AHB Example AMBA SYstem

Technical Reference Manual

ARM®

DDI0170A

AHB Example AMBA SYstem
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
August 1999	A	First release

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
AHB Example AMBA SYstem Technical Reference Manual

Preface

This preface introduces the AHB *Example AMBA SYstem* (EASY) and its reference documentation. It contains the following sections:

About this document

This document is a comprehensive manual for the behavioral HDL model of the *Example AMBA SYstem* (EASY). It gives detailed information about:

- the function of the whole system
- each module in the system
- how to design a new system module.

This document refers to the *Advanced High-performance Bus* (AHB). For information on the *Advanced System Bus* (ASB) refer to the *ASB Example AMBA SYstem Technical Reference Manual*.

Intended audience

This document has been written for experienced hardware and software engineers who wish to incorporate a fully functional AMBA system into their hardware and software design.

Organization

This document is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an overview of the AHB Example AMBA SYstem.

Chapter 2 ***The EASY Microcontroller***

Read this chapter for a description of the modules of the AHB EASY microcontroller.

Chapter 3 ***ARM7TDMI AHB Wrapper***

Read this chapter for a description of the ARM7TDMI AHB wrapper module.

Chapter 4 ***AHB Modules***

Read this chapter for details of the AHB modules that are used in the AHB Example AMBA SYstem.

Chapter 5 ***APB Modules***

Read this chapter for details of the APB modules that are used in the AHB Example AMBA SYstem.

Chapter 6 ***Behavioral Modules***

Read this chapter for details of how to use the behavioral modules, including memory modules and the external AMBA Test Interface Driver module (the *TICBOX*). This chapter contains a description of the *TICTalk* command language.

Chapter 7 ***Designer's Guide***

Read this chapter for details of how to add new bus master, slave and peripheral modules to the AHB EASY microcontroller.

Typographical conventions

The following typographical conventions are used in this document:

bold	Highlights ARM processor signal names within text, and interface elements such as menu names. May also be used for emphasis in descriptive lists where appropriate.
italic	Highlights special terminology, cross-references, and citations.
`typewriter`	Denotes text that may be entered at the keyboard, such as commands, file names and program names, and source code.
`type`writer	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands or functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications by ARM Limited, and by third parties that provide additional information on developing for the ARM processor, and general information on related topics.

ARM publications

AMBA Specification (Rev 2.0) (ARM IHI 0011)

ARM Architecture Reference Manual (ARM DDI 01000)

ARM7TDMI Data Sheet (ARM DDI 0029)

Example AMBA SYstem User Guide (ARM DUI 0092)

ASB Example AMBA SYstem Technical Reference Manual (ARM DDI 0138)

Micropack AHB CPU Wrappers Technical Reference Manual (ARM DDI 0169).

Other publications

IEEE 1149.1 JTAG standard.

Feedback

ARM Limited welcomes feedback both on the AHB Example AMBA SYstem, and on the documentation.

Feedback on this document

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

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

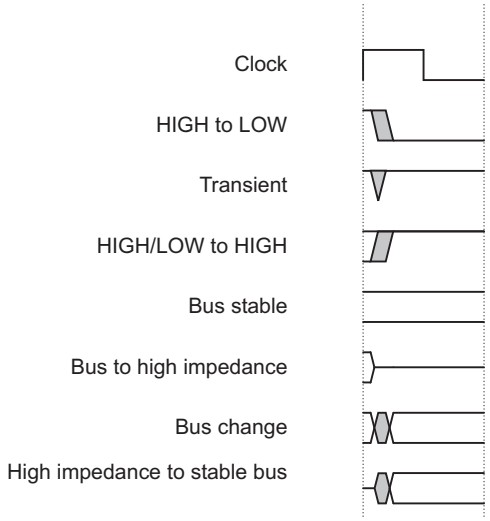
Feedback on the AHB Example AMBA SYstem

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

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

Timing diagram conventions

This manual contains one or more timing diagrams. The following key explains the components used in these diagrams. Any variations are clearly labelled when they occur. Therefore, no additional meaning should be attached unless specifically stated.



Key to timing diagram conventions

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

Chapter 1
Introduction

This chapter introduces the *AHB Example AMBA SYstem* (EASY). It contains the following section:

- *Overview of EASY* on page 1-2.

1.1 Overview of EASY

The EASY microcontroller comprises the building blocks needed to create an example system based on the low-power, generic design methodology of the *Advanced Microcontroller Bus Architecture* (AMBA).

The EASY microcontroller:

- enables custom devices to be developed in very short design cycles
- allows the resulting subcomponents to be easily reused in future designs.

Note

This document refers to the *Advanced High-performance Bus* (AHB). For information on the *Advanced System Bus* (ASB) refer to the *ASB Example AMBA SYstem Technical Reference Manual*.

1.1.1 EASY system blocks

The example design provides all the system modules needed to manage an AMBA system:

- reset controller
- arbiter
- decoder.

These system modules control various aspects of the *Advanced High Performance Bus* (AHB).

1.1.2 EASY components

The example design comprises:

- Two buses:
  - the AHB
  - the *Advanced Peripheral Bus* (APB).
- The ARM processor AHB wrapper, to allow execution of ARM code in an AHB system.
- The *Test Interface Controller* (TIC), to allow external control of the AHB during system test.
- A minimum set of basic microcontroller peripherals. These are supported, and are implemented as low-power designs on the APB. They include:
  - an interrupt controller

  - — a remap and pause controller
  - — a 16-bit timer module.
- The example *Static Memory Interface* (SMI). This demonstrates the minimum requirements for an *External Bus Interface* (EBI).
- A 1KB block of internal memory.

The EASY system consists of a microcontroller with some external memory as shown in Figure 1-1.

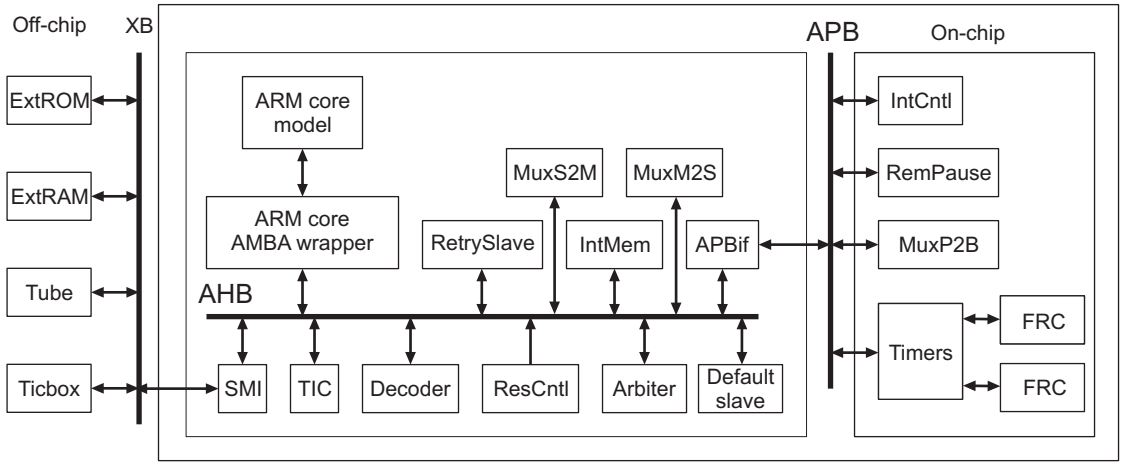


Figure 1-1 EASY system diagram

The descriptions in this manual refer to an AHB-based EASY system. For details of ASB-based EASY system design refer to the *Example AMBA SYstem Technical Reference Manual*.

Chapter 2
The EASY Microcontroller

This chapter describes the microcontroller which is the main unit of the EASY system. It contains the following sections:

2.1 Functional overview

The modules of the EASY microcontroller are grouped in five classes:

AMBA system components
: Used to control the general operation of the system.

Peripherals
: Low-power peripherals, which are connected to the peripheral bus.

Example components
: Demonstration modules that are only simulation models.

System test methodology
: Modules used for testing the system.

Processor core
: The ARM processor core that is built into the EASY microcontroller.

With the exception of the processor core the above modules are fully described in this chapter. For details of the processor core refer to Chapter 3 *ARM7TDMI AHB Wrapper*.

2.2 The AMBA system components

The *Advanced Microcontroller Bus Architecture* (AMBA) system comprises:

- *Reset controller*
- *Arbiter*
- *Decoder*
- *AHB to APB bridge* on page 2-4.

2.2.1 Reset controller

The reset controller consists of a state machine which generates the **HRESETn** signal. This signal indicates the current reset state of the AMBA bus and is used by all the other elements in the EASY microcontroller, primarily for power-on initialization.

Note

All other reset modes, such as standby or warm reset, must be implemented separately.

2.2.2 Arbiter

The arbiter provides arbitration between bus masters competing for access to the AHB. Although there are only two bus masters in the EASY microcontroller, the ARM and the TIC, the arbiter has provision for up to four masters. To extend the number of masters, refer to Chapter 7 *Designer's Guide*. The arbitration is currently assigned with a simple priority system, with the TIC as the highest priority, and the processor as the lowest (also the reset default). The arbitration scheme is not defined in the *AMBA Specification* and can be dependent on implementation.

2.2.3 Decoder

The decoder consists of a simple address decoding logic, which is used to select the system bus slaves based on the address of the current transfer. This module controls the configurable memory map for the system.

2.2.4 AHB to APB bridge

The AHB to APB bridge interface is an AHB slave. When accessed (in normal operation or system test) it initiates an access to the APB. APB accesses are of different duration (three **HCLK** cycles in the EASY for a read, and two cycles for a write). They also have their width fixed to one word, which means it is not possible to write only an 8-bit section of a 32-bit APB register. APB peripherals do not need a **PCLK** input as the APB access is timed with an enable signal generated by the AHB to APB bridge interface. This makes APB peripherals low power consumption parts, because they are only strobed when accessed.

For more information on the APB bus refer to the *AMBA Specification*.

2.3 Reference peripherals

Figure 2-1 shows how the reference peripherals are interconnected within the *Reference Peripherals Specification* (RPS) block, and how they are connected to the bridge.

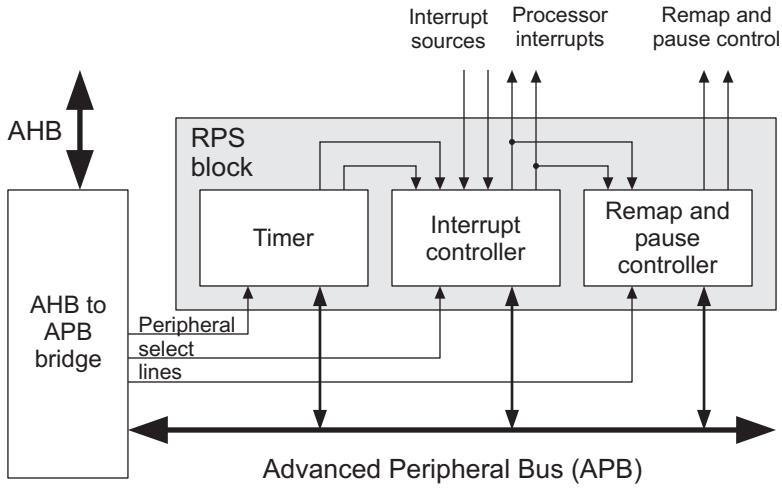


Figure 2-1 Block diagram of the RPS block and bridge

The base addresses of each of the peripherals (timer, interrupt controller, and remap and pause controller) are defined in the AHB to APB bridge interface, which selects the peripheral according to its base address. The whole APB address range is also defined in the bridge.

These base addresses can be implementation-specific. The peripherals standard specifies only the register offsets (from an unspecified base address), register bit meaning, and minimum supported function.

Table 2-1 shows the three bases and their current addresses in the EASY microcontroller.

Table 2-1 Peripherals base addresses

Peripheral	EASY base address
Interrupt controller	0x8000 0000
Timer	0x8400 0000
Remap and pause controller	0x8800 0000

Note

When writing software or test patterns to run on the system, the absolute hex addresses must not be used within the code. Instead, define the base addresses in a header and then use the offset to this base address.

The APB data bus is split into two separate directions:

- read (**PRDATA**), where data travels from the peripherals to the bridge
- write (**PWDATA**), where data travels from the bridge to the peripherals.

This simplifies driving the buses because turnaround time between the peripherals and bridge is avoided.

In the default system, because the bridge is the only master on the bus, **PWDATA** is driven continuously. **PRDATA** is a multiplexed connection of all peripheral **PRDATA** outputs on the bus, and is only driven when the slaves are selected by the bridge during APB read transfers.

It is possible to combine these two buses into a single bidirectional bus, but precautions must be taken to ensure that there is no bus clash between the bridge and the peripherals.

2.3.1 Timer

The timer comprises:

- two 16-bit periodic/free running down counters
- a clock prescaler (divide by 1, 16 or 256)
- a test veneer.

When the counters underflow (passing zero value and reloading) they can generate interrupt requests which are passed to the interrupt controller. Both counter values can be loaded, read, and controlled through addressable registers.

2.3.2 Interrupt controller

The interrupt controller contains a set of registers for controlling eight *interrupt request* (IRQ) sources and one *fast interrupt request* (FIQ) source. These have the following functions:

- enable or disable specific interrupt sources from triggering the ARM **nIRQ** or **nFIQ** interrupt lines
- read the status of all interrupt sources at the inputs of the interrupt controller
- read the status of the interrupt sources enabled to trigger the ARM interrupt lines
- generate a software-triggered **nIRQ** signal to the ARM processor
- isolate the interrupt controller for test.

The number of IRQ sources can easily be extended by increasing the number of IRQ registers.

2.3.3 Remap and pause controller

The remap and pause controller has three functions:

Reset status This enables software to determine whether the last reset was a *Power-On Reset* (POR) or a soft reset. The latter function is redundant in the EASY microcontroller, since it does not have a soft reset. It is implemented only as an example for systems that might provide a soft reset state.

Remap memory On reset the internal RAM is mapped out and bank 4 of the external memory is mapped into location `0x0000 0000` which is the boot location for the ARM processor. The reset memory map is cancelled by writing to a register in this peripheral.

Pause mode The EASY microcontroller only supports one simple power-saving mode, called Pause. This halts all bus activity (but not the system clock) and waits for an interrupt signal from the interrupt controller before restarting the system.

The remap and pause controller also contains an ID register which is currently only a single bit. This block can be extended in many ways including support for software-generated resets, more sophisticated power-saving modes and more detailed ID information.

2.4 Example components

The example components include:

- *Internal memory*
- *Static memory interface*
- *Retry slave*
- *Default slave*.

Typically these blocks must be re-implemented according to the specific system requirements of the microcontroller being developed.

2.4.1 Internal memory

The internal memory is a very basic behavioral model of 1KB of zero wait state static memory, which is not synthesizable. The size of the memory can be extended by altering a setting in the HDL file.

2.4.2 Static memory interface

The SMI is a 32-bit *External Bus Interface* (EBI) that can connect up to four 256MB banks of zero to four wait state *memory* to the EASY microcontroller. However, the number of wait states is set as a constant in the HDL (before synthesis), and is set for all four banks. The example SMI also supports test signals from the TIC. These override the normal operation of the SMI during system test, and directly control the tristate drivers on the **XD** bus.

2.4.3 Retry slave

The retry slave is an example of how to implement an AHB slave that generates retry responses and wait states for read or write accesses. It is used as a template for building slaves that require the use of a retry response.

2.4.4 Default slave

The default slave is used to fill holes in the memory map, so that the system will still function if an invalid area of memory is accessed. This must be modified to suit the memory map of the system, so that all areas of memory will access a system slave.

2.5 System test methodology

Each AHB slave, AHB master, and APB peripheral should be tested in complete isolation. This means that components must be designed with test veneers that allow non-bus signals to be controlled and observed.

When a component is tested, a special test bit is set. This test bit switches these multiplexed signals to test registers (accessible via the AHB or APB), which effectively isolates each component from the rest of the system.

Test vectors should be written to test the component in isolation, making as few assumptions about the rest of the system as possible.

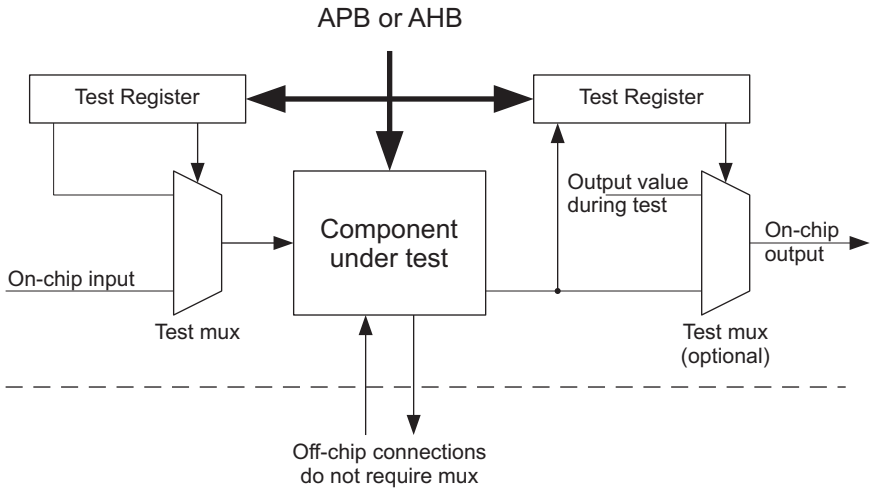


Figure 2-2 Simple test veneer example

A good example of this approach is provided by the test veneer for the ARM processor, which is described in the *AMBA ARM7TDMI Interface Data Sheet*. This approach is also used to test the peripherals on the APB bus.

Under normal conditions, when the TIC is not in use, the current bus master performs transfers to and from any one of the following slaves:

- internal memory
- AHB to APB bridge interface (to access the peripherals)
- example retry slave
- EBI.

However, when test mode is entered, and the TIC is the current master, the following slaves can be accessed:

- internal memory
- AHB to APB bridge interface (to access the peripherals)
- example retry slave
- ARM bus master (test veneer).

Note

Bus masters can become slaves during test mode. The EBI cannot be tested via the TIC due to the way test access is provided to the AHB bus. The TIC is a state machine driven by the test request inputs (**TESTREQA** and **TESTREQB**). It also contains a register that allows it to read address information from the test bus (**TESTBUS**) and drive it onto the AHB address bus (**HADDR**). However, it cannot drive the test bus. Instead, it overrides the normal function of the EBI, forcing it to provide a 32-bit channel between **HRDATA** and **TESTBUS**, passing out read data during a read test vector. Thus, in test mode, the EBI cannot function as a slave.

TESTBUS must be a 32-bit channel. In a system which only supports a 16-bit or 8-bit external data bus, additional external pins such as address lines must be forced into a special test mode in order to supply the full 32-bit bidirectional channel required.

For more information about:

- the test interface, see the *AMBA Specification*
- applying test vectors to an EASY-based microcontroller, see the *EASY User Guide*.

Note

The **TESTREQA**, **TESTREQB** and **TESTBUS** signals are the same as the **TREQA**, **TREQB** and **TBUS** signals described in the *AMBA Specification (Rev 2.0)*.

Chapter 3
ARM7TDMI AHB Wrapper

This chapter describes the ARM7TDMI processor core wrapper that can be used with an AHB-based EASY system.

Note

For details of other supported CPU wrappers refer to the *Micropack AHB CPU Wrappers Technical Reference Manual*.

This chapter contains the following sections:

3.1 About the ARM7TDMI AHB wrapper

The ARM7TDMI AHB wrapper module interfaces between the ARM7TDMI and the AHB bus, allowing the ARM7TDMI to become an AHB bus master. The module also includes a test interface, allowing the ARM7TDMI to be selected as a bus slave and tested via the TIC interface. If, however, an alternative test approach is to be used, the test logic may be removed from the AMBA interface.

The top level block diagram is shown in Figure 3-1, which shows how the wrapper interfaces to the ARM7TDMI. The AHB input signals are routed through the wrapper before becoming inputs to the ARM7TDMI. The outputs are also routed through the wrapper before being driven onto the AHB.

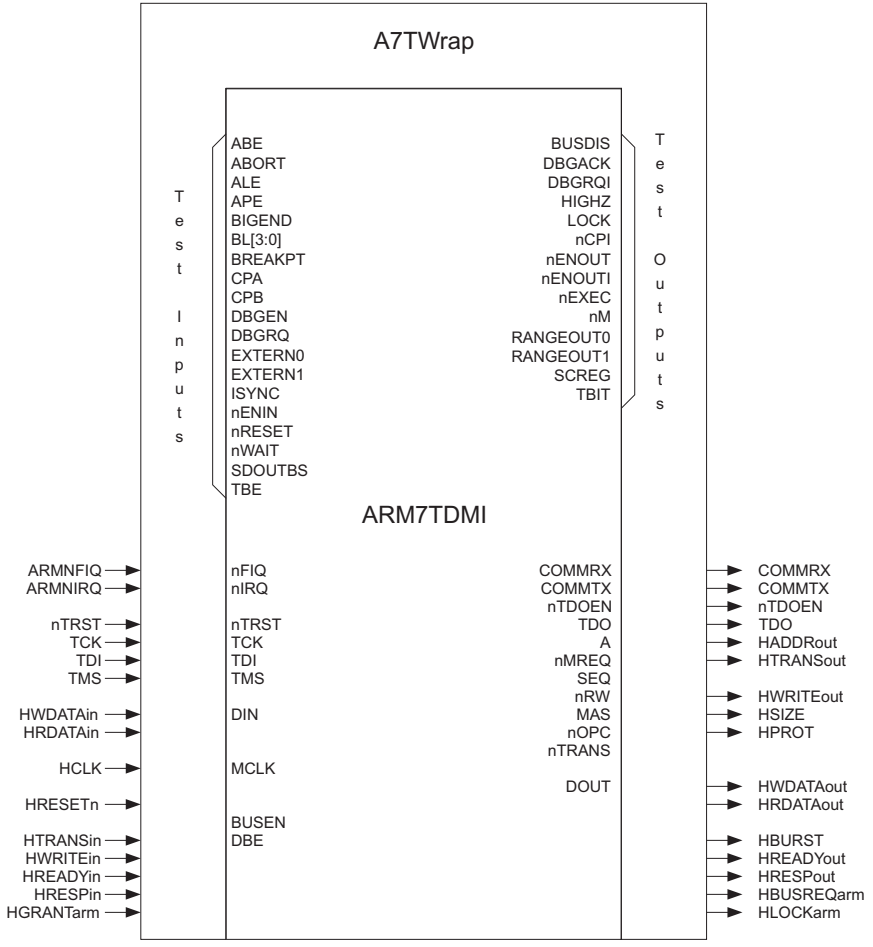


Figure 3-1 ARM7TDMI AHB AMBA wrapper block diagram

3.2 Signal interface

The ARM7TDMI AHB wrapper has a combined AHB master and AHB slave interface. The master interface is used during normal system operation. The slave interface is used during testing of the core when the *Test Interface Controller* (TIC) is acting as the current AHB bus master.

3.3 ARM7TDMI AHB signal descriptions

Table 3-1 describes the signals used by the ARM7TDMI AHB wrapper.

Table 3-1 ARM7TDMI AHB signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers. All signal timings are related to the rising edge of **HCLK**.
HRESETn	Reset	Input	The bus reset signal is active LOW and is used to reset the system and the bus. This is the only active LOW AHB signal.
HADDRout[31:0]	Address bus	Output	This is the 32-bit system address bus.
HTRANSin[1:0] **HTRANSout[1:0]**	Transfer type	Input/ output	These signals indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, or IDLE. The wrapper does not use the BUSY transfer type.
HWRITEin **HWRITEout**	Transfer direction	Input/ output	When HIGH this signal indicates a write transfer and when LOW a read transfer.
HSIZE[2:0]	Transfer size	Output	This signal indicates the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit).
HBURST[2:0]	Burst type	Output	This signal indicates if the transfer forms part of a burst. The ARM core always performs incrementing bursts of unspecified length.
HPROT[3:0]	Protection control	Output	The protection control signals indicate if the transfer is an opcode fetch or data access, and if the transfer is a supervisor mode access or user mode access.
HWDATAin[31:0] **HWDATAout[31:0]**	Write data bus	Input/ output	The write data bus is used to transfer data from the master to the bus slaves during write operations.
HSELArmTest	Slave select	Input	Each AHB slave has its own select signal and this signal indicates that the current transfer is intended for the selected slave. This signal is a combinatorial decode of the address bus.
HRDATAin[31:0] **HRDATAout[31:0]**	Read data bus	Input/ output	The read data bus is used to transfer data from bus slaves to the bus master during read operations.
HREADYin **HREADYout**	Transfer done	Input/ output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.

Table 3-1 ARM7TDMI AHB signal descriptions (continued)

Signal	Type	Direction	Description
HRESPin[1:0] **HRESPout[1:0]**	Transfer response	Input / output	The transfer response provides additional information on the status of a transfer. The wrapper only uses the OKAY response.
HBUSREQarm	Bus request	Output	A signal from the wrapper to the bus arbiter which indicates that it requires the bus. This output signal is always set HIGH as the core requires use of the bus all of the time.
HLOCKarm	Locked transfers	Output	When HIGH this signal indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is LOW.
HGRANTarm	Bus grant	Input	This signal indicates that the ARM core is currently the highest priority master. Ownership of the address/control signals changes at the end of a transfer when **HREADY** is HIGH, so a master gains access to the bus when both **HREADY** and **HGRANTx** are HIGH.
ARMNFIQ	ARM fast interrupt	Input	This is the ARM fast interrupt request, and is routed to the **nFIQ** input on the ARM CPU.
ARMNIRQ	ARM interrupt	Input	This is the ARM interrupt request, and is routed to the **nIRQ** input on the ARM CPU.
COMMRX	Comms receive	Output	When LOW, this signal denotes that the communications channel receive buffer is empty. The communications channel allows serial communication of bytes between the processor and an external device, using the JTAG port as the serial connection.
COMMTX	Comms transmit	Output	When HIGH, this signal denotes that the communications channel transmit buffer is empty.
nTRST	Not test reset	Input	Active LOW reset signal for the boundary scan logic. This pin must be pulsed or driven LOW to achieve normal device operation. This is part of the IEEE 1149.1 JTAG standard.
TCK	Test clock	Input	This is the JTAG clock. This is part of the IEEE 1149.1 JTAG standard.
TDI	Test data in	Input	This is part of the IEEE 1149.1 JTAG standard.

Table 3-1 ARM7TDMI AHB signal descriptions (continued)

Signal	Type	Direction	Description
TMS	Test mode select	Input	This is part of the IEEE 1149.1 JTAG standard.
nTDOEN	Not **TDO** enable	Output	When LOW, this signal denotes that serial data is being driven out on the **TDO** output. **nTDOEN** would normally be used as an output enable for a **TDO** pin in a packaged part.
TDO	Test data out	Output	This is part of the IEEE 1149.1 JTAG standard.

3.4 Overview of the ARM7TDMI wrapper

The ARM7TDMI AHB wrapper (A7TWrap) is made up of the following five blocks:

A7TWrapBurst Controls the AHB address and control output generation during burst transfers.

A7TWrapLock Used to generate the **HLOCK** output, when the core performs a locked transfer (SWP instruction).

A7TWrapMaster Controls the bus master interface to the AHB.

A7TWrapCtrl Contains a test multiplexor used to drive the core control inputs with test data during TIC testing of the core. During normal operation the control inputs are driven with default values. This block is removable.

A7TWrapTest Contains the main test state machine used to control the application of test vectors during core TIC testing. A test register stores the core control inputs during test, which are driven through the A7TWrapCtrl block, and the core control outputs are driven onto the AHB read data bus.

For more details of these blocks, refer to *Description of the ARM7TDMI wrapper blocks* on page 3-14.

Figure 3-2 shows the connections between the blocks that make up the wrapper module.

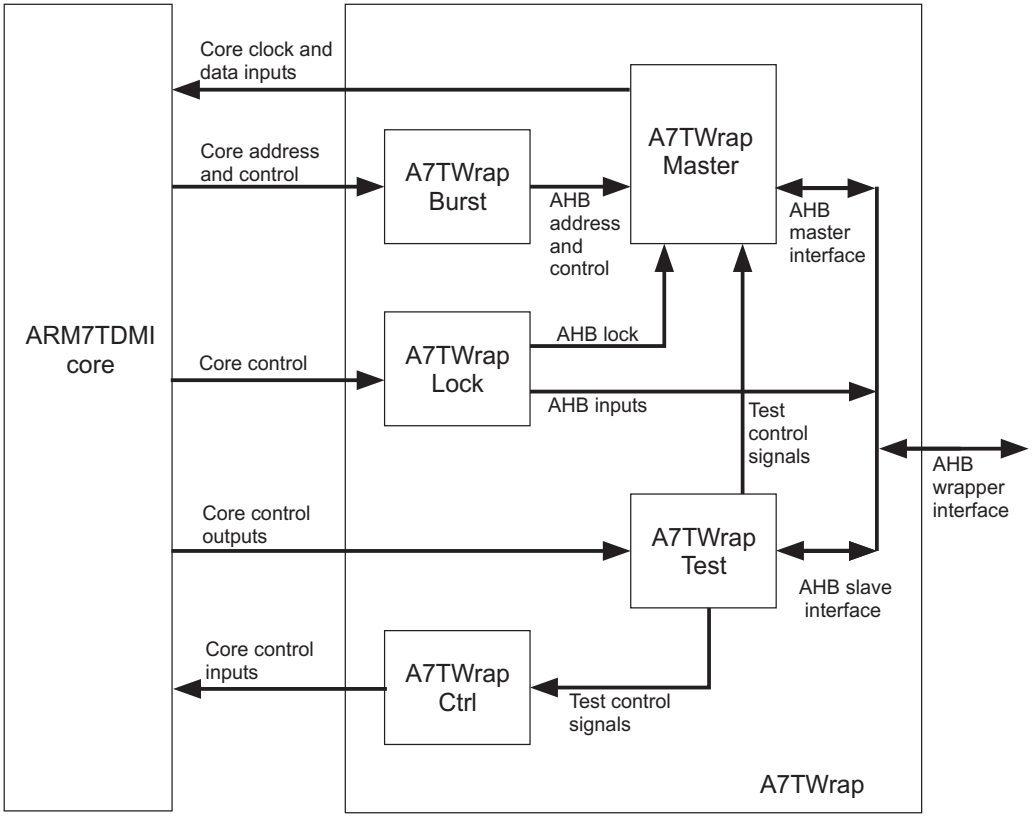


Figure 3-2 ARM7TDMI AHB wrapper block diagram

3.5 Connections to ARM7TDMI core

Table 3-2 shows the connections of the ARM7TDMI core inputs and outputs.

Table 3-2 Connections of ARM7TDMI signals

Signal	Type	Direction	Connected to
A[31:0]	Addresses	Output	**HADDRout[31:0]**
ABE	Address bus enable	Input	Tied HIGH to drive the address and control signals at all times.
ABORT	Memory abort	Input	Generated from slave response when wrapper is bus master.
ALE	Address latch enable	Input	Tied HIGH to allow pipelined addresses from the core.
APE	Address pipeline enable	Input	Tied HIGH to allow pipelined addresses from the core.
BIGEND	Big endian configuration	Input	Default configuration is tied LOW for little-endian operation.
BL[3:0]	Byte latch control	Input	Tied HIGH to latch all 32 bits of the data bus when the core is clocked.
BREAKPT	Breakpoint	Input	Tied LOW as there is no external debug logic.
BUSDIS	Bus disable	Output	Only used for test.
BUSEN	Data bus configuration	Input	Tied HIGH to use the unidirectional data buses.
COMMRX	Communications channel receive	Output	Connected to system output **COMMRX**.
COMMTX	Communications channel transmit	Output	Connected to system output **COMMTX**.
CPA	Coprocessor absent	Input	Tied HIGH as there is no external coprocessor.
CPB	Coprocessor busy	Input	Tied HIGH as there is no external coprocessor.
D[31:0]	Data bus	Input/ output	Unconnected as the unidirectional data buses are used.
DBE	Data bus enable	Input	Tied HIGH to drive the data buses at all times.
DBGACK	Debug acknowledge	Output	Only used for test.
DBGEN	Debug enable	Input	Tied HIGH to allow use of JTAG debug.

Table 3-2 Connections of ARM7TDMI signals (continued)

Signal	Type	Direction	Connected to
DBGRQ	Debug request	Input	Tied LOW as there is no external debug logic.
DBGRQI	Internal debug request	Output	Only used for test.
DIN[31:0]	Data input bus	Input	Comes from **HRDATAin** when a bus master, and from **HWDATAin** when a slave in TIC test mode.
DOUT[31:0]	Data output bus	Output	Used to drive **HWDATAout**.
DRIVEBS	Boundary scan cell enable	Output	Unconnected output.
ECAPCLK	Extest capture clock	Output	Unconnected output.
ECAPCLKBS	Extest capture clock for boundary scan	Output	Unconnected output.
ECLK	External clock input	Output	Unconnected output.
EXTERN[1:0]	External input	Input	Tied LOW as there is no external debug logic.
HIGHZ	-	Output	Only used for test.
ICAPSCLKBS	Intest capture clock	Output	Unconnected output.
IR[3:0]	TAP controller instruction register	Output	Unconnected output.
ISYNC	Synchronous interrupts	Input	Tied HIGH for synchronous interrupts.
LOCK	Locked operation	Output	Only used for test.
MAS[1:0]	Memory access size	Output	Used to generate **HSIZE**.
MCLK	Memory clock input	Output	Main clock input of opposite phase to **HCLK**.
nCPI	Not coprocessor instruction	Output	Only used for test.
nENIN	Not enable input	Input	Tied LOW to enable data buses.
nENOUT	Not enable output	Output	Only used for test.
nENOUTI	Not enable output	Output	Only used for test.
nEXEC	Not executed	Output	Only used for test.
nFIQ	Not fast interrupt request	Input	Connected to system **ARMNFIQ**.
nHIGHZ	Not HIGHZ	Output	Unconnected output.

Table 3-2 Connections of ARM7TDMI signals (continued)

Signal	Type	Direction	Connected to
nIRQ	Not interrupt request	Input	Connected to system **ARMNIRQ**.
nM[4:0]	Not processor mode	Output	Only used for test.
nMREQ	Not processor request	Output	Used to generate **HTRANS**.
nOPC	Not opcode fetch	Output	Used to generate **HPROT** and **HLOCKarm**.
nRESET	Not reset	Input	From system reset **HRESETn**.
nRW	Not read/write	Output	Used to generate **HWRITE**.
nTDOEN	Not **TDO** enable	Output	To system **nTDOEN**.
nTRANS	Not memory translate	Output	Used to generate **HPROT**.
nTRST	Not test reset	Input	From system **nTRST**.
nWAIT	Not wait	Input	Tied HIGH, as wait states are inserted by disabling the core clock **MCLK**.
PCLKBS	Boundary scan update clock	Output	Unconnected output.
RANGEOUT[1:0]	EmbeddedICE macrocell	Output	Only used for test.
RSTCLKBS	Boundary scan reset clock	Output	Unconnected output.
SCREG[3:0]	Scan chain register	Output	Only used for test.
SDINBS	Boundary scan serial input data	Output	Unconnected output.
SDOUTBS	Boundary scan serial output data	Input	Tied LOW as no external scan chains implemented.
SEQ	Sequential address	Output	Used to generate **HTRANS**.
SHCLKBS	Boundary scan shift clock, phase 1	Output	Unconnected output.
SHCLK2BS	Boundary scan shift clock, phase 2	Output	Unconnected output.
TAPSM[3:0]	TAP controller state machine	Output	Unconnected output.
TBE	Test bus enable	Input	Tied HIGH to drive outputs.

Table 3-2 Connections of ARM7TDMI signals (continued)

Signal	Type	Direction	Connected to
TBIT	Thumb state	Output	Only used for test.
TCK	Test clock	Input	From system **TCK**.
TCK1	TCK, phase 1	Output	Unconnected output.
TCK2	TCK, phase 2	Output	Unconnected output.
TDI	Test data input	Input	From system **TDI**.
TDO	Test data output	Output	To system **TDO**.
TMS	Test mode select	Input	From system **TMS**.

3.6 Default signal configurations

Within the wrapper there are a number of control signals that are tied to default values. The following configurations exist:

- **BIGEND** is tied LOW for little-endian operation, but may be tied HIGH for big-endian operation.
- **ISYNC** is tied HIGH for synchronous interrupts, but may be tied LOW if asynchronous interrupts are used.
- The debug input signals (**BREAKPT**, **DBGEN**, **DBGRQ** and **EXTERN[1:0]**) are tied to fixed values. These signals may be used to implement additional debug logic external to the core.
- The coprocessor signals (**CPA**, **CPB**) are tied HIGH, but will be required if an external coprocessor is to be added.
- If an additional boundary scan is to be added, the **SDOUTBS** input will be required.

3.7 Description of the ARM7TDMI wrapper blocks

This section contains descriptions of each of the following blocks:

- *A7TWrap*
- *A7TWrapBurst*
- *A7TWrapLock* on page 3-23
- *A7TWrapMaster* on page 3-27
- *A7TWrapCtrl* on page 3-33
- *A7TWrapTest* on page 3-34
- *Non-standard design practices* on page 3-42.

3.7.1 A7TWrap

This top-level block is purely structural, and connects together all of the blocks within the wrapper.

If the test interface is to be removed, it can be done by removing it from this module and tying the unconnected output signals that are generated to appropriate levels, as described within the HDL code. Removal of the test multiplexor is optional, because when the test interface is removed the multiplexor control input will be tied LOW.

3.7.2 A7TWrapBurst

The burst control block generates the AHB address and control outputs from the core address and control outputs. A simplified diagram of the HDL code is shown in Figure 3-3 on page 3-15.

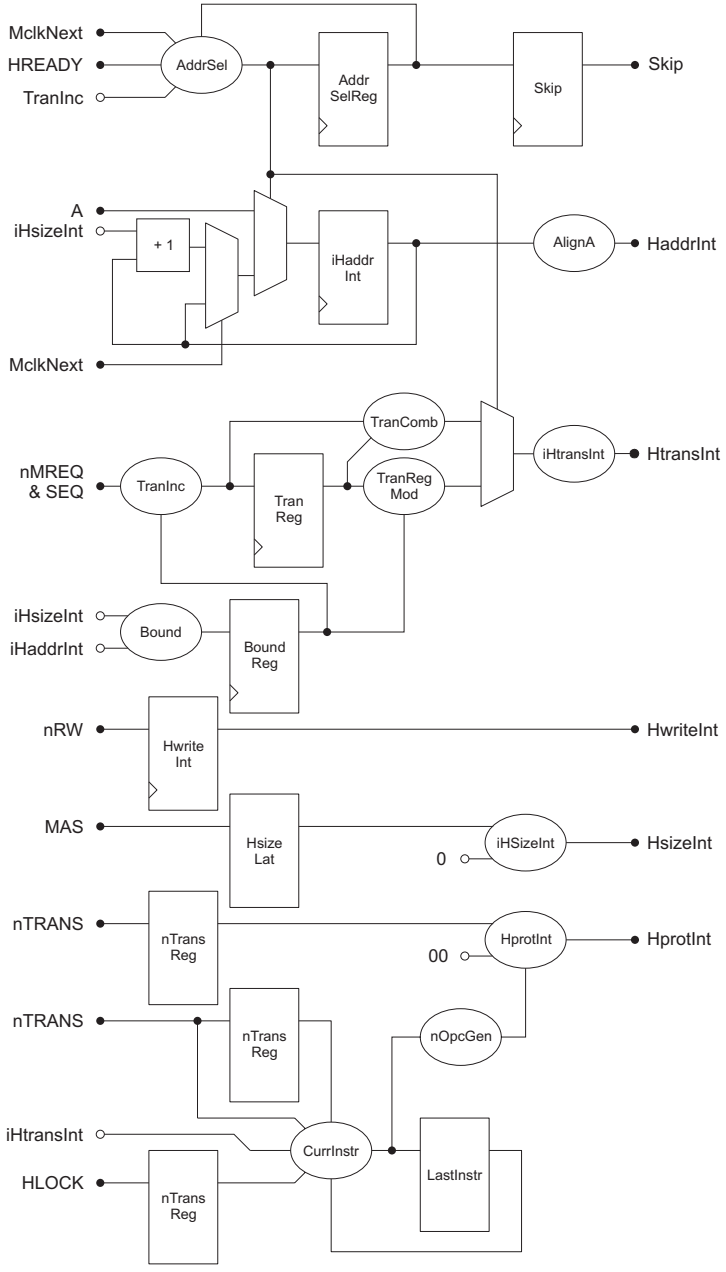


Figure 3-3 A7TWrapBurst block system diagram

There are three main sections to this block:

- *HADDR generation*
- *HTRANS generation* on page 3-20
- *Other control signal generation* on page 3-20.

HADDR generation

The address generation section generates the output address from two sources:

- the core address output **A**
- the internal address incrementer.

These address sources are selected according to the current transfer type, using the state machine shown in Figure 3-4.

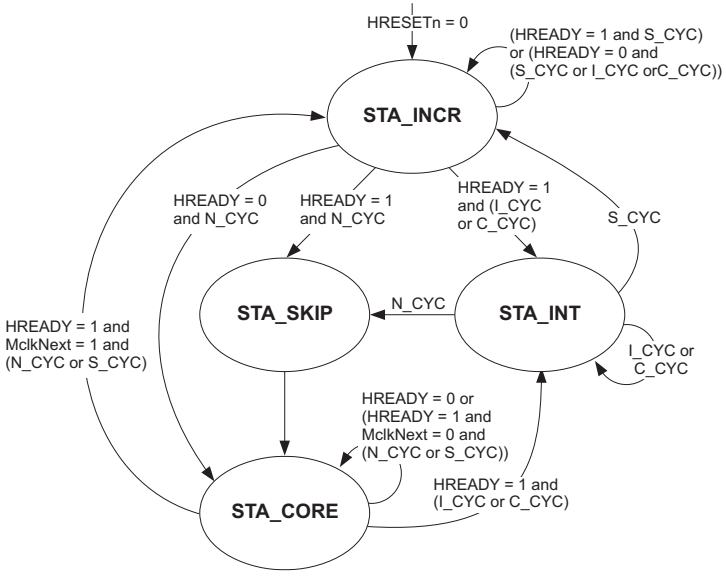


Figure 3-4 Address selection state machine

The four states are described in:

- *STA_INCR* on page 3-17
- *STA_SKIP* on page 3-17
- *STA_INT* on page 3-18
- *STA_CORE* on page 3-18.

- STA_INCR

  During this state the bus is synchronized with the core, and is performing the current core transfer.

  Due to the late address output from the core, the incrementer is used as the current source for the AHB address output, allowing the system address to be driven out much earlier than the core **A** output.

  As the current core transfer is the same as the current bus transfer, the **HTRANS** output is generated combinatorially from the core outputs.

  The STA_INCR state is entered from:

  — reset, when the system is initialized

  — STA_INT when an S_CYC follows an I_CYC or C_CYC

  — STA_CORE when both the bus and the core are clocked after an N_CYC

  — STA_INCR when the bus is clocked with an S_CYC, or when the bus is not clocked and is not performing an N_CYC.

  The next state is:

  — STA_SKIP when the bus is clocked and the core starts an N_CYC

  — STA_INT when the bus is clocked and the core starts an I_CYC or C_CYC

  — STA_CORE when the bus is not clocked and the core starts an N_CYC

  — STA_INCR when the bus is clocked with an S_CYC, or when the bus is not clocked and is not performing an N_CYC.

- STA_SKIP

  During this state the core is running one cycle ahead of the bus.

  When the core starts an N_CYC, a new address value will be driven out by the core that is not related to the address of the previous transfer. An IDLE cycle is inserted on the AHB during the STA_INCR state when the core first started the N_CYC, allowing time for the new core address to be sampled, and then the NONSEQUENTIAL transfer is started on the AHB during this state.

  To allow the AHB to resynchronize with the core, the core clock is stopped during the next cycle, using the skip block output which is passed to the core clock enabling logic.

  The STA_SKIP state is entered from:

  — STA_INCR when the bus is clocked and the core starts an N_CYC

  — STA_INT when the core performs an N_CYC following an I_CYC or C_CYC.

  The next state is always STA_CORE, as the previous IDLE transfer will always receive a zero wait OKAY response.

- STA_INT

  During this state the core and bus are synchronized.

  When the bus is clocked and the core starts an internal cycle, this state is entered. The core address is selected, as if the following transfer is a SEQUENTIAL or internal the address output will not change. If the next transfer is a NONSEQUENTIAL, then a new address will be driven out by the core, and will be sampled when the STA_SKIP state is entered.

  The STA_INT state is entered from:

  — STA_INCR when the bus is clocked and the core starts an I_CYC or C_CYC after an S_CYC

  — STA_CORE when the bus is clocked and the core starts an I_CYC or C_CYC after an N_CYC

  — STA_INT when following an I_CYC or C_CYC the bus is not clocked or the core starts another I_CYC or C_CYC.

  The next state is:

  — STA_INCR when the core starts an S_CYC

  — STA_SKIP when the core starts an N_CYC

  — STA_INT when the core starts an I_CYC or C_CYC or the bus is not clocked.

- STA_CORE

  During this state the bus resynchronizes with the core, as the core is not clocked, and the current core transfer is started on the AHB. The core address output is used to generate **HADDR**.

  When this state is entered from STA_INCR, the STA_SKIP state is bypassed. This is possible as when the bus is not clocked and the core starts an N_CYC, the IDLE cycle will be started on the bus during the **HREADY** LOW cycle. As **HREADY** being LOW causes the core clock to be disabled for one cycle, then the STA_SKIP state is not needed to allow the bus and core to resynchronize, and the STA_CORE state can be entered immediately as the core and bus will now be synchronized again.

  The STA_CORE state is entered from:

  — STA_SKIP following an N_CYC when the bus is clocked

  — STA_INCR following an N_CYC when the bus is not clocked

  — STA_CORE when the bus is not clocked, or when the bus is clocked and the core is not clocked and is performing an S_CYC or N_CYC.

The next state is:

— STA_INCR when both the bus and core are clocked

— STA_INT when the bus is clocked and the core starts an I_CYC or C_CYC

— STA_CORE when the bus is not clocked, or when the bus is clocked and the core is not clocked and is performing an S_CYC or N_CYC.

The output of the state machine is used to control the input to the iHaddrInt registers, selecting either the incremented or the core address. There is also a multiplexor after the incrementer which selects the current AHB address output when the core is not clocked, because the incremented address must not change if the core is not clocked. For example, this happens when the wrapper is not granted the bus but the core wants to perform a transfer.

The 8-bit incrementer uses the **HADDR** output value, and increments against halfword or word boundaries depending on the size of the current transfer. When the address incrementer overflows, the bound signal becomes valid, which causes an IDLE and NONSEQUENTIAL transfer sequence to be inserted, allowing time for the new address value to be sampled from the core.

The **HADDR** output is stored in a register to improve the output timing. This structure may need to be changed, as detailed in the HDL code, depending on the clock frequency that the system is driven with. The default system assumes that the **A** address output from the core becomes valid in time to be sampled on the rising edge of **HCLK** by the iHaddrInt registers. If this is not possible then an array of latches must be used to hold the address, with an array of registers used to store the output of the incrementer, as shown in Figure 3-5.

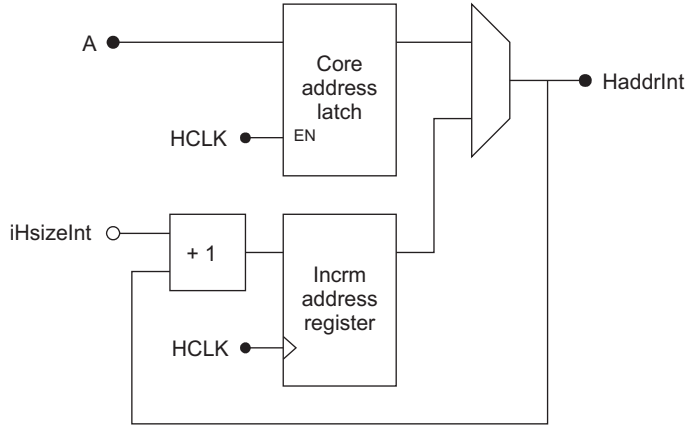


Figure 3-5 Address output latches used with slow core output address

HTRANS generation

The **HtransInt** output is mainly generated from the core transfer signals **nMREQ** and **SEQ**, as well as the address selection state machine.

The **HTRANS** output multiplexor is used to select either the current core transfer output, or the previous transfer held in the TranReg registers, according to the address selection state machine.

When the incremented address is the source of the current AHB address, then the current core transfer outputs are used to generate the AHB transfer outputs. As the incremented address is generated before the core starts the transfer that relates to that address, then the direct core outputs must be used.

When the core address is the current source, then the registered transfer value is used to generate the AHB transfer outputs. When the core address is valid, it relates to the previous transfer that the core was generating, so the registered transfer values must be used.

The core transfer type outputs are modified so that:

- when an incrementer address boundary is crossed, a NONSEQUENTIAL cycle is generated
- when a SEQUENTIAL follows an INTERNAL cycle, a NONSEQUENTIAL is generated
- when the core indicates an INTERNAL or COPROCESSOR cycle, an IDLE is generated.

The output of the multiplexor is then converted from the core transfer type encoding into the AHB **HTRANS** transfer encoding.

Other control signal generation

Registers are used to generate **HwriteInt**, **HsizeInt(1:0)** and **HprotInt(1)** directly from the core outputs, so that they are valid with the correct AHB timing.

It is assumed that the core outputs being used are valid early enough to be sampled on the rising edge of **HCLK** (falling edge of **MCLK**). If these signals cannot be guaranteed to be valid on this edge (for example, the system bus speed is run at a frequency approaching the maximum frequency of the ARM7TDMI core, effectively making the core outputs valid later in the clock cycle), then clock HIGH enabled transparent latches will need to be used to store the signals, so that the control outputs are stable at the end of the address phase on the rising edge of the clock.

The timing of the nOPC output from the core does not naturally match up with the AHB bus control signal timing, so an internally generated version of this signal is used to drive bit zero of the **HprotInt** output. A simple state machine is used to control the generation of this signal, which is shown in Figure 3-6 below:

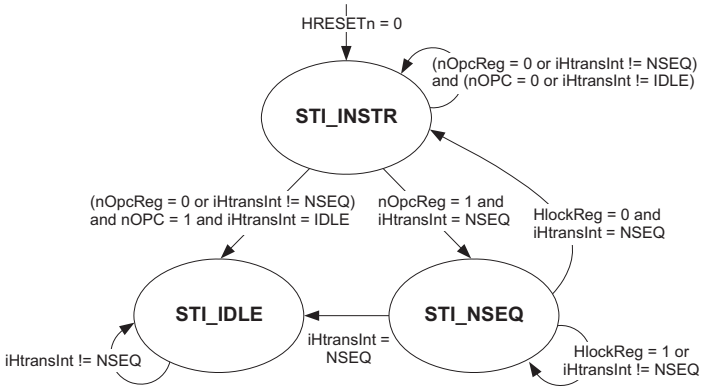


Figure 3-6 Instruction fetch state machine

The three states are described in:

- *STI_INSTR*
- *STI_IDLE* on page 3-22
- *STI_NSEQ* on page 3-22.

STI_INSTR

During this state the wrapper is performing an instruction fetch, and bit zero of **HprotInt** is driven HIGH.

The STI_INSTR state is entered from:

- reset, when the system is initialized
- STI_NSEQ when a NONSEQUENTIAL transfer is started that is not part of a locked transfer
- STI_INSTR when an instruction fetch is being performed.

The next state is:

- STI_IDLE when an IDLE transfer is inserted before a transfer that is not an instruction fetch

- STI_NSEQ when a NONSEQUENTIAL transfer is performed that is not an instruction fetch
- STI_INSTR when an instruction fetch is being performed.

STI_IDLE

During this state the wrapper is performing an IDLE transfer before a NONSEQUENTIAL bus access that is not an instruction fetch. Bit zero of **HprotInt** is driven HIGH, as during an IDLE cycle the value of **HPROT** is not used.

The STI_IDLE state is entered from:

- STI_INSTR when an IDLE transfer is inserted before a transfer that is not an instruction fetch
- STI_IDLE when the NONSEQUENTIAL transfer has not been started yet.

The next state is:

- STI_NSEQ when the NONSEQUENTIAL transfer has started, following the IDLE
- STI_IDLE when the NONSEQUENTIAL transfer has not been started yet.

STI_NSEQ

During this state the wrapper is not performing an instruction fetch, so bit zero of **HprotInt** is driven LOW.

The STI_NSEQ state is entered from:

- STI_INSTR when a NONSEQUENTIAL transfer is performed that is not an instruction fetch
- STI_IDLE when the NONSEQUENTIAL transfer has started, following the IDLE
- STI_NSEQ when a burst of SEQUENTIAL transfers is being performed, or a locked transfer is performed that consists of two back to back NONSEQUENTIAL transfers.

3.7.3 A7TWrapLock

The Lock generation block controls the generation of the **HLOCK** wrapper output, which is only set when the core performs a SWP instruction. A simplified diagram of the HDL code is shown in Figure 3-7.

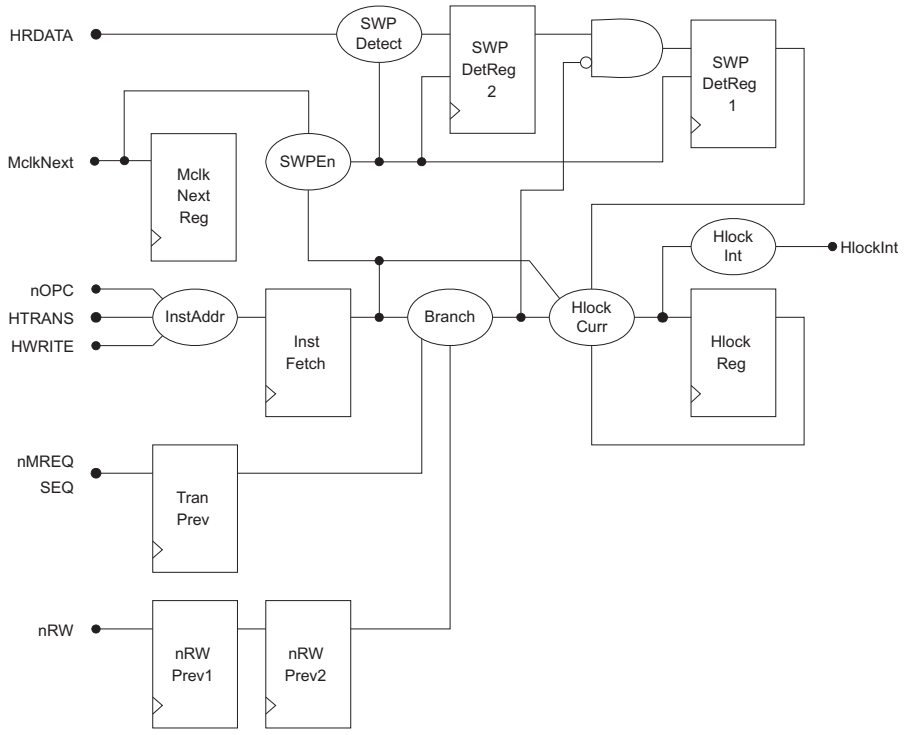


Figure 3-7 A7TWrapLock block system diagram

This block detects when the core is about to perform a SWP instruction by looking at the instructions that are being read in on the **HRDATAin** bus, and then checking that the instruction is actually being executed when it exits the instruction pipeline.

When the block detects a SWP instruction being read in, it is passed along the line of two registers (equivalent to the three-stage pipelining of the ARM7TDMI, with the first stage being the cycle that the instruction is read in). It is then used to set the **HLOCK** output HIGH when the SWP instruction is performed.

The code can branch before the SWP is performed, and this needs to be checked for. This is done by detecting when a NONSEQUENTIAL instruction fetch is performed, as this indicates that a new location in memory has been jumped to. The only case when this does not indicate a branch is when it follows a write cycle. The first instruction fetch after a write will always be NONSEQUENTIAL, and a branch will never immediately follow a write. A read is always followed by an INTERNAL cycle, and the following instruction fetch will be SEQUENTIAL if the code has not branched. So, the branch signal is set HIGH when there is a SWP instruction in the pipeline and the core is performing a NONSEQUENTIAL instruction fetch which is not immediately after a write. This is then used to synchronously reset the last SWP register (register 2 is not cleared as the instruction after the branch may be a SWP) and clear iHlockInt, so that the **HLOCK** output does not get set if the code branches before it reaches the SWP instruction.

If wait states or SPLIT/RETRY cycles are inserted during an instruction fetch, then it is possible for the read data to become available many cycles after the address cycle of the instruction fetch is performed. MclkNext is used to enable the SWP detection registers, ensuring that they are only clocked when the instruction fetch transfer completes.

The lock state machine shown in Figure 3-8 on page 3-25 controls the generation of the **HLOCK** wrapper output. Two locked states are used to indicate the locked read and write transfers that are performed during a SWP instruction.

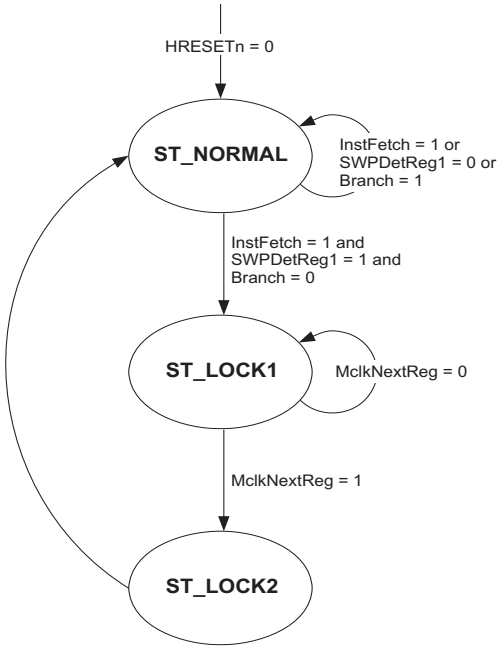


Figure 3-8 Locked state machine

The three states are described in:

- *ST_NORMAL*
- *ST_LOCK1* on page 3-26
- *ST_LOCK2* on page 3-26.

- ST_NORMAL

  This state is used when the core is not performing a locked SWP transfer, and the **HLOCK** output is held LOW.

  The ST_NORMAL state is entered from:

  — reset, when the system is initialized

  — ST_LOCK2 when the transfer after the SWP read has started

  — ST_NORMAL when a locked transfer is not being performed on the bus

  The next state is:

  — ST_LOCK1 when the instruction fetch preceding the SWP transfers is performed and the SWP is not branched past

  — ST_NORMAL when a locked transfer is not being performed on the bus.

- ST_LOCK1

  This state is entered during the transfer before the read transfer of the SWP instruction, which is always an instruction fetch. If a branch has occurred before this instruction fetch has been performed, then this state is not entered, as it will not be necessary to perform a locked transfer.

  The **HLOCK** output is held HIGH, indicating that the next AHB transfer (the SWP read) is locked.

  The ST_LOCK1 state is entered from:

  — ST_NORMAL when the instruction fetch preceding the SWP transfers is performed and the SWP is not branched past

  — ST_LOCK1 when the core has not started the read transfer of the SWP instruction.

  The next state is:

  — ST_LOCK2 when the core has been clocked, and is starting the locked read transfer of the SWP instruction

  — ST_LOCK1 when the core has not started the locked read transfer of the SWP instruction.

- ST_LOCK2

  This state is used during the read of the locked transfer to indicate that the following write transfer is also locked.

  The **HLOCK** output is held HIGH, indicating that the next AHB transfer (the SWP write) is locked.

  The ST_LOCK2 state is always entered from ST_LOCK1 when the core has been clocked and starts the read transfer.

  The next state is always ST_NORMAL, when the locked write transfer is started.

  If multiple SWP instructions are performed sequentially, then the state machine will continue to cycle through all three states as the core performs:

  1. an instruction fetch
  2. the read and write transfers of the SWP
  3. the next instruction fetch and SWP instruction.

3.7.4 A7TWrapMaster

This block controls the interface of the wrapper to the AHB as a bus master. A simplified diagram of the HDL code is shown in Figure 3-9.

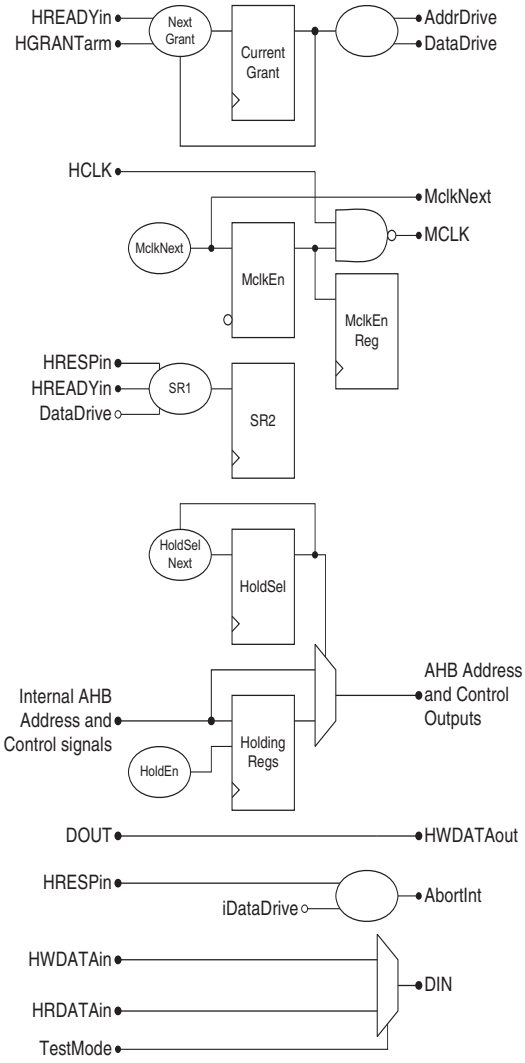


Figure 3-9 A7TWrapMaster block system diagram

The A7TWrapMaster block is made up of many different sections, the main ones being the:

- *Granted state machine*
- *Core clock generation* on page 3-30
- *Address and control holding registers* on page 3-31
- *Address, control and data output drivers* on page 3-32.

Granted state machine

This is used to determine when the wrapper is granted the bus as a bus master, and when it can drive the address, control and data outputs without clashing with other bus masters. The AddrDrive and DataDrive outputs are generated from the current state, and can be used to enable or disable the wrapper address, control and data outputs onto the AHB depending on the interconnection scheme used.

The state machine used is shown in Figure 3-10, and only advances when **HREADYin** is HIGH.

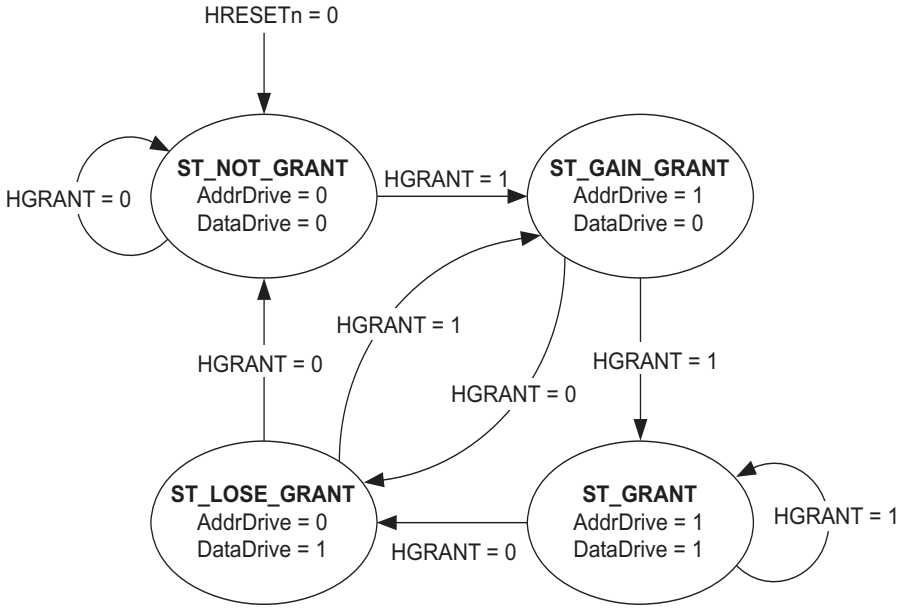


Figure 3-10 A7TWrapMaster block state machine

The four states are described in:

- *ST_NOT_GRANT* on page 3-29
- *ST_GAIN_GRANT* on page 3-29

- *ST_GRANT*
- *ST_LOSE_GRANT* on page 3-30.

- ST_NOT_GRANT

  This state is used when the wrapper is not granted control of the bus, and the address, control and data outputs are all driven LOW to avoid clashing with the current bus master.

  The ST_NOT_GRANT state is entered from:

  — reset, when the system is initialized

  — ST_LOSE_GRANT when the grant input has been set LOW for two completed bus cycles

  — ST_NOT_GRANT when the grant input has been set LOW for at least three completed bus cycles.

  The next state is:

  — ST_GAIN_GRANT when the grant input is first set HIGH

  — ST_NOT_GRANT when the grant input is set LOW.

- ST_GAIN_GRANT

  This state is used when the wrapper has first been granted control of the bus, and can drive the address and control outputs onto the bus. The previously granted master still has control of the read and write data buses from the previous transfer.

  The ST_GAIN_GRANT state is entered from:

  — ST_NOT_GRANT when the grant input is first set HIGH

  — ST_LOSE_GRANT when the wrapper lost control of the bus for one cycle, but has been granted control of the bus again.

  The next state is:

  — ST_GRANT when the grant input is set HIGH

  — ST_LOSE_GRANT when the wrapper was only granted control of the bus for one cycle.

- ST_GRANT

  This state is used when the wrapper has been granted the bus for at least two cycles, and can drive all of the address, control and data outputs without clashing with another bus master.

  The ST_GRANT state is entered from:

  — ST_GAIN_GRANT when the grant input has been set HIGH for two completed bus cycles

  — ST_GRANT when the grant input has been set HIGH for at least three completed bus cycles.

The next state is:

— ST_LOSE_GRANT when the grant input is first set LOW

— ST_GRANT if the grant input is still set HIGH.

- ST_LOSE_GRANT

  This state is used when the grant input has just been set LOW, and the wrapper can only drive the data outputs. The address and control lines will be driven by the currently granted bus master.

  The ST_LOSE_GRANT state is entered from:

  — ST_GRANT when the grant input is first set LOW

  — ST_GAIN_GRANT when the wrapper was only granted control of the bus for one cycle.

  The next state is:

  — ST_NOT_GRANT when the grant input has been set LOW for two cycles, and the wrapper fully loses control of the bus

  — ST_GAIN_GRANT when the wrapper lost control of the bus for one cycle, but has been granted control of the bus again.

Core clock generation

This controls the generation of the **MCLK** input to the core, which is of the opposite phase to **HCLK**.

The core clock is disabled:

- when **HREADYin** is LOW and the core is granted the bus
- when the holding registers contain an unperformed transfer
- when the Skip input from the A7TWrapBurst block is set HIGH, indicating that the clock must be disabled while a new core address is being sampled.

The clock is enabled at all other times.

A combinatorial path is used between the system clock input (**HCLK**), the enabling and inverting logic, and the core clock output (**MCLK**). This is required as both clocks are running at the same frequency, so a registered **MCLK** output cannot be used.

The latched enable signal (**MclkEn**) is gated with the system clock to generate the core clock. A latch is used for the enable to ensure that no glitches are generated after the rising edge of the system clock.

If a rising edge register was used to hold the enable, then when the enable changed from HIGH to LOW, the output of the enable register would not change until after the rising edge of the clock. This would mean that the core clock output would be set LOW for a short time until the enable output became HIGH.

The registered clock enable (**MclkEnReg**) is used in this block to control the operation of the holding registers, and is needed to align the clock enable to the rising edge of the clock.

Address and control holding registers

Holding registers are needed to allow the regeneration of transfers that were stopped due to a SPLIT or RETRY response from the current slave, or when the wrapper loses grant.

During normal operation the outputs of the holding registers are not used, but they are enabled so that they always contain a copy of the previous transfer address and control data. When they are needed, the enable is set LOW, ensuring that the registers hold their current values until after the transfer has been regenerated. The clock signal **Enable** is used to control the loading of the holding registers.

A multiplexor is used to select either the current transfer signals, or the holding register outputs. As it is possible for a transfer to be held for many cycles, a register is used to store the multiplexor control signal. This is set and cleared using the **HoldSet** and **HoldClr** signals.

HoldSel is set HIGH during a split or retry cycle, or when the core has lost grant, and is requesting a SEQUENTIAL or NONSEQUENTIAL transfer.

HoldSel is cleared when the holding registers have been used to regenerate a transfer, which is when the wrapper has been regranted the bus, or during the second phase of a SPLIT or RETRY cycle when the wrapper has not lost grant of the bus.

The **HTRANS** signal is modified before and after the holding registers. As a regenerated transfer will never be part of a burst, then if a SEQUENTIAL transfer is stored in the holding registers it is first converted to a NONSEQUENTIAL. Also, after the holding register selection multiplexors, the transfer type is converted to IDLE during the second cycle of a SPLIT or RETRY transfer, as the current transfer must always be IDLE during this cycle.

The holding register **HLOCK** signal is modified before passing through the holding register selection multiplexers. If an idle transfer is generated after an instruction fetch and immediately before a locked SWP read, it will not be regenerated if the instruction fetch is split or retried. This means that when the instruction fetch is regenerated, the next transfer will be the locked read, so the **HLOCK** output must be set HIGH during the instruction fetch transfer. As the holding registers will not have **HLOCK** set HIGH for the instruction fetch, a combinatorial path must exist between the internal lock signal and the **HLOCK** wrapper output. This is generated by ORing the internal lock signal with the output of the lock holding register.

Address, control and data output drivers

The outputs from the holding register multiplexers are used to directly drive the address and control outputs.

The **HBURST** output is held at 001, as the core only performs incrementing bursts of unspecified length.

The **HBUSREQarm** output is held HIGH, as the wrapper is always requesting use of the bus.

If it is not possible for the wrapper to continuously drive its outputs all of the time without clashing with other masters on the bus (for example, the system uses an OR bus connection scheme), the outputs must be enabled only when the wrapper is granted control of the bus.

This is done using the **AddrDrive** and **DataDrive** outputs from the granted state machine. AddrDrive is used to enable the address and control outputs (**HADDRout**, **HTRANSout**, **HWRITEout**, **HSIZE**, **HPROT** and **HLOCKarm**), and is set during the ST_GAIN_GRANT and ST_GRANT states. DataDrive is used to enable the write data output (**HWDATAout**) during the ST_GRANT and ST_LOSE_GRANT states.

The AHB slave outputs are only used during TIC testing mode. **HREADYout** is driven by the **HreadyInt** signal from the test block. **HRESPout** is always driven to OKAY, as the wrapper will never assert SPLIT, RETRY or ERROR responses.

This section of the block is also used to drive some core inputs. **AbortInt** (which is passed to the A7TWrapCtrl block) is set HIGH when an error response is generated from the currently selected slave, and is used to drive the ABORT core input during normal operation.

DIN, the core data input, is driven with **HWDATAin** during TIC testing mode, or with **HRDATAin** when the wrapper is acting as a standard AHB bus master.

3.7.5 A7TWrapCtrl

This block contains the test wrapper control multiplexor used during TIC testing of the core. A simplified diagram of the A7TWrapCtrl block is shown in Figure 3-11.

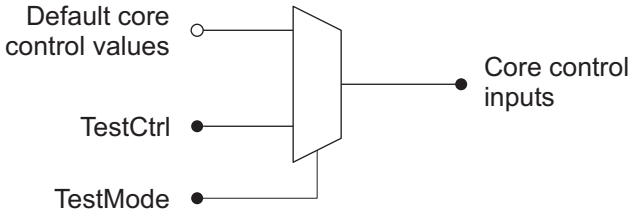


Figure 3-11 A7TWrapCtrl block system diagram

This block is only used during test mode when the wrapper is acting as an AHB slave, and drives the control inputs of the core with the TIC test data. It is separated from the main test block (A7TWrapTest) to allow for easier removal of the test wrapper.

When not in test mode the control inputs are driven to their default values (either HIGH or LOW), or are driven with wrapper inputs, such as the two interrupt lines and the JTAG pins. The **AbortInt** signal is generated in the A7TWrapMaster block.

If the test wrapper is removed, then this multiplexor will be optimized out during synthesis, and the outputs will be driven with their default values. It is also possible to remove this block if the test wrapper is not used. The default connections that the outputs must be tied to are shown in the A7TWrap HDL file.

BUSEN, **DBE** and **nENIN** are all set to constant values, because they do not need to be controlled during normal system use, or during core TIC testing.

3.7.6 A7TWrapTest

The test interface block is used to allow the wrapper module to act as an AHB slave during TIC testing of the core. A simplified diagram of the HDL code is shown in Figure 3-12.

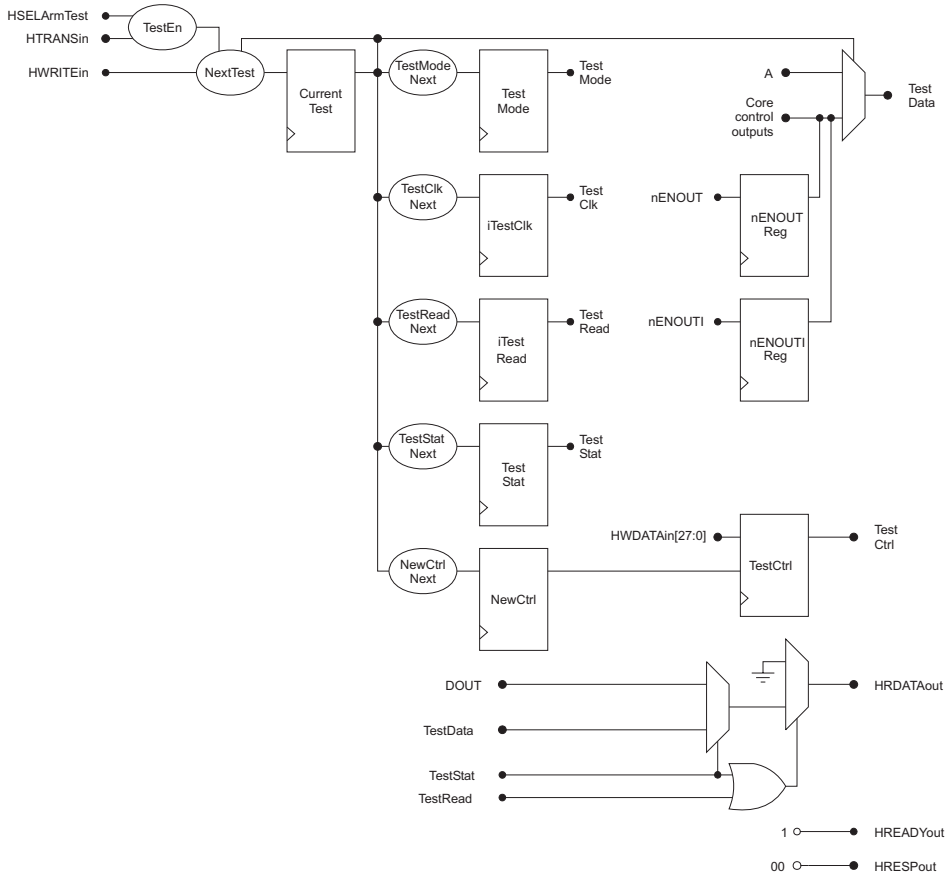


Figure 3-12 A7TWrapTest block system diagram

The main parts of this block are:

- the test state machine, which controls the application of the test vectors
- the 28-bit test register, which stores the value of the control inputs during test.

The state diagram for the test state machine is shown in Figure 3-13 on page 3-35.

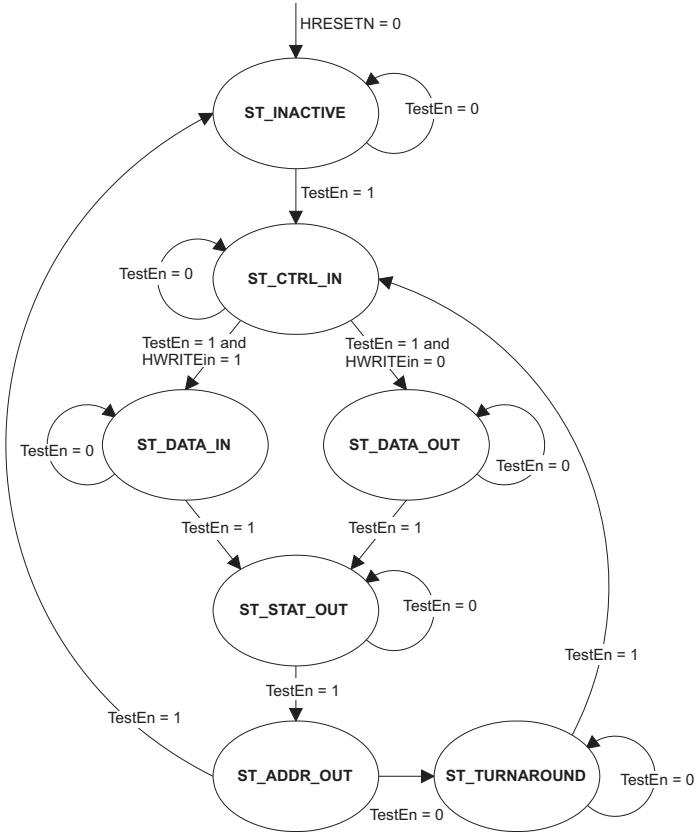


Figure 3-13 A7TWrap Test block state machine

The **TestEn** signal is used to control when test vectors are applied to the core, and therefore controls the transitions through the test state machine. **TestEn** is set HIGH when the core is addressed during a valid transfer, when the **HTRANS** input indicates a NONSEQUENTIAL or a SEQUENTIAL transfer.

The seven states are described in:

- *ST_INACTIVE* on page 3-36
- *ST_CTRL_IN* on page 3-36
- *ST_DATA_IN* on page 3-36
- *ST_DATA_OUT* on page 3-37
- *ST_STAT_OUT* on page 3-37
- *ST_ADDR_OUT* on page 3-37
- *ST_TURNAROUND* on page 3-38.

- ST_INACTIVE

  This state is used when the wrapper is not in test mode, and all test outputs are driven to their default levels. The core is clocked as normal in this state.

  The ST_INACTIVE state is entered from:

  — reset, when the system is initialized

  — ST_ADDR_OUT when the end of the test has been reached.

  The next state is:

  — ST_CTRL_IN when test mode is first entered

  — ST_INACTIVE when the test wrapper is not addressed during a valid transfer.

- ST_CTRL_IN

  This state is used to load the test register with the control data that is currently on the write data bus. This then determines the values of the control signals that will be applied to the core when it is clocked. The core is not clocked during this state.

  The ST_CTRL_IN state is entered from:

  — ST_INACTIVE when test mode is first entered

  — ST_TURNAROUND when the next control vector is being written onto the data bus

  — ST_CTRL_IN when test mode has been entered, but the wrapper is not currently selected.

  The next state is:

  — ST_DATA_IN when write data is being applied to the core

  — ST_DATA_OUT when read data is being loaded from the core.

- ST_DATA_IN

  In this state write data is being applied to the core (the core is performing a read transfer). The core is clocked in this state.

  The ST_DATA_IN state is entered from:

  — ST_CTRL_IN when write data is being applied to the core

  — ST_DATA_IN when write data has been applied to the core, but the wrapper is not currently selected.

  The next state is:

  — ST_STAT_OUT when the wrapper is selected for the output status signals to be read

  — ST_DATA_IN when the wrapper is not currently selected.

- ST_DATA_OUT

  In this state read data is being loaded from the core (the core is performing a write transfer). The core is clocked in this state.

  The ST_DATA_OUT state is entered from:

  — ST_CTRL_IN when read data is being loaded from the core

  — ST_DATA_OUT when read data has been loaded from the core, but the wrapper is not currently selected.

  The next state is:

  — ST_STAT_OUT when the wrapper is selected for the output status signals to be read

  — ST_DATA_OUT when the wrapper is not currently selected.

- ST_STAT_OUT

  This state is used to read the output status signals from the core. The core is not clocked in this state.

  The ST_STAT_OUT state is entered from:

  — ST_DATA_OUT when the previous transfer was a data write to the core

  — ST_DATA_IN when the previous transfer was a data read from the core

  — ST_STAT_OUT when the core output status signals have just been read, but the wrapper is not currently selected.

  The next state is:

  — ST_ADDR_OUT when the wrapper is selected for the address output to be read from the core

  — ST_STAT_OUT when the wrapper is not currently selected.

- ST_ADDR_OUT

  This state is used to read the address output from the core. The core is not clocked in this state.

  The ST_ADDR_OUT state is only entered from ST_STAT_OUT, when the previous transfer was a read of the core status outputs.

  The next state is:

  — ST_INACTIVE when the wrapper is still selected, indicating the end of the test

  — ST_TURNAROUND when the wrapper is not selected, indicating the turnaround cycle before the new control data is written to the test registers.

- ST_TURNAROUND

  This state is used to allow the external data bus time to turnaround between the address read cycle and the control vector write cycle. The core is not clocked in this state.

  The ST_TURNAROUND state is entered from:

  — ST_ADDR_OUT when the wrapper is not selected after the address read cycle

  — ST_TURNAROUND when a turnaround cycle has been inserted on the external data bus, but the wrapper is not currently selected.

  The next state is:

  — ST_CTRL_IN when the turnaround cycle has been inserted, and the next control vector is being written into the test registers

  — ST_TURNAROUND when the wrapper is not currently selected.

  The 28-bit test register that is loaded during the ST_CTRL_IN state determines the control inputs to the core when it is clocked during the ST_DATA_IN or ST_DATA_OUT states. Table 3-3 shows the control input bit positions.

Table 3-3 ARM7TDMI control input bit position

Signal	Description	Bit position	Comments
SDOUTBS	Boundary scan serial output data	27	-
TBE	Test bus enable	26	-
APE	Address pipeline enable	25	-
BL[3:0]	Byte latch control	24:21	ANDed with **TestClk**, and should only be valid during data access cycle.
TMS	Test mode select	20	-
TDI	Test data in	19	-
TCK	Test clock	18	ANDed with **TestClk.**
nTRST	Not test reset	17	-
EXTERN1	External input 1	16	-
EXTERN0	External input 0	15	-
DBGRQ	Debug request	14	-

Table 3-3 ARM7TDMI control input bit position (continued)

Signal	Description	Bit position	Comments
BREAKPT	Breakpoint	13	-
DBGEN	Debug enable	12	-
ISYNC	Synchronous interrupts	11	-
BIGEND	Big-endian configuration	10	-
CPA	Coprocessor absent	9	-
CPB	Coprocessor busy	8	-
ABE	Address bus enable	7	This should normally be set HIGH, as if the address bus is tristated (**ABE** LOW), then it will not be possible to read address values.
ALE	Address latch enable	6	-
DBE	Data bus enable	5	
nFIQ	Not fast interrupt request	4	-
nIRQ	Not interrupt request	3	-
ABORT	Memory abort	2	This should normally be driven when **HRESP** indicates ERROR, and the wrapper has control of the AHB data bus.
nWAIT	Not wait	1	ANDed with **TestClk**, so that the core state can only change during the data access cycle.
nRESET	Not reset	0	-

The test data output multiplexor is found in the A7TWrapCtrl block, but is controlled by the test outputs of this block. It is used to select between:

- core data output during ST_DATA_OUT
- core address output during ST_ADDR_OUT
- core status outputs during ST_STAT_OUT.

The selected output is driven onto the **HRDATAout** output data bus.

Table 3-4 shows the bit positions of the status output signals when driven on the data bus.

Table 3-4 ARM7TDMI status bit positions

Signal	Description	Bit position	Comment
BUSDIS	Bus disable	31	-
SCREG[3:0]	Scan chain register	30:27	These signals are not important to the normal functioning of the core, but are included in this test vector to give a slight improvement in fault coverage during scan and debug testing.
HIGHZ	HIGHZ instruction in TAP controller	26	-
nTDOEN	Not **TDO** enable	25	-
DBGRQ1	Internal debug request	24	-
RANGEOUT0	ICEbreaker Rangeout0	23	-
RANGEOUT1	ICEbreaker Rangeout1	22	-
COMMRX	Communications channel receive	21	-
COMMTX	Communications channel transmit	20	-
DBGACK	Debug acknowledge	19	-
TDO	Test data out	18	This value is often tristate (as indicated by **nTDOEN**), so will usually be masked out.
nENOUT	Not enable output	17	**nENOUT** is only valid during the data access cycle, so **TestClk** is used to clock a register that will capture the correct state.

Table 3-4 ARM7TDMI status bit positions (continued)

Signal	Description	Bit position	Comment
nENOUTI	Not enable output	16	**nENOUTI** is only valid during the data access cycle, so **TestClk** is used to clock a register that will capture the correct state.
TBIT	Thumb state	15	-
nCPI	Not coprocessor instruction	14	-
nM[4:0]	Not processor mode	13:9	-
nTRANS	Not memory translate	8	-
nEXEC	Not executed	7	-
LOCK	Locked operation	6	-
MAS[1:0]	Memory access size	5:4	-
nOPC	Not opcode fetch	3	-
nRW	Not read/write	2	-
nMREQ	Not memory request	1	-
SEQ	Sequential address	0	-

This test interface block may be removed if not required, by removing the A7TWrapTest block from the A7TWrap top level wrapper HDL file. It is then necessary to tie the outputs which were originally generated from this block to fixed values, and these are described in the A7TWrap HDL code.

Removing this block means that the test inputs to the A7TWrapCtrl block will be static, allowing the test multiplexors to be removed during synthesis, or manually removed from the HDL code.

The AHB slave outputs are only used during TIC testing mode. **HREADYout** is always driven HIGH, as the wrapper will never generate wait states. **HRESPout** is always driven to OKAY, as the wrapper will never assert split, retry or error responses.

HRDATAout is generated according to the current test control signal outputs, and is driven to either **DOUT** from the core, **TestData** from the test block (which is comprised of the core control outputs), or LOW.

If an OR bus interconnection scheme is used, then all three of these slave outputs must be driven LOW when the test wrapper has been removed or when the wrapper is not in test mode, so that they do not clash with other slaves on the bus.

3.7.7 Non-standard design practices

This section contains all the non-standard design practices that are used in the ARM7TDMI AHB wrapper.

Signal delays

In some cases delays have been added to some of the wrapper signals. This is necessary to ensure that in a zero delay RTL system simulation using a core model that requires setup and hold times, no timing violations are created on the core inputs. When a full netlist system simulation is run, then these delays are not required, as they will be provided by the cell and interconnect delays of the system.

These delays can be found in the A7TWrap top-level HDL file. They are used to create hold times on the data input to the core **DIN**, the external abort input **ABORT**, the **BL** and **nWAIT** core inputs, and the two interrupt sources **nFIQ** and **nIRQ**.

Transparent latches

Two of the wrapper blocks use transparent latches rather than registers to hold data values.

In the A7TWrapMaster block a transparent LOW latch is used to hold the clock enable value. This removes the chance of generating a glitch on the clock output, which is generated if the system clock is gated with a rising edge registered version of the clock enable.

In the A7TWrapBurst block, it may be necessary to use transparent HIGH latches to generate the **HSIZE** and **HPROT** outputs directly from the core outputs **MAS**, **nTRANS** and **nOPC**. The timing of these core outputs in the default system allows the use of registers, but if the clock speed of the system relative to the maximum clock speed of the core is increased, then latches may be required to sample these core output signals if they become valid after the rising edge of the clock.

Gated clock

The enabled clock output to the core is a gated version of the main system clock. This method of controlling the timing of the core has been chosen as due to the timing of the core and the AHB, it is not possible to use the **nWAIT** core input. This means that wait states must be passed to the core via the clock, so clock enabling and gating must be

used. This clock control logic has been constructed to eliminate the generation of glitches on the clock output, and only adds a single AND gate into the path between the system clock and the core clock.

Chapter 4
AHB Modules

This chapter describes the data sheets for the modules that are connected to the *Advanced High Performance Bus* (AHB). It contains the following sections:

4.1 APB bridge

The AHB to APB bridge is an AHB slave, providing an interface between the high-speed AHB and the low-power APB. Read and write transfers on the AHB are converted into equivalent transfers on the APB. As the APB is not pipelined, then wait states are added during transfers to and from the APB when the AHB is required to wait for the APB. Figure 4-1 shows the block diagram of the APB bridge module.

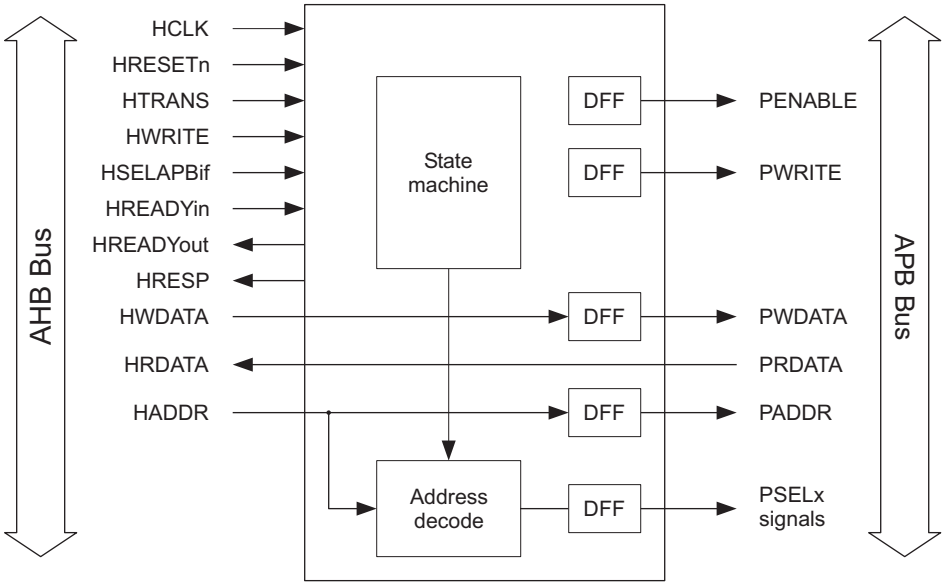


Figure 4-1 Block diagram of bridge module

The main sections of this module are:

- AHB slave bus interface
- APB transfer state machine, which is independent of the device memory map
- APB output signal generation.

To add new APB peripherals, or alter the system memory map, only the address decode sections need to be modified.

4.1.1 Signal descriptions

The APB bridge module signals are described in Table 4-1.

Table 4-1 Signal descriptions for bridge module

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HADDR[31:0]	Address bus	Input	The 32-bit system address bus.
HTRANS[1:0]	Transfer type	Input	This indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE	Transfer direction	Input	When HIGH this signal indicates a write transfer, and when LOW, a read transfer.
HWDATA[31:0]	Write data bus	Input	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HSELAPBif	Slave select	Input	Each APB slave has its own slave select signal, and this signal indicates that the current transfer is intended for the selected slave. This signal is a combinatorial decode of the address bus.
HRDATA[31:0]	Read data bus	Output	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HREADYin **HREADYout**	Transfer done	Input/output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HRESP[1:0]	Transfer response	Output	The transfer response provides additional information on the status of a transfer. This module will always generate the OKAY response.
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).

Table 4-1 Signal descriptions for bridge module (continued)

Signal	Type	Direction	Description
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).
PENABLE	Peripheral enable	Output	This enable signal is used to time 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.
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 will be set LOW at the end of the transfer.
PADDR[31:0]	Peripheral address bus	Output	This is the APB address bus, which may be up to 32 bits wide and is used by individual peripherals for decoding register accesses to that 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, then the address will change to the new value, otherwise it will hold its current value until the start of the next APB transfer.
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.

Timing diagrams showing the relationship between AHB and APB transfers can be found in the *APB Specification*.

4.1.2 Peripheral memory map

The APB bridge controls the memory map for the peripherals, and generates a select signal for each peripheral. The default system memory map is shown in Figure 4-2.

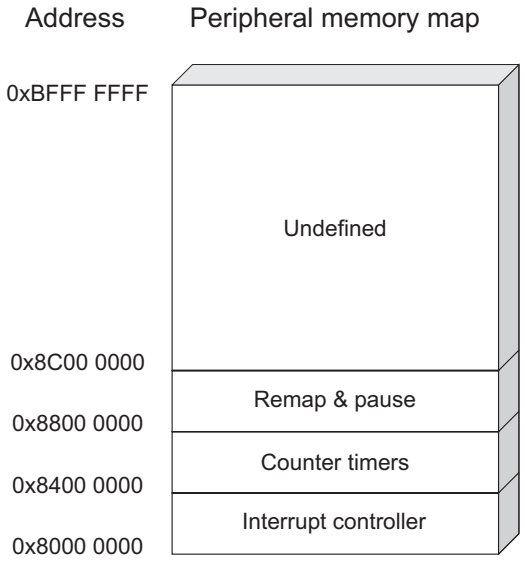


Figure 4-2 Peripheral memory map

4.1.3 Function and operation of module

The APB bridge responds to transaction requests from the currently granted AHB master. The AHB transactions are then converted into APB transactions. The state machine, shown in Figure 4-3 on page 4-6, controls:

- the AHB transactions with the **HREADYout** signal
- the generation of all APB output signals.

The individual **PSELx** signals are decoded from **HADDR**, using the state machine to enable the outputs while the APB transaction is being performed.

If an undefined location is accessed, operation of the system continues as normal, but no peripherals are selected.

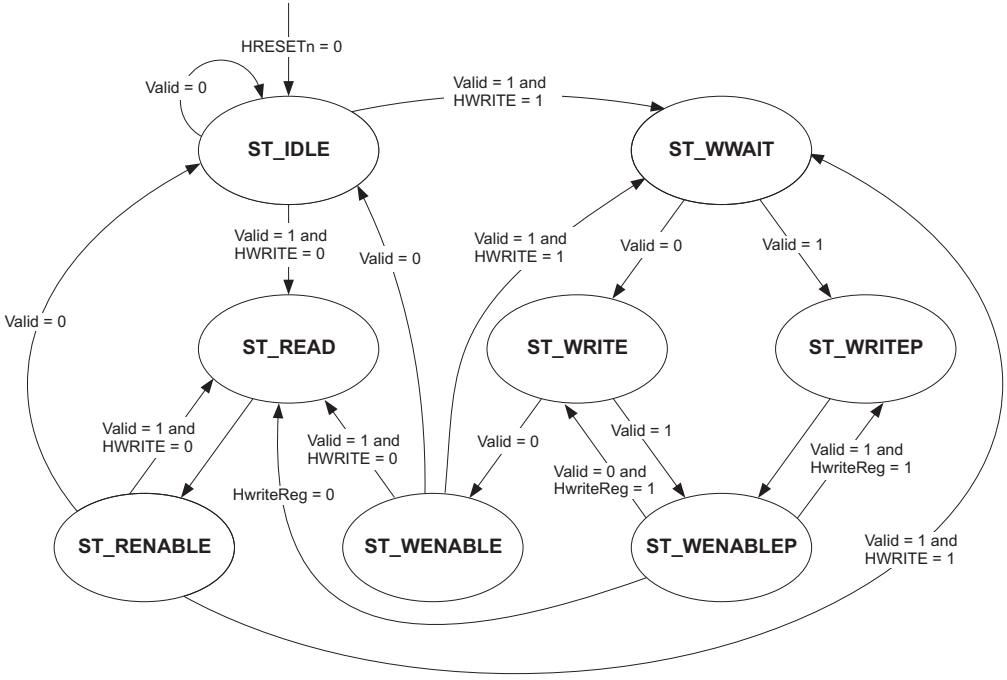


Figure 4-3 State machine for AHB to APB interface

The individual states of the state machine operation are described in the following sections:

- *ST_IDLE* on page 4-7
- *ST_READ* on page 4-7
- *ST_WWAIT* on page 4-7
- *ST_WRITE* on page 4-8
- *ST_WRITEP* on page 4-8
- *ST_RENABLE* on page 4-9
- *ST_WENABLE* on page 4-9
- *ST_WENABLEP* on page 4-9.

ST_IDLE

During this state the APB buses and **PWRITE** are driven with the last values they had, and **PSEL** and **PENABLE** lines are driven LOW.

The ST_IDLE state is entered from:

- reset, when the system is initialized
- ST_RENABLE, ST_WENABLE, or ST_IDLE, when there are no peripheral transfers to perform.

The next state is:

- ST_READ, for a read transfer, when the AHB contains a valid APB read transfer
- ST_WWAIT, for a write transfer, when the AHB contains a valid APB write transfer.

ST_READ

During this state the address is decoded and driven onto **PADDR**, the relevant **PSEL** line is driven HIGH, and **PWRITE** is driven LOW. A wait state is always inserted to ensure that the data phase of the current AHB transfer does not complete until the APB read data has been driven onto **HRDATA**.

The ST_READ state is entered from ST_IDLE, ST_RENABLE, ST_WENABLE, or ST_WENABLEP during a valid read transfer.

The next state will always be ST_RENABLE.

ST_WWAIT

This state is needed due to the pipelined structure of AHB transfers, to allow the AHB side of the write transfer to complete so that the write data becomes available on **HWDATA**. The APB write transfer is then started in the next clock cycle.

The ST_WWAIT state is entered from ST_IDLE, ST_RENABLE, or ST_WENABLE, during a valid write transfer.

The next state will always be ST_WRITE.

ST_WRITE

During this state the address is decoded and driven onto **PADDR**, the relevant **PSEL** line is driven HIGH, and **PWRITE** is driven HIGH.

A wait state is not inserted, as a single write transfer can complete without affecting the AHB.

The ST_WRITE state is entered from:

- ST_WWAIT, when there are no further peripheral transfers to perform
- ST_WENABLEP, when the currently pending peripheral transfer is a write, and there are no further transfers to perform.

The next state is:

- ST_WENABLE, when there are no further peripheral transfers to perform
- ST_WENABLEP, when there is one further peripheral write transfer to perform.

ST_WRITEP

During this state the address is decoded and driven onto **PADDR**, the relevant **PSEL** line is driven HIGH, and **PWRITE** is driven HIGH. A wait state is always inserted, as there must only ever be one pending transfer between the currently performed APB transfer and the currently driven AHB transfer. See the write transfer timing diagrams in the *AMBA Specification (Rev 2.0)* for more details.

The ST_WRITEP state is entered from:

- ST_WWAIT, when there is a further peripheral transfer to perform.
- ST_WENABLEP, when the currently pending peripheral transfer is a write, and there is a further transfer to perform.

The next state will always be ST_WENABLEP.

ST_RENABLE

During this state the **PENABLE** output is driven HIGH, enabling the current APB transfer. All other APB outputs remain the same as the previous cycle.

The ST_RENABLE state is always entered from ST_READ.

The next state is:

- ST_READ, when there is a further peripheral read transfer to perform
- ST_WWAIT, when there is a further peripheral write transfer to perform
- ST_IDLE, when there are no further peripheral transfers to perform.

ST_WENABLE

During this state the **PENABLE** output is driven HIGH, enabling the current APB transfer. All other APB outputs remain the same as the previous cycle.

The ST_WENABLE state is always entered from ST_WRITE.

The next state is:

- ST_READ, when there is a further peripheral read transfer to perform
- ST_WWAIT, when there is a further peripheral write transfer to perform
- ST_IDLE, when there are no further peripheral transfers to perform.

ST_WENABLEP

A wait state is inserted if the pending transfer is a read because, when a read follows a write, an extra wait state must be inserted to allow the write transfer to complete on the APB before the read is started.

The ST_WENABLEP state is entered from:

- ST_WRITE, when the currently driven AHB transfer is a peripheral transfer
- ST_WRITEP, when there is a pending peripheral transfer following the current write.

The next state is:

- ST_READ, when the pending transfer is a read
- ST_WRITE, when the pending transfer is a write, and there are no further transfers to perform
- ST_WRITEP, when the pending transfer is a write, and there is a further transfer to perform.

4.1.4 System description

This section describes how the HDL code for the APB bridge is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This should be read in conjunction with the HDL code.

Figure 4-4 shows the APB bridge module block diagram.

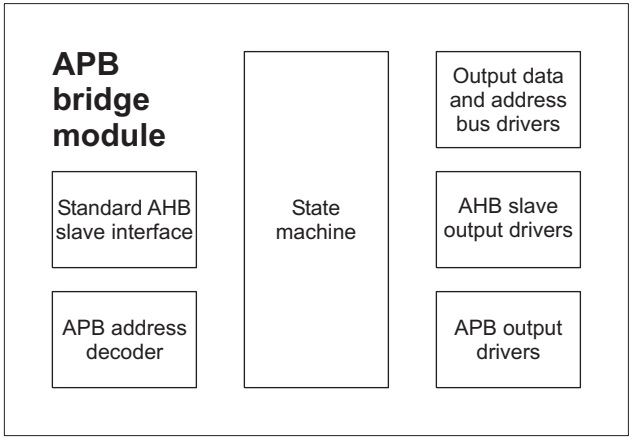


Figure 4-4 APB bridge module block diagram

The AHB to APB bridge comprises a state machine, which is used to control the generation of the APB and AHB output signals, and the address decoding logic which is used to generate the APB peripheral select lines.

All registers used in the system are clocked from the rising edge of the system clock **HCLK**, and use the asynchronous reset **HRESETn**.

Figure 4-5 on page 4-11 shows the APB bridge HDL file.

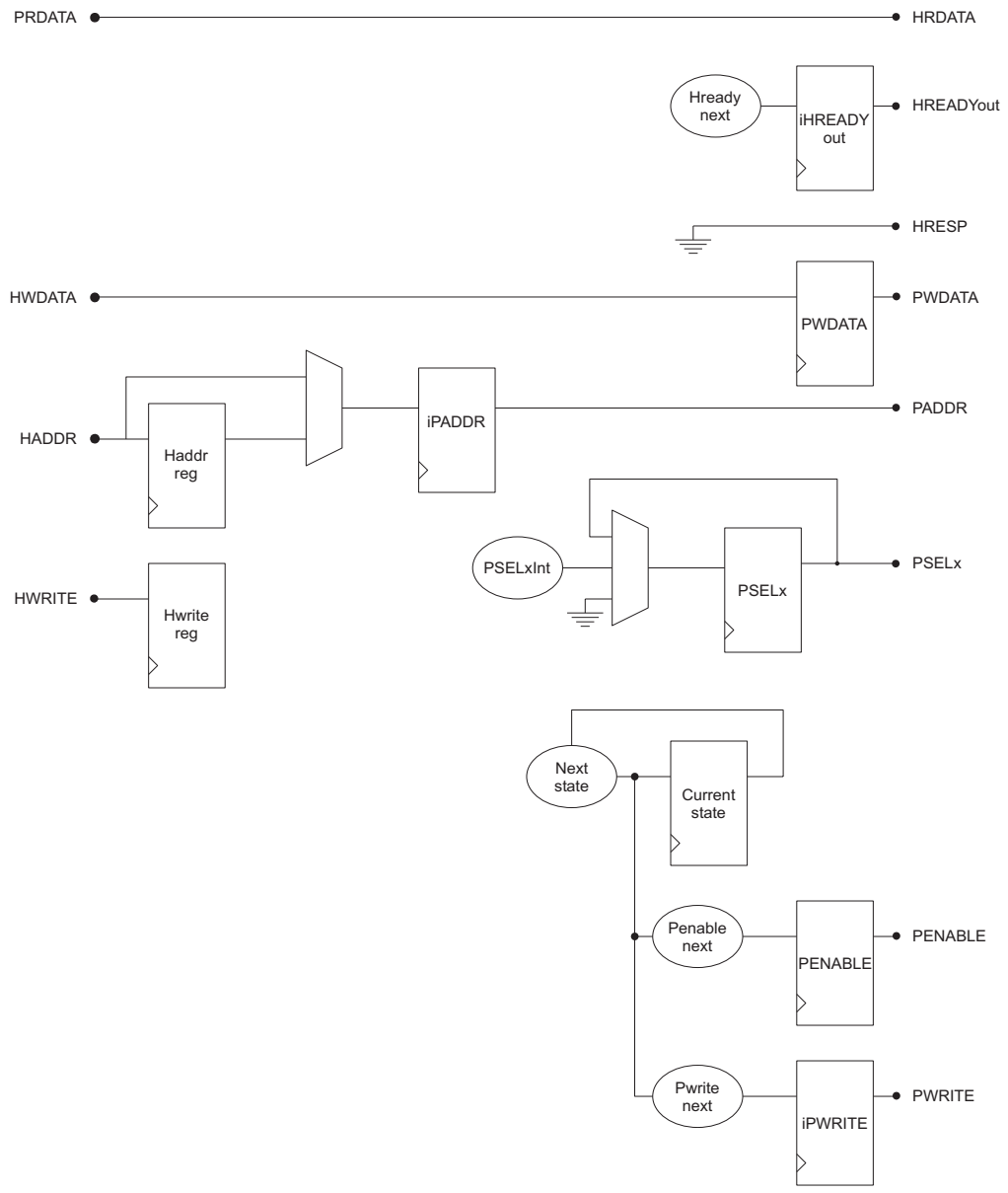


Figure 4-5 APB bridge module system diagram

The main sections in this module are explained in the following paragraphs:

- *Constant definitions*
- *AHB slave bus interface*
- *APB transfer state machine*
- *APB output signal generation* on page 4-13
- *AHB output signal generation* on page 4-13.

Constant definitions

The constant PADDRWIDTH sets the width of the peripheral address bus that is used, up to a maximum of 32 bits. This size depends on the size of address that is needed by the peripherals in the system. The default value is a 16-bit address bus.

The next two constants define the state machine states, and the top four address bits that are used to decode the peripheral select outputs. If the peripheral address map is changed from the default, then these constants must be modified to match the changes.

AHB slave bus interface

This module uses the standard AHB slave bus interface, which comprises:

- the valid transfer detection logic which is used to determine when a valid transfer is accessing the slave
- the address and control registers, which are used to store the information from the address phase of the transfer for use in the data phase.

Due to the different AHB to APB timing of read and write transfers, either the current or the previous address input value is needed to correctly generate the APB transfer. A multiplexor is therefore used to select between the current address input or the registered address, for read and write transfers respectively.

APB transfer state machine

The transfer state machine is used to control the application of APB transfers based on the AHB inputs. The state diagram in Figure 4-3 on page 4-6 shows the operation of the state machine, which is controlled by its current state and the AHB slave interface signals.

APB output signal generation

The generation of all APB output signals is based on the status of the transfer state machine:

- **PWDATA** is a registered version of the **HWDATA** input, which is only enabled during a write transfer. As the bridge is the only bus master on the APB, then it can drive **PWDATA** continuously.
- **PENABLE** is only set HIGH during one of three enable states, in the last cycle of an APB transfer. A register is used to generate this output from the next state of the transfer state machine.
- The **PSELx** outputs are decoded from the current transfer address. They are only valid during the read, write and enable states, and are all driven LOW at all other times so that no peripherals are selected when no transfers are being performed.
- **PADDR** is a registered version of the currently selected address input (**HADDR** or the address register) and only changes when the read and write states are entered at the start of the APB transfer.
- **PWRITE** is set HIGH during a write transfer, and only changes when a new APB transfer is started. A register is used to generate this output from the next state of the transfer state machine.
- The **APBen** signal is used as an enable on the **PSEL**, **PWRITE** and **PADDR** output registers, ensuring that these signals only change when a new APB transfer is started, when the next state is ST_READ, ST_WRITE, or ST_WRITEP.

AHB output signal generation

A standard AHB slave interface consists of the following three outputs:

- **HRDATA** is directly driven with the current value of **PRDATA**. APB slaves only drive read data during the enable phase of the APB transfer, with **PRDATA** set LOW at all other times, so bus clash is avoided on **HRDATA** (assuming OR bus connections for both the AHB and APB read data buses).
- **HREADYout** is driven with a registered signal to improve the output timing. Wait states are inserted by the APB bridge during the ST_READ and ST_WRITEP states, and during the ST_WENABLEP state when the next transfer to be performed is a read.
- **HRESP** is continuously held LOW, as the APB bridge does not generate SPLIT, RETRY or ERROR responses.

4.2 Arbiter

The AMBA bus specification is a multi-master bus standard. As a result, a bus arbiter is needed to ensure that only one bus master has access to the bus at any particular point in time. Each bus master requests control of the bus, and the arbiter decides which has the highest priority and issues a grant signal accordingly.

Every system must have a default bus master which is granted use of the bus when no other bus master requires control. Figure 4-6 shows the arbiter block diagram.

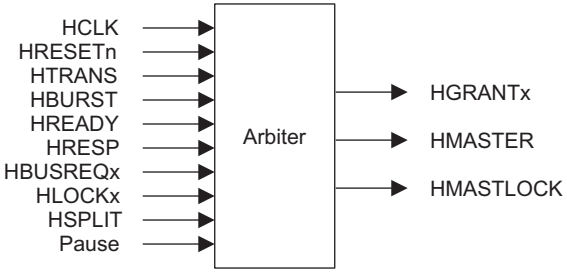


Figure 4-6 Arbiter block diagram

The default arbiter included in the EASY design can support up to four bus masters, although only two are used. It is expandable up to a maximum of fifteen bus masters, excluding the default master.

The main sections of this module are:

- the split transfer control logic
- the locked transfer control logic
- the arbitration scheme
- the grant output signal generation.

4.2.1 Signal descriptions

Table 4-2 contains a list of signals used by the arbiter.

Table 4-2 Signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW and is used to reset the system and the bus.
HTRANS[1:0]	Transfer type	Input	Indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL or BUSY.
HBURST[2:0]	Burst type	Input	Indicates if the transfer forms part of a burst. Both 4-beat and 8-beat bursts are supported and the burst can be either incrementing or wrapping.
HREADY	Transfer done	Input	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HRESP	Transfer response	Input	The transfer response provides additional information on the status of a transfer. This input is used to detect Split or Retry transfers.
HBUSREQx	Bus request	Input	A signal from the bus master to the bus arbiter which indicates that the master requires the bus.
HLOCKarm	Locked transfers	Input	When HIGH this signal indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is LOW.
HLOCKx	Locked transfers	Input	Lock signal from the bus master.
HSPLITx[15:0]	Split completion request	Input	The 16-bit split bus is used by a split-capable slave to indicate to the arbiter which bus masters should be allowed to re-attempt a split transaction. Each bit of this split bus corresponds to a single bus master.
Pause	Pause mode	Input	This signal allows the processor system to enter a low-power, *wait for interrupt* state, when the system does not require the processors to be active.

Table 4-2 Signal descriptions (continued)

Signal	Type	Direction	Description
HGRANTx	Bus grant	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 **HGRANTx** are HIGH.
HMASTER[3:0]	Master number	Output	These signals from the arbiter indicate which bus master is currently performing a transfer. This is used by slaves which support split transfers to determine which bus master is attempting an access.
HMASTLOCK	Locked sequence	Output	Indicates that the current master is performing a locked sequence of transfers. This signal has the same timing as the **HMASTER** signals.

4.2.2 Function and operation of arbiter module

The arbiter is used to ensure that, at any point in time, only one master has access to the bus. The arbiter performs this function by observing all of the bus master requests to use the bus, and deciding which is currently the highest priority. The arbiter also receives requests from slaves that wish to complete split transfers, which are used to modify the priority of the master request inputs.

The arbiter has a standard interface to all bus masters and split-capable slaves in the system.

A bus master may request the bus during any cycle by setting its **HBUSREQ** output HIGH. This is then sampled by the arbiter on the rising edge of the clock, and passed through the priority algorithm to decide which master will have access to the bus during the next cycle. The **HGRANT** outputs then change to indicate which master currently is granted control of the bus.

The **HLOCK** signals may be used to ensure that, during an indivisible transfer, the current grant outputs do not change. **HLOCK** must be asserted at least one cycle before the locked transfer to prevent the arbiter from changing the grant signals.

The following arbitration priorities (from highest to lowest) are implemented in the default system:

- TIC (highest)
- bus master 3
- bus master 4
- ARM processor (lowest and default).

During split transfers the above priorities will be changed to allow other masters access to the bus. When a split transfer is indicated by a split-capable slave that cannot complete the current transfer immediately, the bus request input for the current master is masked out. This has the effect of changing the priorities of the bus request inputs, allowing a lower priority master to be granted control of the bus. When the slave is ready to complete the split transfer, it drives the **HSPLIT** bus with the number of the bus master that was performing the transfer. This number was sampled by the slave, from the **HMASTER** arbiter output, when it started the split transfer. The arbiter then uses this input to unmask the bus request of the master, allowing it to be regranted the bus so that the transfer can complete.

During reset, and when no other masters are requesting control of the bus, the ARM core is selected as the currently granted master. This minimizes the delay required for the core to perform a transfer on the bus, as it does not have to wait to be granted control of the bus before it can start a new transfer.

The system also requires a default master, which is selected when no masters are granted control of the bus, for example, when all system bus masters are waiting for split transfers to complete. The default master performs IDLE transfers while it is granted control of the bus.

The default master is also selected during pause mode when the **Pause** input is set HIGH, indicating that the system has entered a low-power mode, and no transfers will be started on the bus.

The bus grant outputs may change while **HREADY** is LOW, but the newly granted master may only drive the bus when the current transfer has completed. This requires that bus masters only drive the bus after they detect that both their **HGRANT** and **HREADY** inputs are set HIGH.

4.2.3 System description

This section describes how the HDL code for the system arbiter is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This part should be read in conjunction with the HDL code.

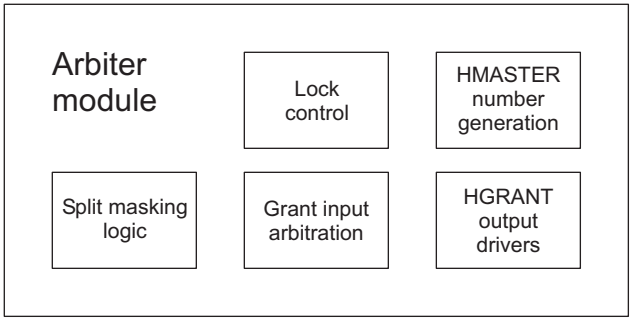


Figure 4-7 Arbiter module block diagram

The arbiter comprises:

- split grant masking logic
- locked transfer control
- arbitration scheme logic
- **HGRANT** output drivers
- **HMASTER** output generation.

All registers used in the system are clocked from the rising edge of the system clock **HCLK**, and use the asynchronous reset **HRESETn**.

Figure 4-8 on page 4-19 shows the arbiter HDL file.

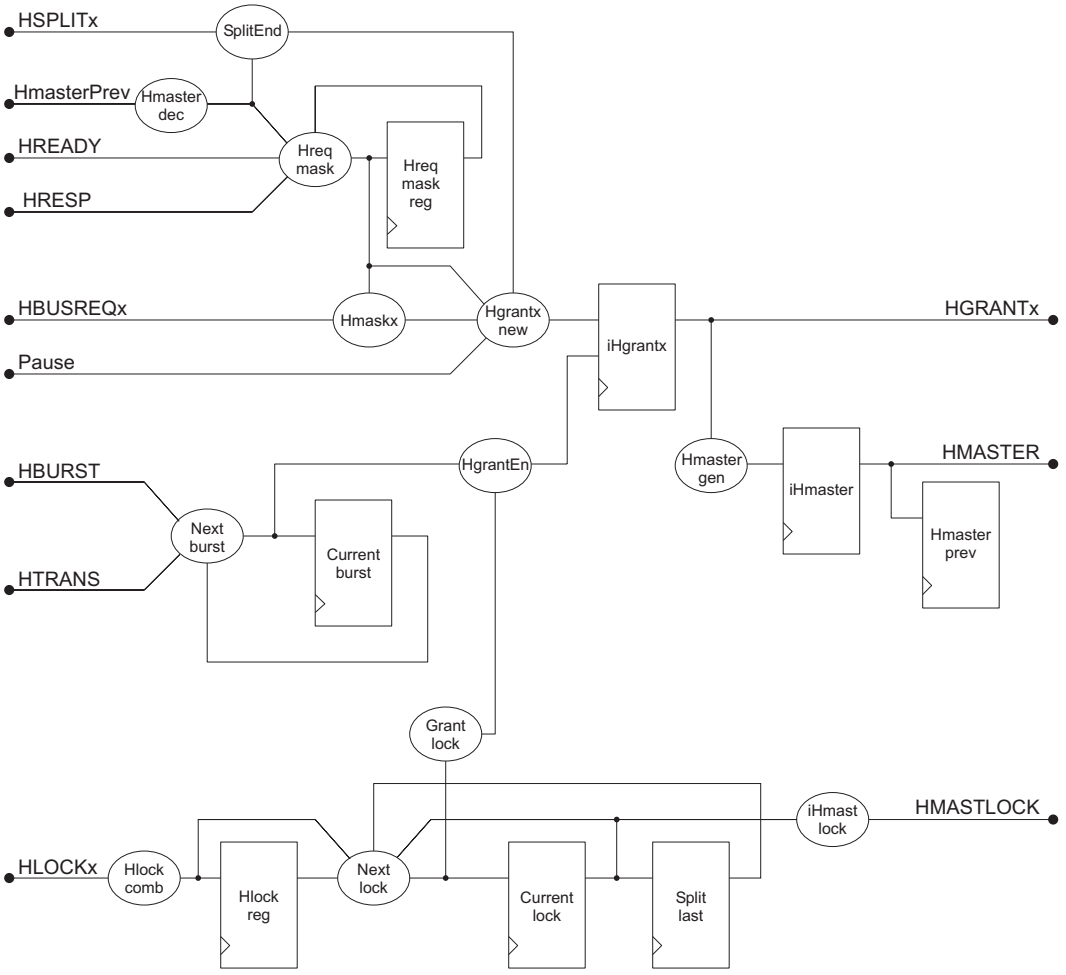


Figure 4-8 Arbiter module system diagram

The main sections in this module are explained in:

- *Split grant masking* on page 4-20
- *Locked state machine* on page 4-20
- *Arbitration scheme* on page 4-24
- *Output registers* on page 4-24.

Split grant masking

The split grant masking logic is comprised of a set of registers which hold the current mask value, and a combinational logic that is used to control the setting of the registers.

When a split transfer is detected, a single bit in the mask register is cleared, which blocks the bus request input of the split master from reaching the arbitration logic. This allows lower priority masters access, if they are requesting use of the bus. A decoded 16-bit version of the previous 4-bit **HMASTER** output is used to determine which bit of the mask to clear, when a split response is detected.

When a split transfer is resumed, the 16-bit **HSPLIT** input is used to set the bit in the mask register, allowing the bus request line to be used to generate the bus grant outputs. The split master will then be regranted the bus as normal and will be able to complete the split transfer.

The encoding of the **HSPLIT** input allows multiple bits of the grant mask to be set at the same time.

The grant mask value is ANDed with the **HGRANTx** inputs to generate the internal **Hmaskx** signals, which are then fed to the arbitration logic.

Locked state machine

A state machine is used to control the operation of the arbiter during a locked transfer.

First, the **HLOCKx** inputs from the system bus masters are masked with the current grant outputs, generating an internal lock signal. This shows when the currently granted master is requesting a locked transfer, and ignores the lock status of the other system bus masters. This signal is then passed to the locked state machine, which is shown in Figure 4-9 on page 4-21.

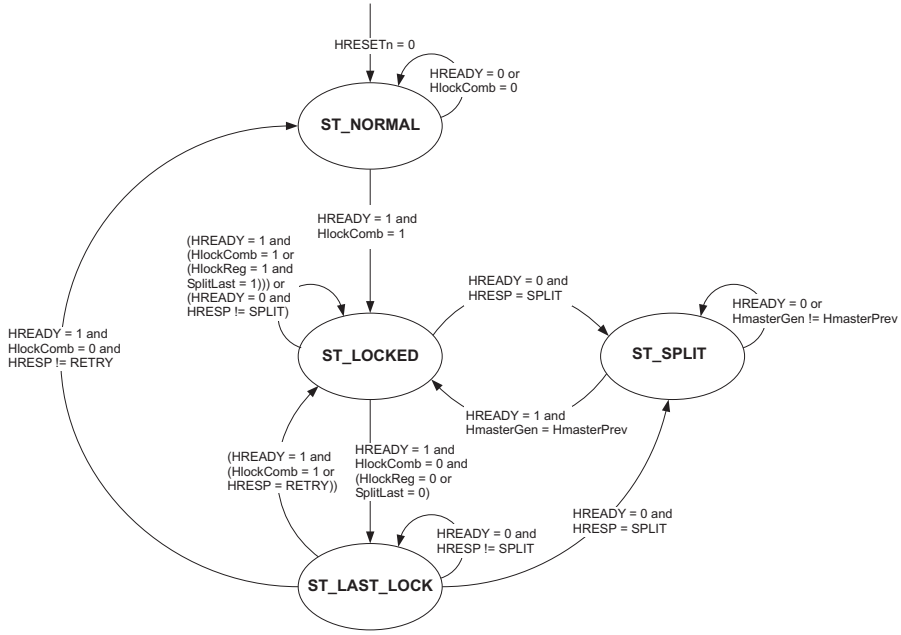


Figure 4-9 Locked state machine

The four states are described in:

- *ST_NORMAL* on page 4-22
- *ST_LOCKED* on page 4-22
- *ST_LAST_LOCK* on page 4-23
- *ST_SPLIT* on page 4-23.

- ST_NORMAL

  During this state, the arbiter is operating normally and no locked transfers are being performed. The **HMASTLOCK** output is set LOW.

  The ST_NORMAL state is entered from:

  — reset, when the system is initialized

  — ST_LAST_LOCK, when the data phase of the last locked transfer has completed

  — ST_NORMAL, when no locked transfers are being performed.

  The next state is:

  — ST_LOCKED, when the currently granted master sets its **HLOCK** output HIGH

  — ST_NORMAL, when no locked transfers are being performed.

- ST_LOCKED

  During this state, the currently performed transfer is locked, and the grant outputs will not change. The **HMASTLOCK** output is set HIGH to indicate that the current transfer is locked.

  The ST_LOCKED state is entered from:

  — ST_NORMAL, when the currently granted master sets its **HLOCK** output HIGH

  — ST_SPLIT, when a split locked transfer has been restarted

  — ST_LOCKED, when there is another locked transfer to perform

  — ST_LAST_LOCK, during the second locked transfer in a locked-unlocked-locked sequence, or when the last locked transfer (**HLOCK** is LOW) has received a RETRY response.

  The next state is:

  — ST_SPLIT, when the current locked transfer receives a SPLIT response

  — ST_LAST_LOCK, when the currently granted master sets its **HLOCK** output LOW

  — ST_LOCKED, when there is another locked transfer to perform.

- ST_LAST_LOCK

  This state is used to add an extra locked transfer after the currently granted master has set its **HLOCK** output LOW, ensuring that the grant outputs do not change until the data phase of the last locked transfer has completed, even if it has received a SPLIT or RETRY response. The **HMASTLOCK** output is set LOW.

  The ST_LAST_LOCK state is always entered from ST_LOCKED when the currently granted master sets its **HLOCK** output LOW.

  The next state is:

  — ST_NORMAL, when the data phase of the last locked transfer has completed

  — ST_LOCKED, during the second locked transfer in a locked-unlocked-locked sequence, or when the last locked transfer (**HLOCK** is LOW) has received a RETRY response

  — ST_SPLIT, when the last locked transfer receives a SPLIT response

  — ST_LAST_LOCK, when the bus is waited and the last locked transfer has not received a SPLIT response.

- ST_SPLIT

  This state is used when a locked transfer receives a SPLIT response. As the transfer is locked, no new masters may be granted control of the bus, but as it has been split the currently granted master may not have control of the bus until the slave indicates that it is ready to resume the transfer. So, the default master is granted while in this state. The **HMASTLOCK** output is set LOW.

  The ST_SPLIT state is entered from:

  — ST_LOCKED, when a locked transfer receives a SPLIT response

  — ST_LAST_LOCK, when the last locked transfer receives a SPLIT response

  — ST_SPLIT, when the slave is not ready to resume the transfer.

  The next state is:

  — ST_LOCKED, when the split locked transfer has been restarted

  — ST_SPLIT, when the slave is not ready to resume the transfer.

Arbitration scheme

This section of the code defines the arbitration scheme that is used by the system, the default being a priority-based system. The order that the inputs are checked in the `if` statement defines the priority order of the system. This section should be modified if the arbitration scheme of the system is to be changed from the default. The **HgrantxNew** signals are generated by this section, which are then registered to generate the **HGRANT** outputs.

Output registers

The **HGRANT** registers sample the outputs from the arbitration scheme logic when the **HgrantEn** enable signal is set HIGH. This is used to control the loading of the grant output registers during locked, split and burst transfers.

The grant outputs do not change:

- during locked transfers
- during the first *n*-1 transfers of a fixed length burst of *n* transfers.

The grant outputs only change:

- when the system is not performing a locked transfer
- when a locked transfer receives a SPLIT response, to allow the default master to be selected
- during a locked split transfer, when the slave indicates that the transfer may resume, to allow the locked master to be selected
- during the last transfer of a fixed-length burst.

HMASTER is generated from the current **HGRANT** outputs, encoding the 16 possible master grant signals into a 4-bit number. This is registered, and is valid during the address phase of the transfer it relates to. A previous value is also generated, which is used to control the operation of the system during split transfers, and is valid during the data phase of the transfer.

HMASTLOCK is directly generated from the current state of the locked state machine, and is valid during the address phase of the locked transfer.

4.3 Decoder

The system decoder is used to decode the address bus and generate select lines to each of the system bus slaves, indicating that a read or write access to that slave is required. Figure 4-10 shows the decoder module interface block diagram.

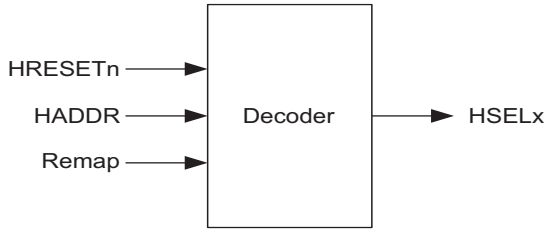


Figure 4-10 Decoder module interface diagram

This module only contains a combinatorial decode of the system address bus, using the **Remap** input to control the selection of the internal and external memory.

4.3.1 Signal description

Table 4-3 shows the signal descriptions for the decoder module

Table 4-3 Decoder module signal descriptions

Signal	Type	Direction	Description
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HADDR[31:0]	Address bus	Input	The 32-bit system address bus.
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 `0x0000 0000` which normally contains the system startup code. In normal operation this signal is HIGH, allowing use of the internal memory.
HSELx	Slave select	Output	Slave select to each system bus slave.

.

4.3.2 System memory map

The decoder controls the memory map of the system, and generates a slave select signal for each memory region.

The **Remap** signal is used to provide a different memory map at reset, when ROM is required at address `0x0000 0000`, and during normal operation, when internal RAM may be used at address `0x0000 0000`.

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

Figure 4-11 shows both the normal and reset memory maps.

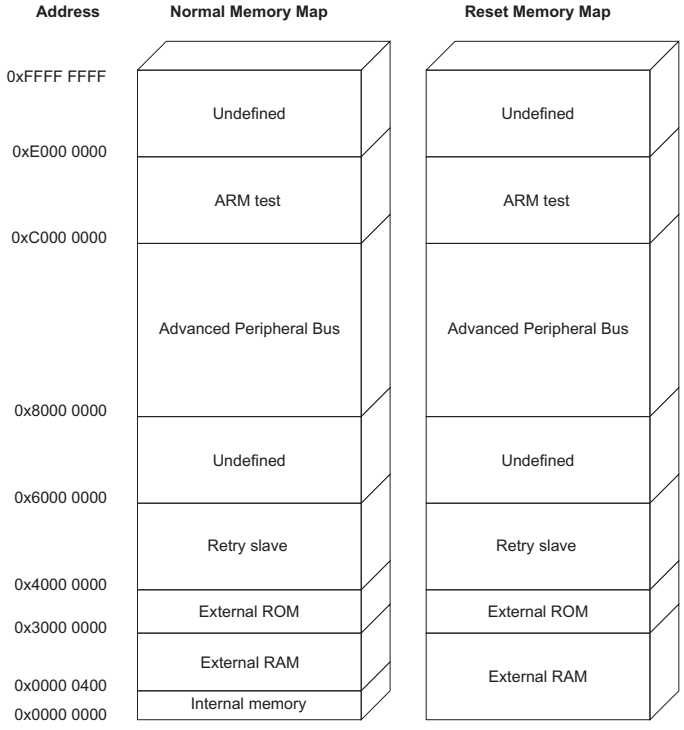


Figure 4-11 System memory map

4.3.3 Function and operation of the decoder module

The decoder continuously performs a combinatorial decode of the system address bus, updating the slave select outputs whenever the address or system **Remap** inputs change value.

The default slave is used to control the operation of the system when a transfer is made to an undefined area of memory, and is selected when an invalid address is generated.

4.3.4 System description

The following paragraphs give a description of how the HDL code for the decoder is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the inputs, and outputs used in the module. This part should be read together with the HDL code.

Figure 4-12 shows the decoder module block diagram.

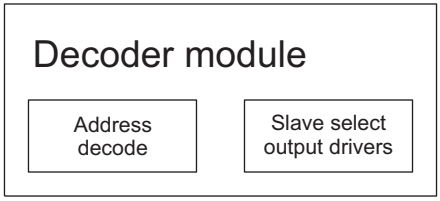


Figure 4-12 Decoder module block diagram

The decoder comprises a simple block of combinational logic, which is used to decode the address and system remap inputs to directly generate the slave select outputs.

Figure 4-13 shows the decoder HDL file.

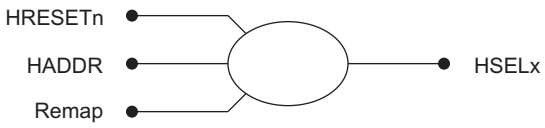


Figure 4-13 Decoder module system diagram

The whole of the decode logic is contained in one `if` statement. During reset, the default slave is selected, and at all other times, the **HADDR** and **Remap** inputs are decoded and used to generate the **HSELx** outputs.

The minimum number of address bits needed to select a slave are used, keeping the combinational logic as small as possible.

This section of code is used to define the memory map for the whole system, and if modules are added, removed, or moved to new locations, the code must be modified to match these system changes, ensuring that the correct slave is selected for each address used.

4.4 Default slave

The default slave is used to respond 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 4-14 shows the default slave module interface diagram.

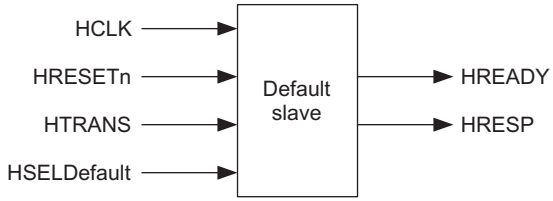


Figure 4-14 Default slave module interface diagram

This module contains a standard AHB slave response interface, using the **HREADY** and **HRESP** outputs to respond to transfers.

4.4.1 Signal descriptions

Table 4-4 shows the signal descriptions for the default slave module

Table 4-4 Default slave module signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HTRANS[1:0]	Transfer type	Input	Indicated the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.

Table 4-4 Default slave module signal descriptions (continued)

Signal	Type	Direction	Description
HSEL	Default slave select	Input	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.
HREADYout	Transfer done	Output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal is only driven LOW to generate a two cycle error response.
HRESP[1:0]	Transfer response	Output	The transfer response provides additional information on the status of a transfer. This module will only generate the OKAY and ERROR responses.

.

4.4.2 Function and operation of module

The default slave only responds to transfers when it is selected by the decoder with the **HSEL** input when an undefined region of memory is accessed. The response generated depends on the type of transfer that is performed.

If an IDLE or BUSY transfer is performed, then the default slave must provide a zero wait OKAY response as the master will not expect to receive any data back from these transfers.

If a NONSEQUENTIAL or SEQUENTIAL transfer is performed, then an ERROR response is generated, as there is nothing at the current location that can be written to or read from. The standard two-cycle ERROR response is provided with one wait state.

4.4.3 System description

This section describes how the HDL code for the default slave is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This should be read together with the HDL code.

Figure 4-15 shows the default slave module block diagram.

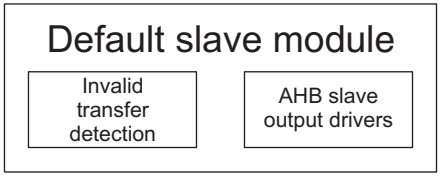


Figure 4-15 Default slave module block diagram

The default slave comprises the invalid transfer detection logic and two simple sets of combinational logic and registers, which are used to generate the **HREADY** and **HRESP** outputs.

Figure 4-16 shows the decoder HDL file.

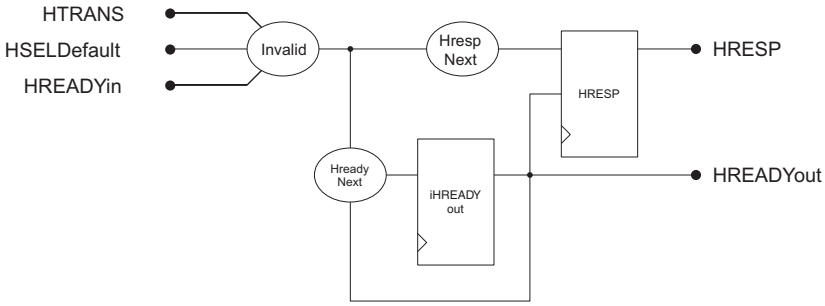


Figure 4-16 Default slave module system diagram

The internal signal Invalid is set HIGH during the final cycle of the address phase of an invalid transfer (when **HREADYin** is set HIGH, a NONSEQUENTIAL or SEQUENTIAL transfer is performed, and the default slave is selected), and is set LOW at all other times.

This signal is then passed to the response generation logic, which is split into two sections for the **HREADYout** and **HRESP** outputs. This logic generates the response values for the output registers. **HREADYout** is set LOW during the first cycle of the data phase, as is required for the two cycle ERROR response, and **HRESP** is set to ERROR for the two-cycles of the data phase.

At all other times, the default slave generates a zero wait OKAY response.

4.5 Master to slave multiplexor

The master to slave multiplexor is used to connect all of the system bus masters to the bus slaves, using the current **HMASTER** number to select the bus master outputs to use. It is also used to generate the default master outputs when no other masters are selected. Figure 4-17 shows an interface diagram of the master to slave multiplexor module.

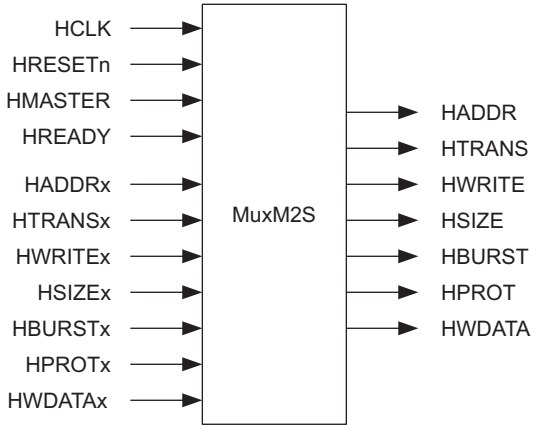


Figure 4-17 Master to slave multiplexor module interface diagram

The module has the address, control and data outputs of all system bus masters as its inputs, and has a single set of these signals as its outputs, which are connected to the inputs of all system slaves. When masters are added to, or removed from the system, the input connections to this module must be altered to account for the changes.

4.5.1 Signal descriptions

Table 4-5 lists signal descriptions for the master to slave multiplexor module.

Table 4-5 Master to slave multiplexor signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HMASTER[3:0]	Master number	Input	These signals from the arbiter indicate which bus master is currently performing a transfer, and is used by slaves which support split transfers to determine which master is attempting an access.
HREADY	Transfer done	Input	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HADDRx[31:0] **HADDR[31:0]**	Address bus	Input/ output	The 32-bit system address bus.
HTRANSx[1:0] **HTRANS[1:0]**	Transfer type	Input/ output	These signals indicate the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITEx **HWRITE**	Transfer direction	Input/ output	When HIGH this signal indicates a write transfer, and when LOW, a read transfer.
HSIZEx[2:0] **HSIZE[2:0]**	Transfer size	Input/ output	These signals indicate the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit). The protocol allows for larger transfer sizes up to a maximum of 1024 bits.
HBURSTx[2:0] **HBURST[2:0]**	Burst type	Input/ output	These signals indicate if the transfer forms part of a burst. Both four beat and eight beat bursts are supported and the burst may be either incrementing or wrapping.
HPROTx[3:0] **HPROT[3:0]**	Protection control	Input/ output	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection.
HWDATAx[31:0] **HWDATA[31:0]**	Write data bus	Input/ output	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended, however this can easily be extended to allow for higher bandwidth operation.

4.5.2 Function and operation of module

The master to slave multiplexor controls the routing of address, control and data signals from the system bus masters to the bus slaves. The arbiter determines which master currently has control of the bus, and the multiplexor is used to connect the outputs of the selected master to the inputs of the bus slaves.

The address and control signals are switched during the address phase of a transfer using the **HMASTER** arbiter output.

The write data signals are switched during the data phase of a transfer using a registered version of **HMASTER**.

When no masters are selected, the default master signals are selected and the module drives all outputs LOW, performing IDLE transfers until another master is granted control of the bus.

4.5.3 System description

This section describes how the HDL code for the master to slave multiplexor is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This should be read together with the HDL code.

Figure 4-18 shows the master to slave module block diagram.

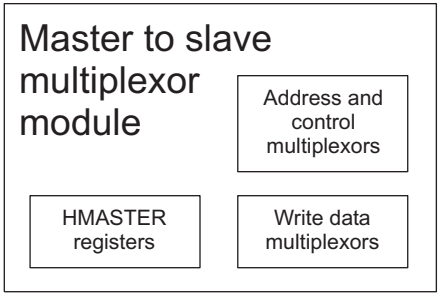


Figure 4-18 Master to slave multiplexor module block diagram

The master to slave multiplexor module comprises a set of multiplexors for each address, control and data output from the system bus masters. A set of registers is also used to hold the previous value of the **HMASTER** input.

Figure 4-19 shows the master to slave multiplexor HDL file.

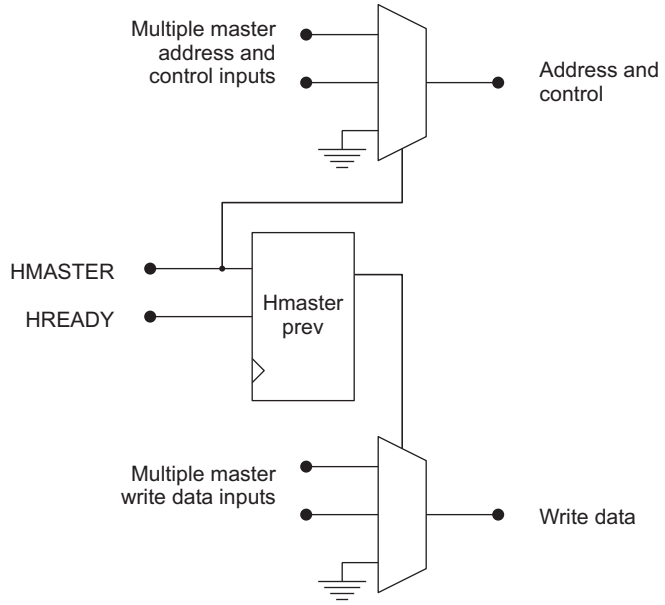


Figure 4-19 Master to slave multiplexor module system diagram

The multiplexor for each master signal has an input for each system bus master, and a ground connection for the default master signal values. The master number is decoded, and used to select the correct input signal.

The multiplexors are constructed using case statements, ensuring that there is no priority to the master selection logic.

An **HREADY** enabled register is used to hold the previous value of **HMASTER**, because the **HWDATA** master outputs are always running one cycle behind the other address and control signals, due to the pipelined bus. The enable is used to ensure that the value is only updated when the previous transfer has completed.

4.6 Slave to master multiplexor

The slave to master multiplexor is used to connect the read data and response signals of the system bus slaves to the bus masters, using the current decoder **HSELx** outputs to select the bus slave outputs to use. Figure 4-20 shows the slave to master multiplexor module.

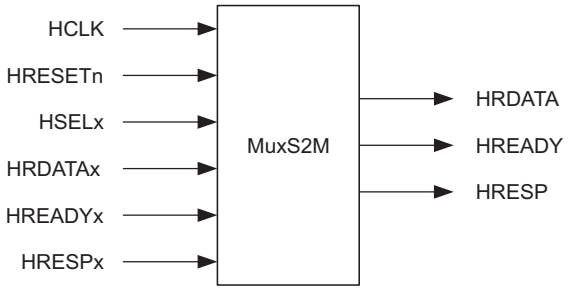


Figure 4-20 Slave to master multiplexor module interface diagram

This module has the read data and response outputs of all system bus slaves as its inputs, and has a single set of these signals as its outputs, which are connected to the inputs of all system masters. When slaves are added to the system or removed, the input connections to this module must be altered to account for the changes.

4.6.1 Signal descriptions

Table 4-6 shows the signal descriptions for the slave to master multiplexor module.

Table 4-6 Slave to master multiplexor signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HSELx	Slave select	Input	Each AHB slave has its own slave select signal and this signal indicates that the current transfer is intended for the selected slave.

Table 4-6 Slave to master multiplexor signal descriptions (continued)

Signal	Type	Direction	Description
HRDATAx[31:0] **HRDATA[31:0]**	Read data bus	Input/ output	The read data bus is used to transfer data from bus slaves to the bus master during read operations.
HREADYx **HREADY**	Transfer done	Input/ output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HRESPx[1:0] **HRESP[1:0]**	Transfer response	Input/ output	The transfer response provides additional information on the status of a transfer.

4.6.2 Function and operation of module

The slave to master multiplexor controls the routing of read data and response signals from the system bus slaves to the bus masters. The decoder determines which is the currently selected slave, and the multiplexor is used to connect the outputs of the selected slave to the inputs of the bus masters.

The read data and response signals are switched during the data phase of a transfer, so a registered version of the slave select signals is used.

The default slave inputs are used when no other slaves are selected.

4.6.3 System description

This section describes how the HDL code for the slave to master multiplexor is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This part should be read together with the HDL code.

Figure 4-21 on page 4-38 shows the slave to master module block diagram.

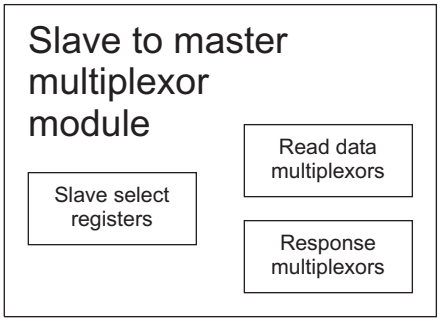


Figure 4-21 Slave to master multiplexor module block diagram

The slave to master multiplexor module comprises a set of registers to store the previous slave select values, and a set of multiplexors for the read data and slave response signals.

Figure 4-22 shows the slave to master multiplexor HDL file.

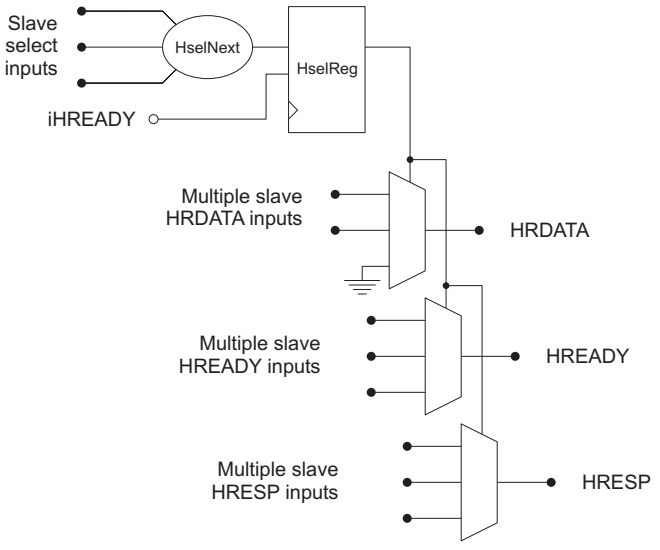


Figure 4-22 Slave to master multiplexor module system diagram

To allow the use of case statements for the multiplexors, the **HSEL** slave select inputs are combined to create a multi-bit bus signal. This bus is then registered, and used as the select control on the three multiplexors, one each for the read data and two response signals. The select register is enabled with the internal **HREADY** signal, ensuring that the outputs only change when the previous transfer has finished.

As the default slave does not generate any read data, one input to the **HRDATA** multiplexor is tied LOW, so that when the default slave is selected, no read data appears on **HRDATA**.

4.7 Reset controller

The reset controller is used to generate the system reset signal from an external reset input as shown in Figure 4-23.

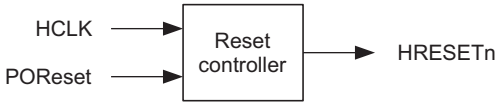


Figure 4-23 Reset controller module interface diagram

This module is based around a state machine, which is used to detect the external reset being asserted, and is used to generate the system reset output.

4.7.1 Signal descriptions

Table 4-7 shows the signal descriptions for the reset controller.

Table 4-7 Reset controller signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
POReset	Power-on reset	Input	Power-on reset input. This active LOW signal causes a cold reset when LOW. May be asserted asynchronously to **HCLK**.
HRESETn	Reset	Output	The bus reset signal is active LOW, and is used to reset the system and the bus.

The source of the **POReset** signal is implementation-dependent.

4.7.2 Function and operation of module

HRESETn is asserted LOW, and is used to indicate a reset condition where all bus and system states should be initialized. This signal is suitable as an asynchronous clear into state machine flip-flops, and for resetting any peripheral registers that require initialization.

During reset, the arbiter grants the bus to the default reset bus master, and the decoder selects the default slave.

Assertion (the falling edge) of **HRESETn** is asynchronous to **HCLK**. De-assertion (the rising edge) of **HRESETn** is synchronous to the rising edge of **HCLK**. **HRESETn** is only asserted during a power-on reset condition, caused by the assertion of the **POReset** signal. The **POReset** input is an asynchronous input, so a synchronizing register is required to eliminate propagation of metastable values. Figure 4-24 shows the operation of the **HRESETn** signal with respect to an example **POReset** input signal and the system clock.

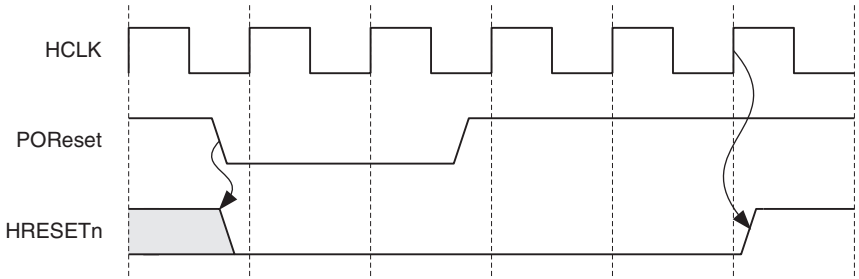


Figure 4-24 Reset signal timing

The reset controller contains a state machine running from the rising edge of **HCLK**. The **HRESETn** signal directly reflects a single bit of the current state, minimizing the combinational logic applied to the reset output.

Figure 4-25 shows the state machine for the reset controller.

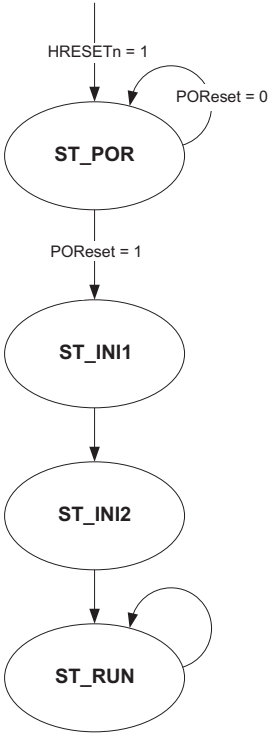


Figure 4-25 State machine for reset controller

The four states are described in:

- *ST_POR* on page 4-43
- *ST_INI1* on page 4-43
- *ST_INI2* on page 4-43
- *ST_RUN* on page 4-43.

ST_POR

During this state, the system is initialized when the reset line is asserted. This state should be preserved by a power on reset cell or controller, until the system bus clock is running and stable, and the system power supply has reached its correct operating voltage (within its allowed limits).

The ST_POR state is entered from:

- reset, when the external reset input is first asserted LOW
- ST_POR when the external reset input is still asserted and the system clock is running.

The next state is:

- ST_INI1 when the external reset input is de-asserted
- ST_POR when the external reset input is still asserted and the system clock is running.

If there is a clock valid signal in the system, this should be used to prevent the ST_POR state from being exited until the clock is valid.

ST_INI1

This state is used to hold the **HRESETn** output LOW for an extra cycle after the external reset is de-asserted.

This state is always entered from ST_POR on the first rising edge of the clock that the external reset is HIGH.

The next state is always ST_INI2.

ST_INI2

This state is used in the same manner as ST_INI1.

This state is always entered from ST_INI1.

The next state is always ST_RUN.

ST_RUN

This state is used during normal system operation when the **HRESETn** output is set HIGH.

This state is held until the external reset is re-asserted.

The default reset controller implementation asserts **HRESETn** for two cycles after the external reset is de-asserted, but this may be altered by adding extra ST_INI states to the state machine, so that it takes more cycles to reach the final ST_RUN state.

4.7.3 System description

The following paragraphs give a description of how the HDL code for the reset controller is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This part should be read together with the HDL code.

Figure 4-26 shows the reset controller module block diagram.

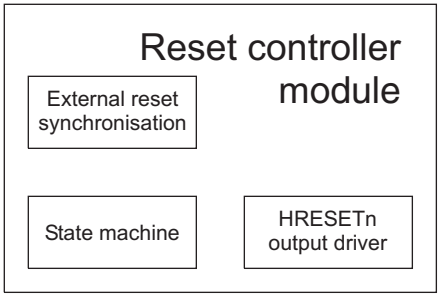


Figure 4-26 Reset controller module block diagram

The reset controller is comprised of a register used to synchronize the external reset input, and a state machine used to control the generation of the system reset output.

All registers used in the system are clocked from the rising edge of the system clock **HCLK**.

Figure 4-27 shows the reset controller HDL file.

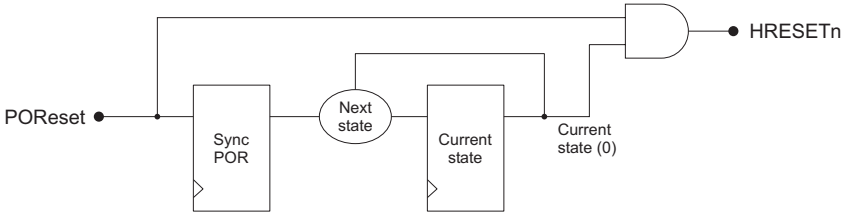


Figure 4-27 Reset controller module system diagram

The main sections in this module are explained in the following paragraphs:

- *Asynchronous reset input synchronization*
- *Reset state machine*
- *Reset output generation*.

Asynchronous reset input synchronization

The asynchronous external reset is first passed through a rising-edge-triggered register. This is to avoid metastability, due to the arrival time of the input relative to the system clock when used in the state machine.

Reset state machine

The state machine shown in Figure 4-25 on page 4-42 is used to control the generation of the system reset output, based on the status of the synchronized external reset input and the system clock.

The number of cycles the module holds **HRESETn** asserted after the de-assertion of the external reset may be changed by altering the number of initialization states between the first and last states.

Reset output generation

The reset output is generated directly from bit 0 of the state machine registers, gated with the external reset input. This allows asynchronous assertion of the reset output when the external reset input is set LOW and the system clock is not running, but ensures that de-assertion is synchronous to the rising edge of the clock.

4.8 Retry slave

The retry slave is a rudimentary module that is used to demonstrate how to build an AHB slave. The example contains very little functionality and consists of four 32-bit wide registers. The slave generates various logic functions of these registers, which can be read from different locations.

One of the most important features of the slave is that the response that it gives can be varied according to the high order address lines. Figure 4-28 shows the retry slave block diagram.

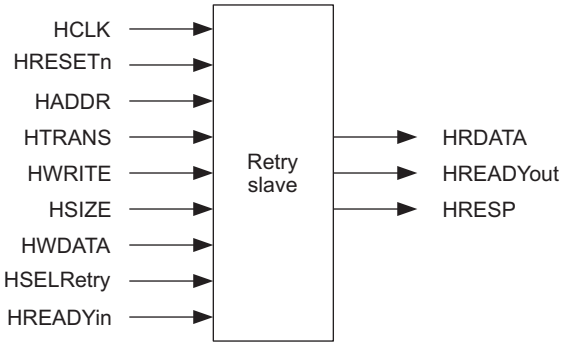


Figure 4-28 Retry slave block diagram

The main sections of this module are:

- the AHB slave bus interface
- the internal read/write registers
- the wait state and retry cycle generation logic
- the read data value generation.

4.8.1 Signal descriptions

Table 4-8 contains a list of signals used by the retry slave.

Table 4-8 Signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HADDR[31:0]	Address bus	Input	The 32-bit system address bus.
HTRANS[1:0]	Transfer type	Input	Indicated the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE	Transfer direction	Input	When HIGH this signal indicates a write transfer, and when LOW, a read transfer.
HWDATA[31:0]	Write data bus	Input	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended, however, this may easily be extended to allow for higher bandwidth operation.
HSELRetry	Slave select	Input	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 a combinatorial decode of the address bus.
HRDATA[31:0]	Read data bus	Output	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended, however this may easily be extended to allow for higher bandwidth operation.
HREADYin **HREADYout**	Transfer done	Input/output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HRESP[1:0]	Transfer response	Output	The transfer response provides additional information on the status of a transfer. This module only generates OKAY and RETRY responses.

4.8.2 Function and operation of module

This example module contains four 32-bit wide registers, which can be accessed using byte, halfword or word, read or write transfers. Extra read only locations are provided that generate logical combinations of these four registers. The module memory map in Table 4-9 shows the logical functions that the slave can provide, and the addresses at which the functions and four read/write registers are accessed.

Table 4-9 Memory map of the example AHB retry slave

Address	Read location	Write location
0x00	R0	R0
0x04	R1	R1
0x08	R2	R2
0x0C	R3	R3
0x10	Not R0	-
0x14	R0 and R1	-
0x18	R1 or R2	-
0x1C	R2 xor R3	-
0x20	R0 and R1 and R2 and R3	-
0x24	R0 or R1 or R2 or R3	-
0x28	R0 xor R1 xor R2 xor R3	-

All addresses shown in the memory map are offsets from the module base address. In the default system the retry slave module occupies memory locations 0x4000 0000 to 0x5FFF FFFF.

When any of the memory locations are accessed, the high order address lines are used to determine the response that the slave will provide, inserting wait states or retry cycles.

The address lines that are used are:

- **HADDR[11:8]**, number of wait states to be inserted
- **HADDR[13:12]**, number of times a retry response will be generated.

The number of wait states inserted for each read or write module access can be varied from 0 to 15, and the number of times the slave provides a retry response can be varied from 0 to 3.

When the slave is programmed to provide a retry response, the number of wait states to insert must be set to a value greater than zero, as all retry responses require two cycles, with a wait state inserted during the first cycle.

4.8.3 System description

The following paragraphs give a description of how the HDL code for the example retry slave is set out. A basic block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs and outputs used in the system. This part should be read together with the HDL code.

Figure 4-29 shows a basic block diagram of the retry slave module system.

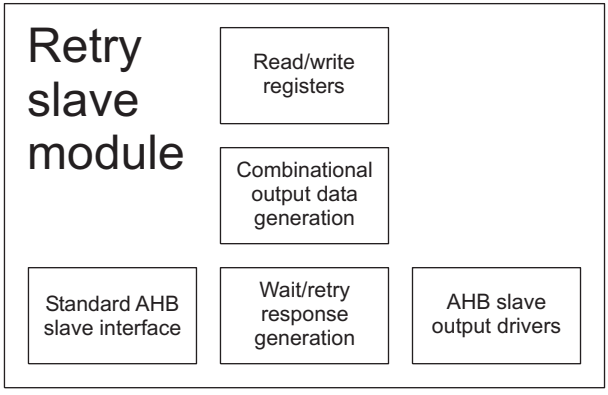


Figure 4-29 Retry slave module block diagram

The retry slave comprises a set of read/write registers, and programmable wait/retry generation logic.

All registers used in the system are clocked from the rising edge of the system clock **HCLK**, and use the asynchronous reset **HRESETn**.

Figure 4-30 shows the retry slave HDL file.

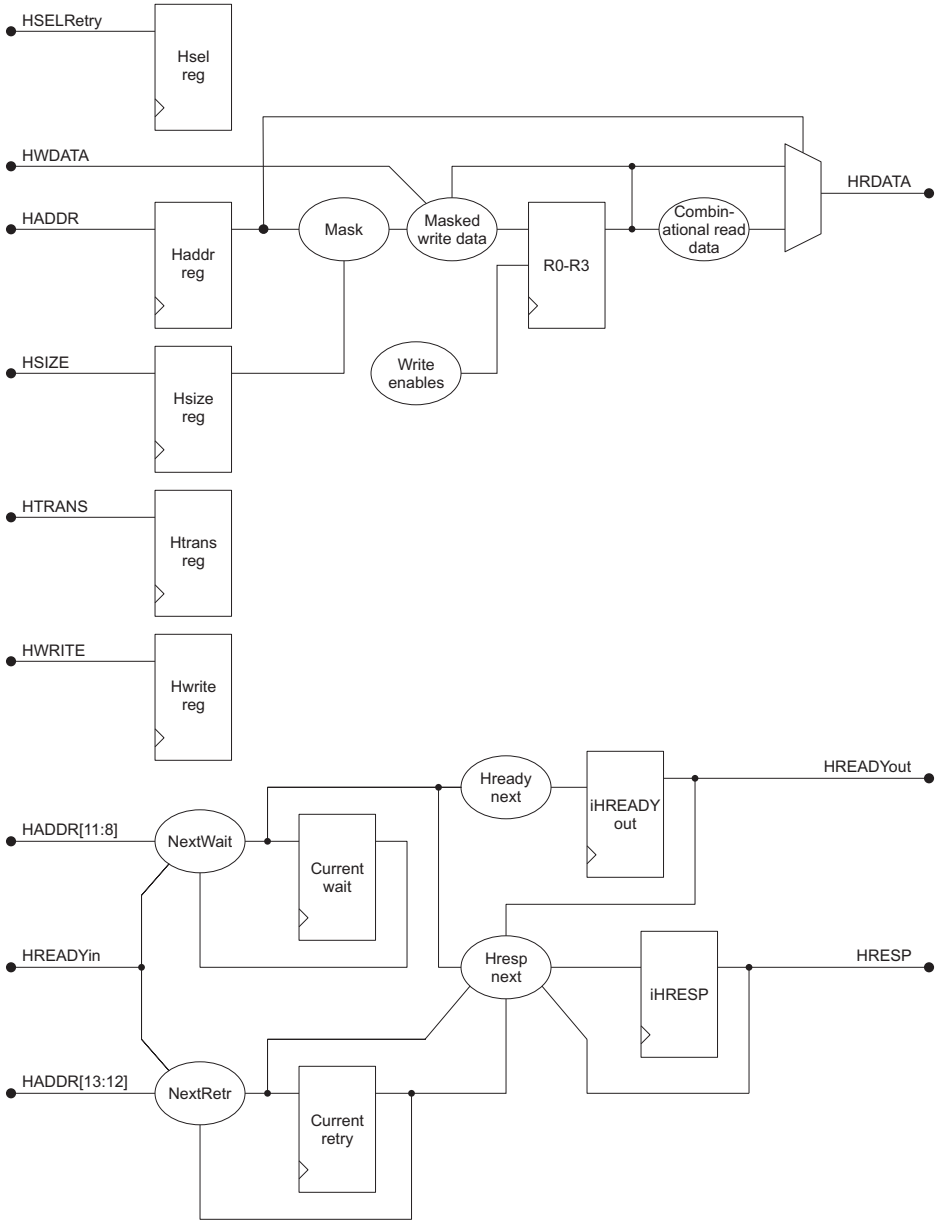


Figure 4-30 Retry slave module system diagram

The main sections in this module are explained in the following paragraphs:

- *AHB slave bus interface*
- *Write data mask*
- *Read/write registers*
- *Response generation logic* on page 4-52
- *Read data generation* on page 4-52.

AHB slave bus interface

This module uses the standard AHB slave bus interface, which comprises the valid transfer detection logic, and the address and control registers, which are used to store the information from the address phase of the transfer for use in the data phase.

Write data mask

The amount of data written to the four internal registers depends on the transfer size setting. The mask is used to control which bytes of data are written to the 32-bit registers, and which bytes are left unchanged. A single mask value is used to allow one set of size decoding logic to be used for all registers in the module, rather than having a set of decoding logic for each register.

The bytes of data that will change are set LOW in the mask, and all other bits are set HIGH.

Read/write registers

Four 32-bit registers are used to store user data, all initializing to zero. They are only enabled when addressed during a write transfer, and when any wait states or retry cycles have ended. The data mask is used to control writes of byte, halfword and word, by masking out the bits of the current write data that are not needed, and ORing it with a masked version of the current register data. This ensures that only the required bytes of the read data are used, and the unchanged register bytes are reloaded with the previous register value.

Response generation logic

This logic is used to control the generation of wait states and retry cycles.

Wait states are inserted when the address of the current transfer has a nonzero value in bits [11:8]. This value, from zero to fifteen, is loaded into the CurrentWait register, and then decremented each clock cycle until zero is reached. This counter value is used to hold the **HREADYout** output LOW until zero is reached, when **HREADYout** is set HIGH and the transfer can complete.

Retry cycles are inserted when the address of the current transfer has a nonzero value in bits [13:12] and [11:8], as all retry cycles require at least one wait state. This value is loaded into the CurrentRetry register, and is decremented each time the transfer is retried until zero is reached. The input to the iHRESP register is set according to the state of the retry logic and the wait logic, so that if more than one wait state is inserted, the **HRESP** output only changes during the last **HREADY** LOW cycle. Retry responses are generated until the counter reaches zero, when the **HRESP** output indicates that the transfer may complete normally.

Read data generation

Different read data values must be generated according to the address of the current transfer, selecting output data from one of the four registers or one of the seven combinational outputs. This section of the code selects a data source during the data phase of a valid transfer, and then directly drives the output data bus **HRDATA** with this selected data value.

This combinational output path allows a zero wait state response to be possible, as data written to a register can be read the following cycle with a zero wait state transfer. If a registered output data path is used, then reads from registers that were written to in the previous cycle must have at least one wait state inserted, to allow for the internal data register to sample the write data, and then for the data register output to be sampled by the output read data register, before being driven onto the output read data bus.

4.9 Static memory interface

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

The SMI connects the AMBA AHB to the external memory bus of an AMBA microcontroller. This allows the connection of up to three 256MB banks of 32-bit wide static memory (for example, SRAM and ROM) and also provides 32-bit test access to the AMBA system in conjunction with the TIC. Figure 4-31 shows the block diagram of the SMI.

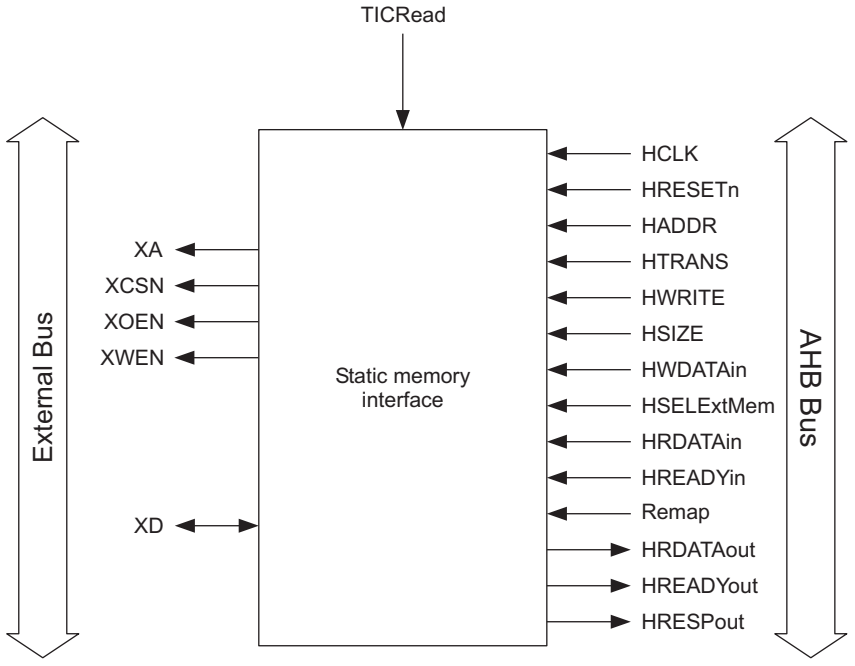


Figure 4-31 Static memory interface block diagram

The main sections of this module are:

- the AHB slave bus interface
- the data and address bus registers and drivers
- the external memory access control logic.

4.9.1 Signal descriptions

Table 4-10 describes the signals used by the SMI.

Table 4-10 Signal descriptions

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HADDR[31:0]	Address bus	Input	The 32-bit system address bus.
HTRANS[1:0]	Transfer type	Input	This indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE	Transfer direction	Input	When HIGH this signal indicates a write transfer, and when LOW, a read transfer.
HSIZE[2:0]	Transfer size	Input	Indicates the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit). The protocol allows for larger transfer sizes up to a maximum of 1024 bits.
HWDATAin[31:0	Write data bus	Input	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended, however, this can easily be extended to allow for higher bandwidth operation.
HSELExtMem	Slave select	Input	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 a combinatorial decode of the address bus.
HRDATAin[31:0] **HRDATAout[31:0]**	Read data bus	Input/output	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended, however this can easily be extended to allow for higher bandwidth operation.
HREADYin **HREADYout**	Transfer done	Input/output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HRESP[1:0]	Transfer response	Output	The transfer response provides additional information on the status of a transfer. This module will always generate the OKAY response.

Table 4-10 Signal descriptions (continued)

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 `0x0000 0000`, which normally contains the system startup code. In normal operation this signal is HIGH, allowing use of the internal memory.
TicRead	Drive out read data	Input	This signal controls the SMI to drive the current read data from **HRDATA** to **XD**.
XD[31:0]	External data bus	Input/output	This is the bidirectional external data bus. In normal operation it is driven by the external bus when **XOEN** is LOW, and by this module when **XOEN** is HIGH. During system test this becomes the test bus **TESTBUS** and its direction is controlled by the TIC control signals.
XA[30:0]	External address bus	Output	The external address bus becomes valid during the **HCLK** LOW phase of the transfer and remains valid throughout the rest of the transfer.
XCSN[3:0]	External chip select	Output	These signals are active LOW chip enables for each of the three banks (0-1, 3) of static memory. **XCSN[3]** should be connected to the memory containing the startup program (boot ROM/BIOS) for the system.
XOEN	External output enable	Output	This is the output enable for devices on the external bus. This is LOW during reads from external memory, during which time the selected bank should drive the **XD** bus.
XWEN[3:0]	External write enable	Output	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. The SMI is configured to have a minimum of two wait states when writing to memory. **XWEN** is only valid during the second cycle of the write transfer.

4.9.2 Functional description of the SMI

The SMI has five functions in the example system described in the following paragraphs:

- *External bus control*
- *Memory bank select* on page 4-57
- *Memory write control* on page 4-58
- *Configurable memory access wait states* on page 4-59
- *System test access* on page 4-59.

External bus control

To perform a read from external memory, **XOEN** must be LOW and the **XD** output is tristated, allowing it to be driven with read data by the external memory.

Figure 4-32 shows the timing of a read from memory with zero wait states.

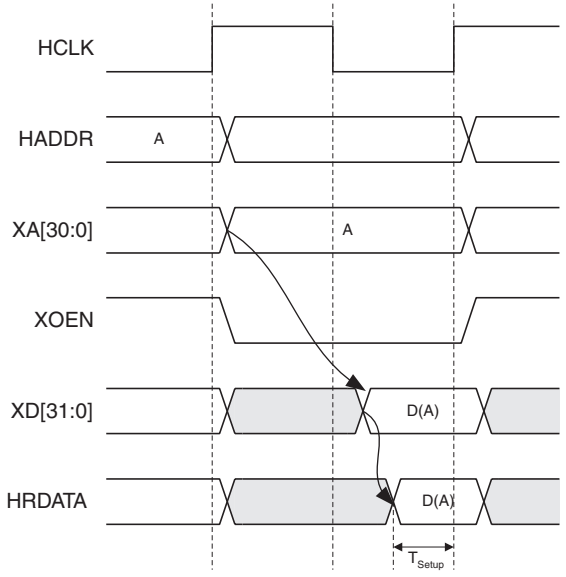


Figure 4-32 Zero wait memory read

Note

The data must be valid on the **XD** bus in time for the signal to propagate on-chip so that the **HRDATA** bus becomes valid before the next rising edge of **HCLK**. If this setup time cannot be achieved, the access will require wait states.

To perform a write to the external memory, **XOEN** must be HIGH, to allow **XD** to be driven by the SMI with a registered version of **HWDATA**.

The SMI requires at least two wait states to be added for a write to memory, to allow for the timing of the **XWEN** write enable signal relative to the **XA** and **XD** buses. When **XWEN** is LOW **XA** must be stable and, on the rising edge of **XWEN**, **XD** must be valid.

Figure 4-33 shows the timing of a write to memory with two wait states.

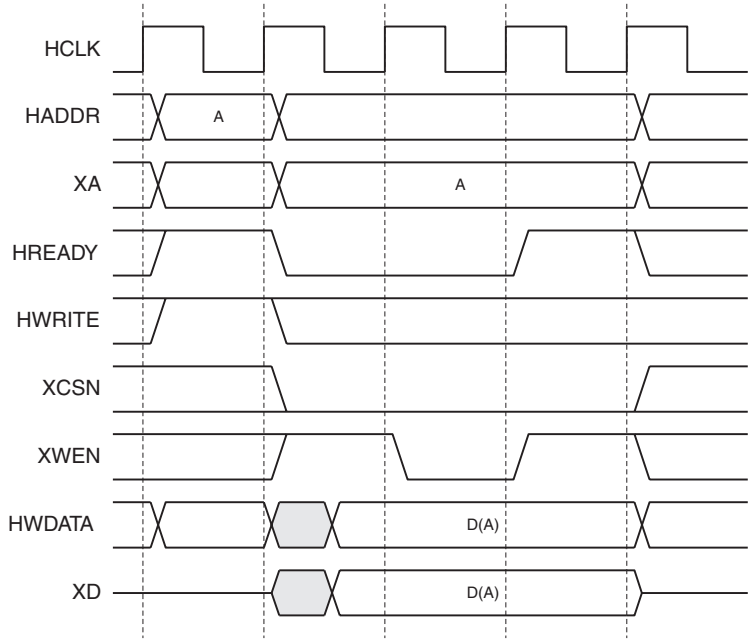


Figure 4-33 Memory write with two wait states

Memory bank select

The **XCSN** chip select lines are controlled by the address of a valid transfer, and the system memory map mode. Before the system memory is remapped, the boot ROM at `0x3000 0000` is also mapped to the base address of `0x0000 0000`.

Table 4-11 shows the relationship between the inputs and the generated value of **XCSN**.

Table 4-11 XCSN coding

Input HSELExtMem	Input Remap	Input HADDR[29:28]	Output XCSN[3:0]
0	X	XX	1111
1	0	00	0111
1	0	01	1101
1	0	10	1011
1	0	11	0111
1	1	00	1110
1	1	01	1101
1	1	10	1011
1	1	11	0111

XCSN is also held in the 1111 state asynchronously during reset.

Memory write control

The 4-bit **XWEN** write enable signal allows the four bytes in the 32-bit wide word to be written independently. The byte assignments are:

- **XWEN[0]** controls **XD[7:0]**
- **XWEN[1]** controls **XD[15:8]**
- **XWEN[2]** controls **XD[23:16]**
- **XWEN[3]** controls **XD[31:24]**.

The SMI controls **XWEN** for writes in word (32-bit), halfword (16-bit) and byte (8-bit) quantities. The SMI uses **HSIZE[1:0]** and **HADDR[1:0]** to select the width and order of each write to memory. This information must be valid before **XWEN** is asserted.

Table 4-12 shows the bytes selected according to the **HSIZE** and **HADDR[1:0]** inputs.

Table 4-12 XWEN coding

HSIZE[1:0]	HADDR[1:0]	XWEN[3:0]
10 (word)	XX	0000
01 (half word)	0X	1100
01 (half word)	1X	0011
00 (byte)	00	1110
00 (byte)	01	1101
00 (byte)	10	1011
00 (byte)	11	0111

Configurable memory access wait states

The SMI only supports global (the same for every bank) wait states for read and write accesses. This is configurable (in the HDL model, not in synthesized hardware) between zero and three waits for reads, and between two and three for writes. Figure 4-33 on page 4-57 shows a memory transfer with two wait states. A transfer with more wait states causes further wait cycles to be added. The external address and data information remains valid until the memory access cycle is completed. For writes, the **XWEN** signal is extended, going LOW during the first wait, and not going HIGH until the final cycle of the transfer. Before synthesis, the wait states can be configured by altering the 2-bit wide constants READWAIT and WRITEWAIT. WRITEWAIT must be value 2 or greater.

System test access

During system TIC testing, the external bus output of the SMI is controlled by the active HIGH **TicRead** signal from the TIC. This is used to pass read data from the **HRDATAin** bus onto the external test bus **XD** During normal operation this signal is held LOW.

4.9.3 System description

The following paragraphs give a description of how the HDL code for the module is set out. A basic system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs and outputs used in the module. This part should be read together with the HDL code.

A basic block diagram of the static memory interface system is shown in Figure 4-34.

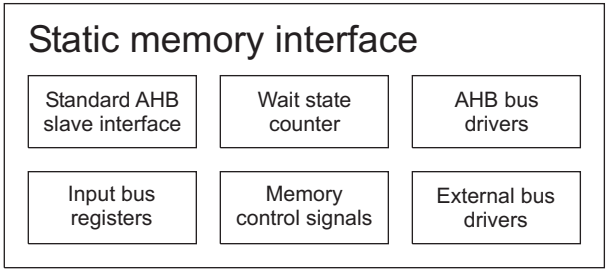


Figure 4-34 Static memory interface module block diagram

The static memory interface module comprises the input bus registers, the wait state counter used to insert wait states, and the external memory control signal generation.

All registers used in the system are clocked from the rising edge of the system clock **HCLK**, and use the asynchronous reset **HRESETn**.

Figure 4-34 on page 4-60 shows the static memory interface HDL file.

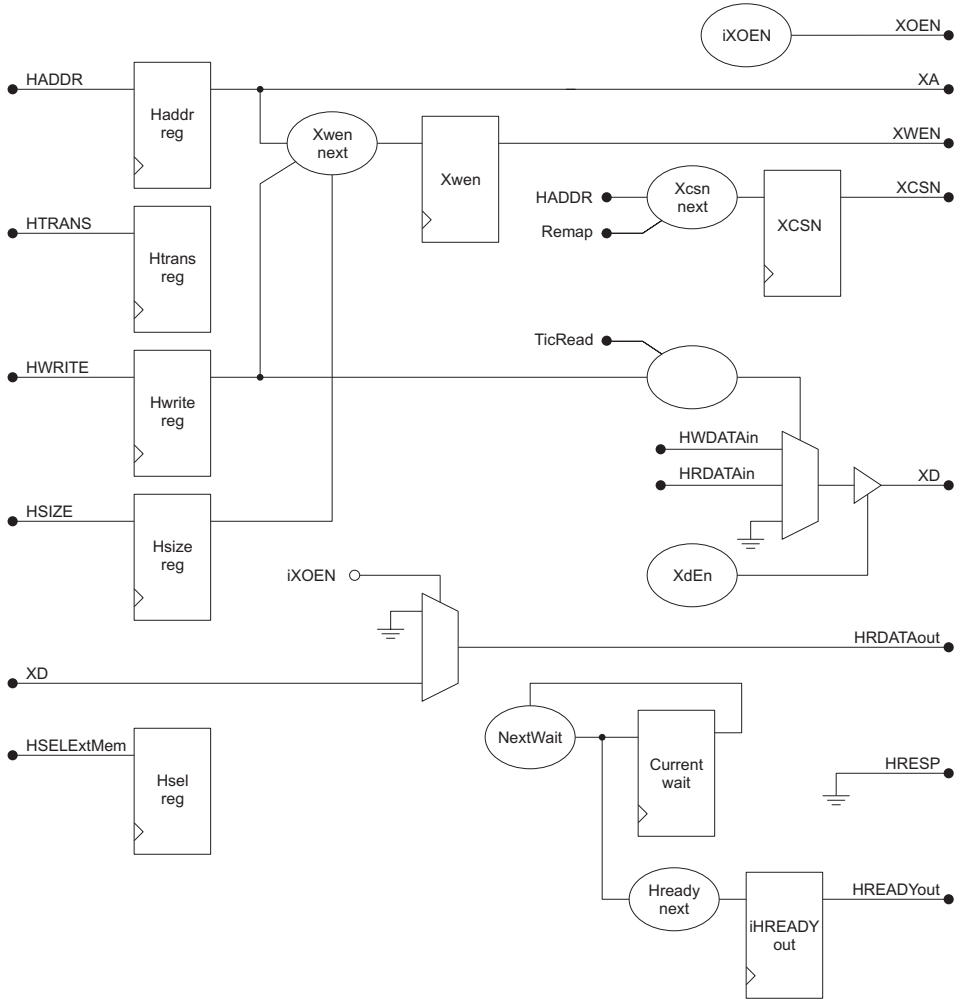


Figure 4-35 Static memory interface module system diagram

The main sections in the SMI module are explained in more detail in the following paragraphs:

- *Constant definitions*
- *AHB slave bus interface*
- *Wait state generation*
- *AHB output data bus generation* on page 4-63
- *External bus output generation* on page 4-63.

Constant definitions

The constants READWAIT and WRITEWAIT are used to set the number of wait states that are inserted when a read and write transfer is performed. The value of zero to three for reads, and two to three for writes, is set for all transfers to all memory banks, and although configurable in the HDL code, it is permanently set when synthesized.

AHB slave bus interface

This module uses the standard AHB slave bus interface, which comprises:

- the valid transfer detection logic
- the address and control registers, which are used to store the information from the address phase of the transfer for use in the data phase.

The default address setting of the module is external RAM from `0x0000 0000` to `0x1FFF FFFF`, and external boot ROM from `0x3000 0000` to `0x3FFF FFFF`. When the **Remap** signal is HIGH, indicating that remapped memory is in use, external RAM is mapped from `0x0000 0400` to `0x1FFF FFFF`, with internal memory being mapped in the first `0x000` to `0x400` region.

Wait state generation

The counter register is used to insert wait states according to the values set in the READWAIT and WRITEWAIT constants. The counter is loaded with the relevant value when a read or write transfer begins, and decrements the value until no more wait states need to be added. The counter value is used to generate the input to the **HREADYout** register, which is set LOW while the counter is not zero.

AHB output data bus generation

The **HRDATAout** output is driven to **XD** during a normal external memory read transfer, to propagate the read data value from the external bus onto the AHB. **HRDATAout** is driven LOW at all other times.

The registered **HREADYout** output is driven LOW while the current value of the wait state counter is not zero.

The **HRESP** output is held LOW, because the SMI will always generate an OKAY response to all transfers.

External bus output generation

This section contains the signals that are driven onto the external bus:

- **XD** is generated from either the AHB read or write data buses, depending on the current system mode of operation. **HWDATAin** is used during a normal external memory write transfer, and **HRDATAin** is used during a TIC testing read cycle. As **XD** is a tristate bus, then it is only driven by the SMI when the current transfer is a standard write or a TIC testing read, allowing **XD** to be driven by any external modules at all other times.
- **XA** is driven with a registered version of bits [30:0] of **HADDR**, as the full system address range is not required on the external bus.
- **XCSN** is generated from the input address during a valid read or write transfer. Bits [29:28] of the address are decoded as shown in Table 4-11 on page 4-58. When **Remap** is LOW, the boot ROM is mapped at the base address, as well as its standard address. External RAM access is not dependant on the **Remap** input. During reset, or when the memory is not addressed, all **XCSN** output bits are set HIGH to deselect all banks of external memory.
- **XOEN** is set LOW during a valid read transfer, and is set HIGH at all other times.
- **XWEN** is generated from the size and address settings for a write transfer, selecting the transfer size and byte lane to use, as shown in Table 4-11 on page 4-58. A registered output is used to avoid the generation of glitches, which can cause incorrect values to be written to the external ROM.

4.10 Test interface controller

The *Test Interface Controller* (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 **TESTBUS** (**XD**), and uses the *External Bus Interface* (EBI) to drive the external bus with test read data, allowing the use of only one set of output tristate buffers onto **TESTBUS**.

The TIC is used to convert externally applied test vectors into internal transfers on the AHB bus. A three-wire external handshake protocol is used, 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, which ensures test access under all conditions.

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

Figure 4-36 shows the TIC module interface diagram.

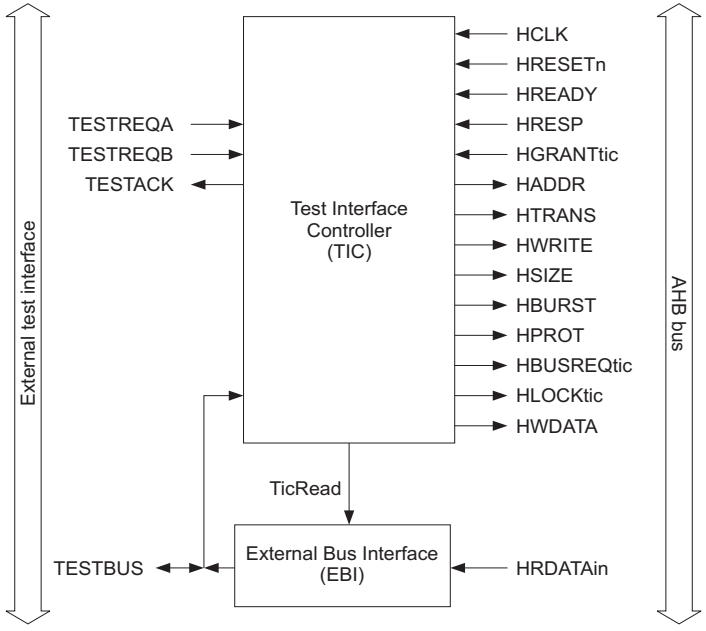


Figure 4-36 TIC module interface diagram

Figure 4-36 on page 4-64 represents a TIC module in a system where the external data bus becomes the test bus when performing test mode accesses. 16-bit and 8-bit data bus systems require, for example, 16 or 24 address lines to be reconfigured as bidirectional test port signals for the test mode access.

4.10.1 Signal descriptions

The TIC has three primary interfaces:

- the AHB bus master interface, to control the operation of the system during test
- the external test interface, to read the type of vector being applied and control the application of new vectors
- the datapath interface, to control the operation of the EBI to drive the external data bus.

Table 4-13 shows the TIC module signal descriptions for an AHB-based system.

Table 4-13 TIC signal descriptions for AHB

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers. All signal timings are related to the rising edge of **HCLK**.
HRESETn	Reset	Input	The bus reset signal is active LOW and is used to reset the system and the bus. This is the only active LOW signal.
HADDR[31:0]	Address bus	Output	The 32-bit system address bus.
HTRANS[1:0]	Transfer type	Output	Indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL or IDLE. The TIC does not use the BUSY transfer type.
HWRITE	Transfer direction	Output	When HIGH this signal indicates a write transfer and when LOW a read transfer.
HSIZE[2:0]	Transfer size	Output	Indicates the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit). The TIC does not support larger transfer sizes.
HBURST[2:0]	Burst type	Output	Indicates if the transfer forms part of a burst. The TIC always performs incrementing bursts of unspecified length.

Table 4-13 TIC signal descriptions for AHB (continued)

Signal	Type	Direction	Description
HPROT[3:0]	Protection control	Output	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 bufferable.
HWDATA[31:0]	Write data bus	Output	The write data bus is used to transfer data from the master to bus slaves during write operations. A minimum data bus width of 32 bits is recommended, however this can easily be extended to allow for higher bandwidth operation.
HREADY	Transfer done	Input	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HRESP[1:0]	Transfer response	Input	The transfer response provides additional information on the status of a transfer. Four different responses are provided, OKAY, ERROR, RETRY and SPLIT.
HBUSREQtic	Bus request	Output	A signal from the TIC to the bus arbiter which indicates that it requires the bus.
HLOCKtic	Locked transfers	Output	When HIGH this signal indicates that the master requires locked access to the bus and no other master should be granted the bus until this signal is LOW.
HGRANTtic	Bus grant	Input	This signal indicates that the TIC is currently the highest priority master. Ownership of the address and control signals changes at the end of a transfer when **HREADY** is HIGH, so a master gains access to the bus when both **HREADY** and **HGRANTx** are HIGH.
TESTBUS	Test data bus	Input	This is the bidirectional external data bus. In normal operation it is driven by the external bus interface. During system test it becomes the test data bus and its direction is controlled by the test bus request A and B signals.
TESTREQA	Test bus request A	Input	This is the test bus request A input signal and is required as a dedicated device pin. During normal system operation the **TESTREQA** signal is used to request entry into the test mode. During test **TESTREQA** is used, in combination with **TESTREQB**, to indicate the type of test vector that will be applied in the following cycle.

Table 4-13 TIC signal descriptions for AHB (continued)

Signal	Type	Direction	Description
TESTREQB	Test bus request B	Input	During test this signal is used, in combination with **TESTREQA**, to indicate the type of test vector that will be applied in the following cycle.
TESTACK	Test acknowledge	Output	The test bus acknowledge signal gives external indication that the test bus has been granted and also indicates when a test access has completed. When **TESTACK** is LOW the current test vector must be extended until **TESTACK** becomes HIGH.
TicRead	Drive out read data	Output	This signal controls the EBI to drive the current read data from **HRDATA** to **TESTBUS**.

4.10.2 Function and operation of module

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 4-14 shows the operation of the external test pins to change the TIC mode from normal operation into test mode.

Table 4-14 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 are used to control the operation of the TIC. The operation of these pins is shown in Table 4-15.

Table 4-15 Test control signals during test mode

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

In test mode, the internal **HCLK** is driven from the external **TESTCLK** source. This pin may be the normal clock oscillator source input or a port replacement signal. The system bus clock must not glitch when switching between normal and 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

There are five types of test vector 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, which 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. The following rules may be used to determine which type of vector 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 and control vectors are applied they are all address vectors, apart from the last which is a control vector.

Control vectors

The control vector is used to determine the types of transfer the TIC can perform, by setting the values of the **HSIZE**, **HPROT** and **HLOCK** AHB master outputs.

The default TIC bus master transfer type is:

- 32-bit transfer width, **HSIZE[2:0]** signifies word transfer
- privileged system access, **HPROT[3:0]** signifies supervisor data access, uncacheable and unbufferable.

Bit 0 of the control vector is used to indicate if the control vector is valid. Thus, if a control vector is applied with bit 0 LOW, the vector will be ignored and will not update the control information. This mechanism allows address vectors which have bit 0 LOW to be applied for many cycles without updating the control information.

Although the default settings will be sufficient for testing many embedded system designs, the control vector can be used to change the control signals of the transfer, and can also be used to determine whether the TIC should generate fixed addresses or incrementing addresses.

Table 4-16 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 4-16 Control vector bit definitions

Bit position	Description
0	Control vector valid
1	Reserved
2	**HSIZE[0]**
3	**HSIZE[1]**

Table 4-16 Control vector bit definitions (continued)

Bit position	Description
4	**HLOCK**
5	**HPROT[0]**
6	**HPROT[1]**
7	Address increment enable
8	Reserved
9	**HPROT[2]**
10	**HPROT[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 and therefore **HSIZE[2]** cannot be altered and will always be LOW.

In order to support burst accesses using the test interface the TIC may support incrementing of the bus address. The TIC increments eight address bits and the address range that can be covered by this incrementer is dependent on the size of the transfers being performed.

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

The address incrementer is disabled by default and must be enabled using a control vector prior to use.

Note

The control vector is primarily used to change signals which have the same timing as the address bus. However the control vector also allows the lock signal to be changed, which is actually required before the locked transfer commences. If the **HLOCK** signal is used during testing it should be set a transfer before it is required. This difference in timing on the **HLOCK** signal may in some cases cause an additional transfer to be locked both before and after the sequence intended to be locked.

4.10.3 Test vector sequences

The following test vector sequences are described:

- *Entering test mode*
- *Write vectors* on page 4-72
- *Read vectors* on page 4-73
- *Control vector* on page 4-74
- *Burst vector* on page 4-75
- *Read-to-write and write-to-read transfers* on page 4-76
- *Exiting test mode* on page 4-77.

Entering test mode

In normal operating mode **TESTREQA** will be LOW, indicating that test access is not required and the test bus will be used as required for normal operation, which will usually be part of the external bus interface. Entering test mode allows test vectors to be applied externally that will cause transfers on the internal bus.

The following sequence, required in order to enter test mode, is illustrated in Figure 4-37 on page 4-72:

1. **TESTREQA** is asserted to request test bus access.
2. Test mode is entered when the TIC has been granted the internal bus and this is indicated by the assertion of the **TESTACK** signal.
3. At this point **TESTCLK** will become the source of the internal **HCLK** signal.
4. When test mode has been entered **TESTREQB** is asserted to initiate an address vector.
5. The TIC will not perform any internal transfers until a valid address vector has been applied.

A synchronous tester would not be expected to poll **TESTACK** for the bus. Normally the **TESTREQA** signal would be asserted for a minimum number of cycles guaranteed to gain access to the bus (completion of the longest wait-state peripheral access or the maximum number of cycles for all bus masters to have completed their current instruction).

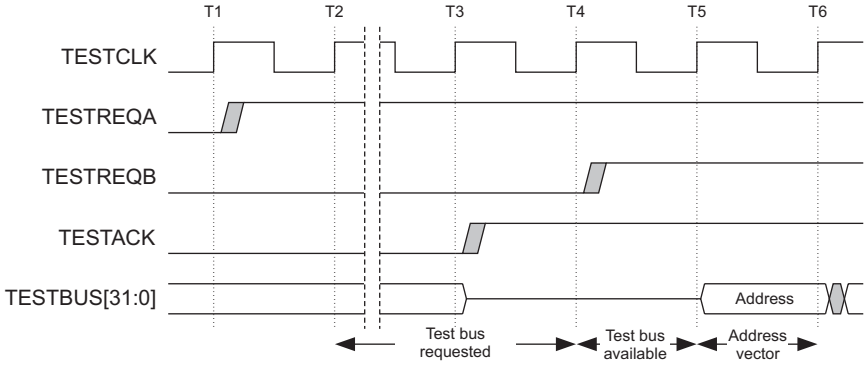


Figure 4-37 Test start sequence

Write vectors

Figure 4-38 shows the sequence of events when applying a set of write test vectors. Initially an address vector is applied and this is followed by a write test vector.

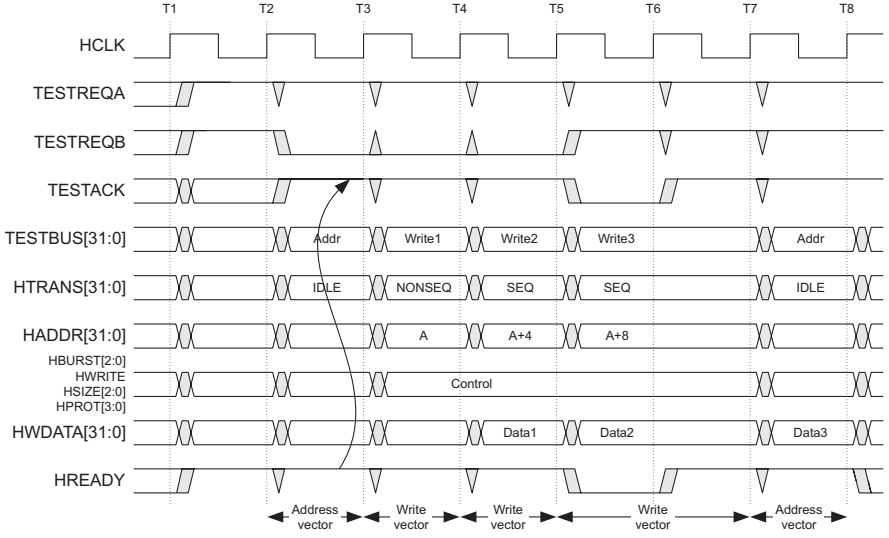


Figure 4-38 Write test vectors

The **TESTREQA** and **TESTREQB** signals are pipelined and are used to indicate what type of vector will be applied in the following cycle.

Figure 4-38 on page 4-72 shows an example of a number of write transfers being performed.

The TIC samples the address, **TESTREQA**, and **TESTREQB** signals at time T3, and following this it can initiate the appropriate transfer on the AHB. In the following cycle the write data is driven onto **TESTBUS** and it is then sampled on the following clock edge, T4, and driven onto the internal bus.

If the internal transfer is not able to complete then the **TESTACK** signal is driven LOW and this indicates that the external test vector must be applied for another cycle.

Read vectors

Read transfers are more complex because they require **TESTBUS** to be driven in the opposite direction, and therefore additional cycles are required to prevent bus clash when changing between different drivers of **TESTBUS**. Figure 4-39 shows a typical test sequence for reads.

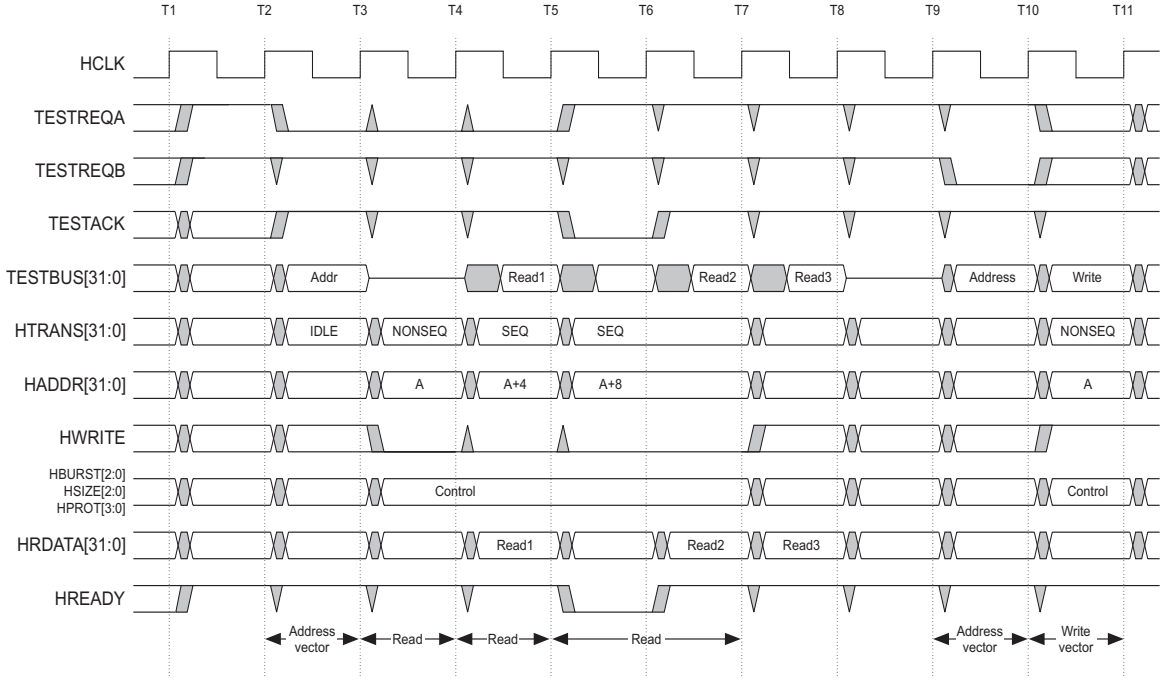


Figure 4-39 Read test vectors

The **TESTREQA** and **TESTREQB** signals are used in the same way as for write transfer. Initially **TESTREQA** and **TESTREQB** are used to apply an address vector and then in the following cycle they are used to indicate that a read transfer is required. For the first cycle of a read **TESTBUS** must be tristated, which ensures that the external equipment driving **TESTBUS** has an entire cycle to tristate its buffers before the TIC will enable the on-chip buffers to drive out the read data.

At the end of a burst of reads it is also necessary to allow time for bus turnaround. In this case the TIC must turn off the internal buffers and an entire cycle is allowed before the external test equipment starts to drive **TESTBUS**.

The end of a burst of reads is indicated by both **TESTREQA** and **TESTREQB** being HIGH, as for an address vector. In fact they must indicate an address vector for two cycles, which allows for the turnaround cycle at the start of the burst and also the turnaround cycle at the end of the burst.

Control vector

The operation of the TIC may be modified by the use of a control vector. Whenever more than one address vector is applied in succession then the last vector is considered to be a control vector and is not latched as the address. Bit 0 of the control vector is used to determine whether or not the control vector should be considered valid, which allows multiple address vectors to be applied without changing the control information.

Figure 4-40 on page 4-75 shows the process of inserting a control vector.

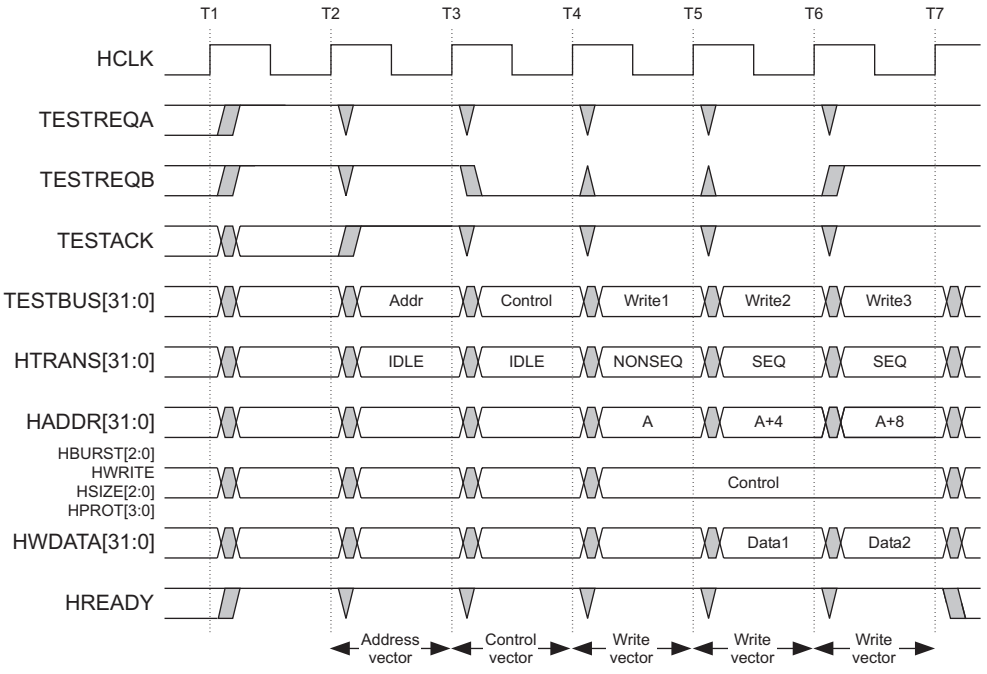


Figure 4-40 Control vector

At time T4 the TIC can determine that **TESTBUS** contains a control vector. This is because the previous cycle was an address vector, and **TESTREQA** and **TESTREQB** are indicating that the following cycle is either a read or a write and therefore the current cycle must be a control vector.

Burst vector

The examples of read and write transfers shown in Figure 4-40 also show how additional transfers can be used to form burst transfers on the bus. The TIC has limited capabilities for burst transfers and can only perform undefined length incrementing bursts.

The TIC contains an 8-bit incrementer and if an attempt is made to perform a burst which crosses the incrementer boundary then the address will simply wrap and the TIC will signal the transfer as NONSEQUENTIAL. The exact boundary at which this will occur is dependent on the size of the transfer. For word transfers the incrementer will overflow at 1KB boundaries, for halfword transfers it will overflow at 512-byte boundaries and for byte transfers the overflow will occur at 256-byte boundaries.

Read-to-write and write-to-read transfers

It is possible to switch between read transfers and write transfers without applying a new address vector. Usually this is done with the address incrementer disabled, so that both the read transfers and the write transfers are to the same address. It is also possible to do this with the incrementer enabled if the test circumstances require it.

When moving from a read transfer to a write transfer it is also necessary to allow two cycles for bus handover and therefore **TESTREQA** and **TESTREQB** should signal an address vector for two cycles after the read. This will not cause the address to be changed unless it is followed by a third address vector.

Figure 4-41 illustrates the sequence of events.

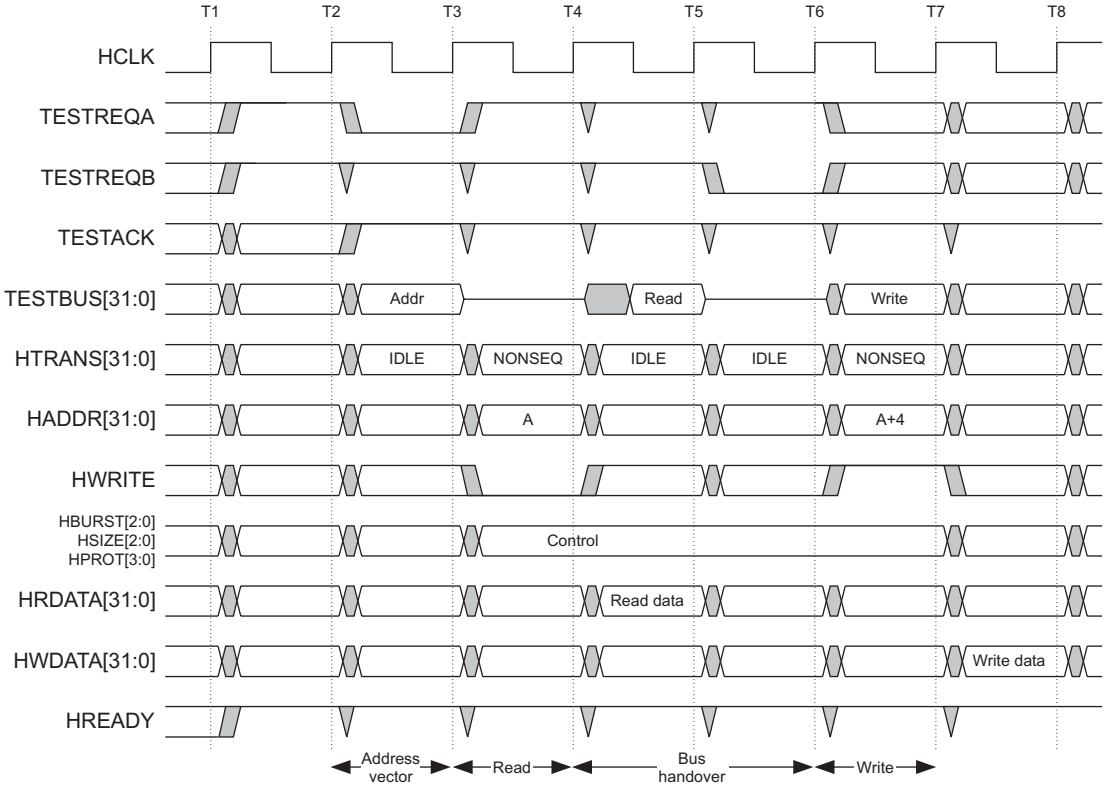


Figure 4-41 Read vector followed by write vector

Exiting test mode

Test mode is exited using the following sequence:

1. Apply a single cycle of address vector, which causes an IDLE cycle internally. This ensures any internal transfers have been completed and an ADDRESS-ONLY transfer is performed on the internal bus.
2. **TESTREQA** and **TESTREQB** are both driven LOW to indicate that test mode is to be exited.
3. When the test interface has been configured for normal system operation, **TESTACK** will go LOW to indicate that test mode has been exited.

It is important that test mode can be entered and exited cleanly so that the TIC can be used for diagnostic test during system operation, as well as during production testing.

4.10.4 System description

This describes how the HDL code for the TIC is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This should be read together with the HDL code.

Figure 4-42 shows the TIC module block diagram.

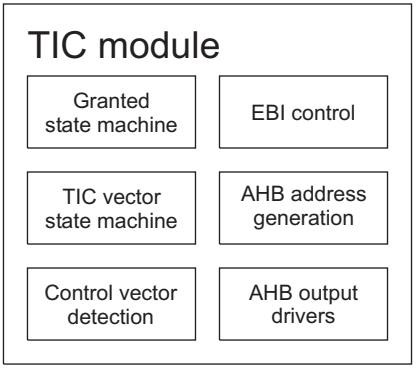


Figure 4-42 TIC module block diagram

The TIC comprises two state machines, which are used to control the access to the AHB of the master interface, and the application of test vectors from the external bus to the system.

All registers in the system are clocked from the rising edge of the system clock **HCLK**, and use the asynchronous reset **HRESETn**.

A diagram of the TIC HDL file is shown in Figure 4-43.

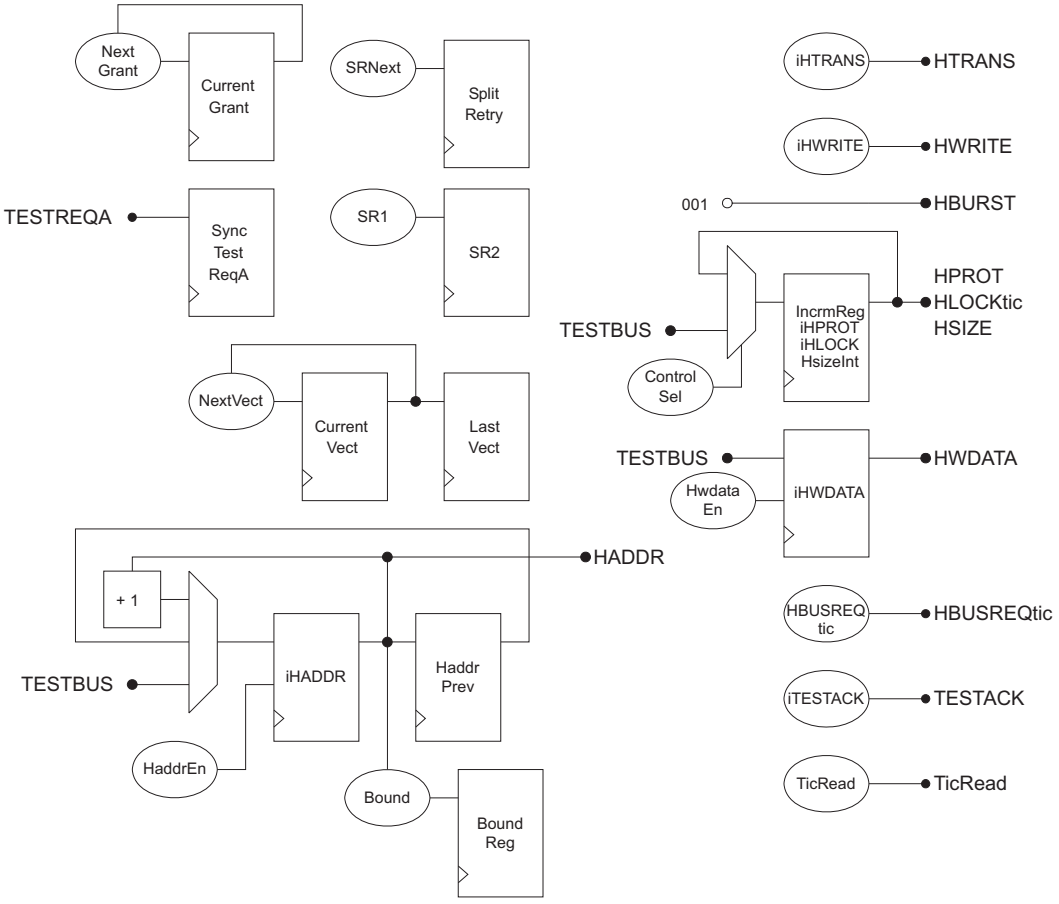


Figure 4-43 TIC module system diagram

The main sections of the code are explained in the following paragraphs:

- *Granted state machine*
- *TIC vector state machine*
- *AHB address generation* on page 4-82
- *Control vector detection* on page 4-83
- *Read data control* on page 4-83
- *Split or retry detection* on page 4-83
- *AHB bus master output signal generation* on page 4-84.

Granted state machine

This is part of the standard AHB bus master interface, and is used to determine when the TIC is granted the bus, and when it can drive the address, control and data outputs.

The state machine is shown in Figure 4-44, and only advances when the **HREADY** input is set HIGH.

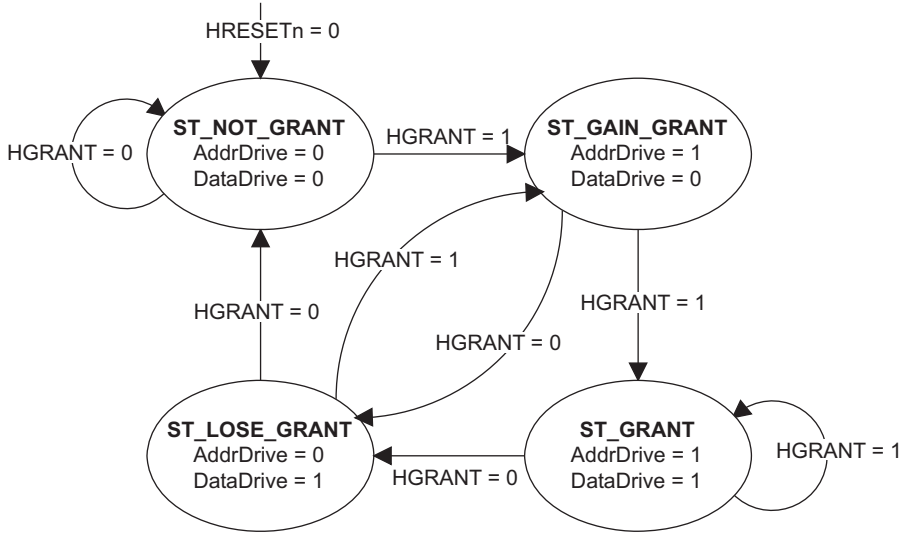


Figure 4-44 TIC module granted state machine

TIC vector state machine

This section of the code is used to control the application of test vectors from the external tester onto the AHB.

Figure 4-45 illustrates the operation of the TIC vector state machine.

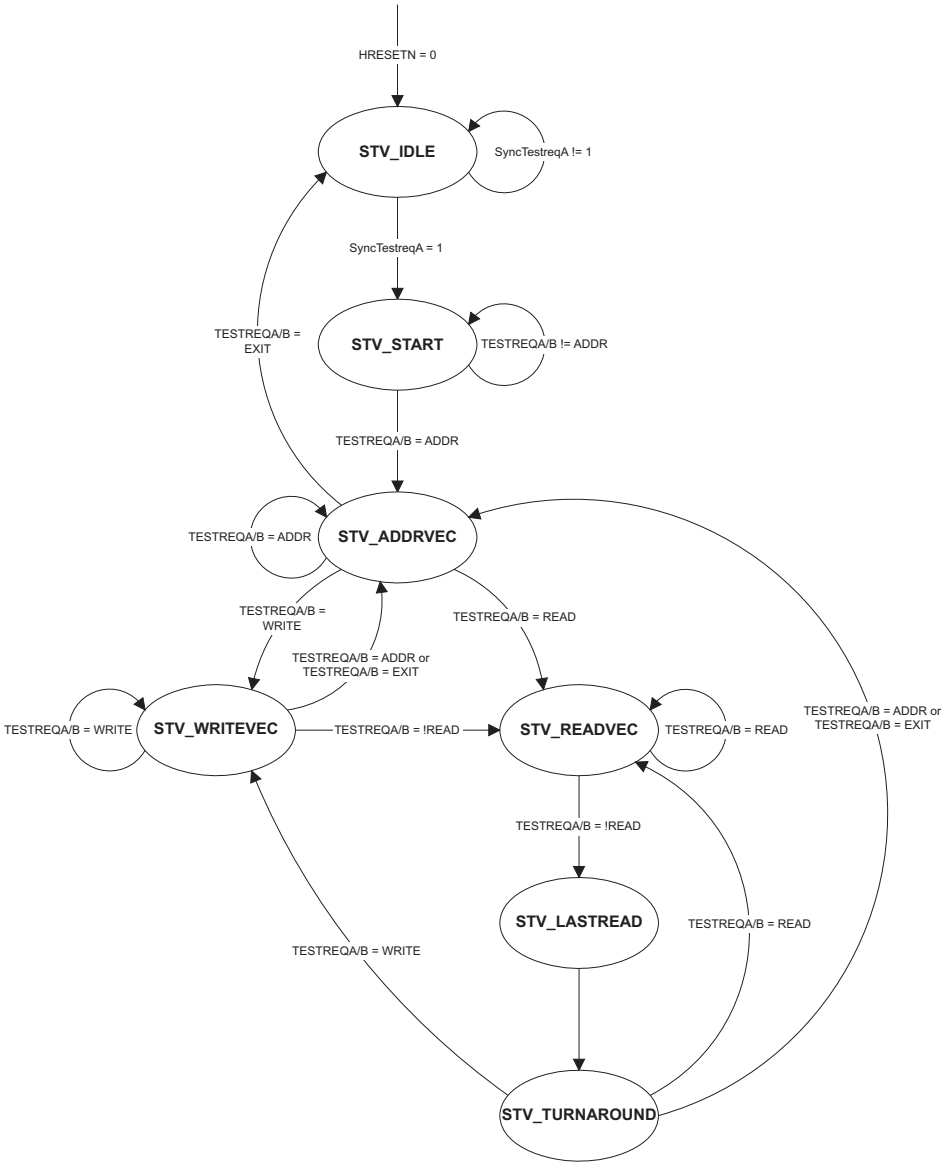


Figure 4-45 TIC vector state machine

At reset the TIC is in the IDLE state and will not be requesting use of the AHB. When in the IDLE state **TESTACK** is driven LOW to indicate that the test interface cannot be used.

The **TESTACK** signal controls all transactions around the state machine, except for the transition from IDLE to START. In all other cases the state machine remains in the same state if the **TESTACK** signal is low.

The **TESTREQA** signal moves from the IDLE state to the START state. The state of **TESTREQB** is not checked when moving from normal operation to test mode.

In some system implementations it will be necessary to switch from an internal clock source to an external clock **TESTCLK** which is used during test mode. When **TESTREQA** first goes HIGH this can be used as an indication that the clock source should be changed. A return signal that indicates when the clock switch has occurred successfully can be used to prevent the move into the START state until the test clock is in use.

If clock switching is being used then it is possible that **TESTREQA** is asynchronous to the on-chip clock before test mode is entered. Therefore a synchronizer is used to generate a synchronized version of **TESTREQA** to control the movement from the IDLE state to the START state.

The START state ensures that the first vector applied is an address vector to prevent read and write vectors occurring before the address has been initialized. The START state is only exited when **TESTREQA** and **TESTREQB** indicate an address vector and the following state is ADDRVEC.

In the ADDRVEC state the TIC registers the address on the **TESTBUS**. The ADDRVEC state is used for both address and control vectors, so additional logic is required to determine whether the value on **TESTBUS** should be considered as an address or as a control vector. If the previous cycle was an address vector and the following cycle (as indicated by **TESTREQA** and **TESTREQB**) is not an address vector then the current cycle is a control vector.

It is possible to stay in the ADDRVEC state for a number of cycles, but usually an address vector will be followed by either read or write transfers.

If a write transfer is being performed, the TIC moves into the WRITEVEC state at the same time that it initiates the transfer on the bus. Multiple write transfers can be performed by remaining in the WRITEVEC state. Usually the WRITEVEC will be followed by an address vector. However, it is also possible to move directly to a read transfer by moving to the READVEC state.

When a read, or a burst of reads is performed, the TIC enters the READVEC state. This state indicates that the TIC is starting a read transfer on the bus and it is not until the following cycle that the read data will appear. When the READVEC state is first entered the **TESTBUS** will be tristated, but will become driven during additional cycles in the READVEC state.

All read vectors must be followed by two turnaround vectors. For the first of these cycles the TIC will move into the LASTREAD state, during which the last read of the transfer will complete and will be driven out on to the external **TESTBUS**. During the LASTREAD state no internal transfers will be started and the TIC will perform IDLE transfers on the bus.

Following the LASTREAD state the TIC moves into the TURNAROUND state, during which time the external **TESTBUS** will be tristated. The TURNAROUND state will usually be followed by an address vector, but it is also possible to go immediately to a write vector or another read.

The usual method to exit from test is to return to the ADDRVEC state and then set both **TESTREQA** and **TESTREQB** LOW to return to IDLE and effectively exit from test. In fact, at any point the test mode can be exited by setting both **TESTREQA** and **TESTREQB** LOW, and eventually this will cause the TIC to exit from test.

Note

When applying TIC vectors it is theoretically possible to assert the **HLOCK** output and then exit from the test. If this happens and then the TIC is granted the bus under normal operation it will effectively lock up the bus. No protection is provided within the TIC to prevent this occurrence.

AHB address generation

There are four main sources of the **HADDR** output in the TIC:

- current address registers
- previous address registers
- external data bus
- incrementer.

The current address is held during a standard read or write cycle, as the address loaded during the previous address vector is used for all subsequent read and write transfers.

The previous address is only used when a split or retry response has been generated by the currently selected slave, and the TIC is set in incrementing mode. When the transfer is regenerated, the incremented address will have moved on for the next transfer, so the previous address must be stored for use.

When an address vector is applied, the TIC must read in the new address from the external data bus **TESTBUS**. This new value is stored in the iHADDR registers, and used for the following read and write transfers.

If address incrementing is enabled, then sequential read and write vectors will increment the address according to the transfer size that has been set. The first read or write transfer after an address vector will be to that address, then subsequent transfers will have their address incremented. This continues until a control vector is used to disable address incrementing.

Sequential incrementing read and write vectors are signalled as SEQUENTIAL transfers on the AHB, but a NONSEQUENTIAL transfer is added when the address incrementer crosses an 8-bit boundary, set by the current transfer size.

Control vector detection

This part is used to detect a control vector, and contains the control registers. A control vector is the last address vector in a burst of addresses, so is only detected when **TESTREQA** and **TESTREQB** indicate that the next transfer is a read or write vector, and there have been two or more address vectors. The TIC vector state machine is used to detect this, when LastVect and CurrentVect are set to address vector, and NextVect is either a read or a write vector. Also, bit 0 of the control vector (on **TESTBUS**) must be set HIGH for it to be valid, allowing for bursts of addresses.

Once it has been detected, the control vector is written to the registers used to hold the transfer settings for **HSIZE**, **HLOCK**, **HPROT,** and if address incrementing is enabled. These values are then held until the next control vector is detected and stored.

Read data control

TicRead is used to enable the EBI to drive the current read data value from **HRDATA** onto **TESTBUS**. It is set HIGH when the last vector was a read, allowing time for the read data to be driven onto the AHB. This output is disabled when the TIC is not granted control of the bus, allowing the EBI to function normally.

Split or retry detection

The TIC must know when the currently selected slave has generated a split or retry, and this section is used to detect that response. If the TIC loses grant before the transfer has been regenerated, then the value of the **SplitRetry** signal is held until the TIC has gained control of the bus again.

SR1 and SR2 are also used to indicate the first and second cycles of a SPLIT/RETRY response. SR2 is registered to remove a combinational path from **HRESP** to **HTRANS**.

AHB bus master output signal generation

As the TIC is an AHB bus master, it must drive all of the output signals needed to control the operation of AHB slaves on the bus, and also the bus grant request output. This section generates these outputs, and controls when they can be driven out.

HTRANS is generated according to the granted state machine, the TIC vector state machine, the split or retry status, and the incrementer boundary condition.

NONSEQUENTIAL transfers are generated:

- during a read or write following an address
- during a read or write when the TIC has just gained control of the bus
- during a regenerated read or write that has been split or retried
- when the address incrementer has crossed an 8-bit boundary during a sequential read or write.

SEQUENTIAL transfers are generated in incrementing mode:

- when a read follows a read or a write
- when a write follows a write.

IDLE transfers are generated at all other times, as no bus transfers need to be performed.

HWRITE is set HIGH when the current transfer is a write, and is set LOW at all other times. During a regenerated split/retry transfer, the last vector is used.

HBUSREQtic is set LOW when the TIC vector state machine is in the IDLE state, and is set HIGH at all other times, as the bus is only requested when test mode has been entered.

Chapter 5
APB Modules

This chapter describes the modules that comprise the *Advanced Peripheral Bus* (APB). It contains the following sections:

5.1 Interrupt controller

The interrupt controller is an APB slave, providing a simple software interface to the interrupt system. It consists of:

- source status and interrupt request status
- separate enable set and enable clear registers to allow independent bit enable control of interrupt sources
- level-sensitive interrupts
- programmable interrupt source.

Figure 5-1 shows the interrupt controller module block diagram.

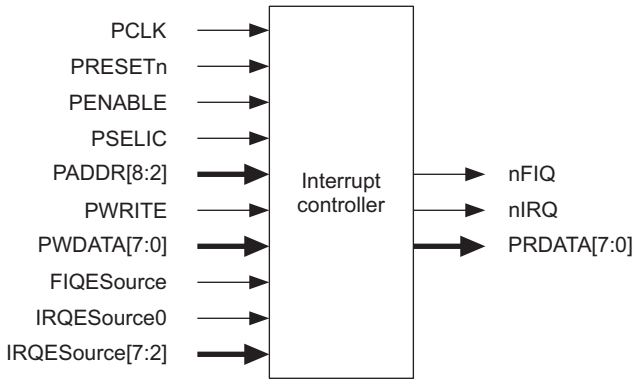


Figure 5-1 Interrupt controller module block diagram

5.1.1 Hardware interface and signal description

The interrupt controller module is connected to the APB bus. Table 5-1 shows the signal descriptions for the interrupt controller.

Table 5-1 APB signal descriptions for interrupt controller

Signal	Type	Direction	Description
PCLK	Peripheral clock	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **PCLK** are used to control transfers.
PRESETn	Peripheral reset	Input	The bus reset signal is active LOW and is used to reset the system.
PENABLE	Peripheral enable	Input	This enable signal is used to time all accesses on the peripheral bus.
PSELIC	Peripheral slave select	Input	When HIGH, this signal indicates that this module has been selected by the APB bridge. This selection is a decode of the system address bus.
PADDR[8:2]	Peripheral address	Input	This is the peripheral address bus, which is used for decoding register accesses. The addresses become valid before **PENABLE** goes HIGH and remains valid after **PENABLE** goes LOW.
PWRITE	Peripheral transfer direction	Input	This signal indicates a write when HIGH and a read when LOW. It has the same timing as the peripheral address bus.
PWDATA[5:0]	Peripheral write data bus	Input	The write peripheral data bus is driven by the bridge at all times.
PRDATA[7:0]	Peripheral read data bus	Output	The read peripheral data bus is driven by this block during read cycles (when **PWRITE** is LOW and **PSELIC** is HIGH).
FIQESource	FIQ interrupt source	Input	**FIQ** interrupt signal into the interrupt module. This active HIGH signal indicates that a fast interrupt request has been generated.
IRQESource[0] **IRQESource[7:2]**	IRQ interrupt sources	Input	**IRQ** interrupt signals into the interrupt module. These active HIGH signals indicate that interrupt requests have been generated. (**IRQESource[1]** is internally generated in the interrupt controller module and is used to provide a software triggered **IRQ**.)
nFIQ	FIQ output	Output	Active LOW fast interrupt request input to the ARM core.
nIRQ	IRQ output	Output	Active LOW interrupt request input to the ARM core.

5.1.2 Function and operation of the interrupt controller module

The interrupt controller provides a simple software interface to the interrupt system. Certain interrupt bits are defined for the basic functionality required in any system. The remaining bits are available for use by other devices in any particular implementation. 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.

Ideally, in an ARM system, only a single FIQ source is in use at any particular time. This provides a true low-latency interrupt, because a single source ensures that the interrupt service routine may be executed directly without the need to determine the source of the interrupt. It also reduces the interrupt latency because the extra banked registers, which are available for FIQ interrupts, may be used to maximum efficiency by preventing the need for a context save.

Separate interrupt controllers are used for FIQ and IRQ. Only a single bit position is defined for FIQ, which is intended for use by a single interrupt source, while up to 32 bits are available in the IRQ controller. The standard configuration only makes eight interrupt request lines available. This can be extended to up to 32 sources by altering the IRQSize constant setting and increasing the width of the **PWDATA** and **PRDATA** lines to the interrupt controller.

The IRQ interrupt controller uses a bit position for each different interrupt source. Bit positions are defined for a software-programmed interrupt, a communications channel, and counter-timers. Bit 0 is unassigned in the IRQ controller so that it may share the same interrupt source as the FIQ controller.

All interrupt source inputs must be active HIGH and level-sensitive. Any inversion or latching required to provide edge sensitivity must be provided at the generating source of the interrupt.

No hardware priority scheme nor any form of interrupt vectoring is provided, because these functions can be provided in software.

A programmed interrupt register is also provided to generate an interrupt under software control. Typically this may be used to downgrade an FIQ interrupt to an IRQ interrupt.

Interrupt control

The interrupt controller provides:

- interrupt status
- raw interrupt status
- an enable register.

The enable register is used to determine whether or not an active interrupt source should generate an interrupt request to the processor.

The raw interrupt status indicates whether or not the appropriate interrupt source is active prior to masking and the interrupt status indicates whether or not the interrupt source is causing a processor interrupt.

The enable register has a dual mechanism for setting and clearing the enable bits. This allows enable bits to be set or cleared independently, with no knowledge of the other bits in the enable register.

When writing to the enable set location, each data bit that is HIGH sets the corresponding bit in the enable register. All other bits of the enable register are unaffected. Conversely, the enable clear location is used to clear bits in the enable register while leaving other bits unaffected.

Figure 5-2 shows the structure for a single segment of the interrupt controller.

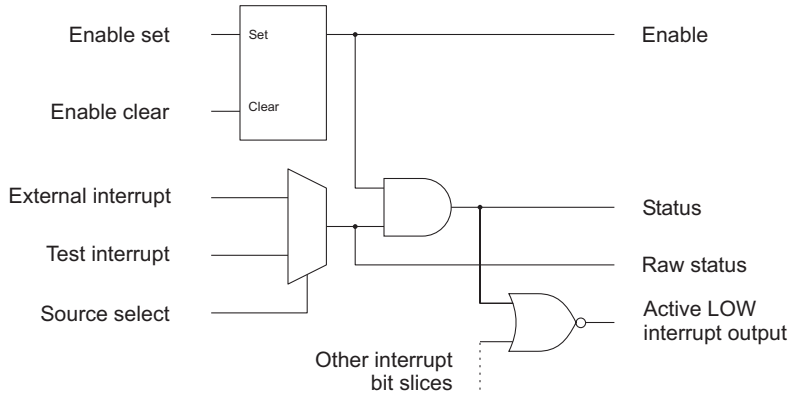


Figure 5-2 Single bit slice of the interrupt controller

The IRQ controller will usually have a larger number of bit slices, where the exact size is dependent on the system implementation.

The FIQ interrupt controller consists of a single bit slice, located on bit 0.

5.1.3 Register memory map

The base address of the interrupt controller is not fixed and may be different for any particular system implementation. However, the offset of any particular register from the base address is fixed. Table 5-2 shows the register memory map.

Table 5-2 Register memory map of the interrupt controller APB peripheral

Address	Read location	Write location
IntBase + 0x000	IRQStatus	-
IntBase + 0x004	IRQRawStatus	-
IntBase + 0x008	IRQEnable	IRQEnableSet
IntBase + 0x00C	-	IRQEnableClear
IntBase + 0x010	-	IRQSoft
IntBase + 0x100	FIQStatus	-
IntBase + 0x104	FIQRawStatus	-
IntBase + 0x108	FIQEnable	FIQEnableSet
IntBase + 0x10C	-	FIQEnableClear
IntBase + 0x014	IRQTestSource	IRQTestSource
IntBase + 0x018	IRQSourceSel	IRQSourceSel
IntBase + 0x114	FIQTestSource	FIQTestSource
IntBase + 0x118	FIQSourceSel	FIQSourceSel

5.1.4 Register descriptions

The following registers are provided for both FIQ and IRQ interrupt controllers:

Enable Read-only. The enable register is used to mask the interrupt input sources and defines which active sources will generate an interrupt request to the processor. This register is read-only, and its value can only be changed by the enable set and enable clear locations. If certain bits within the interrupt controller are not implemented, the corresponding bits in the enable register must be read as undefined.

An enable bit value of 1 indicates that the interrupt is enabled and will allow an interrupt request to reach the processor. An enable bit value of 0 indicates that the interrupt is disabled. On reset, all interrupts are disabled.

EnableSet Write-only. This location is used to set bits in the interrupt enable register. When writing to this location, each data bit that is HIGH causes the corresponding bit in the enable register to be set. Data bits that are LOW have no effect on the corresponding bit in the enable register.

EnableClear Write-only. This location is used to clear bits in the interrupt enable register. When writing to this register, each data bit that is HIGH causes the corresponding bit in the enable register to be cleared. Data bits that are LOW have no effect on the corresponding bit in the interrupt enable register.

RawStatus Read-only. This location provides the status of the interrupt sources to the interrupt controller. A HIGH bit indicates that the appropriate interrupt request is active prior to masking.

Status Read-only. This location provides the status of the interrupt sources after masking. A HIGH bit indicates that the interrupt is active and will generate an interrupt to the processor.

Soft Write only. A write to bit 1 of this register sets or clears a programmed interrupt. Writing to this register with bit 1 set HIGH generates a programmed interrupt, while writing to it with bit 1 set LOW clears the programmed interrupt. The value of this register may be determined by reading bit 1 of the source Status register. Bit 0 of this register is not used.

Two extra read/write registers are defined for both FIQ and IRQ to allow testing of the interrupt controller module using the AMBA test methodology. They must not be accessed during normal operation.

TestSource Same size as RawStatus, and used to load RawStatus with test data.

SourceSel 1-bit wide (bit 0). When set, the value in TestSource is multiplexed into RawStatus.

5.1.5 Standard configuration of registers

The FIQ interrupt controller is one bit wide and is located on bit 0. The source of this interrupt is implementation-dependent.

The interrupt controller will be customized to fit into each application. The following is an example minimum set of interrupt bits assigned in a system:

- Bits 1 to 5 in the IRQ interrupt controller are defined in the standard EASY world.
- Bit 0 and Bits 6 up to 31 are available for use as required. Bit 0 is left available so that the FIQ source may also be routed to the IRQ controller in an identical bit position.

Table 5-3 gives a typical example allocation of IRQ sources.

Table 5-3 Example of IRQ sources

Bit	Interrupt source
0	Undefined
1	Programmed Interrupt
2	Comms Rx
3	Comms Tx
4	Timer 1
5	Timer 2

5.1.6 System description

This section describes how the HDL code for the interrupt controller is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of all the registers, inputs and outputs used in the system. This section should be read together with the HDL code.

Figure 5-3 on page 5-9 shows the interrupt controller module block diagram.

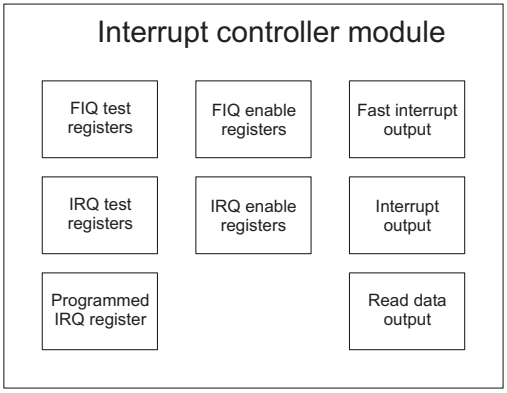


Figure 5-3 Interrupt controller module block diagram

The interrupt controller comprises sets of interrupt registers and test registers that are used to control the generation of the two interrupt outputs to the ARM core, based on the interrupt inputs.

All registers used in the system are clocked from the rising edge of the system clock **PCLK**, and use the asynchronous reset **PRESETn**.

Two diagrams are used to show the interrupt controller HDL file. Figure 5-4 shows the layout of the bit slices that are used for bit 0 of the FIQ and bits 0 and [5:2] of the IRQ.

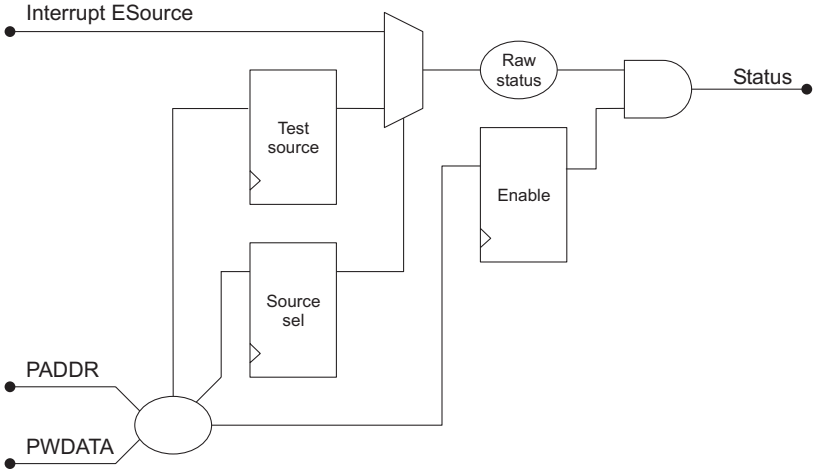


Figure 5-4 Interrupt controller slice system diagram

Figure 5-5 shows the layout of the whole system.

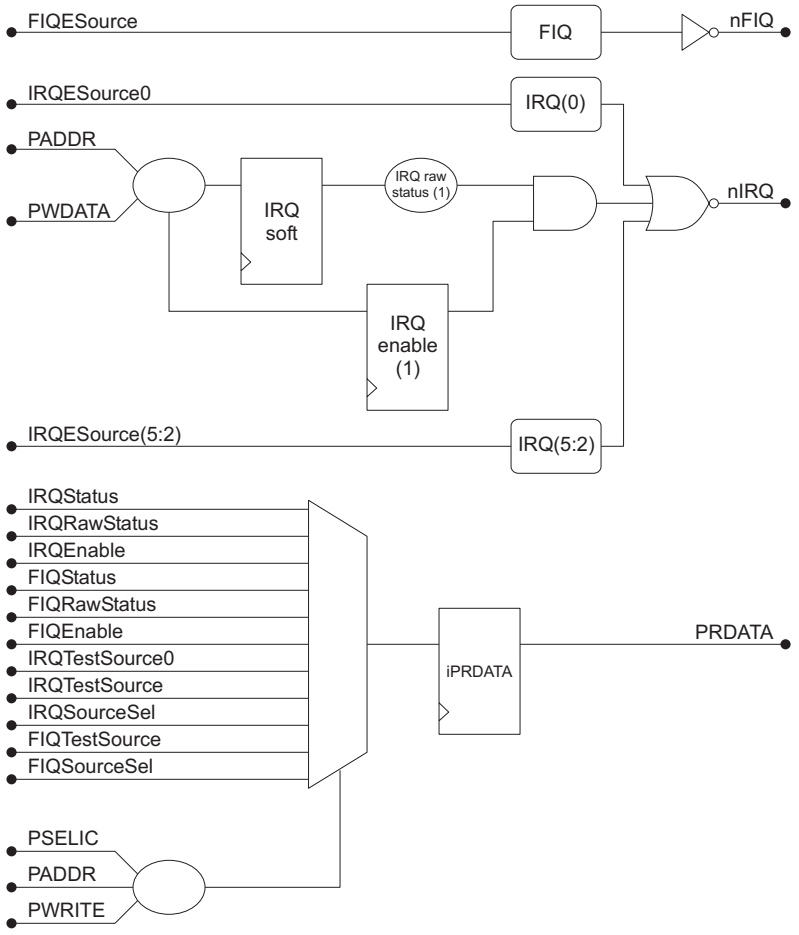


Figure 5-5 Interrupt controller module system diagram

The main sections in this module are explained in more detail in the following paragraphs:

- *Constant definitions* on page 5-11
- *IRQ generation* on page 5-11
- *FIQ generation* on page 5-11
- *Output data generation* on page 5-11.

Constant definitions

The first two constants that are specified (IRQSIZE and FIQSIZE), are used to set the number of IRQ and FIQ lines that are used in the system. The defaults are for eight IRQ lines and one FIQ line. These constants should only be changed when the number of interrupt input sources are changed.

The other constants are used to set the relative addresses of the interrupt controller registers from the base address.

IRQ generation

Figure 5-4 on page 5-9 shows the structure of the IRQ generation logic from the external interrupt sources.

The read/write TestSource register is used to hold the test value. This is passed through a multiplexor, and then used to switch between the external and internal test interrupt sources. This is the read-only RawStatus value, which is gated with the output of the enable register, and used to generate the Status output.

All of the IRQ sources are then combined to generate the active LOW **nIRQ** output, which is set LOW when any of the IRQ lines are set HIGH.

FIQ generation

The FIQ logic is similar to the IRQ logic, but in the default system is only one bit wide, and does not have a software programmable source. The **nFIQ** output is directly generated from the single interrupt source bit, using an inverter.

Output data generation

This section is used to decode the current address during a read, and generate the correct data to be driven onto the APB data bus. The address is compared with all of the register addresses, and the value of PRDATANext is set accordingly. This is then stored in the iPRDATA register to help decrease the output propagation time by using a registered output, rather than an output with the combinational delay of the large multiplexor. This register also synchronizes the reading of all raw interrupt inputs to the rising edge of the clock. The **PRDATA** output is then driven by the register.

5.2 Remap and pause controller

The remap and pause controller is an APB slave, providing control of the system boot behavior and low-power *wait for interrupt* mode.

The main sections of this module are:

- defined boot behavior with power-on reset detection
- a wait for interrupt pause mode
- an identification register.

A block diagram of the remap and pause module is shown in Figure 5-6.

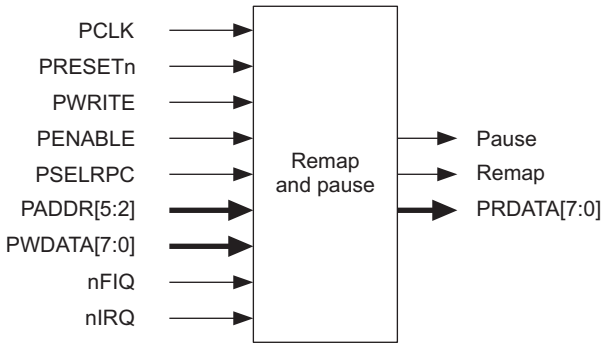


Figure 5-6 Remap and pause module block diagram

5.2.1 Signal descriptions

Table 5-4 describes the APB signals used and produced by the remap and pause controller.

Table 5-4 APB signal descriptions for remap and pause controller

Signal	Type	Direction	Description
PCLK	Peripheral clock	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **PCLK** are used to control transfers.
PRESETn	Peripheral reset	Input	The bus reset signal is active LOW and is used to reset the system.
PENABLE	Peripheral enable	Input	This enable signal is used to time all accesses on the peripheral bus.

Table 5-4 APB signal descriptions for remap and pause controller (continued)

Signal	Type	Direction	Description
PSELRPC	Peripheral slave select	Input	When HIGH, this signal indicates that this module has been selected by the APB bridge. This selection is a decode of the system address bus.
PADDR[5:2]	Peripheral address bus	Input	This is the peripheral address bus, which is used for decoding register accesses. The addresses become valid before **PENABLE** goes HIGH and remains valid after **PENABLE** goes LOW.
PWRITE	Peripheral transfer direction	Input	This signal indicates a write when HIGH and a read when LOW. It has the same timing as the peripheral address bus.
PWDATA[7:0]	Peripheral write data bus	Input	The write peripheral data bus is driven by the bridge at all times.
PRDATA[7:0]	Peripheral read data bus	Output	The read peripheral data bus is driven by this block during read cycles (when **PWRITE** is LOW and **PSELRPC** is HIGH).
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.

5.2.2 Functions and operations of the remap and pause module

The remap and pause control is the combination of four separate functions:

Pause Defines a method of allowing the processor system to enter a low-power, *wait for interrupt* state, when the system does not require the processor to be active.

Identification Provides an indication of the system configuration.

Reset memory map Provides a method of overlaying the system base memory at reset.

Reset status Provides an indication of the cause of the most recent reset condition. A minimum implementation is defined.

5.2.3 Register memory map

The base address of the remap and pause controller memory is not fixed and may be different for any particular system implementation. However, the offset of any particular register from the base address is fixed. Table 5-5 shows the remap and pause controller memory map.

Table 5-5 Memory map of the remap and pause controller APB peripheral

Address	Read location	Write location
`RemapBase + 0x00`	-	Pause
`RemapBase + 0x10`	Identification	-
`RemapBase + 0x20`	-	ClearResetMap
`RemapBase + 0x30`	ResetStatus	ResetStatusSet
`RemapBase + 0x34`	-	ResetStatusClear

5.2.4 Remap and pause register descriptions

Pause

Write-only. Writing to the pause location causes the system to enter a wait for interrupt state, by setting the **Pause** output HIGH.

The exact effect of writing to this location is not defined, but typically this would prevent the processor from fetching further instructions until the receipt of an interrupt or a power-on reset. Further registers may be added to provide more sophisticated power-saving modes.

Identification

Read-only. The identification location provides identification information about the system. Only a single-bit implementation (bit 0) is required, which is used to indicate if there is further ID information:

0 = no further ID information

1 = further ID information is available.

If bit zero of the identification register is set, further bits are required to provide more detailed system identification information.

ClearResetMap Write-only. Writing to the clear reset memory map location changes the system memory map. It changes from that required during boot-up to that required during normal operation. This is done by setting the **Remap** output to HIGH. Once the reset memory map has been cleared and the normal memory map is in use, there is no method of resuming the reset memory map, other than undergoing a power-on reset condition. A typical system implementation is to map the system ROM to location `0x0000 0000` at reset, but to change the memory map after reset, such that RAM is located at location `0x0000 0000` for normal operation. In a system where such remapping does not occur, writing to this register has no effect.

ResetStatus Read-only. The reset status location provides the reset status. Only one bit of this register is defined in this specification and this is bit 0, which provides the power-on reset status. Further bits in the ResetStatus register may be implemented to provide more detailed reset information. The ResetStatus register has a dual mechanism for setting and clearing bits, allowing independent bits to be altered with no knowledge of the other bits in the register. This is done by using the ResetStatusClear and the ResetStatusSet registers.

The single bit defined in this specification is the power-on reset bit, which may be used to determine if the most recent reset was caused by initial power-on, or if a warm reset has occurred:

0 = no POR since flag was last cleared

1 = POR.

ResetStatusClear Write-only. This location is used to clear reset status flags. When writing to this register each data bit that is HIGH causes the corresponding bit in the ResetStatus register to be cleared. Data bits that are LOW have no effect on the corresponding bit in the ResetStatus register.

ResetStatusSet Write-only. This location is used to set reset status flags. When writing to this register each data bit that is HIGH causes the corresponding bit in the ResetStatus register to be set. Data bits that are LOW have no effect on the corresponding bit in the ResetStatus register. The power-on reset status bit (bit 0) cannot be set by software, as it can only be set during a system reset. The extra bits of the register are included in the specification to ensure the reset status functionality can easily be expanded.

5.2.5 System description

The following paragraphs describe how the HDL code for the remap and pause controller module is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of all the registers, inputs and outputs used in the system. This section should be read together with the HDL code.

A basic block diagram of the remap and pause controller module is shown in Figure 5-7.

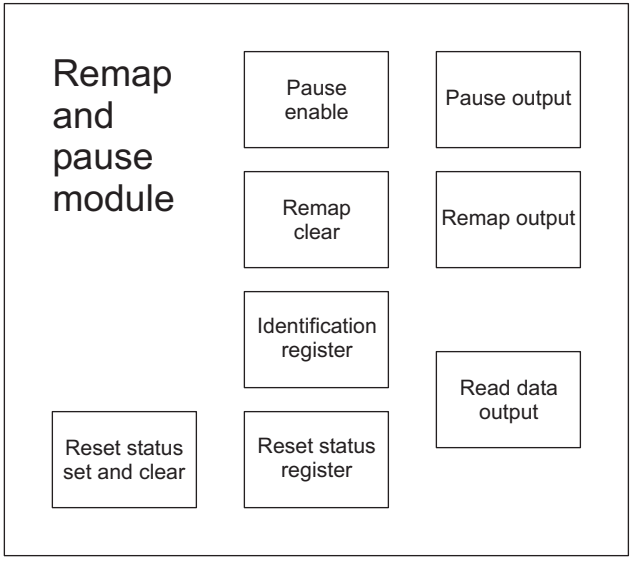


Figure 5-7 Remap and pause module block diagram

The remap and pause controller comprises registers to generate the **Remap** and **Pause** outputs, and logic to allow the reading of the identification and reset status values.

All registers used in the system are clocked from the rising edge of the system clock **PCLK**, and use the asynchronous reset **PRESETn**. The Pause register also uses the two interrupt inputs as asynchronous resets, allowing the value to be cleared while the system is not clocked.

A diagram of the remap and pause HDL file is shown in Figure 5-8.

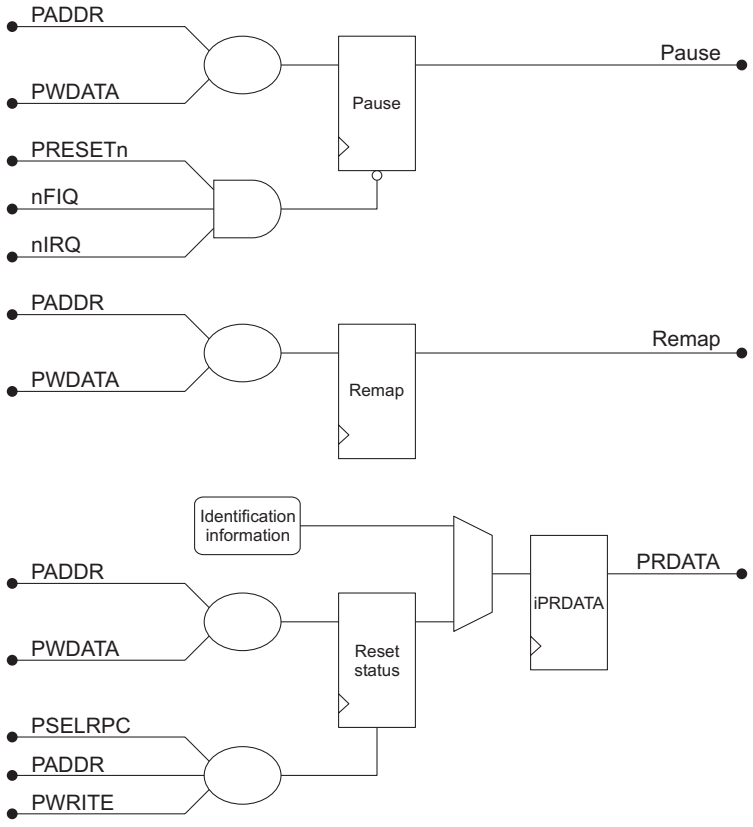


Figure 5-8 Remap and pause module system diagram

The main sections in this module are explained in more detail in the following sections:

- *Constant definitions* on page 5-18
- *ResetStatus value generation* on page 5-18
- *Pause output generation* on page 5-18
- *Remap output generation* on page 5-19
- *Output data generation* on page 5-19.

Constant definitions

The constant IDENTIFICATION holds the identification information about the system. The default setting for this value is all zero. The maximum size for this value is the width of the read and write data buses of the module.

ResetStatus value generation

This register is modified through the ResetStatusSet and ResetStatusClear addresses. When writing to the set location, each data bit that is HIGH sets the corresponding bit in the ResetStatus register. All other bits of the register are unaffected. Each data bit that is set HIGH when writing to the clear location will clear the corresponding bit in the ResetStatus register, leaving all other bits unaffected.

The power-on-reset bit (bit 0) cannot be set by writing to the set location, as it is only set HIGH during system reset. It can be cleared in the same manner as the other register bits.

Pause output generation

A register is used to hold the *wait for interrupt* state value. The **Pause** output is synchronously set HIGH (on the rising edge of **PCLK**) when the Pause location is written to, with any value, and is asynchronously set LOW by **PRESETn**, **nFIQ** or **nIRQ**. Once set HIGH, it can only be set LOW with a reset or an interrupt.

Figure 5-9 on page 5-19 shows the operation of setting and clearing the Pause registered output.

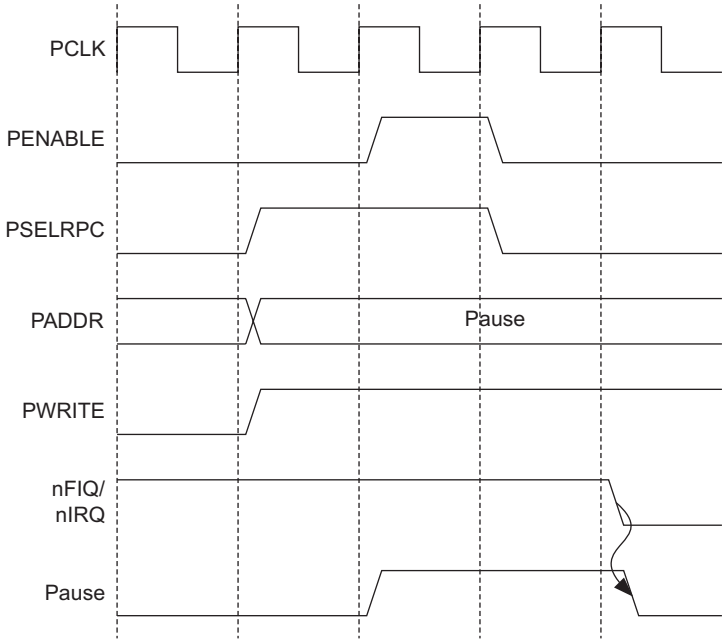


Figure 5-9 Pause signal timing

Remap output generation

This register is used to hold the system memory map state value. The **Remap** output is set LOW on reset, indicating that the reset memory map is in use. It is set HIGH when the ClearResetMap location is written to with any value, indicating that the normal system memory map is in use. Once set HIGH, it can only be set LOW by a system reset.

Output data generation

This section is used to decode the current address during a read, and generate the correct data to be driven onto the APB read data bus. The address is compared with all of the register addresses, and the value of PRDATANext is set accordingly. This is then stored in the iPRDATA register to help decrease the output propagation time by using a registered output, rather than an output with the combinational delay of the large multiplexor. The **PRDATA** output is then driven by the register.

5.3 Timers

The timers module is an APB slave, providing access to two interrupt generating programmable 16-bit *Free-Running decrementing Counters* (FRCs).

The main sections of the timers module are:

- two identical instantiations of a programmable 16-bit free-running counter
- prescale for each counter clock
- interrupt generation based on counter value.

The timers module is shown in Figure 5-10.

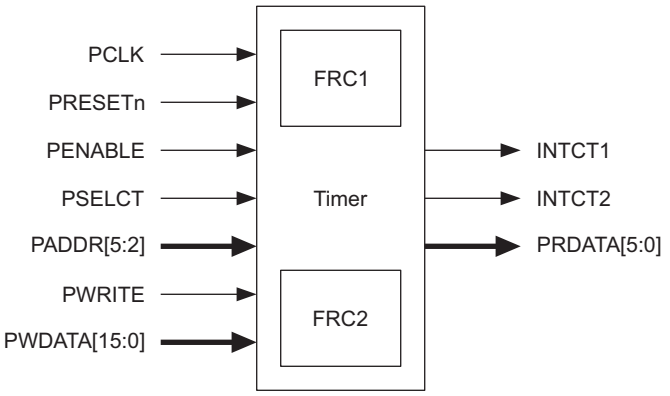


Figure 5-10 Timer module block diagram

5.3.1 Signal descriptions

The two sets of signals associated with the timers module are:

- the external connections to the rest of the EASY world
- the internal connections between the timers module and the two FRC modules.

The signal descriptions for the timers module are listed in Table 5-6 on page 5-21.

Table 5-6 APB signal descriptions for timer

Signal	Type	Direction	Description
PCLK	Peripheral clock	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **PCLK** are used to control transfers.
PRESETn	Peripheral reset	Input	The bus reset signal is active LOW and is used to reset the system.
PENABLE	Peripheral enable	Input	This enable signal is used to time all accesses on the peripheral bus.
PSELCT	Peripheral slave select	Input	When HIGH, this signal indicates that this module has been selected by the APB bridge. This selection is a decode of the system address bus.
PADDR[5:2]	Peripheral address bus	Input	This is the peripheral address bus, which is used for decoding register accesses. The addresses become valid before **PENABLE** goes HIGH and remains valid after **PENABLE** goes LOW.
PWRITE	Peripheral transfer direction	Input	This signal indicates a write when HIGH and a read when LOW. It has the same timing as the peripheral address bus.
PWDATA[15:0]	Peripheral write data bus	Input	The write peripheral data bus is driven by the bridge at all times.
PRDATA[15:0]	Peripheral read data bus	Output	The read peripheral data bus is driven by this block during read cycles (when **PWRITE** is LOW and **PSELCT** is HIGH).
INTCT	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 been decremented to zero.
INTCT2	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 been decremented to zero.

5.3.2 Function and operation of module

Two counters are defined as the minimum provided within a system, although this may easily be expanded. The same principle of simple expansion has been applied to the register configuration, allowing more complex counters to be used.

Two 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.

5.3.3 Timer operation

The timer is loaded by writing to the Load register and, if enabled, counts down to zero. When zero is reached, an interrupt is generated. The interrupt may be cleared by writing to the **Clear** register.

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 Control register.

At any point, the current counter value may be read from the 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 one respectively.

Figure 5-11 on page 5-23 is a block diagram showing timer operation.

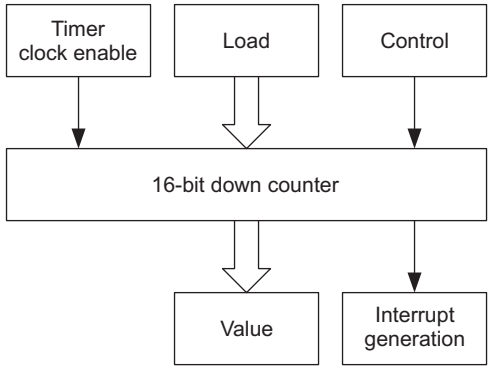


Figure 5-11 Timer operation

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 5-12 shows how the timer clock frequency is selected in the prescale unit.

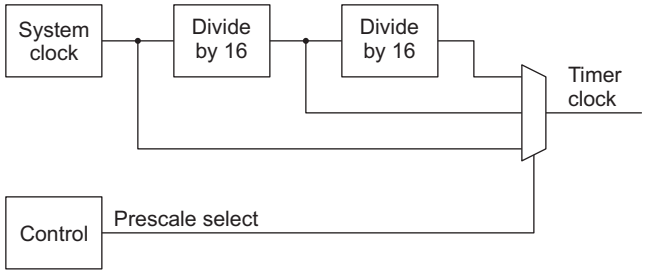


Figure 5-12 Prescale clock enable generation

5.3.4 Register memory map

The base address of the timers module is not fixed and may be different for any particular system implementation. However, the offset of any particular register from the base address is fixed.

Table 5-7 Memory map of the time APB peripheral

Address	Read location	Write location
TimerBase + 0x00	Timer1Load	Timer1Load
TimerBase + 0x04	Timer1Value	-
TimerBase + 0x08	Timer1Control	Timer1Control
TimerBase + 0x0C	-	Timer1Clear
TimerBase + 0x20	Timer2Load	Timer2Load
TimerBase + 0x24	Timer2Value	-
TimerBase + 0x28	Timer2Control	Timer2Control
TimerBase + 0x2C	-	Timer2Clear
TimerBase + 0x10	Timer1Test	Timer1Test
TimerBase + 0x30	Timer2Test	Timer2Test

5.3.5 Timer register descriptions

TimerXLoad Read/write. This register contains the initial value to be loaded into the counter and is also used as the reload value in periodic mode. This register is the same width as the counter (default is 16 bits).

TimerXValue Read-only. This location gives the current value of the counter.

TimerXClear Write-only. Writing to this location clears an interrupt generated by the counter.

TimerXControl Read/write. This register provides enable/disable, mode and prescale configurations for the counter.

Figure 5-13 shows the control register.

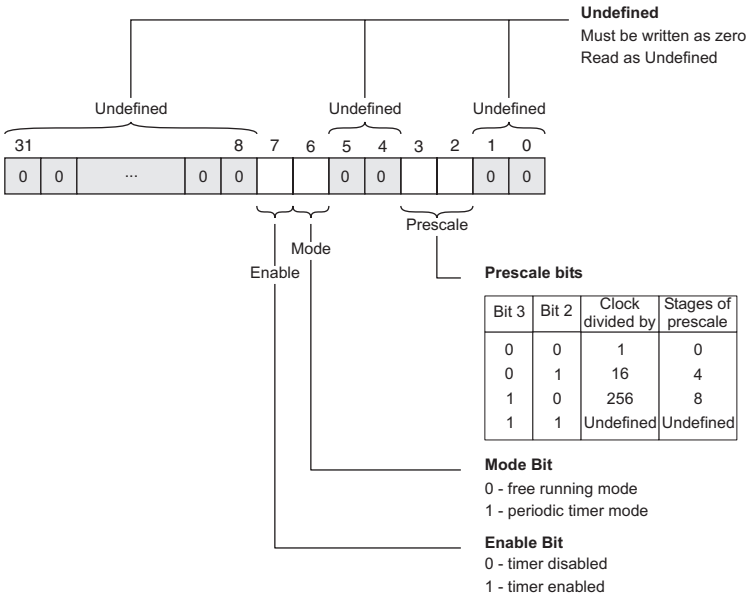


Figure 5-13 The control register

TimerXTest Two special registers are provided for validation purposes, Timer1Test and Timer2Test. These locations should not be accessed during normal system operation.

Both registers are read/write and are 2 bits wide, as shown in Table 5-8.

Table 5-8 Test register bit functions

Bit	Name	Function
0	Test	Counter test mode
1	TestClkSel	Test clock select

The counter test mode bit is stored in a register in both FRCs. The test clock select bit is stored in a single register in the top-level timers module, but can be accessed from either test address.

When the counter test mode bit is set, the selected 16-bit counter is divided into four separate 4-bit counters that continually loop round from 15 to 0. This reduces the testing time needed to ensure that the correct counting sequence is performed. Clearing this bit (default) brings the selected timer back to normal operation.

When the test clock select bit is set in either of the two test registers, a special test clock (NOT **PENABLE** ANDed with **PSELCT**) is fed into the prescale unit instead of the system clock (therefore both counters have to be using the same clock source, either normal or test). Clearing this bit (default) selects the system clock as the prescale clock input (normal operation).

5.3.6 System description

This section describes how the HDL code for the timers module is set out. A basic system block diagram, with information about the main parts of the HDL code, is followed by details of all the registers, inputs and outputs used in this module. This should be read together with the HDL code.

A basic block diagram of the timers module is shown in Figure 5-14.

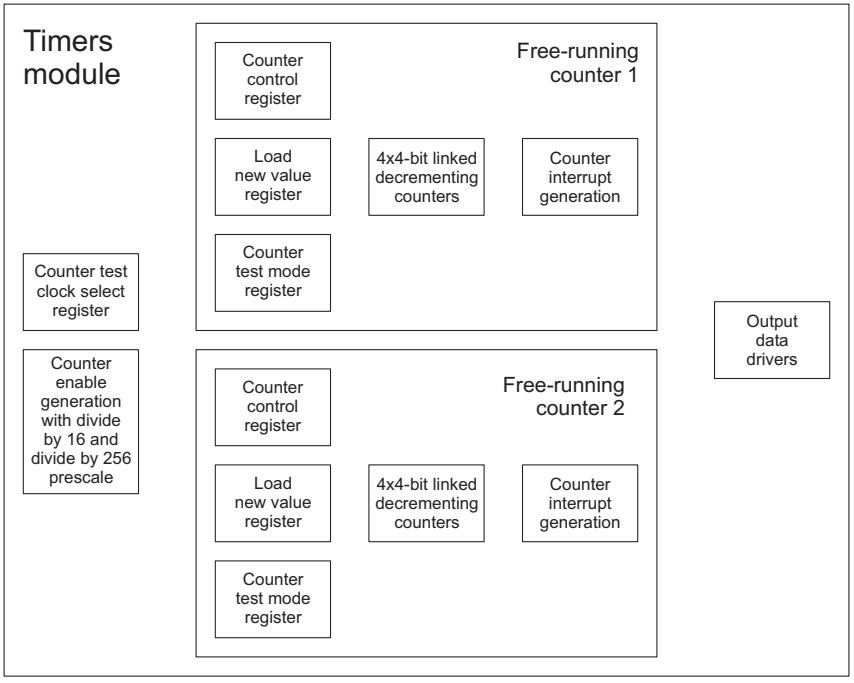


Figure 5-14 Timers module block diagram

The timers module comprises two 16-bit programmable free-running counters, and clock prescale enable generation logic. The free-running counters comprise four linked 4-bit counters, interrupt generation logic and counter control registers.

All registers used in the system are clocked from the rising edge of the system clock **PCLK** and use the asynchronous reset **PRESETN**.

5.3.7 Timer system description

A diagram of the timers module HDL file is shown in Figure 5-15.

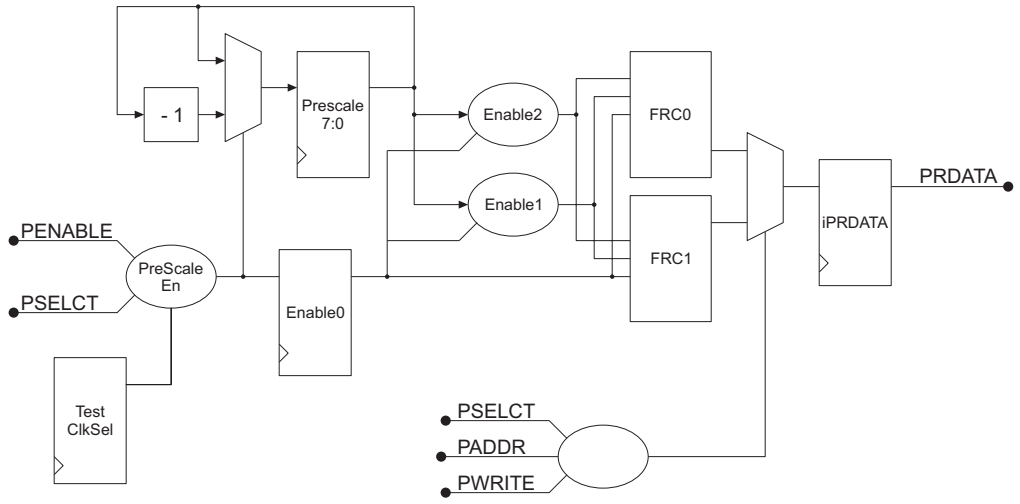


Figure 5-15 Timers module system diagram

The main sections in this module are explained in the following paragraphs:

- *Address decoder*
- *Test clock select generation* on page 5-28
- *Clock prescaler* on page 5-28
- *Output clock enable generation* on page 5-28
- *Output data generation* on page 5-29.

Address decoder

This section is used to generate the **TestSel** signal, which is used to indicate an access to either of the test registers, and the **Frcsel** select lines to the FRCs based on the current address. As there are two instantiations (in the default system) of an identical FRC module, then part of the address decoding must be done at the previous system level.

Test clock select generation

This register is used to store the current value of bit 1 of all counter test registers. A read or write to any of the test register addresses will access this single register.

Clock prescaler

The 8-bit prescale registers are used to generate the two prescale signals of divide by 16 and divide by 256, by decrementing the current value of the registers. The enable signal **PreScaleEn** is used to control the operation of the registers, which by default is always set, but in test clock mode is a combination of **PENABLE** and **PSELCT**, allowing an output clock pulse to be generated for each read or write access to the timers module.

Output clock enable generation

The three different clock enable signals (equivalent to the system clock, the system clock divided by 16, and the system clock divided by 256) enable the timer clocks in the two FRC modules, based on the amount of prescale that is required.

Figure 5-16 and Figure 5-17 on page 5-29 show the timing of these enable signals.

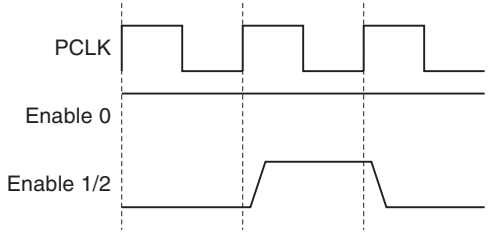


Figure 5-16 Timer module counter enable timing - system clock selected

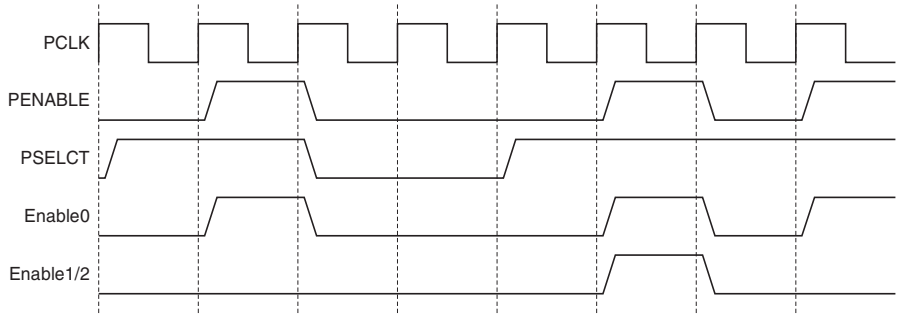


Figure 5-17 Timer module counter enable timing - test clock selected

Output data generation

This section is used to decode the current address during a read, and generate the correct data to be driven onto the APB read data bus. The address is compared with all of the register addresses, and the value of PRDATANext is set accordingly. This is then stored in the iPRDATA register to help decrease the output propagation time by using a registered output, rather than an output with the combinational delay of the large multiplexor. The **PRDATA** output is then driven by the register.

The read data is based on the FRC data outputs, with the local Test Clock Select register output also used when reading from a test location.

5.3.8 FRC system description

Two identical instances of the free-running counter block are included in the timers module.

The basic block diagram of the free-running counter block is shown in Figure 5-18.

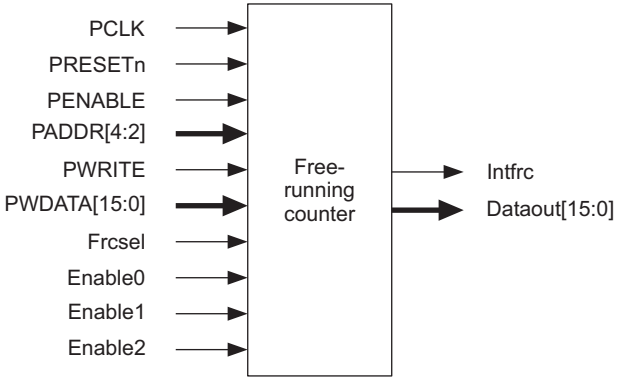


Figure 5-18 FRC module block diagram

5.3.9 FRC signal descriptions

Table 5-9 shows descriptions for the FRC signals.

Table 5-9 Signal descriptions for FRC

Signal	Type	Direction	Description
PCLK	Peripheral clock	Input	Direct connection from timers module.
PRESETn	Peripheral reset	Input	Direct connection from timers module.
PENABLE	Peripheral enable	Input	Direct connection from timers module.
PADDR[4:2]	Peripheral address	Input	Direct connection from timers module.
PWRITE	Peripheral transfer direction	Input	Direct connection from timers module.
PWDATA[15:0]	Peripheral write data bus	Input	Direct connection from timers module.
Frcsel	FRC register select	Input	FRC register select, driven HIGH when a register in this FRC is addressed. There is a select line for each counter in the timers module.
Enable0	Enable prescale 0	Input	Counter clock enable, divide by 1.
Enable1	Enable prescale 4	Input	Counter clock enable, divide by 16.

Table 5-9 Signal descriptions for FRC (continued)

Signal	Type	Direction	Description
Enable2	Enable prescale 8	Output	Counter clock enable, divide by 256.
Intfrc	Interrupt output	Output	Interrupt output from the counter, generated when 16-bit counter reaches zero. There is an interrupt output for each counter in the timers module.
Dataout	Read data output	Output	Read data output used to generate **PRDATA** for register reads. There is a read data output for each counter in the timers module.

Figure 5-19 shows the FRC HDL file.

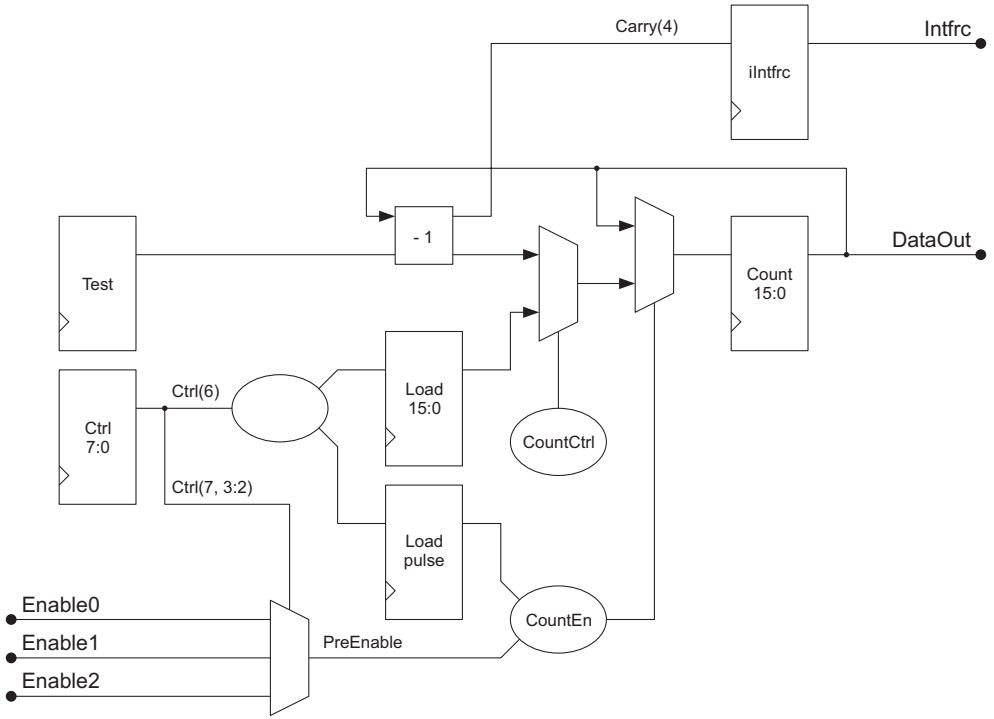


Figure 5-19 FRC module system diagram

The main sections in this module are described in:

- *Control, Test and Load registers*
- *Counter enable selection*
- *16-bit counter*
- *Interrupt generation* on page 5-33
- *Output data generation* on page 5-34.

Control, Test and Load registers

The Control, Test (bit zero only) and Load registers only change when written to, and hold their values at all other times.

Counter enable selection

The enable input to use is selected according to the prescale mode setting in the control registers. The selected input is then used to generate an internal enable, which is also gated with the enable bit of the control registers. An additional signal ensures that the load data value is clocked into the counters when a load operation is performed.

16-bit counter

The counter is split up into four 4-bit parts (nibbles) to allow efficient testing. Each nibble is used to generate a carry signal (when the 4-bit counter overflows), which is passed to the next nibble as an enable. When Counter Test Mode is selected, all carry enable signals are set HIGH, forcing all four nibbles to count at the same time.

The 16-bit counter value is stored in registers, which are enabled using the externally generated counter enable. The input to the registers is normally the output from the four 4-bit decrementers, but when a new value is written to the Load registers, or when the counter reaches zero and periodic mode is set, the current value of the Load registers is stored in the counter registers.

The operation of the counter is shown in Figure 5-20 on page 5-33.

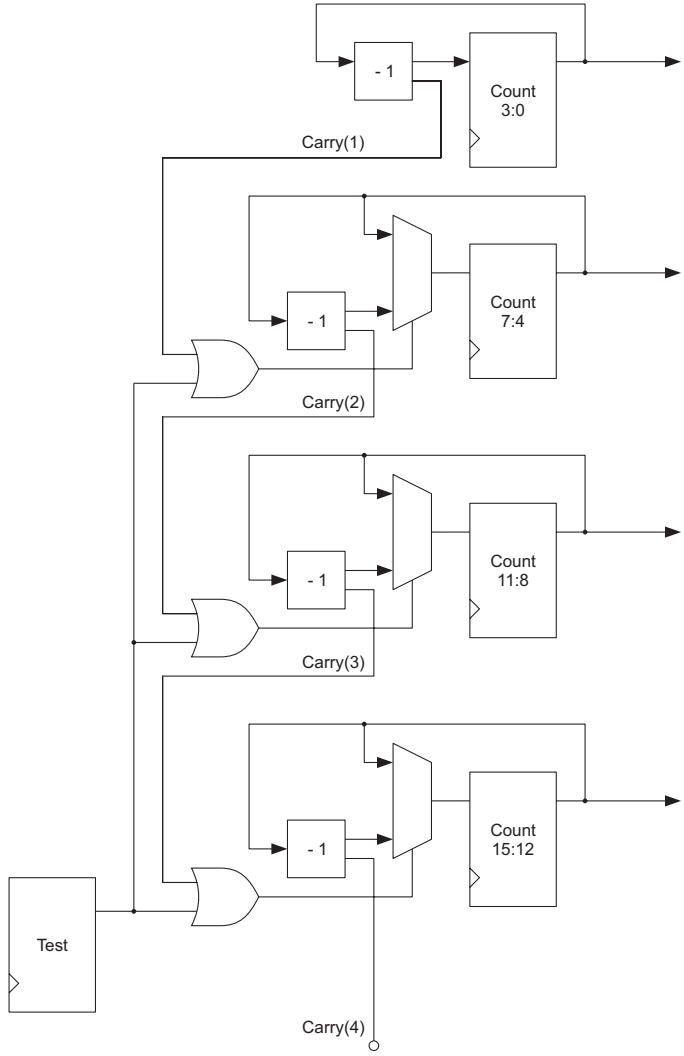


Figure 5-20 FRC module count down diagram

Interrupt generation

An interrupt is generated when the full 16-bit counter reaches zero, and is only cleared when the TimerClear location is written to. A register is used to hold the value until the interrupt is cleared. The most significant carry bit of the counter is used to detect the counter reaching zero.

Output data generation

The current address is used to generate the internal read data value for the Test, Load, Value and Control locations. As the Test and Control registers are not 16-bits, then the read values are padded out with the Load register value, minimizing the number of output changes when different registers are read.

This read data value is then passed to the timers module, and then driven onto the APB read data bus.

5.4 Peripheral to bridge multiplexor

The peripheral to bridge multiplexor module is used to connect the read data outputs of the peripheral bus slaves to the peripheral bus bridge module, using the **PSELx** select signals to select the bus slave outputs to use. Figure 5-21 shows an interface diagram for the peripheral to bridge multiplexor module.

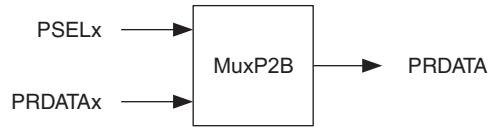


Figure 5-21 Peripheral to bridge multiplexor module interface diagram

This module is a simple multiplexor, with the read data buses from all peripheral bus slaves as the inputs, using the slave select bridge outputs as the select inputs, with a single read data bus as the output to the bridge module. When slaves are added to the system or removed, the input connections to this module must be altered to account for the changes.

5.4.1 Signal descriptions

Table 5-10 shows the signal descriptions for the peripheral to bridge multiplexor module.

Table 5-10 Signal descriptions for peripheral to bridge multiplexor module

Signal	Type	Direction	Description
PSELx	Slave select	Input	Each APB slave has its own slave select signal, and this signal indicates that the current transfer is intended for the selected slave.
PRDATAx[31:0] **PRDATA[31:0]**	Read data bus	Input/ output	The read data bus is used to transfer data from bus slaves to the bridge during read operations.

5.4.2 Function and operation of module

The peripheral to bridge multiplexor controls the routing of read data from the peripheral bus slaves to the bridge. The bridge determines which is the currently selected slave, and the multiplexor is used to connect the output of the selected slave to the input of the bridge.

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.

5.4.3 System description

The following paragraphs give a description of how the HDL code for the peripheral to bridge multiplexor is set out. A simple system block diagram, with information about the main parts of the HDL code, is followed by details of the registers, inputs, and outputs used in the module. This part should be read together with the HDL code.

Figure 5-22 shows the peripheral to bridge module block diagram.

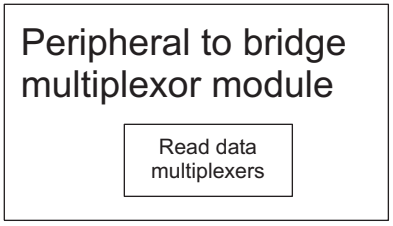


Figure 5-22 Peripheral to bridge multiplexor module block diagram

The peripheral to bridge multiplexor module is comprised of a set of multiplexors for the slave read data.

A diagram of the peripheral to bridge multiplexor HDL file is shown in Figure 5-23.

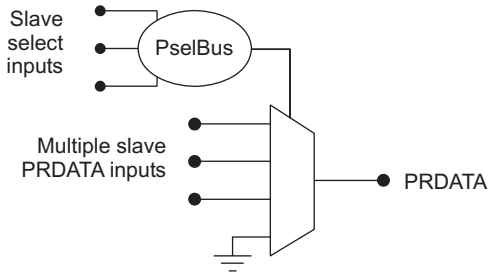


Figure 5-23 Peripheral to bridge multiplexor module system diagram

To allow the use of case statements for the multiplexors, the **PSEL** slave select inputs are combined to create a multi-bit bus signal. This bus is then used as the select control on the read data multiplexor.

One input to the PRDATA multiplexor is tied LOW, so that when no peripheral slaves are selected, no read data appears on **PRDATA**.

Chapter 6
Behavioral Modules

This chapter describes the behavioral modules found in the *Example AMBA SYstem* (EASY). The behavioral modules are only available for use during system simulation, as they all read in or generate locally stored data files. This chapter contains the following sections:

6.1 External RAM

The external RAM module is a simple model of a 32K x 8 off-chip SRAM, which can be initialized with data from a local file.

Figure 6-1 shows the external RAM module interface.

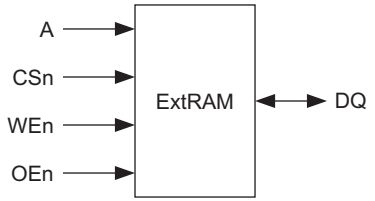


Figure 6-1 External RAM module interface diagram

The main sections of this module are:

- memory initialization from local data file
- memory read and write from external bus.

6.1.1 Signal descriptions

Table 6-1 shows the signal descriptions for the external RAM module.

Table 6-1 Signal descriptions for the external RAM module

Signal	Type	Direction	Description
A[14:0]	External address	Input	The external address input.
DQ[7:0]	External data I/O	Input/ output	The external data bus, sampled during write transfers and driven during read transfers.
CSn	Chip enable	Input	When LOW this signal indicates that the chip has been selected and should respond to the current transfer.
WEn	Write enable	Input	When LOW this signal indicates a write transfer.
OEn	Output enable	Input	When LOW this signal indicates a read transfer, and enables the module to drive data onto **DQ**.

6.1.2 User-defined settings

Table 6-2 shows the user-defined settings for the external RAM module.

Table 6-2 User-defined settings for the external RAM module

Signal	Type	Default setting	Description
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 allow all memory to be addressed.

6.1.3 Function and operation of module

Operations described are:

- *Memory initialization from local data file*
- *Memory read and write from external bus*.

Memory initialization from local data file

On simulation initialization, the external RAM module loads in data from the file specified in the instantiating top-level memory module. This must be stored as a two-hex character per line data file, which cannot contain more data than the model will support. An example file `ram.dat` is shown in Example 6-1.

Example 6-1

```
00
01
0F
F7
```

The default configuration for the external RAM modules is in groups of four, which are used to allow memory accesses of full 32-bit words, with a byte stored in each memory module.

Memory read and write from external bus

The external RAM is accessed by transfers through the static memory interface module, allowing both reads from memory and writes to memory. These are performed as 32-bit word transfers, with each byte connected to one of the four memory models.

Refer to *Static memory interface* on page 4-53 in Chapter 4 *AHB Modules*, for timing diagrams showing read and write transfers to external memory.

6.2 External ROM

The external ROM module is a simple model of a 16K x 8 off-chip EPROM, which can be initialized with data from a local file.

Figure 6-2 shows the external ROM module interface.

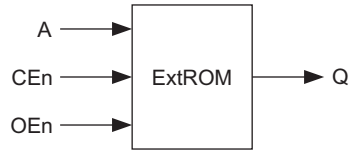


Figure 6-2 External ROM module interface diagram

The main sections of this module are:

- memory initialization from local data file
- memory read from external bus.

6.2.1 Signal descriptions

Table 6-3 shows signal descriptions for the external ROM module.

Table 6-3 Signal descriptions for the external ROM module

Signal	Type	Direction	Description
A[13:0]	External address	Input	The external address input.
Q[7:0]	External data out	Output	The external data bus, driven during read transfers.
CEn	Chip enable	Input	When LOW this signal indicates that the chip has been selected and should 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**.

6.2.2 User-defined settings

Table 6-4 shows user-defined settings for the external ROM module

Table 6-4 User-defined settings for the external ROM module

Signal	Type	Default setting	Description
ROMDEPTH	Memory depth	16	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 allow all memory to be addressed.

6.2.3 Function and operation of module

Operations described are:

- *Memory initialization from local data file*
- *Memory read from external bus*.

Memory initialization from local data file

On simulation initialization, the external ROM module loads in data from the file specified in the instantiating top-level memory module. This must be stored as a two-hex character per line data file, which cannot contain more data than the model will support. An example file `rom.dat` is shown in Example 6-2.

Example 6-2

```
00
01
0F
F7
```

The default configuration for the external ROM modules is in groups of four, which are used to allow memory accesses of full 32-bit words, with a byte stored in each memory module.

Memory read from external bus

The external ROM is accessed by transfers through the static memory interface module, allowing reads from memory. These are performed as 32-bit word transfers, with each byte connected to one of the four memory models.

Refer to *Static memory interface* on page 4-53 in Chapter 4 *AHB Modules*, for timing diagrams showing read and write transfers from external memory.

6.3 Internal RAM

The internal RAM is a simple little-endian model of a 1KB x 32 on-chip SRAM, which can be initialized with data from a local file. As this module is connected to the main system bus, there are AHB and ASB versions available. The AHB version is shown in Figure 6-3.

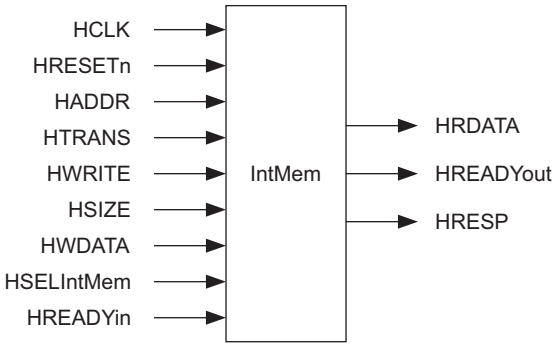


Figure 6-3 AHB internal RAM module interface diagram

The main sections of this module are:

- memory initialization from local data file
- memory read and write from system bus.

6.3.1 AHB signal descriptions

Table 6-5 shows signal descriptions for the AHB internal RAM module.

Table 6-5 Signal descriptions for the AHB internal RAM module

Signal	Type	Direction	Description
HCLK	Bus clock	Input	This clock times all bus transfers.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HADDR[31:0]	Address bus	Input	The 32-bit system address bus.
HTRANS[1:0]	Transfer type	Input	Indicated the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE	Transfer direction	Input	When HIGH this signal indicates a write transfer, and when LOW, a read transfer.

Table 6-5 Signal descriptions for the AHB internal RAM module (continued)

Signal	Type	Direction	Description
HSIZE[2:0]	Transfer size	Input	Indicates the size of the transfer, which is typically byte (8-bit), halfword (16-bit) or word (32-bit). The protocol allows for larger transfer sizes up to a maximum of 1024 bits.
HWDATA[31:0]	Write data bus	Input	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HSELIntMem	Slave select	Input	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.
HRDATA[31:0]	Read data bus	Output	The read data bus is used to transfer data from bus slaves to the bus master during read operations. A minimum data bus width of 32 bits is recommended. However this may easily be extended to allow for higher bandwidth operation.
HREADYin HREADYout	Transfer done	Input / output	When HIGH the **HREADY** signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer.
HRESP[1:0]	Transfer response	Output	The transfer response provides additional information on the status of a transfer. This module will always generate the OKAY response.

6.3.2 User-defined settings

Table 6-6 shows user-defined settings for the external RAM module

Table 6-6 User-defined settings for the external RAM module

Signal	Type	Default setting	Description
MemSize	Memory size	1	This sets the memory size in KB.
FileName	Input filename	`intram.dat`	This points to the local input data file that is read in after reset.

.

6.3.3 Function and operation of module

Operations described are:

- *Memory initialization from local data file*
- *Memory read and write from system bus* on page 6-11.

Memory initialization from local data file

On simulation initialization, the internal RAM module loads in data from the file specified in the `FileName` setting. This must be stored as an 8-character Verilog `$readmemh` format data file (for both VHDL and Verilog format models), which cannot contain more data than the model will support. Address lines (starting with `@`) and single line comments (starting with `//`) are valid, but all other non-value characters are not allowed. 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. An example `intram.dat` file is shown in Example 6-3.

Example 6-3

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

The internal RAM module stores data as 32-bit words, and in default configuration is 256 words deep, which is equivalent to 1KB. This is only accessible once the normal memory map is in use (**Remap** set HIGH), and occupies the address range from `0x0000 0000` to `0x0000 03FF`. If the size of the internal memory is modified, then the address range that it occupies will also change. This will require the system decoder to be updated so that it only selects the internal RAM module over the correct address range.

Memory read and write from system bus

The internal RAM module is accessed by standard system bus transfers, allowing both reads from memory and writes to memory. These can be performed as 32-bit word, 16-bit halfword or 8-bit byte transfers. Each byte lane of the transfer is treated separately, so a byte write to byte zero will not alter the values stored in the other three bytes at that word address. Data reads are all treated the same, and the full 32-bit word at the selected word address will be driven out onto the system data bus.

All transfers are performed with zero wait states. An ERROR response is never generated.

6.4 Test interface driver

The test interface driver (Ticbox) is an external module which 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 should be familiar with AMBA and its test interface protocol. If not, refer to the *AMBA Specification* for further information. Figure 6-4 shows an interface diagram of the ticbox module.

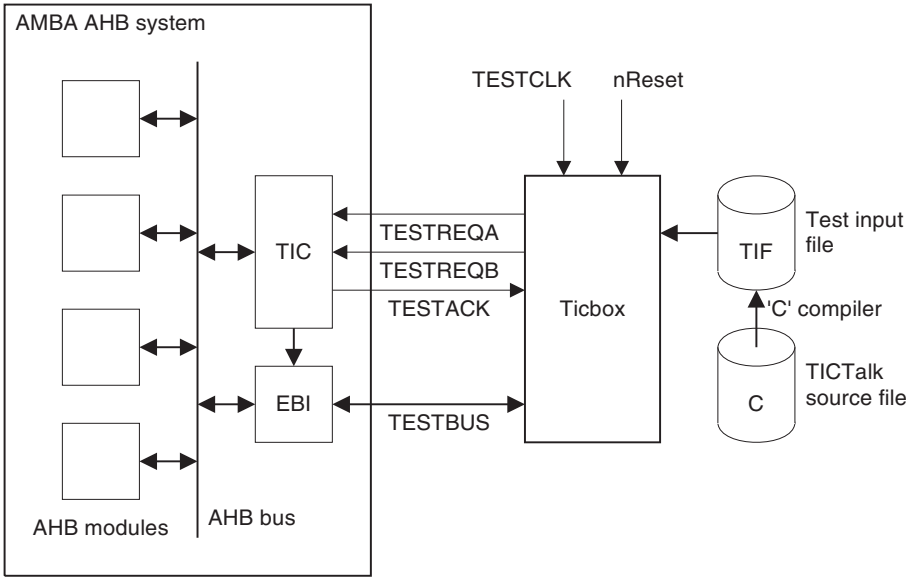


Figure 6-4 Ticbox module interface diagram

The main sections of this module are:

- the input file reader
- output vector generation
- read data expected value checking.

6.4.1 Signal descriptions

Table 6-7 shows signal descriptions for the Ticbox module

Table 6-7 Signal descriptions for the Ticbox module

Signal	Type	Direction	Description
TESTCLK	Test mode clock	Input	This is the system clock **HCLK** in test mode. All the test interface transactions are timed using this signal.
nReset	External reset	Input	Active LOW external reset input. Used to control the operation of the Ticbox module.
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. Refer to 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**.
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.

.

6.4.2 User-defined settings

Table 6-8 shows user-defined settings for the Ticbox module.

Table 6-8 User-defined settings for the Ticbox module

Name	Type	Default setting	Description
FileName	Input filename	`infile.tif` (VHDL) `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 is used to control the operation of the module when a read error is detected. When set FALSE, a warning message will be displayed showing the read error, and if set TRUE, the simulation will be halted when a read error is detected.
Verbosity	Comment display	TRUE	Controls the displaying of input vector file comments. When set TRUE, comments are displayed, and when set FALSE, comments are not displayed. This does not affect the displaying of other system messages.

6.4.3 Function and operation of module

The AHB and ASB versions of the Ticbox are internally different, even though the external ports are identical. Due to the pipelined nature of the AHB, the read data becomes available one cycle later than the ASB read data relative to the generation of the read test vector, so the AHB Ticbox includes an extra delay stage in the read data checking logic.

Once the external system reset input has been de-asserted, the Ticbox requests access to the system. This is done by asserting **TESTREQA** HIGH and **TESTREQB** LOW. The *Test Interface Controller* (TIC) then indicates when test mode has been entered by asserting **TESTACK** HIGH. Once 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 once 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.

A typical simulation output display while running a TIC program is shown in Example 6-4 on page 6-15.

Example 6-4

```
#    Time: 2603 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Addressing location 80000614

#    Time: 2703 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Writing data 00000005

#    Time: 3003 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Addressing location 80000618

#    Time: 3103 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Reading. Expected: 00000010. Mask 0000003F

#    Time: 3403 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Addressing location 8000061c

#    Time: 3703 ns Iteration: 0 Instance:/u_ticbox
# ** Warning: Error on vector read. Expected: 00000010 Actual: 00000011 Mask:
0000003F
#    Time: 3753 ns Iteration: 0 Instance:/u_ticbox

#    Time: 4003 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Addressing location 80000584

#    Time: 4303 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Writing data 00000000

#    Time: 4603 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Addressing cycle at end

#    Time: 4903 ns Iteration: 0 Instance:/u_ticbox
# ** Note: ; Exiting Test Mode

#    Time: 5203 ns Iteration: 0 Instance:/u_ticbox
# ** Failure: Vector run completed: halting simulation
#    Time: 77703 ns Iteration: 0 Instance:/u_ticbox
# Break at ticbox.vhd line 288
```

In Example 6-4 on page 6-15 you will note that a read error has occurred, but the error message is broadcast later in the simulation. This is because there are a number of clock cycles between when the read is requested, and when the information is sampled by the Ticbox to be compared with the expected value. The example simulation has been run without HaltOnMismatch set, and therefore the program does not stop after the error has been detected. Verbosity is set, as all TIF vector comments have been displayed in the simulation output.

6.4.4 TICTalk command language

TICTalk is a very simple set of commands that allows the development of validation programs for the AMBA blocks. The TICTalk language is a small library of C functions. Once a TICTalk program is compiled and run, it produces a test input file in what is called the *TIC Interface Format* (TIF) which may be applied using the Ticbox module to test the desired block.

The AMBA test interface is able to perform the following actions:

- address vector
- write vector
- burst of write vectors
- read vector
- burst of read vectors
- change from write to read and read to write.

The TICTalk language performs these actions by combining together a number of basic commands. These commands are described in the following sections.

6.4.5 TICTalk commands

The basic TICTalk commands are described in the following sections:

- *Write address vector (A)* on page 6-17
- *Write test vector (W)* on page 6-17
- *Read test vector (R)* on page 6-17
- *Burst read test vector (B)* on page 6-17
- *Repeat last command (L)* on page 6-17
- *Include the string message into the TIF (C)* on page 6-17
- *Exit test mode (E)* on page 6-17.

Write address vector (A)

The `A(int32 address_vector)` command is used to address a new location in the system. It will always be followed by a write test vector, or a read test vector command in order to perform the required action (write or read data) at that location.

Write test vector (W)

The `W(int32 write_vector)` command generates a data vector write. It can be used after an address vector (single write), another write test vector (burst write) or a read test vector (change from reads to writes).

Read test vector (R)

The `R(int32 expected_value, int32 mask_value)` command generates a data vector read. The read value is masked with the specified `mask_value` and compared with the `expected_value`. If the comparison is false, an error message will be broadcast. It can be used after an address vector (single read), or a write test vector (change from writes to reads), and to indicate the last read on a burst, but it cannot be used after another read test vector. To signal a burst sequence of reads, the burst read vector command should be used instead.

Burst read test vector (B)

The `B(int32 expected_value, int32 mask_value)` command is similar to the read test vector. The only difference is that it can only be used if the next action is another read. This is because, in this case, a change of bus direction is not needed. Otherwise the function performed is the same.

Repeat last command (L)

The `L(int32 number_of_loops)` command signals that the last action should be repeated the specified number of times. This is useful when, for example, a burst of reads or writes from the same address location needs to be performed.

Include the string message into the TIF (C)

The `C(char * message)` command is used to add extra simulation comments.

Exit test mode (E)

The `E()` command should always be used at the end of a program so the Ticbox can signal the end of the test.

6.4.6 Programming with TICTalk commands

The possible combinations that are available when using the TICTalk commands are:

Single writes The command sequence will be: `A-W A-W A-W`, and so on.

Single reads The command sequence will be: `A-R A-R A-R`, and so on.

Burst of writes The command sequence will be: `A-W-W-W`, and so on.

If the value to be written is always the same, the command sequence could also be `A-W-L`, specifying on the `L` command the number of writes required.

Burst of reads This is a special case. After the `A` command, `B` (burst read vector) should be used on consecutive reads, and only on the last read of the burst do we apply the `R` command. Therefore the sequence will be: `A-B-B-B-R A-B-B-....-B-R`, and so on.

If the value to be read is expected always to be the same, or there is no need to check it against an expected value, the sequence could also be `A-B-L-R`, with the `L` command specifying the number of reads required.

Change from read to write

This change can only be made after a `R` command (`R-W`), and not after a `B` command.

Change from write to read

If the change is for a single read, the sequence `W-R` is used. On the other hand if the change is for a read burst, the `W-B` sequence is used (`W-B-B-...-B-R`).

6.4.7 The TICTalk file

An example C program using the TICTalk commands is shown in Example 6-5.

Example 6-5

```
#define CT1Load     Counter_Base + 0x00
#define CT1Value    Counter_Base + 0x04
#define CT1Control  Counter_Base + 0x08
#define CT1Clear    Counter_Base + 0x0C
#define CT1Test     Counter_Base + 0x10

#define MaskAll     0x00000000
#define NoMask      0xFFFFFFFF
#define MaskControl 0x000000CC
#define MaskValue   0x0000FFFF
#define DUMMY       0x12345678

#include “header.h”
#include “ticmacros.h”

int main()
{
  A(CT1Load)
  W(0x55555555)
  A(CT1Control)
  W(0x000000C0) /* Counter Enabled, Periodic Mode, Prescale 0 */
  A(CT1Value)
  R(0x55555547, MaskValue)
  A(CT1Load)
  W(0xDADADADA)
  B(0xDADADADA, MaskValue)    /* Read CT1Value */
  R(0x000000C0, MaskControl)  /* Read CT1Control */
  A(CT1Value)
  R(0xAAAAAAB8, MaskAll)
  W(0x000000C4)          /* Write to CT1Control */
  W(DUMMY)               /* Write to CT1Clear   */
  L(5)                   /* Repeat last write 5 times */
  E()
}
```

Example 6-5 on page 6-19 shows the TICTalk commands accept 32-bit integers as arguments. These can be specified using the `#define` directive, immediate values or normal C variables. This C-like approach provides the flexibility to develop more elaborate tests and new extended functions. For example, the basic commands could be used to build a pair of functions for reading and writing vectors that automatically take care of bus turnaround and address vectors.

The `ticmacros.h` file includes all the macro definitions for each command. These macros are expanded to generate a test input file in a format that can be read by the Ticbox.

The `header.h` file contains the base address definitions for the different blocks in the system. This is where the `Counter_Base` constant should be defined. This ensures portability of the test program to other systems with different peripheral address mapping.

6.4.8 Generating a test input format file

To generate a TIF file, the TICTalk program should be C compiled (using `gcc` for example) in the following way:

```
gcc -ansi source_file ticmacros.c -o object_file
```

Afterwards the object_file should be run and its output redirected to a file with the same name as the generic variable `FileName` defined in the Ticbox, for example:

```
object_file > infile.tif
```

This output file should then be copied or linked to the directory where the Ticbox simulation model exists.

6.4.9 TIF format file

The TIF file is very similar to the TICTalk file as shown in Example 6-6 on page 6-21, with the difference that all the constant definitions have been substituted with their hexadecimal values and each line reflects a single test cycle. The previous example compiled and executed will output the following TIF. Lines preceded with a semicolon (;) are comments that the simulator will print on the screen while the test is being executed.

Example 6-6

```
; Addressing location 84000000
A 84000000
; Writing data 55555555
W 55555555
; Addressing location 84000008
A 84000008
; Writing data 000000C0
W 000000C0
; Addressing location 84000004
A 84000004
; Reading. Expected: 55555547. Mask: 0000FFFF
R 55555547 0000FFFF
A ZZZZZZZZ
; Addressing location 84000000
A 84000000
; Writing data DADADADA
W DADADADA
; Reading. Expected: DADADADA. Mask: 0000FFFF
R DADADADA 0000FFFF
; Reading. Expected: 000000C0. Mask: 000000CC
R 000000C0 000000CC
A ZZZZZZZZ
; Addressing location 84000004
A 84000004
; Reading. Expected: AAAAAAB8. Mask: 00000000
R AAAAAAB8 00000000
A ZZZZZZZZ
; Writing data 000000C4
W 000000C4
; Writing data 12345678
W 12345678
; Looping for 5 cycles
L 5
; Addressing cycle at end
A 00000000
; Exiting Test Mode
E ZZZZZZZZ
```

6.4.10 SIM format file

The Verilog Ticbox requires the input file to be in the SIM format, which is formatted as a Verilog `include input file, with each test vector calling a task in the Verilog Ticbox behavioral module.

A SIM file is generated from a TIF file using the conversion script tif2sim in the following manner:

```
tif2sim infile.tif > infile.sim
```

Comments use the Verilog style double slash (//), and due to the properties of Verilog `include files, are not displayed in the simulation output. The Verilog Ticbox directly generates the simulation comments based on the test vector that is being run.

The TIF file above is shown in Example 6-7 in SIM format:

Example 6-7

```
// Addressing location 84000000
A(32'h84000000);

// Writing data 55555555
W(32'h55555555);

// Addressing location 84000008
A(32'h84000008);

// Writing data 000000C0
W(32'h000000C0);

// Addressing location 84000004
A(32'h84000004);

// Reading. Expected: 55555547. Mask: 0000FFFF
R(32'h55555547, 32'h0000FFFF);
A(32'hZZZZZZZZ);

// Addressing location 84000000
A(32'h84000000);

// Writing data DADADADA
W(32'hDADADADA);

// Reading. Expected: DADADADA. Mask: 0000FFFF
R(32'hDADADADA, 32'h0000FFFF);
```



```
// Reading. Expected: 000000C0. Mask: 000000CC
R(32'h000000C0, 32'h000000CC);
A(32'hZZZZZZZZ);

// Addressing location 84000004
A(32'h84000004);

// Reading. Expected: AAAAAAB8. Mask: 00000000
R(32'hAAAAAAB8, 32'h00000000);
A(32'hZZZZZZZZ);

// Writing data 000000C4
W(32'h000000C4);

// Writing data 12345678
W(32'h12345678);

// Looping for 5 cycles
L(32'd5);

// Addressing cycle at end
A(32'h00000000);

// Exiting Test Mode
E(32'hZZZZZZZZ);
```

6.5 Tube

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

Figure 6-5 shows the tube module interface.

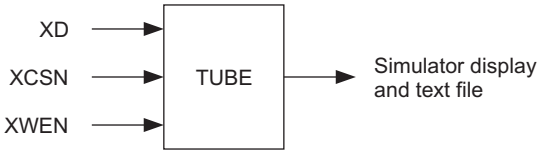


Figure 6-5 Tube module interface diagram

The main sections of this module are:

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

6.5.1 Signal descriptions

Table 6-9 shows signal descriptions for the tube module.

Table 6-9 Signal descriptions for the tube module

Signal	Type	Direction	Description
XD[31:0]	External data	Input	This is the external data bus, which is sampled by this module during write transfers.
XCSN[3:0]	External chip select	Input	These signals are active LOW chip enables.
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.

6.5.2 User-defined settings

Table 6-10 shows user-defined settings for the tube module.

Table 6-10 User-defined settings for the tube module

Signal	Type	Default setting	Description
OutFile	Output filename	`Tube.txt`	This points to the local output data file that is written to during simulation when messages are passed to the tube.

6.5.3 Function and operation of module

The tube module is used to perform program message and simulation termination control. It acts as a one-way communications port through which ASCII information can be passed.

Messages are written, one byte at a time, to the tube model location. In the default system this is address range `0x2000 0000` to `0x2FFF FFFF`, detected by the model using the external enable **XCSN[2]**. These bytes are buffered until a terminating control character is written to the tube, or the buffer overflows (default buffer length is 80 characters). The message is then printed by the simulator, and written to the output text file. An example message is:

```
# ** Note: TUBE: Hard Reset
```

In this example the message `Hard Reset` has been passed to the tube. The program running on the microcontroller can also terminate simulation by writing a control character to the tube with no message to produce the following assertion:

```
# ** Failure: TUBE: Program exit
```

All user messages sent to the simulator display are also recorded in the output text file.

The tube module only accepts the ASCII control characters shown in Table 6-11.

Table 6-11 Valid tube ASCII control characters

ASCII character	Decimal value	Tube function
Control D (^D)	04	Exit test
Linefeed	10	Print output
Carriage return	13	Print output

Other standard alphanumeric characters will be stored in the buffer until displayed. The values for commonly used display characters are shown in Table 6-12.

Table 6-12 Commonly used ASCII alphanumeric characters

ASCII character	Decimal value
0-9	48-57
a-z	97-122
A-Z	65-90
space	32
_	95
#	35

Chapter 7
Designer's Guide

This chapter briefly describes adding new modules to the EASY microcontroller. Since AMBA has been designed specifically to be modular, little change needs to be made to other elements when a component is added or removed. The chapter contains the following sections:

7.1 Adding bus masters

For bus masters, the arbiter is the only block that requires changes.

The arbiter currently has facilities for up to two more masters without any modification. A new master needs to be connected to the appropriate **HBUSREQx** and **HGRANTx** signals. This can be done by altering the top-level HDL file, which connects all AHB modules together.

Note

If a system requires more than four masters, the arbiter HDL file will also need to be modified.

7.1.1 Arbiter modifications

When modifying the arbiter the following rules must be followed:

- The ARM core should be the default master (granted on reset), and granted when no masters are requesting the bus.
- The *Test Interface Controller* (TIC) should have the highest priority (to allow test access under all conditions).
- Only one master should tie its **HBUSREQx** permanently HIGH.
- Currently the ARM bus master always asserts **HBUSREQx**, thus no other bus master should constantly request the bus. Consequently the ARM must be the lowest priority master, as masters of lower priority than the ARM will never get granted.

If more sophisticated *round-robin* type arbitration schemes are used, the latter point will no longer be valid. Alternative arbitration schemes are not considered further in this document.

7.1.2 Bus master requirements

New designs of bus master must drive all the relevant signals at appropriate times. For more information consult the *AMBA Specification*.

7.2 Adding AHB slaves

When a slave is added, the decoder needs to be modified. This will add an **HSELx** signal for the new slave. The central slave to master multiplexor must also have extra connections added for the new slave.

7.2.1 AHB slave modifications

When adding new AHB slaves, care should be taken to:

- plan the slave position in the memory map
- consider any issues concerning the remapping of memory to allow the external boot ROM to appear at location zero
- decode as few address lines as possible, to keep the slave address decode section gate count low
- ensure that all areas of address space have one, and only one, slave selected.

The default slave must be set so that all holes in the memory map are filled. If any holes are left without a slave to drive the **HREADY** line, then any accesses to this area will cause the system to lock, with **HREADY** staying LOW until a system reset.

7.2.2 Slave requirements

These vary according to the function of the slave. Special cases like external bus interfaces (which must also consider the requirements of the TIC), or the AHB to APB bridge interface have more complex requirements. For more information consult the *AMBA Specification*.

7.3 Adding APB peripherals

When adding a peripheral, the APB bridge needs to be modified. This will add a new **PSELx** signal for the new peripheral. The central peripheral to bridge multiplexor must also have extra connections added for the new peripheral.

7.3.1 APB bridge modifications

When adding new **PSELx** lines similar steps should be taken to those outlined in *AHB slave modifications* on page 7-3, although reset memory map will not be an issue for APB peripherals.

7.3.2 Peripheral requirements

When designing APB peripherals, ensure that the resulting hardware has a low power consumption. The following guidelines should be followed where possible:

- Do not use **PCLK** in peripherals unless absolutely necessary as its use will dramatically increase power consumption.
- Ensure that peripherals cannot drive **PRDATA[31:0]** during reset (by including a **PRESETn** term on the output enable control).

Designers familiar with conventional circuits connected to free-running clocks may find this design approach difficult. However, it will result in small circuits with low power consumption.

Index

D

E

F

G

I

M

P

R

S

T

W

