGH interrupt signal to the interrupt controller module. This signal indicates an interrupt has been generated by either counter having being decremented to zero, and is the logical OR of TIMINT1 and TIMINT2.

Note

For a description of the AMBA signals used by the timer, see *AMBA signals* on page 1-3.

Chapter 5
Behavioral Models

This chapter describes the behavioral models in the *AMBA Design Kit* (ADK). It contains the following sections:

5.1 External RAM,

The external RAM module, `ExtRAM`, is a simple model of a 32K x 8 off-chip SRAM, that can be initialized with data from a local file. Figure 5-1 shows the external RAM module.

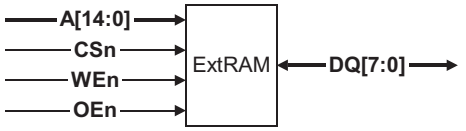


Figure 5-1 AHB external RAM module interface diagram

5.1.1 Programmer's model

Table 5-1 lists the user-defined settings for the external RAM module.

Table 5-1 User-defined settings for the external RAM module

Signal	Type	Default setting	Description
FILENAME	Input filename	-	This points to the local input data file that is read in after reset.
RAMDEPTH	Memory depth	32	This sets the memory depth in KB. If the value is increased from the default setting, then the address input bus A must also be increased to enable all memory to be addressed.

5.1.2 Signal descriptions

Table 5-2 lists the non-AMBA signals used by the external RAM module.

Table 5-2 External RAM module signals

Signal	Type	Direction	Description
A[14:0]	External address	Input	The external address input
CSN	Chip enable	Input	When LOW, this signal indicates that the chip has been selected and must respond to the current transfer

Table 5-2 External RAM module signals (continued)

Signal	Type	Direction	Description
DQ[7:0]	External data I/O	Input/ output	The external data bus, sampled during write transfers and driven during read transfers
OEN	Output enable	Input	When LOW, this signal indicates a read transfer, and enables the module to drive data onto **DQ**
WEN	Write enable	Input	When LOW, this signal indicates a write transfer

5.2 Internal memory

The internal memory, `IntMem32` and `IntMem64`, is a simple little-endian model of a variable-depth 32 or 64-bit wide on-chip SRAM that can be initialized with data from a local file.

Note

Initialization is completed only once at simulation start up, and is not carried out based on the module reset signal.

Figure 5-2 shows the internal memory module.

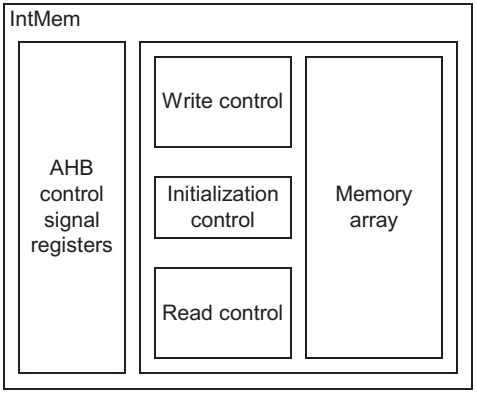


Figure 5-2 AHB internal memory module components

5.2.1 Programmer's model

Table 5-3 lists the user-defined settings for the internal RAM module

Table 5-3 User-defined settings for the internal RAM module

Signal	Type	Default setting	Description
FILENAME	Input filename	`intram.dat`	This signal points to the local input data file that is read in after reset
MEMBITS	Memory address width	12	This signal sets the number of used address bits

Memory initialization from local data file

On simulation initialization, the internal memory module loads in data from the file specified in the `FileName` setting. This file must be stored as an 8-character Verilog `$readmemh` format data file, for Verilog format models, that cannot contain more data than the model supports. Address lines, starting with `@`, and single line comments, starting with `//`, are valid, but all other non-value characters are not. Loading starts from address zero, and continues incrementing on word boundaries until an address line is found in the file. Loading then continues from that address. All values are initialized to zero before loading is started. Example 5-1 shows an example `intram.dat` file.

Example 5-1 intran.dat file

```
ea00000b
ea000005
// Data values stored at 0x00000200
@00000200
01234567
89ABCDEF
```

5.2.2 Signal descriptions

The internal memory module uses only AMBA signals. For a description of the AMBA signals, see *AMBA signals* on page 1-3.

5.3 External ROM

The external ROM module, `ExtROM`, is a simple model of a 16K x 8 off-chip EPROM, that can be initialized with data from a local file. Figure 5-3 shows the external ROM module.

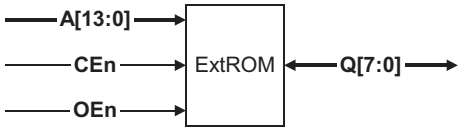


Figure 5-3 External ROM module interface diagram

5.3.1 Programmer's model

Table 5-4 lists the user-defined settings for the external ROM module.

Table 5-4 User-defined settings for the external ROM module

Signal	Type	Default setting	Description
FILENAME	Input filename	-	This signal points to the local input data file that is read in after reset.
ROMDEPTH	Memory depth	16	This signal sets the memory depth in KB. If the value is increased from the default setting, then the address input bus A must also be increased to enable all memory to be addressed.

5.3.2 Signal descriptions

Table 5-5 lists the non-AMBA signals used by the external ROM module.

Table 5-5 External ROM module signals

Signal	Type	Direction	Description
A[13:0]	External address	Input	The external address input
CEN	Chip enable	Input	When LOW, this signal indicates that the chip has been selected and must respond to the current transfer
OEN	Output enable	Input	When LOW, this signal indicates a read transfer, and enables the module to drive data onto **Q**
Q[7:0]	External data out	Output	The external data bus, driven during read transfers

5.4 Tube

The system messaging tube, `Tube`, is a simple method of passing system messages from a test program to the display, and enables a test program to stop the simulation.

Figure 5-4 shows the tube module interface.

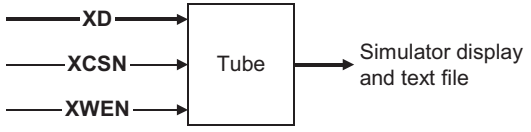


Figure 5-4 Tube module interface diagram

The main sections of this module are:

- message output to simulator
- message output to file
- simulation termination control.

5.4.1 Signal descriptions

Table 5-6 lists the non-AMBA signals used by the tube module.

Table 5-6 Tube module signals

Signal	Type	Direction	Description
XCSN[3:0]	External chip select	Input	These signals are active LOW chip enables.
XD[31:0]	External data	Input	This is the external data bus, that is sampled by this module during write transfers.
XWEN[3:0]	External write enable	Input	This is the active LOW memory write enable. For little-endian systems, **XWEN[0]** controls writes to the least significant byte and **XWEN[3]**, the most significant. The example system is configured to be little-endian.

5.5 AHB file reader master

The AHB *File Reader Master* (FRM), `FileRdMaster32` and `FileRdMaster64`, enables designers to simulate AHB systems quickly and efficiently by using it to generate explicit bus transfers. The FRM can operate in the ADK system with or without an ARM core present.

The FRM is a generic AHB *Bus Functional Model* (BFM) that directly controls bus activity by interpreting a stimulus file. The FRM facilitates the efficient validation of blocks or systems.

The 64-bit FRM, FileRdMaster64, is split into two parts:

- AHB-Lite file reader
- AHB-Lite to AHB wrapper.

The 32-bit FRM, FileRdMaster32, has an additional part, the funnel, that converts 32-bit transfers on a 64-bit bus to 32-bit transfers on a 32-bit bus.

The AHB-Lite file reader can:

- perform all AHB burst types at data widths of 8, 16, 32, and 64 bits
- insert BUSY states during bursts
- perform idle transfers
- compare received data with the expected data and report differences during simulations.

The AHB-Lite file reader is controlled entirely through the stimulus file at simulation run time. It does not have a slave interface, and therefore cannot be addressed by another AHB master.

The human-readable input stimulus file must be transformed to a data file in Verilog hexadecimal format by the preprocessor script `fm2conv.pl`.

The FRM is designed so that, wherever possible, RTL code is used to describe its logic. All RTL code is written for synthesis with pragmas where necessary to enable the block to pass through synthesis tools.

Figure 5-5 on page 5-9 shows a block diagram of the two versions of the FRM.

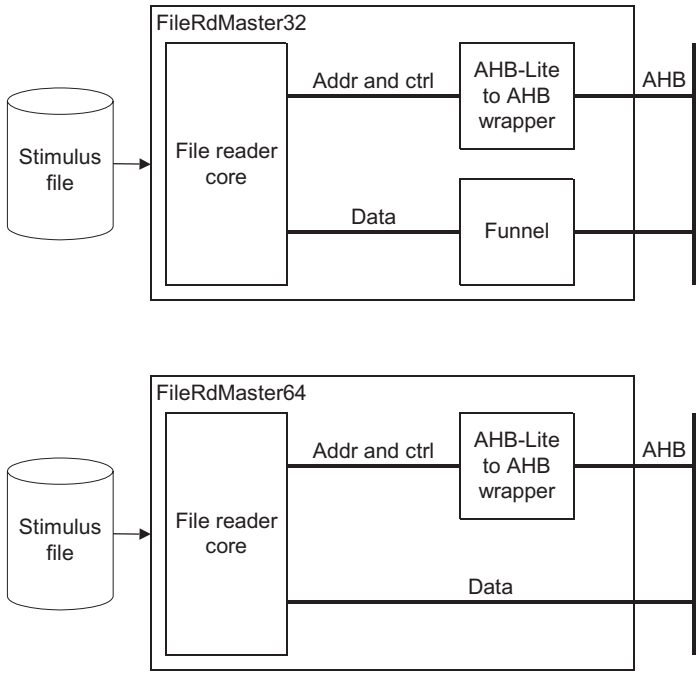


Figure 5-5 File reader bus master

5.5.1 Programmer's model

The `FileRdMaster` uses the following Verilog parameters:

`InputFileName`

This is the name of the stimulus data file to be read. If the file is not found, simulation is aborted. The default file name is `filestim.m2d`.

`MessageTag` A string that is prepended to all stimulation messages from this `FileRdMaster`. This tag can be used to differentiate between messages from multiple file reader masters in a system. The default message tag is `FileReader:`.

`StimArraySize`

The size, in words, of the internal array used to store the stimulus data. This value has a direct effect on the simulation startup time and memory requirement. This value must be large enough to store the whole data file. If the data file is larger than the array, simulation is aborted. The default value is 5000.

The following AHB-Lite FRM functions are described in this section:

- *Write command*
- *Read command* on page 5-11
- *Sequential command* on page 5-12
- *Busy command* on page 5-13
- *Idle command* on page 5-14
- *Poll command* on page 5-16
- *Loop command* on page 5-17
- *Resp field* on page 5-18
- *Clock and reset* on page 5-18
- *Error reporting at runtime* on page 5-18
- *End of stimulus* on page 5-19.

Write command

The `write` command `W` starts a write burst and can be followed by one or more S vectors. For bursts of fixed length, the Burst field determines the number of S vectors. For bursts of undefined length, there can be any number of S vectors as long as they do not cause the address to cross a 1K boundary.

Figure 5-6 shows the `write` command timing diagram.

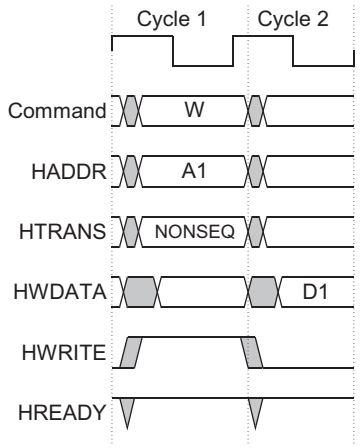


Figure 5-6 Write command timing

The `write` command operates as follows:

Cycle 1 The file reader sets up the control signals from the command and asserts **HWRITE**. **HTRANS** is NONSEQ to indicate the first transfer of the burst. The Data field is stored and ready to be driven during the data phase.

If **HREADY** is asserted, the file reader proceeds to the second control phase.

Cycle 2 This is the first data phase in which the data is driven for the previous cycle. Unless the Burst field specifies a single transfer, the file reader calculates the next address based on the Size and Burst values.

Read command

The `read` command `R` starts a read burst and can be followed by one or more S vectors. For bursts of fixed length, the Burst field determines the number of S vectors. For bursts of undefined length, there can be any number of S vectors as long as they do not cause the address to cross a 1K boundary.

Figure 5-7 shows the `read` command timing diagram.

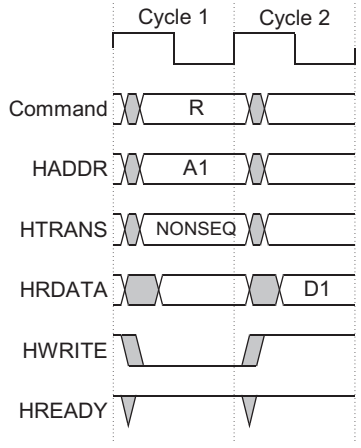


Figure 5-7 Read command timing

The `read` command operates as follows:

Cycle 1 The file reader sets up the control signals from the command and deasserts **HWRITE**. **HTRANS** is NONSEQ to indicate the first transfer of a burst.

If **HREADY** is asserted, the file reader proceeds to the second control phase.

Cycle 2 The data read for the previous cycle is compared with the Data field after applying the mask and byte lane strobes. Any differences are reported to the simulation environment. Unless the Burst field indicates a single transfer, the file reader calculates the next address based on the Size and Burst values.

Sequential command

The `sequential` command `S` is a vector that provides data for a single beat within the burst. The file reader calculates the required address. A `sequential` command is valid when a burst transfer is started by a `read` or `write` command.

Figure 5-8 shows the `sequential` command timing diagram.

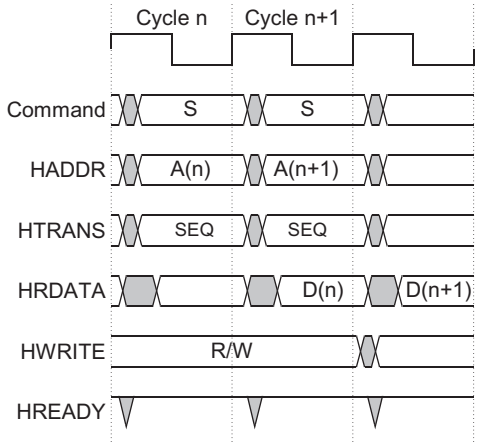


Figure 5-8 Sequential command timing

A `sequential` command is valid when a burst transfer has been started by a Read or Write command:

Cycle n The file reader drives the calculated address, and **HTRANS** is SEQ to indicate the remaining transfers of the burst.

If **HREADY** is asserted, the file reader proceeds to the second control phase.

Cycle n + 1 In a write burst, the file reader drives the Data field data and ignores the Mask field.

In a read burst, the file reader applies the Mask and Bstrb fields to the input data and then compares the Data field with the input data. The file reader reports differences between the expected data and the read data to the simulation environment.

Busy command

The `busy` command `B` inserts either a BUSY transfer or a BUSY cycle, depending on the Wait field. A `busy` command is valid when a burst transfer is started by a `read` or `write` command.

During a burst with the Wait field not specified, the `busy` command inserts a single **HCLK** BUSY transfer on the AHB. An INCR burst can have a `busy` command after its last transfer while the master determines whether it has another transfer to complete.

Figure 5-9 shows the busy command timing diagram.

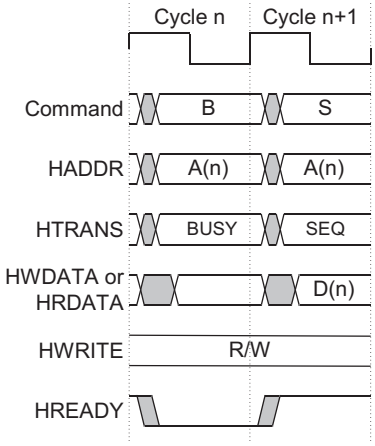


Figure 5-9 Busy transfer timing

Cycle n The file reader drives the calculated address and **HTRANS** is BUSY.

Cycle n + 1 The file reader proceeds to the next control phase. Data is ignored.

During a burst with the Wait field specified, the `busy` command inserts a complete AHB transfer as Figure 5-10 on page 5-14 shows.

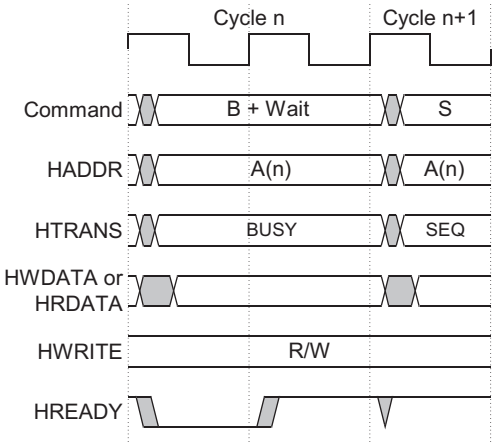


Figure 5-10 Busy cycle timing

The address phase is extended by wait states because of the data phase of a previous transfer, if present.

Idle command

The `idle` command `I` performs either an IDLE transfer or an IDLE cycle, depending on the Wait field. The options enable you to set up the control information during the IDLE transfer, and to specify if the transfer is locked or unlocked.

If the Wait field is not specified, the `idle` command inserts a single **HCLK** cycle IDLE transfer on the AHB, as Figure 5-11 on page 5-15 shows.

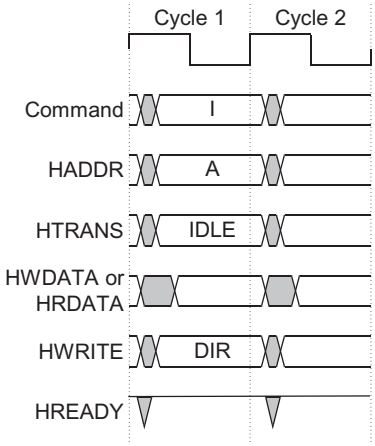


Figure 5-11 Idle transfer timing

Cycle 1 **HTRANS** is IDLE and the control signals take the default values, except for those specified in the command.

Cycle 2 The file reader proceeds to the next control phase. Data is ignored.

If the Wait field is specified, the `idle` command inserts a complete AHB transfer as Figure 5-12 on page 5-16 shows. The address phase is extended by wait states due to the data phase of a previous transfer, if present.

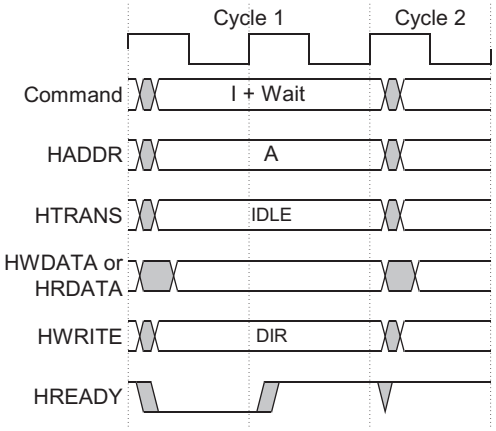


Figure 5-12 Idle cycle timing

Cycle 1 **HTRANS** is set to IDLE and the control signals are set to the default values, except for those specified in the command.

Cycle 2 If the Wait field is not specified, or the Wait field is specified and **HREADY** is asserted, then the file reader proceeds to the next control phase. Data is ignored.

Poll command

The `poll` command `P` continually reads the input data until it matches the value in the Data field or until the number of reads equals the number in the Timeout field. If the input data does not match after the Timeout number, the file reader reports an error. Not specifying a TimeOut value or specifying a Timeout value of 0 causes the `poll` command to read continually until the data matches the required value. The `poll` command is only for INCR or SINGLE burst types and for aligned addresses.

Figure 5-13 on page 5-17 shows the `poll` command.

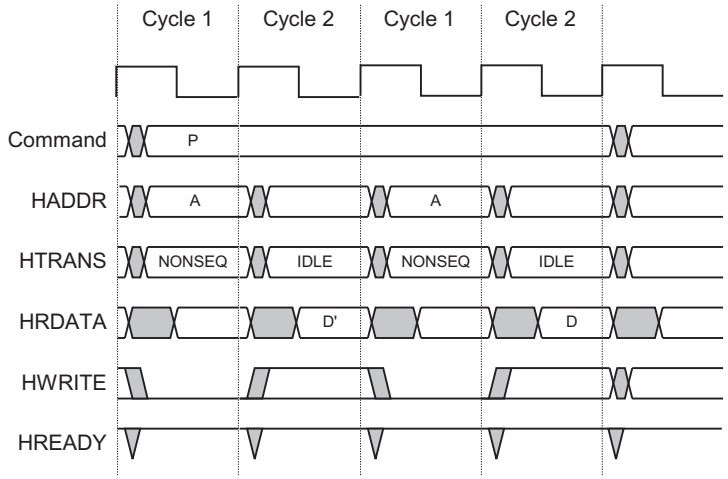


Figure 5-13 Poll command timing

The poll command operates as follows:

Cycle 1 The file reader sets up a read of the single address in the Address field.

If **HREADY** is asserted, the file reader proceeds to the second control phase.

Cycle 2 The file reader issues an IDLE transfer, reads the data for the previous address value.

Loop command

The `loop` command `L` repeats the last command a number of times. When the burst type is INCR or SINGLE, a `loop` command must follow only a `write` or `read`. Because the file reader has a 32-bit counter, the maximum number of loops is 2^{32} -1.

Note

Commands that do not directly represent bus transactions, for example, the simulation comment command `C`, are not looped. Consecutive loops are cumulative and not multiplicative. For example:

```
I 0x4000
C Commencing IDLES
L 1000
L 1000
```

This sequence performs an IDLE to address `0x4000`, generates the comment, and then performs 2000 further IDLE access to address `0x4000`.

Comment command

The `comment` command `C` sends a message to the simulation window.

Quit command

The `quit` command `Q` causes the simulation to terminate immediately. Additionally, the `quit` command gives a summary of the number of commands and errors.

Resp field

The `Resp` field tests for the expected response. The `Resp` field must be present on a command that is expected to receive an Error response from a slave.

If the `Resp` field is set to `Errorcont` or `Errorcanc` and an ERROR response is received, no warning is given. If the `Resp` field is set to `Errorcont` or `Errorcanc` and an ERROR response is not received, a simulation warning is generated.

If an error occurs during a burst transfer and the `Resp` field is set to `Errorcont`, the burst continues.

If an error occurs during a burst transfer and the `Resp` field is set to `Errorcanc`, the burst is cancelled. No attempt is made to retransmit the erroneous transfer. The stimulus file does not have to contain the remaining transfers in the burst. An IDLE transfer is always inserted during the ultimate cycle of the error response if the burst is to be cancelled.

Clock and reset

The file reader is synchronous with the AHB bus clock signal **HCLK** and is reset by the AHB reset signal **HRESETn**.

Error reporting at runtime

An error can occur in the following circumstances:

- a read transfer where the expected data does not match the actual data
- a transfer that receives an AHB ERROR response and the Error field is not set
- a transfer where the Error field is set and the transfer does not receive an AHB ERROR response
- a `Poll` command where the expected data is not received within the timeout number of attempts

- the stimulus file is longer than the array size allocated.

When an error is reported, the line number of the corresponding command on the input file is reported to the simulation window.

End of stimulus

A summary of the number of commands and errors is given when any of the following is reached:

- a quit command
- end of input file
- end of the internal command array.

Simulation is terminated when a Q command is encountered.

If the end of the stimulus is reached, all AHB signals are set to zero. This implies IDLE read transfers to address `0x00`.

If the end of the internal command array is reached and the end of the stimulus file has not been reached, a warning is given, and all AHB signals are set to zero. This implies IDLE read transfers to address `0x00`.

5.5.2 Command syntax

The filename of the stimulus data file is specified using a parameter Verilog, at the point of instantiation in the HDL code.

The syntax uses a single letter for each command followed by a number of fields.

Command syntax

Table 5-7 lists the stimulus command syntax.

Table 5-7 Stimulus command syntax

Cmd	Fields								
`W`	Address	Data		[Size]	[Burst]	[Prot]	[Lock]	[Resp]	[Comment]
`R`	Address	Data	[Mask]	[Size]	[Burst]	[Prot]	[Lock]	[Resp]	[Comment]
`S`		Data	[Mask]					[Resp]	[Comment]
`B`								[Wait]	[Comment]

Table 5-7 Stimulus command syntax (continued)

Cmd	Fields								
I	[Address]	[Dir]		[Size]	[Burst]	[Prot]	[Lock]	[Wait]	[Comment]
P	Address	Data	[Mask]	[Size]	[Burst]	[Prot]	[Timeout]		[Comment]
L	Number								[Comment]
C	Message								[Comment]
Q									[Comment]

Note

Items in brackets [] are optional. See Table 5-8 on page 5-21 for default values.

The commands are:

W The write command starts a write burst and can be followed by one or more S vectors. The number of S vectors is set by the size and burst fields for fixed length bursts. There is no limit to the number of S vectors for undefined length bursts, as long as it does not cause the address to cross a 1k boundary.

R The read command starts a read burst and can be followed by one or more S vectors. The number of S vectors is set by the size and burst fields for fixed length bursts. There is no limit to the number of S vectors for undefined length bursts, as long as it does not cause the address to cross a 1k boundary.

S The sequential vector provides data for a single beat in the burst. The file reader calculates the address required.

B The busy command inserts either a BUSY cycle or a BUSY transfer mid burst, depending on the value of the Wait field. An INCR burst can have a busy after its last transfer, while the master determines whether it has another transfer to complete. It is not valid to have a busy command when a burst is not in progress.

I The Idle command performs either an IDLE cycle or an IDLE transfer, depending on the value of the Wait field. The options enable you to set up the control information during the idle transfer, and to specify if the transfer is locked or unlocked.

P Poll command performs a read transfer that repeats until the data matches the required value. If it repeats this Number times and the value is not read, then an error is reported. Either omitting TimeOut or setting to value zero causes the Poll to repeat continually until the data matches the required value. The poll vector can only be used for INCR or SINGLE burst types and for aligned addresses.

L Loop command repeats the last command a number of times. An L command must only follow a W or R when the burst type is INCR or SINGLE.

C The `comment` command `C` sends a message to the simulation window.

Q The `quit` command `Q` causes the simulation to terminate immediately. Additionally, the quit command gives a summary of the number of commands and errors.

Table 5-8 shows the stimulus command fields.

Table 5-8 Command fields

Field	Default	Values	Prefix	Description
Address	`0x00000000`	32-bit hex value	`0x` (optional)	First address of burst.
Data	-	8, 16, 32, or 64-bit hex value	`0x` (optional)	Data field for `read`, `write`, `sequential`, and `poll` commands. The width of the Data field must match either the specified transfer size or the bus width of the FRM.
Mask	`0xFF` for each active byte lane as determined by Address and Size, and `0x00` for inactive byte lanes, or `0xFFFFFFFF` if `adk1` switch is set	8, 16, 32, or 64-bit hex value	`0x` (optional)	Bit mask. Enables masking of read data when testing against required data. You must write the Mask and Data fields as the same size. They must match either the specified transfer size or the bus width of the FRM.
Size	`word` or `doubleword` depending on user-defined `-buswidth` switch	`b | byte | size8 | h | hword | size16 | w | word | size32 | d | dword | size64`	-	Data size for `read`, `write`, `sequential`, and `poll` commands.

Table 5-8 Command fields (continued)

Field	Default	Values	Prefix	Description
Burst	`incr`	`sing` | `single` | `incr` | `incr4` | `wrap4` | `incr8` | `wrap8` | `incr16` | `wrap16`	-	Burst type for `read`, `write`, and `idle` transfer commands. For `poll` commands, the only permitted values for Burst are `sing`, `single`, or `incr`.
Prot	0000	4-bit binary	`p` | `P`	Indicates the **HPROT** value for the transfer.
Lock	`nolock`	`nolock` | `lock`	-	Transfers lock.
Resp	`okay`	`okay` | `ok` | `errcanc` | `errcanc`	-	When `errcont` is specified:[a] • If an ERROR response occurs, no warning is generated. A burst in progress continues. • If no ERROR response occurs, a warning is generated. A burst in progress continues. When `errcanc` is specified: • If an ERROR response occurs, no warning is generated. A burst in progress is cancelled.[b] • If no ERROR response occurs, a warning is generated. A burst in progress continues.
Dir	`read`	`read` | `write`	-	Controls the value of **HWRITE** during an `idle` command.
Number	-	Decimal value from 1-(2^32-1)	-	Loops repeat value.
Timeout	`0`	Decimal value from 0-(2^32-1)	`t` | `T`	Number of times the `poll` command repeats data check before generating an error when data does not match expected value. Specifying 0 repeats continuously.

Table 5-8 Command fields (continued)

Field	Default	Values	Prefix	Description
Wait	`nowait`	`wait | nowait`	-	Waits for **HREADY** before continuing. Makes an IDLE or BUSY cycle.
Message	-	1 to 80 characters and symbols. See Table 5-9 for supported characters.	comment contained within double quotes	Sends a user-defined comment to the simulation window.
Comment Delimiter	-	; | # | // | --	-	All common comment delimiters are valid.

a. The value `err` or `error` can be used as a synonym for `errcont` for compatibility with legacy BFM versions but is not recommended for use in new development.
b. An IDLE transfer is always inserted in the last cycle of the ERROR response if the burst is cancelled. No attempt is made to retransmit the erroneous transfer. The stimulus file does not have to contain the remaining transfers of the burst.

Table 5-9 lists the keyboard characters that are supported by the comment command. Any other characters are replaced with a - (dash) by the script.

Table 5-9 Characters supported by comment command

Character	Symbol							
a-z (lower case)	!	$	%	^	&	*	(	)
A-Z (upper case)	_	-	+	=	{	}	[	]
0-9	:	;	@		'~	#	<	>
(white space)	,	.	?	/	|			

5.5.3 File preprocessing

The stimulus file is converted into a format that can be fed directly into the HDL code using the script `fm2conv.pl`. This script verifies that the syntax of the input file is correct. This script provides useful error messages when the syntax is not correct. Figure 5-14 on page 5-24 shows the process of stimulus file conversion.

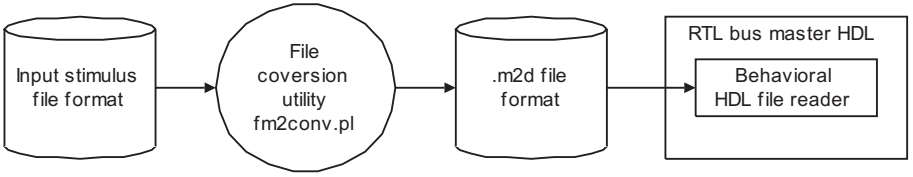


Figure 5-14 Stimulus file conversion

Loops

The `fm2conv.pl` script unfolds loops of S vectors but relies on the FRM functionality for other commands.

Note

Large loops of S vectors can create large stimulus data files.

Data and mask representations

Data and mask values can be specified as either:

- the bus width
- with the `-buswidth` switch to `fm2conv.pl`
- the same length as the transfer size with or without a `0x` prefix.

The byte lanes are driven according to both the least significant address bits, and the specified endian organization. The default is little-endian.

If the data or mask is represented as fewer bits than the data bus, then the transfer size is implicitly set to be that width. If this value conflicts with an explicit Size field, then an error is generated. The following examples show data and mask representations of with fewer bits than the data bus:

`R 00000002 DD`

Read transfer with burst type INCR and implied size Byte. The Data mask is `0x0000000000FF0000` in default little-endian mode.

`R 0000ABCD 0x0123456789ABCDEF AB`

The Data field is 64 bits (bus width), and the Mask field is BYTE, so the transfer size is BYTE.

`R 00000004 EEEEEEEE FF`

Invalid stimulus on a 64-bit system. The Data field implies a word transfer while the Mask field implies BYTE.

`W 000000C0 AB WORD`

Invalid stimulus. The Data field implies a byte transfer while the Size field specifies word.

FRM versions

Because the FRM and `fm2conv.pl` utilities are closely coupled through the stimulus data file, you must take care to ensure the correct versions are used. Table 5-10 lists the compatibility between versions.

Table 5-10 Compatibility between versions of FRM and fm2conv.pl

fm2conv.pl version	
ADK1v1	**ADK2**
File reader version ADK1v1	-
-	File reader version ADK2

Because of enhancements in FRM functionality and stimulus extensions, the stimulus files and data files for the AHB file reader are incompatible with previous versions of the file reader. The versions can be identified by their ADK version keyword:

- ADK_REL1v for previous versions
- ADK2v for the FRM described in this document.

Table 5-11 lists the compatibility between versions.

The file preprocessor can translate ADK1v1 stimulus files using the corresponding command-line switch.

Table 5-11 Compatibility between versions of stimulus file and fm2conv.pl

fm2conv.pl version	
ADK1v1	ADK2
Stimulus file version ADK1v1	Stimulus file version ADK1v1[a]
-	Stimulus file version ADK2

a. Using `-adk1` command-line switch.

Endianness

The preprocessor script by default assumes little-endian data organization. Therefore, if only a single byte of data is specified for a byte access, it is placed on the byte lane determined by little-endian addressing.

Big-endian mode is supported for AMBA (Rev 2). The type of big-endian is legacy big-endian, also called ARM big-endian or BE-32. The data and mask must be specified in the same way as for little-endian mode. The preprocessor script places the data and mask bytes in the correct lanes.

Stimulus file size

When the file reader simulation begins, the entire stimulus file is read into an array. Ensure that the array size is large enough to store the entire stimulus file. The `fm2conv.pl` utility reports the array size required and the total number of vectors in a summary of the stimulus file conversion. A warning is generated if the array size is too small for the resulting stimulus file.

If the array size in the RTL file reader bus master is changed from the default value, you can set the array size through a generic parameter in the FRM HDL by using the command line switch `-stimarraysize` with the `fm2conv.pl` utility.

File preprocessor usage

Table 5-12 lists the command-line switches accepted by the preprocessor, `fm2conv.pl`.

Table 5-12 Preprocessor command-line options

Switch	Options	Default	Description
`-help`	-	-	Displays the usage message.
`-quiet`	-	-	Suppresses warning messages.
`-adk1`	-	-	Translates an ADK1v1 stimulus file. This option can also be specified within the stimulus file.
`-endian = <endianness>`	`little` or `big`	`little`	The endianness determines the byte lanes that are driven for sparsely declared Data and Mask fields. Not supported for v6 stimulus; instead, the full bus width must be specified for big-endian transfers. The big-endian option is implemented as ARM big-endian.
`-infile = <filename>`	-	`filestim.m2i`	Input file name.

Table 5-12 Preprocessor command-line options (continued)

Switch	Options	Default	Description
-outfile = <filename>	-	filestim.m2d	Output file name. This name must match the definition specified in the file reader bus master HDL.
-buswidth = <width>	32 or 64	64	Specifies the data bus width of the target FRM.
-arch = <arch>	ahb2 or V6	ahb2	Specifies the ARM processor architecture version of the target FRM. **Note** V6 is not supported by this version of the FRM.
-StimArraySize = <size>	<size>	5000	The size of the file reader bus master file array. This size must match the value set in the FRM HDL.

Error reporting during file preprocessing

The script performs additional checks to ensure correct FRM operation. Table 5-13 lists the error checks. File conversion is aborted if an error with the command-line options is found. File conversion continues if any other error is found, so that you can generate non-AMBA compliant stimulus for test purposes, if required.

Table 5-13 fm2conv.pl error messages

Error number	Description
17	Input file is unreadable, does not exist or has incorrect permissions.
20	Input file has same name as output file.
21	Cannot create output file.
32	Unrecognized commands within the file.
36	Required fields are missing or in the wrong format.
37	Loop command has Number field missing.
38	Comment command requires a string within double quotes.
40	Size value exceeds the data bus width. Maximum value is dword | size64 for the ADK2 64-bit version FRM, and word | size32 for the ADK2 32-bit version FRM.
43	Loop Number field out of range.

Table 5-13 fm2conv.pl error messages (continued)

Error number	Description
44	Poll `TimeOut` field out of range.
48	`Data` field length exceeds FRM data bus width.
49	`Data` field has invalid length.
52	`Mask` field length exceeds FRM data bus width.
53	`Mask` field has invalid length.
56	Mismatch between transfer size, whether specified or implicitly set by Data or Mask width, and Data or Mask field.
64	Address is not aligned with the size of the transfer.
80	For Poll commands, burst types are not the valid incr or single.
84	S or B vectors before a defined-length or undefined-length burst has started.
88	Burst exceeds 1kB address boundary, for both defined and undefined-length bursts.
89	Loop number exceeds number of remaining transfers, for each defined-length burst type.

Most common AMBA protocol violations are detected by the file preprocessor script, but the absence of errors and warnings does not guarantee that the stimulus are compliant with the AMBA protocol.

Table 5-14 lists the warnings. File conversion continues when a warning is issued.

Table 5-14 fm2conv.pl warnings

Warning number	Description
128	Perl version older than 5.005. Command line switches not supported
132	Invalid data bus width selected.
133	Invalid architecture selected.
134	Adk1 architecture selected and data bus width not specified as 32-bits.
136	Output file length exceeds specified size of stimarraysize.
144	EOF found during burst: expected further transfers.

Table 5-14 fm2conv.pl warnings (continued)

Warning number	Description
164	An optional field has an invalid value.
165	Invalid character in comment string.
168	Comment command has a string of length greater than 80 characters.
169	Consecutive blank or commented lines exceeds 63 for line number reporting to work.
216	Number of S vectors following a `W | R` command is incorrect for a fixed length burst (a burst is terminated early). This enables the simulation of early-terminated bursts.
240	Unsupported command is encountered, `Memory`.
241	Unsupported field is encountered, `AltMaster`, and entire line is ignored.
242	Unsupported field is encountered, `DeGrant`, and is ignored.
248	A feature in development status.
254	Currently unsupported value in field, for example, size > 64.
255	Internal or debug error. Not expected to occur in normal usage.

Errors and warnings have the following numbering scheme:

[7] Severity.

[6:4] Error or warning type.

[3:2] Error or warning subtype.

[1:0] Enumerator.

Table 5-15 to Table 5-17 on page 5-31 list the numbering scheme for errors and warnings.

Table 5-15 Numbering scheme for bit 7, severity

Value	Meaning
0	Error
1	Warning

Table 5-16 Numbering scheme for bits [6:4] and [3:2], error and warning type and subtype

Value bits [6:4]	Meaning	Value bits [3:2]	Meaning
000	Command line	00	Environment
		01	Options
		10	-
		11	-
001	File input/output	00	Input file
		01	Output file
		10	-
		11	-
010	Syntax	00	Command
		01	Field
		10	Range
		11	-
011	Transfer size	00	Data
		01	Mask
		10	Mismatch
		11	-
100	Alignment	00	Address
		01	-
		10	-
		11	-

Table 5-16 Numbering scheme for bits [6:4] and [3:2], error and warning type and subtype (continued)

Value bits [6:4]	Meaning	Value bits [3:2]	Meaning
101	Burst	00	Within burst
		01	Outside burst
		10	Length
		11	-
110	Reserved	-	-
111	Reserved	-	-

Table 5-17 Numbering scheme for bits [1:0], enumerator

Value	Meaning
(any)	Creates unique identifier in conjunction with bits [7:2]

5.6 Test interface driver

The test interface driver, `Ticbox`, is an external module that drives the test interface lines to gain access to the AHB bus, and then applies test vectors from a test input file. This test input file is the output from a C program written with the TICTalk command language.

Before reading this section, you must be familiar with AMBA and its test interface protocol. If not, refer to the *AMBA Specification* for further information.

Figure 5-15 shows an interface diagram of the Ticbox module.

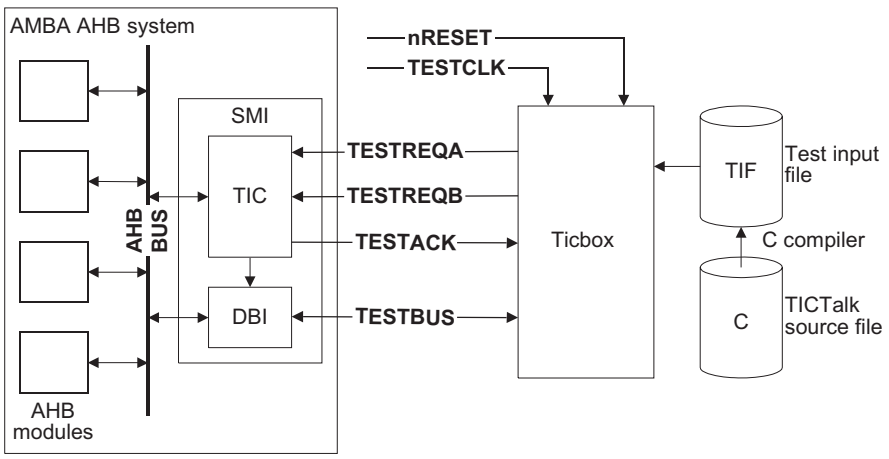


Figure 5-15 Ticbox module interface diagram

The main sections of this module are:

- the input file reader
- output vector generation
- read data expected value checking.

When the external system reset input has been deasserted, the Ticbox requests access to the system. This is done by asserting **TESTREQA** HIGH and **TESTREQB** LOW. The TIC then indicates when test mode has been entered by asserting **TESTACK** HIGH. When in test mode, the test input file is then read and translated by the Ticbox into AMBA test interface transactions, using the **TESTREQA** and **TESTREQB** signals.

The Ticbox applies test vectors to the system every time the **TESTACK** line indicates the system is ready. On read cycles the value is masked and then compared with the masked expected value given in the test vector file. An error message is given if the

comparison fails. System testing ends when the end of the input vector file is reached, and the Ticbox indicates this by asserting both **TESTREQA** and **TESTREQB** LOW to end the simulation.

5.6.1 Signal descriptions

Table 5-18 lists the non-AMBA signals used by the Ticbox module.

Table 5-18 Ticbox module signals

Signal	Type	Direction	Description
nRESET	External reset	Input	Active LOW external reset input. Used to control the operation of the Ticbox module.
TESTACK	Test acknowledge	Input	Indicates that the test bus has been granted and also that a test access has been completed.
TESTBUS[31:0]	Test data bus	Input/output	32-bit bidirectional test port.
TESTCLK	Test mode clock	Input	This is the system clock **HCLK** in test mode. All the test interface transactions are timed using this signal.
TESTREQA	Test request A	Output	Indicates test vector mode. Refer to the test interface chapter in the *AMBA Specification* for further information about the test protocol. It is driven early in the LOW phase of **TESTCLK** and held to the falling edge of **TESTCLK**.
TESTREQB	Test request B	Output	Indicates test vector mode. See the test interface chapter of the *AMBA Specification* for further information about the test protocol. It is driven early in the LOW phase of **TESTCLK** and held to the falling edge of **TESTCLK**.

5.6.2 User-defined settings

Table 5-19 lists the user-defined settings for the Ticbox module

Table 5-19 User-defined settings for the Ticbox module

Name	Type	Default setting	Description
FILENAME	Input filename	`infile.sim` (Verilog)	This points to the local input vector file that is read in a line at a time as each vector is performed.
HALTONMISMATCH	Read error setting	FALSE	This controls the operation of the module when a read error is detected. When set to FALSE, a warning message is displayed showing the read error, and if set to TRUE, the simulation is halted when a read error is detected.
VERBOSITY	Comment display	TRUE	Controls the displaying of input vector file comments. When set to TRUE, comments are displayed, and when set to FALSE, comments are not displayed. This does not affect the displaying of other system messages.

Chapter 6
PrimeCell GPIO

This chapter describes how the PrimeCell *General Purpose Input/Output* (GPIO) is integrated within the ADK. It contains the following sections:

6.1 Operation

The PrimeCell GPIO is an AMBA-compliant *System-on-Chip* (SoC) peripheral. It is an AMBA slave module that connects to the *Advanced Peripheral Bus* (APB) compliant with *AMBA Specification (Rev 2.0)* onwards.

The GPIO offers:

- compliance to the *AMBA Specification (Rev 2.0)* onwards for easy integration into SoC implementation
- eight individually programmable input/output pins
- scalability by multiple instantiations to 16, 24, 32, 40, or more bits
- programmable interrupt generation capability, from a transition or a level condition, on any number of pins
- hardware control capability of PrimeCell GPIO lines for different system configurations
- bit masking in both read and write operations through address lines
- identification registers that uniquely identify the PrimeCell GPIO.

The direction registers for port A and port B are programmable. Additional test registers and modes are implemented for functional verification and manufacturing test.

All block registers are cleared during power-on reset, **PRESETn** LOW. This enables the input drivers for both ports A and B that default to inputs on reset.

For each port, there is a data register and a data direction register. On reads, the data register contains the current status of corresponding port pins, and whether they are configured as input or output. Writing to a data register only affects the pins that are configured as outputs.

Additional test registers and modes are implemented for functional verification and manufacturing test.

For full details of the features provided by the PrimeCell GPIO, see the *General Purpose Input/Output (PL061) Technical Reference Manual*.

6.2 Integration within ADK

For details on how the GPIO integrates with the ADK, see the *AMBA Design Kit User Guide*.

Appendix A
AHB-Lite Overview

This appendix provides an overview of the AHB-Lite. It contains the following sections:

A.1 About AHB-Lite

AHB-Lite is a subset of the full AHB specification and is intended for use in designs where only a single bus master is used. This can either be a simple single-master system, as shown in Figure A-1, or a multi-layer AHB system where there is only one AHB master per layer.

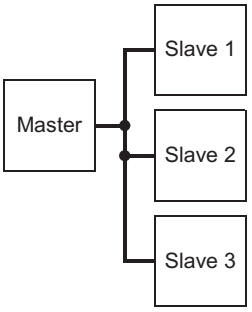


Figure A-1 AHB-Lite single-master system

AHB-Lite simplifies the AHB specification by removing the protocol required for multiple bus masters, and includes:

- request and grant protocol to the arbiter
- SPLIT and RETRY responses from slaves.

Masters designed to the AHB-Lite interface specification can be significantly simpler in terms of interface design, compared to a full AHB master. AHB-Lite enables faster design and verification of these masters and the addition of a standard off-the-shelf bus mastering wrapper can be used to convert an AHB-Lite master for use in a full AHB system.

Any master that is already designed to the full AHB specification can be used in an AHB-Lite system with no modification.

The majority of AHB slaves can be used interchangeably in either an AHB or AHB-Lite system. This is because AHB slaves that do not use either the SPLIT or RETRY response are automatically compatible with both the full AHB and the AHB-Lite specification. It is only existing AHB slaves that do use SPLIT and RETRY responses that require an additional standard off-the-shelf wrapper to be used in an AHB-Lite system.

Any slave designed for use in an AHB-Lite system works in both a full AHB and an AHB-Lite design.

A.1.1 Specification

The AHB-Lite specification differs from the full AHB specification in the following ways:

- It is a single-master system. There is only one source of address, control, and write data, so no master-to-slave multiplexor is required.
- There is no arbitration. The AHB-Lite master always has control of the bus.
- There is no master **HBUSREQ** output. If such an output exists on a master, it is left unconnected.
- There is no master **HGRANT** input. If such an input exists on a master, it is tied HIGH.
- There is no SPLIT or RETRY slave responses. The AHB-Lite master deals only with a slave ERROR response.
- The AHB-Lite lock signal **HMASTLOCK**, compared with **HLOCK** for full AHB, and it has the same timing as the address bus and other control signals. If a master has an **HLOCK** output, it can be retimed to generate **HMASTLOCK**.
- The AHB-Lite lock signal, **HMASTLOCK**, must remain stable throughout a burst of transfers, in the same way that other control signals must remain constant throughout a burst.

Using the AHB-Lite interface makes the bus transfers generated by the AHB-Lite file reader easier to understand and easier to debug. Because the AHB-Lite is a single master protocol, an AHB-Lite master always has control of the bus. Unlike AHB, AHB-Lite has no request phase. Consequently, the AHB-Lite bus might be subject to wait states during the request phase of the AHB bus.

A.1.2 Compatibility

Table A-1 on page A-4 lists how masters and slaves designed for use in either full AHB or AHB-Lite can be used interchangeably in different systems.

Table A-1 AHB-Lite interchangeability

Component	Full AHB system	AHB-Lite system
Full AHB master	Yes	Requires **HLOCK** retiming to create **HMASTLOCK**
AHB-Lite master	Use ADK Lite2AHB master wrapper	Yes
AHB slave, no SPLIT and RETRY	Yes	Yes
AHB slave, with SPLIT and RETRY	Yes	Use AHB slave wrapper

A.2 AHB-Lite master

An AHB-Lite master has the same signal interface as a full AHB bus master, except that it does not support **HBUSREQx** and **HGRANTx**.

The Lock functionality is still required because the master might be performing a transfer to a multi-port slave. The slave must be given an indication that no other transfer should occur to the slave when the master requires locked access.

An AHB-Lite master is not required to support either the SPLIT or RETRY response and only the OKAY and ERROR responses are required, so the AHB-Lite master interface does not require the **HRESP[1]** input.

A.2.1 AHB-Lite advantages

The advantage of using the AHB-Lite protocol is that the bus master does not have to support the following cases:

- Losing ownership of the bus. The clock enable for the master can simply be derived from the **HREADY** signal on the bus.
- Early terminated bursts. There is no requirement for the master to rebuild a burst due to early termination, because the master always has access to the bus.
- SPLIT or RETRY transfer responses. There is no requirement for the master to retain the address of the last transfer to be able to restart a previous transfer.

A.2.2 AHB-Lite conversion to full AHB

A standard wrapper is available to convert an AHB-Lite master to make it a full AHB master. This wrapper adds support for the features described in *AHB-Lite advantages*.

Because the AHB-Lite master has no bus request signal available, the wrapper generates this directly from the **HTRANS** signals.

A.3 AHB-Lite slaves

AHB slaves that do not use either the SPLIT or RETRY response can be used in either a full AHB or AHB-Lite system.

Any slave that does use SPLIT or RETRY responses can be used in an AHB-Lite system by adding a standard wrapper. This wrapper provides the ability to store the previous transfer in the case of a SPLIT and RETRY response and restart the transfer when appropriate. This wrapper is very similar to that required to convert an AHB-Lite master for use in a full AHB system.

For compatibility with Multi-layer AHB, it is required that all AHB-Lite slaves still retain support for early terminated bursts.

Figure A-2 shows a more detailed block diagram, including Decoder and slave-to-master multiplexor connections.

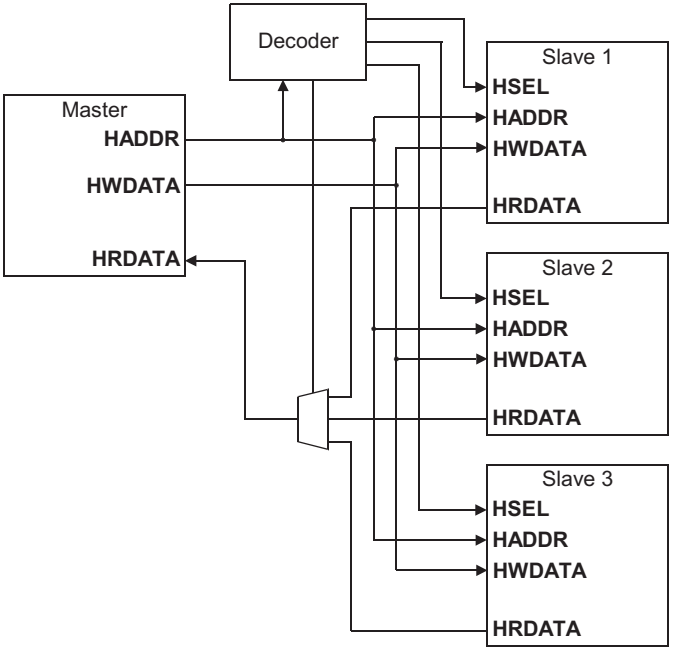


Figure A-2 AHB-Lite components

Glossary

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

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.

See also Advanced High-performance Bus and AHB-Lite.

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-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
: Refers to data items stored so that their address is divisible by the highest power of two that divides their size. Aligned words and halfwords therefore have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore refer to addresses that are divisible by four and two respectively. The terms byte-aligned and doubleword-aligned are defined similarly.

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.

ARM state
: A processor that is executing ARM (32-bit) instructions is operating in ARM state.

  See also Thumb state.

Banked registers
: The physical registers whose use is defined by the current processor mode. The banked registers are r8 to r14.

Big-endian
: Memory organization in which the least significant byte of a word is at a higher address than the most significant byte.

  See also Little-endian and Endianness.

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 lane strobe An AHB signal, **HBSTRB**, that is used for unaligned or mixed-endian data accesses to determine the byte lanes that are active in a transfer. One bit of **HBSTRB** corresponds to eight bits of the data bus.

Cache A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions and/or data. This is done to increase the average speed of memory accesses and therefore to increase processor performance.

Central Processing Unit (CPU)
The part of a processor that contains the ALU, the registers, and the instruction decode logic and control circuitry. Also commonly known as the processor core.

Cold reset Also known as power-on reset. Starting the processor by turning power on. Turning power off and then back on again clears main memory and many internal settings. Some program failures can lock up the processor and require a cold reset to enable the system to be used again. In other cases, only a warm reset is required.

See also Warm reset.

Coprocessor A processor that supplements the main CPU. It carries out additional functions that the main CPU cannot perform. Usually used for floating-point math calculations, signal processing, or memory management.

Core reset *See* Warm reset.

CPU *See* Central Processing Unit.

Domain A memory division that is made up of supersections, sections, large pages, or small pages of memory, that can have their access permissions switched rapidly by writing to the Domain Access Control Register, CP15 register 3.

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

Endianness Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory.

See also Little-endian and Big-endian.

Little-endian Memory organization where the least significant byte of a word is at a lower address than the most significant byte.

See also Big-endian and Endianness.

Memory bank One of two or more parallel divisions of interleaved memory, usually one word wide, that enable reads and writes of multiple words at a time, rather than single words. All memory banks are addressed simultaneously and a bank enable or chip select signal determines the bank that is accessed for each transfer. Accesses to sequential word

addresses cause accesses to sequential banks. This enables the delays associated with accessing a bank to occur during the access to its adjacent bank, speeding up memory transfers.

Power-on reset *See* Cold reset.

Processor A contraction of microprocessor. A processor includes the CPU or core, plus additional components such as memory, and interfaces. These are combined as a single macrocell, that can be fabricated on an integrated circuit.

Region A partition of instruction or data memory space.

Register A temporary storage location used to hold binary data until it is ready to be used.

Remapping Changing the address of physical memory or devices after the application has started executing. This is typically done to enable RAM to replace ROM when the initialization has been done.

Reserved A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as zero and are read as zero.

Scan chain A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

TAP *See* Test Access Port.

Test Access Port (TAP)
The collection of four mandatory terminals and one optional terminal that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**.

Unaligned Memory accesses that are not appropriately word-aligned or halfword-aligned.

See also Aligned.

Undefined Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architectural Reference Manual* for more information on ARM exceptions.

Warm reset Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

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. Each entry in the write buffer can contain the address of a data item to be stored to main memory, the data for that item, and a sequential bit that indicates if the next store is sequential or not.

Write completion

The memory system indicates to the CPU 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.