ASB Example AMBA™ SYstem

Technical Reference Manual

ARM®

ARM DDI 0138D

ASB Example AMBA SYstem
Technical Reference Manual

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
October 1998	A	Non-Confidential	First release.
July 1999	B & C	Non-Confidential	Note: Issues B and C were not released.
August 1999	D	Non-Confidential	Name change to ASB EASY.

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
ASB Example AMBA SYstem Technical Reference Manual

Preface

This preface introduces the *ASB Example AMBA SYstem Techincal Reference Manual*. It contains the following sections:

About this manual

This document is a comprehensive manual for the behavioral HDL model of the ASB Example AMBA SYstem (EASY) It gives detailed information about the function of the whole system, each module in the system, and describes how to design a new system module.

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

Using this manual

This manual is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the ASB Example AMBA SYstem (EASY).

Chapter 2 ***Microcontroller***

Read this chapter for a description the microcontroller, which is the main unit of the EASY system.

Chapter 3 ***ASB Modules***

Read this chapter for a description of the data sheets for the modules that are connected to the *Advanced System Bus* (ASB)

Chapter 4 ***APB Modules***

Read this chapter for a description of the modules that comprise the *Advanced Peripheral Bus* (APB).

Chapter 5 ***Test Interface Driver***

Read this chapter for a description of the use of the external AMBA Test Interface Driver module. It includes a description of the TICTalk command language.

Chapter 6 ***Designer's Guide***

Read this chapter for a basic look at adding new modules to the EASY microcontroller.

Conventions

Conventions that this manual can use are described in:

- *Typographical* on page vii

- *Timing diagrams*
- *Signals* on page viii
- *Numbering* on page ix.

Typographical

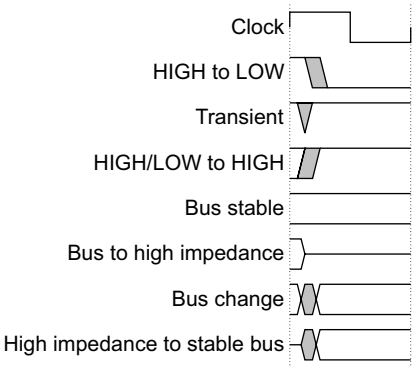
The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

Timing diagrams

The figure named *Key to timing diagram conventions* on page viii explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

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.



Key to timing diagram conventions

Single-bit signals are sometimes shown as HIGH and LOW at the same time and they look similar to the bus change shown in *Key to timing diagram conventions*. If a single-bit signal is shown like this then its value does not affect the accompanying description.

Signals

The signal conventions are:

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

Lower-case n At the start or end of a signal name denotes an active-LOW signal.

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

Prefix AR Denotes AXI read address channel signals.

Prefix AW Denotes AXI write address channel signals.

Prefix B Denotes AXI write response channel signals.

Prefix C Denotes AXI low-power interface signals.

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

Prefix P Denotes Advanced Peripheral Bus (APB) signals.

Prefix R Denotes AXI read data channel signals.

Prefix W Denotes AXI write data channel signals.

Numbering

The numbering convention is:

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

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

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

Additional reading

This section lists publications by ARM and by third parties.

See `http://infocenter.arm.com/help/index.jsp` for access to ARM documentation.

ARM publications

This manual contains information that is specific to the Abbreviated device name AMBA SYstem (EASY). See the following documents for other relevant information:

- *AMBA® Specification (Rev 2.0)* (ARM IHI 0011)
- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM7TDMI Data Sheet* (ARM DDI 0029)
- *AMBA ARM7TDMI Interface Data Sheet* ((ARM DDI 045)
- *Example AMBA SYstem User Guide* (ARM DUI 0092)
- *AHB Example AMBA SYstem Technical Reference Manual* (ARM DDI 0170).

Other publications

This section lists relevant documents published by third parties:

- IEEE 1149.1 JTAG standard

Feedback

ARM welcomes feedback on the ASB Example AMBA SYstem and its documentation.

Feedback on this product

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

- the product name
- a concise explanation.

Feedback on this manual

If you have any comments on this manual, send an e-mail to `errata@arm.com`. Give:

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

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the ASB Example AMBA SYstem (EASY). it contains the following:

1.1 Overview

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 sub-components to be easily re-used in future designs.

Note

This document refers to the *Advanced System Bus* (ASB).For information on the Advanced High-performance Bus (AHB) refer to the AHB 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 the various aspects of the ASB.

1.1.2 EASY components

The example design contains:

- Two bus masters, the ASB, and the *Advanced Peripheral Bus*.
- The ARM processor, to allow execution of the ARM code.
- The *Test Interface Controller (TIC)*, to allow external control of the ASB 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.

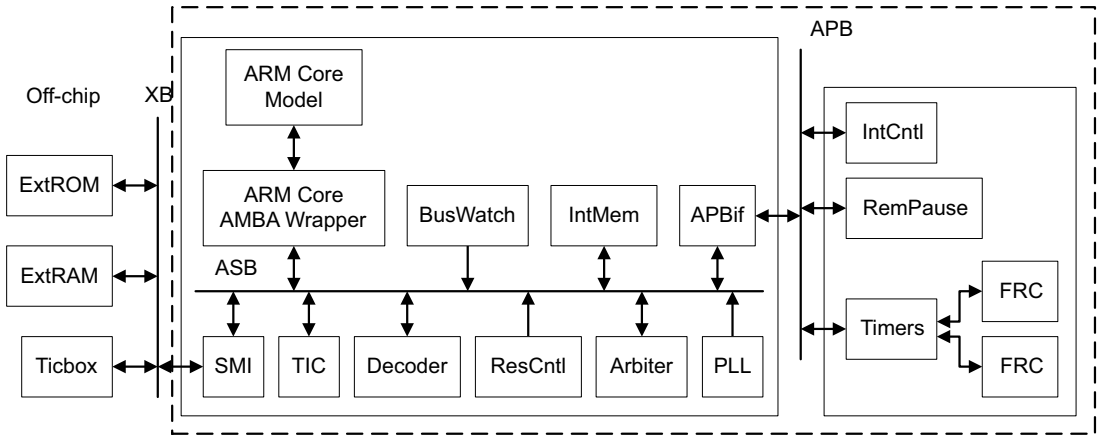


Figure 1-1 EASY system diagram

Chapter 2
Microcontroller

The microcontroller is the main unit of the EASY system. This chapter describes the microcontroller under the headings:

2.1 Functional overview

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

AMBA system components These are used to control the general operation of the system.

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

Example components

Demonstration parts 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, please refer to the relevant documentation.

2.2 The AMBA system components

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

- reset controller
- arbiter
- decoder
- *Advanced System Bus (ASB) to Advanced Peripheral Bus* (APB) Bridge.

The functions of each component are described below.

2.2.1 Reset controller

The reset controller consists of a state machine which controls the **BnRES** 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 ASB. Although there are only two bus masters in the EASY microcontroller (the ARM and the *Test Interface Controller* (TIC), the arbiter has provision for up to four masters. To extend the number of masters refer to *Adding bus masters* on page 6-2. The arbitration is currently assigned with a simple priority system, with the TIC as the highest priority, and the processor as the lowest reset default. The arbitration scheme is not defined in the *AMBA Specification* and can be dependent on implementation.

2.2.3 Decoder

The decoder manages all transfers on the ASB bus. Each bus transfer requires three components to act:

- a bus master to start the transfer
- the decoder to control the operation of the transfer
- a bus slave to accept a write transfer or control a read transfer.

The functions of these components are:

Bus Master This initiates a read or write transfer by driving **BTRAN[1:0]** (transfer type) and **BA[31:0]** (AMBA address bus) and control signals. This component drives **BD[31:0]** for a write transfer.

Decoder **BTRAN** is used to determine how the transfer should proceed, whether a slave should be selected, if and in which state slave responses (**BWAIT**, **BERROR** and **BLAST**) should be driven. The high order bits of **BA** are used to generate the corresponding slave select line (**DSEL**).

Bus Slave If selected, the slave will drive the slave responses. This component drives **BD[31:0]** for a write transfer.

Each transfer takes one or more cycles of the system clock (**BCLK**). The last cycle of each transfer occurs when **BWAIT** is driven LOW by the decoder or slave.

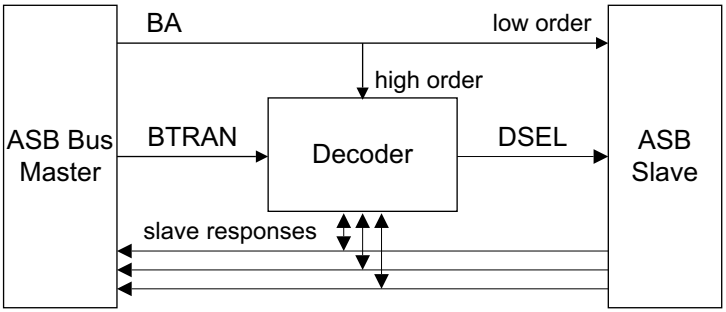


Figure 2-1 The role of the decoder in the AMBA bus

The EASY system provides a configurable decoder block, with or without decode cycles. A decode cycle can be inserted to improve the performance of the system.

Note

In some systems, typically those with a low clock frequency, transfers may occur without the addition of a decode cycle.

For more information on ASB transfers, see the *AMBA Specification*.

2.2.4 ASB to APB bridge

The ASB to APB bridge interface is an ASB slave. When accessed (in normal operation or system test) it initiates an access to the APB. All APB accesses are of the same duration (two **BCLK** cycles in the EASY). 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 **BCLK** input as the APB access is timed with a strobe signal generated by the ASB 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 2-2 shows how the reference peripherals are interconnected within the *Reference Peripherals Specification* (RPS) block, and how they are connected to the bridge.

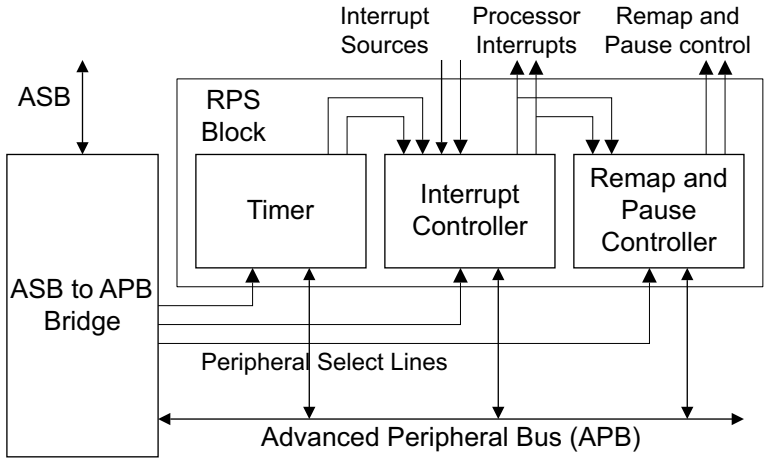


Figure 2-2 Block diagram of the RPS block and bridge

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

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	`0x80000000`
Timer	`0x84000000`
Remap and pause controller	`0x88000000`

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) Data travels from the peripherals to the bridge.

write (PWDATA) 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 **PWDATA** is the only master on the bus, it is driven continuously by the bridge. **PRDATA** is tristated by all peripherals on the bus, and is only driven when they 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.

Inside the RPS module is an APB bus keeper model for simulation purposes only (it is not synthesizable). Bus keepers must be included on a real implementation to prevent **PRDATA** from floating, since a given peripheral might not drive all the data bits. Although bus keepers are important to ensure low power consumption on the APB read data bus, they should not be relied upon to hold valid values.In designs based on EASY, bus keepers should be instantiated from the appropriate cell library.

2.3.1 Timer

The timer consists of two 16-bit periodic/free running down counters, a clock prescaler (divide by 1, 16 or 256) and a test veneer. When the counters underflow (passing zero value and reloading) they can generate *Interrupt Requests* (IRQs) which are connected 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 using six IRQ sources and one *Fast Interrupt Request* (FIQ) source. These have the following functions:

- to enable or disable specific interrupt sources from triggering the ARM **NIRQ** or **NFIQ** interrupt lines
- to read the status of all interrupt sources at the inputs of the interrupt controller

- to read the status of the interrupt sources enabled to trigger the ARM interrupt lines
- to generate a software-triggered **NIRQ** signal to the ARM processor
- to 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 7 of the external memory is mapped into location `0x00000000` 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 the internal memory and the *Static Memory Interface* (SMI).

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 the `MemSize` setting in the HDL file `behavioural/IntMem`. If this is done the decoder must also be altered. Refer to *Choosing a decoder implementation* on page 6-5).

2.4.2 Static memory interface

The SMI is a 32-bit *External Bus Interface* (EBI) that can connect up to 2GB of zero to four wait state *Static Random Access Memory* (SRAM) 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 eight banks of SRAM. The Example SMI also supports four signals from the TIC. These override the SMI's normal operation during a system test and directly control the tristate drivers and latches on the **XD** and **BD** buses.

2.5 System test methodology

Each ASB slave, ASB 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 ASB), 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.

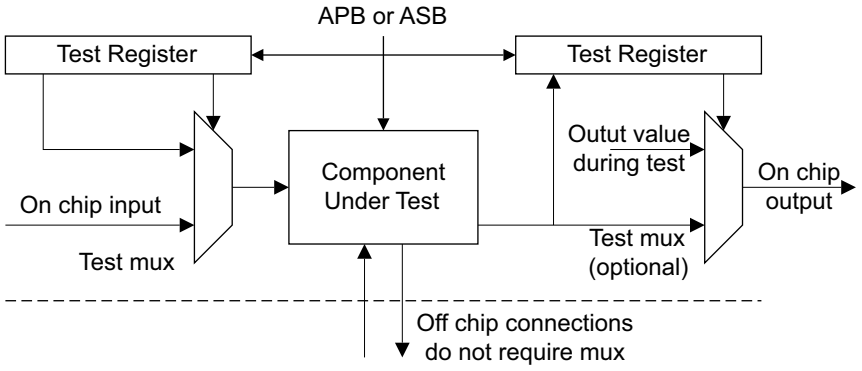


Figure 2-3 Simple test veneer example

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
- ASB to APB bridge interface (to access the peripherals)
- external bus interface.

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

- internal memory
- ASB to APB bridge interface (to access the peripherals)
- ARM bus master (test veneer).

Note

Bus masters can become slaves in the test mode. The SMI *cannot* be tested via the TIC. This is due to the method used to provide the test access to the ASB. During TIC testing, the normal function of the SMI is overridden, and it becomes a bidirectional channel between **TBUS** and **BD**. This means that during TIC testing the SMI cannot function as a slave.

The EBI **ca**nnot be tested via the TIC because of the way test access is provided to the ASB bus. The TIC is a state machine driven by the test request inputs (**TREQA** and **TREQB**). It also contains a latch that allows it to read address information from the test bus (**TBUS**) and drive it onto **BA**. However, it cannot drive **BD**. Instead, it overrides the normal function of the EBI, forcing it to provide a 32-bit, bidirectional channel between **TBUS** and **BD**. Thus in test mode the EBI cannot function as a slave.

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

For more information on:

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

Chapter 3
ASB Modules

This chapter describes the data sheets for the modules that are connected to the *Advanced System Bus* (ASB). It contains the following:

3.1 APB bridge

The APB bridge provides an interface between the ASB and the *Advanced Peripheral Bus* (APB). It continues the pipelining of the ASB by inserting wait cycles on the ASB only when they are needed. It inserts them for burst transfers or read transfers when the ASB must wait for the APB.

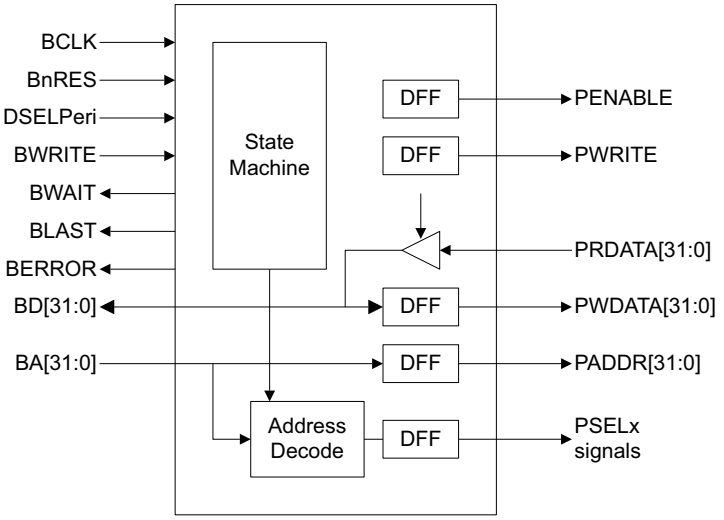


Figure 3-1 Block diagram of bridge module

The implementation of this block contains:

- a state machine, which is independent of the device memory map
- ASB address, and data bus latching
- combinatorial address decoding logic to produce **PSELx** signals.

To add new peripherals, or alter the system memory map only the address decode section needs to be modified.

3.1.1 Hardware interface and signal description

This module converts ASB transactions to APB transactions.

Table 3-1 Signal descriptions for bridge module

Name	Type	Description
BCLK	Input	This clock times all bus transfers.
BnRES	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
BA[31:0]	Input	The system address bus, which is driven by the current bus master.
BWRITE	Input	When HIGH this signal indicates a write transfer, when LOW, a read transfer. This signal is driven by the active bus master.
DSELPeri	Input	This signal indicates that the peripheral bus controller has been selected. It becomes valid during the **BCLK** HIGH phase before the data transfer, and remains active until the last **BCLK** HIGH phase of the transfer.
BD[31:0]	Input/Output	This is the bidirectional system data bus. The data bus is driven by the current bus master during write transfers, and by this block during read transfers from peripherals.
BWAIT	Output	This signal is driven by the selected bus slave during reads and writes to indicate whether the current transfer may complete. When selected, the peripheral bus controller drives this signal in the LOW phase of **BCLK,** and it is valid set up to the rising edge of **BCLK**.
BERROR	Output	A transfer error is indicated by the selected bus slave using the **BERROR** signal. This signal is not used by the bridge, so when selected, it is always driven LOW in the LOW phase of **BCLK**. It is valid set up to the rising edge of **BCLK**.
BLAST	Output	This signal is driven by the selected bus slave to indicate if the current transfer should be the last of a burst sequence. This signal is not used by the bridge, so when selected, it is always driven LOW in the LOW phase of **BCLK**. It is valid set up to the rising edge of **BCLK**.
PRDATA[31:0]	Input	The peripheral read data bus is driven by the selected peripheral bus slave during read cycles (when **PWRITE** is LOW).
PWDATA [31:0]	Output	The peripheral write data bus is continuously driven by this block, changing during write cycles (when **PWRITE** is HIGH).
PENABLE	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.

Table 3-1 Signal descriptions for bridge module (continued)

Name	Type	Description
PSELx	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. This signal 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]	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	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.

The timing and inputs and outputs of the module are shown in Figure 3-2 on page 3-5, Figure 3-3 on page 3-6 and Figure 3-4 on page 3-7.

Figure 3-2 on page 3-5 shows a single write cycle. There are four cycles between the start of the transfer on the ASB and the end of the transfer on the APB, but only one wait state is inserted on **BWAIT**. The write data (on **BD**) is sampled on the **BCLK** rising edge after it becomes valid.

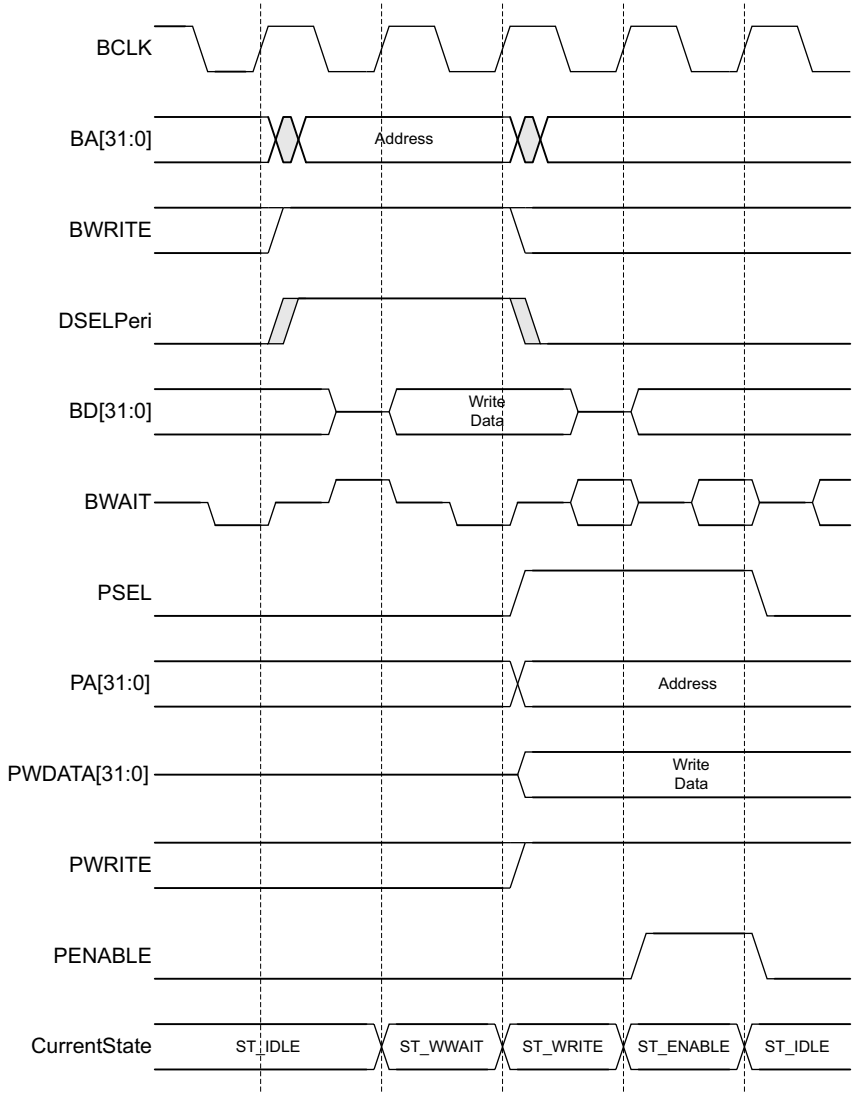


Figure 3-2 APB write cycle

Figure 3-3 on page 3-6 shows a single APB read transfer. There are three cycles between the start of the transfer on the ASB and the end of the transfer on the APB, but only one wait state is inserted on **BWAIT**. The APB peripheral read data (**PRDATA**) is generated on the rising edge of **BCLK,** and sampled on the next falling edge of **BCLK**

by the ARM core. If the APB peripherals used cannot generate **PRDATA** early enough to be sampled on the falling edge, then the bridge must be modified to insert an extra ASB wait state.

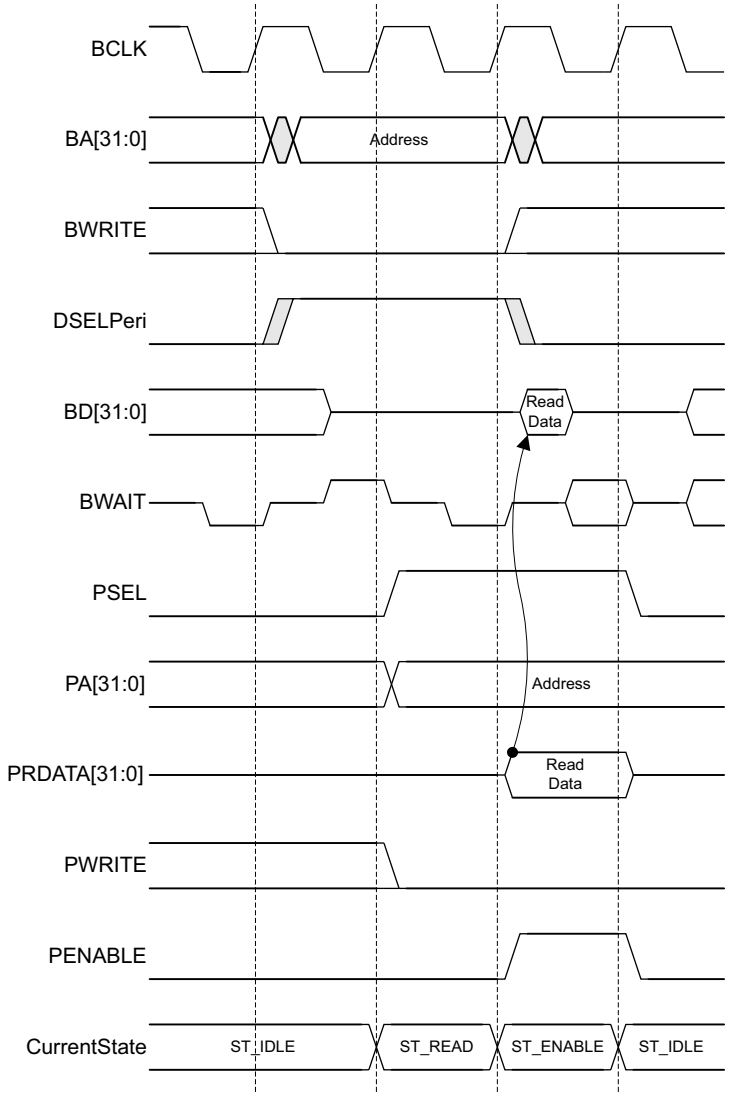


Figure 3-3 APB read cycle

Figure 3-4 on page 3-7 shows an APB write followed by an APB read. Two wait states are added before the APB read, this allows the APB write transfer time to end before the read starts.

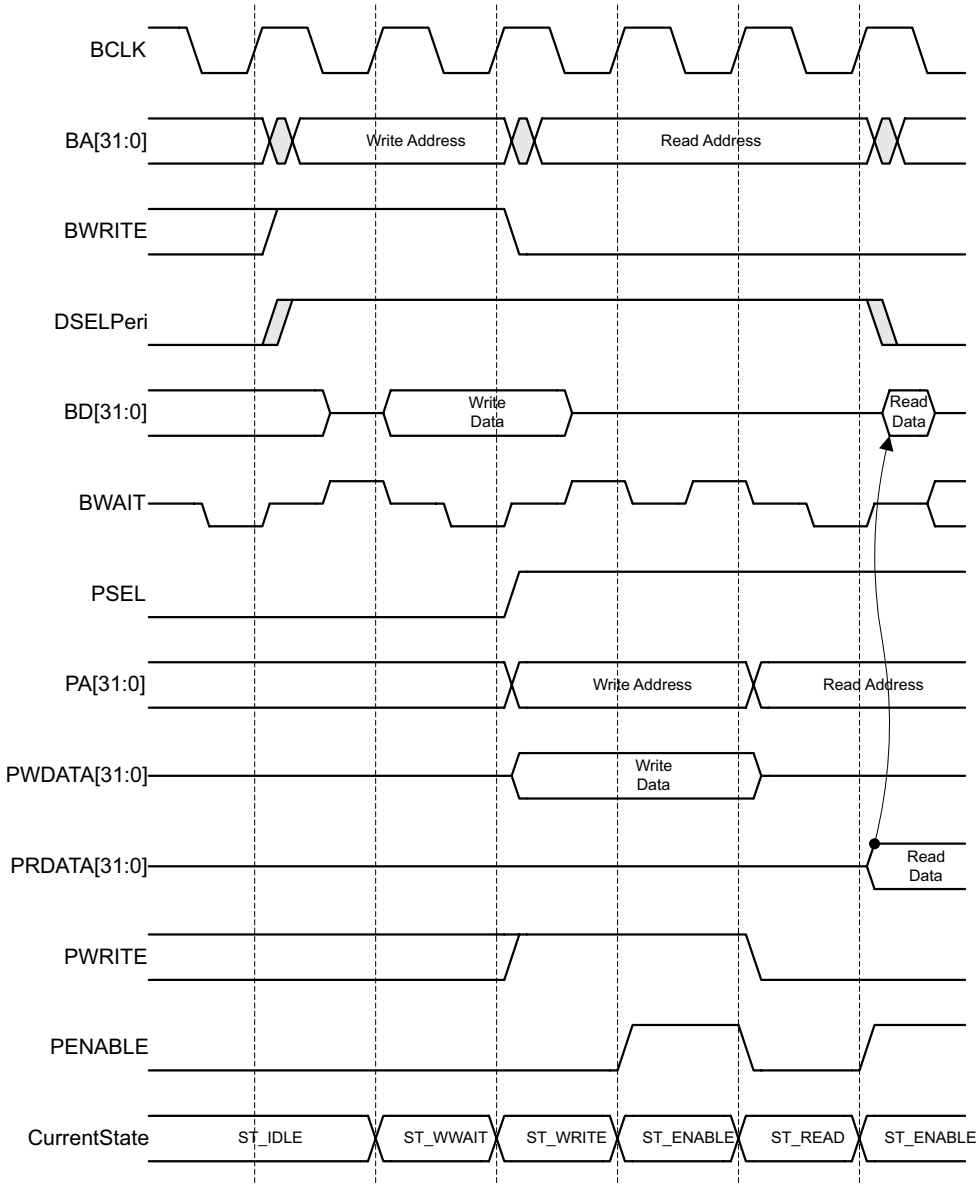


Figure 3-4 APB burst cycle

3.1.2 Peripheral memory map

The 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 3-5.

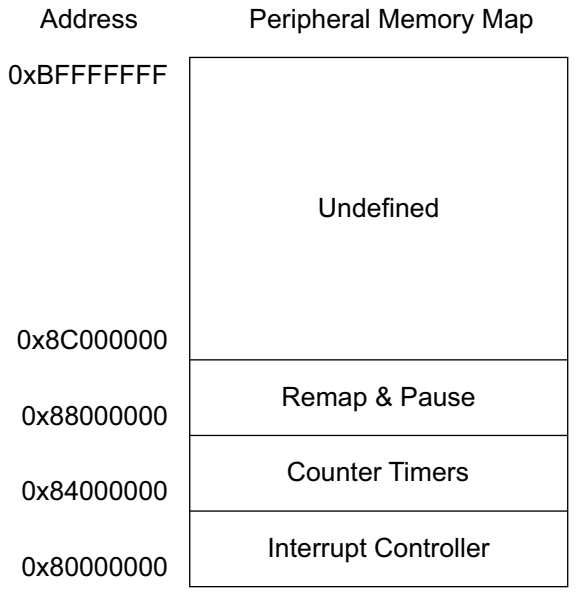


Figure 3-5 Peripheral memory map

3.1.3 Function and operation of block

The APB bridge responds to transaction requests from the currently enabled ASB master. The ASB transactions are converted into APB transactions. The state machine, shown in Figure 3-6 on page 3-9, controls:

- the ASB transactions with the slave response signals
- the register enables for the **PADDR** and **PWDATA** buses
- the tristate driver controls for the **BD** bus.

It also produces the **PENABLE** signal. This design uses the **DSELPeri** signal from a centralized decoder to select the peripheral bus controller as an ASB slave.

The individual **PSELx** signals are decoded from **BA**, 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.

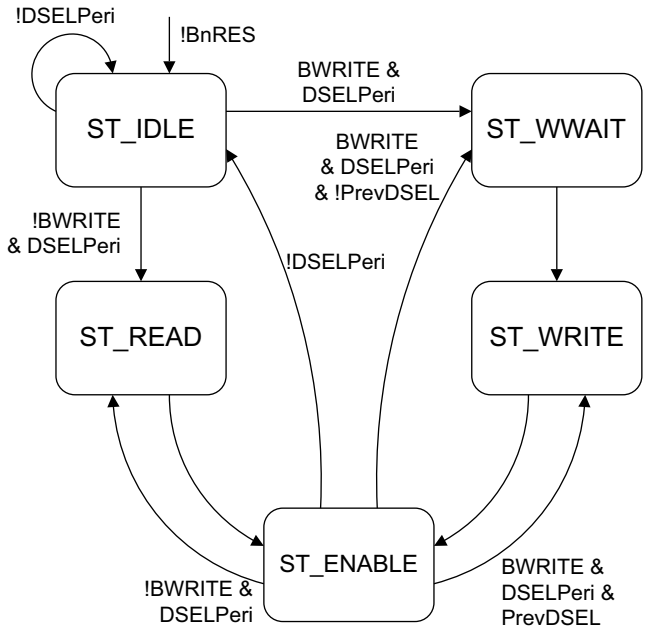


Figure 3-6 State machine for APB controller

ST_IDLE

The **ST_IDLE** state is entered from:

- reset, when the system is initialized
- **ST_IDLE** when there are no peripheral transactions being performed.

The APB buses and **PWRITE** are driven with the last values they had, and the **PSEL** and **PENABLE** lines are all set to zero.

The next state is:

- **ST_READ** for a read transfer, when **DSELPeri** is HIGH and **BWRITE** is LOW
- **ST_WWAIT** for a write transfer, when both **DSELPeri** and **BWRITE** are HIGH.

A wait state (setting **BWAIT** HIGH) is always inserted on exit from **ST_IDLE** when **DSELPeri** is HIGH.

ST_READ

The **ST_READ**state is entered from:

- **ST_IDLE** during a nonsequential read cycle
- **ST_ENABLE** during a sequential read cycle.

The address is decoded and driven onto **PADDR**, the relevant **PSEL** line is driven, and **PWRITE** is driven LOW.

The next state will always be **ST_ENABLE**.

ST_WWAIT

The **ST_WWAIT** state is entered from

- **ST_IDLE** during a nonsequential write cycle
- **ST_ENABLE** when there has been a single address cycle between two consecutive nonsequential writes.

Adding this state to a WRITE operation allows time for **BD** to be driven with the write data, ready to be sampled on the next rising edge of the clock.

The next state will always be **ST_WRITE**.

ST_WRITE

The **ST_WRITE**state is entered from:

- **ST_WWAIT** during a nonsequential write cycle
- **ST_ENABLE** during a sequential write cycle.

If a single write is being performed then BWAIT is set LOW. For a burst of writes, BWAIT is set HIGH (to delay the ASB transfer).

As for **ST_READ**, the address is decoded and **PSEL** is driven, but **PWRITE** is driven HIGH. The next state will always be **ST_ENABLE**.

ST_ENABLE

The **ST_ENABLE**state is entered from

- **ST_READ** during a read cycle
- **ST_WRITE** during a write cycle.

If the reset transaction is a sequential read cycle, a wait cycle is added to allow the APB line to finish the current transaction before starting the read cycle. If the next transaction is a sequential write, no wait state is added as one will be inserted in the **ST_WRITE** state.

For nonsequential writes, where **DSELPeri** is not held constantly HIGH (for example, when decode cycles are being inserted by the decoder), a wait state is inserted to allow time for **BD** to be driven with valid data.

The next state is:

- **ST_READ** or **ST_WRITE** for a burst transfer
- **ST_WWAIT** for a burst of nonsequential writes with decode cycles
- **ST_IDLE** for the last cycle in a burst, or for a single read or write cycle.

3.1.4 System description

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

Figure 3-7 shows the APB bridge module block diagram.

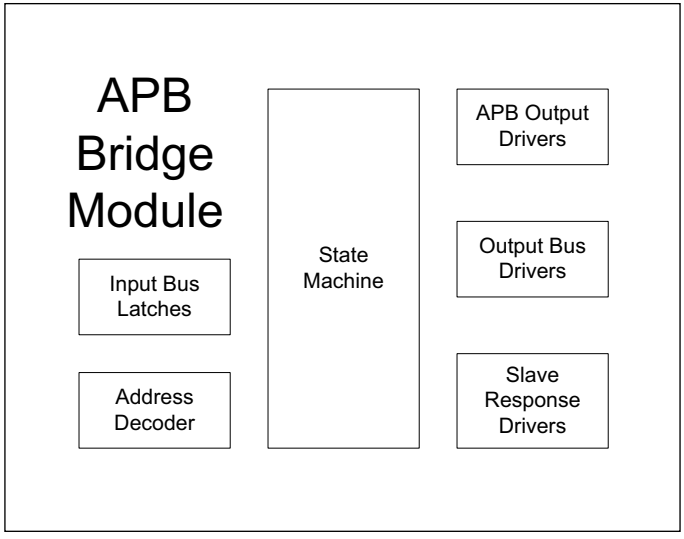


Figure 3-7 APB bridge module block diagram

The APB bridge system comprises:

- the state machine which is used to control the generation of the output signals

- the address decoding 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 **BCLK**. Enable signals are used to control the loading of the registers. All registers use the asynchronous reset **BnRES**.

A diagram of the APB bridge HDL file is shown in Figure 3-8.

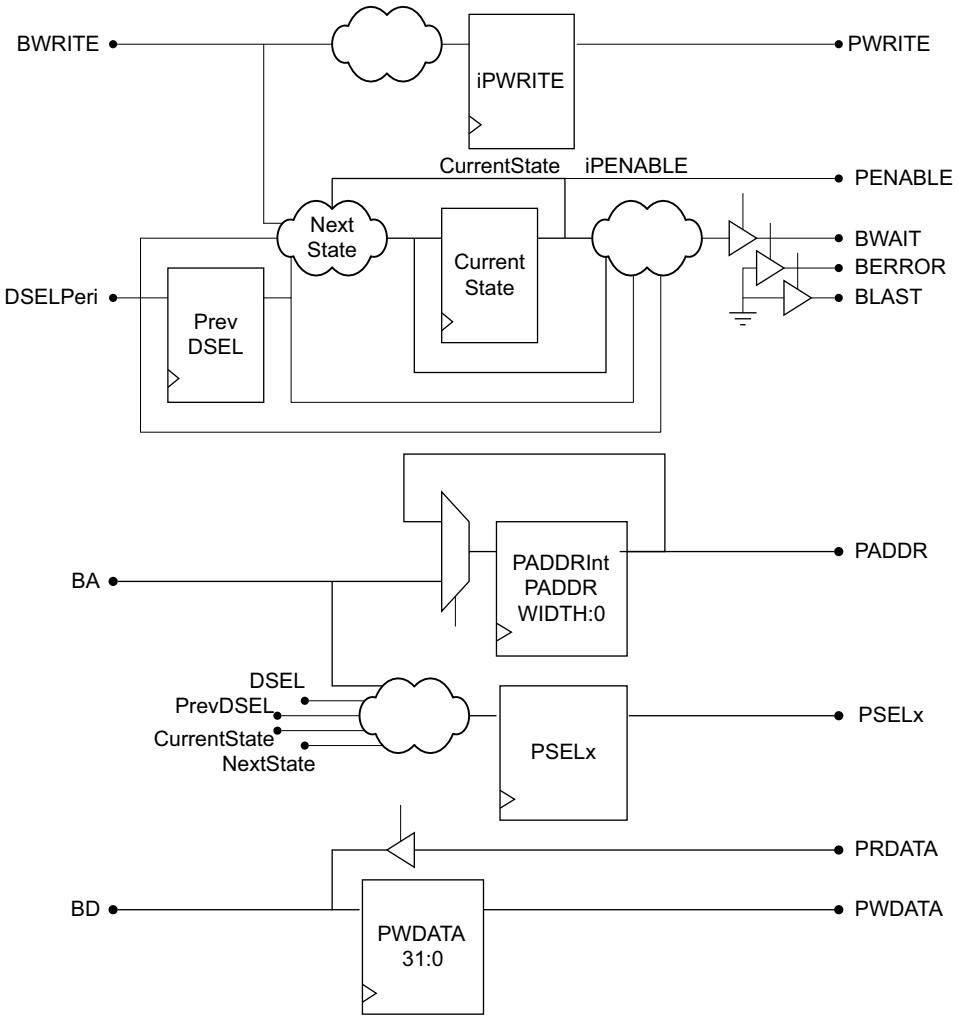


Figure 3-8 APB bridge module system diagram

The main parts and processes in the code are:

- *Address decoder* on page 3-13.

- *Next state logic* on page 3-14.
- *State machine* on page 3-14.
- *Current state decoding* on page 3-14.
- *PWRITEInt generation* on page 3-15.
- *Registered BA* on page 3-15.
- *Internal PENABLE generation* on page 3-15.
- *BD tristate enable* on page 3-15.
- *Slave response tristate enable* on page 3-15.
- *PWDATA generation* on page 3-15.
- *APB output signals* on page 3-15.
- *Tristate output drivers* on page 3-16.

Each of these is explained in the following paragraphs.

Constant and signal definitions

The constant **PADDRWIDTH** sets the width of the peripheral address bus that is used, up to a maximum of 32 bits wide. 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 five constants are the state machine states. They are explicitly defined so that the **ST_ENABLE** state bit is not used by any other state, ensuring that there is no chance of getting any glitches on the **PENABLE** line. The signals that are used inside the module are then defined.

Address decoder

The address decoder decodes the current address on **BA,** and generates the internal **PSELxInt** signals that are used to drive the output ports on the APB. Constants are set with the address values for the peripherals (default uses bits 29:26 of the address), which are then compared with the current value of **BA** and the relevant peripheral select line is set.

For addresses outside the specified ranges, none of the **PSELxInt** lines will be set, but the read/write transfer will continue as normal.

Registered DSELPeri

A rising edge registered copy of **DSELPeri** is required to generate **NextState** and **BWAITInt**, which is called **PrevDSEL**.

Next state logic

The next state logic is the combinational part of the state machine. The next state of the state machine is calculated using the current state of the state machine and the **DSELPeri** and **BWRITE** inputs to the bridge. Figure 3-6 on page 3-9 shows the operation of the state machine.

The system resets into the **ST_IDLE** state, and stays in this state until a transfer is requested. If the current cycle is a read or a write, the next state becomes **ST_READ** or **ST_WWAIT**, which is followed by **ST_ENABLE,** or **ST_WRITE** and **ST_ENABLE**. If the transfer is only a single read or write operation, the system goes back to the **ST_IDLE** state. If the transfer is a burst of reads or writes, the next state becomes **ST_READ** or **ST_WRITE**, and this continues until the transfer ends. For bursts of nonsequential writes, the state machine changes from **ST_ENABLE** to **ST_WAIT** to allow time for the write data to become valid on BD.

State machine

With the state machine register **NextState** is loaded into **CurrentState** on the rising edge of **BCLK**.

Current state decoding

The current state decoding is used to generate two signals, **BWAITInt** and **APBEn**.

BWAITInt is set HIGH during four possible conditions:

- the start of the first APB transfer, when **DSELPeri** is set HIGH and the current state is still **ST_IDLE**
- during a burst of sequential writes when the current state is **ST_WRITE** and **DSELPeri** is still set HIGH
- during a burst of sequential reads when the current state is **ST_ENABLE** and the next state will be **ST_READ**.
- during a burst of nonsequential writes when the current state is **ST_ENABLE, DSELPeri** is HIGH but **PrevDSEL** is LOW.

This signal is then used to drive the external **BWAIT** signal.

APBEn is set HIGH when a read or a write is being performed on the APB. The **APBEn** signal is then used to enable the **PWRITE**, **PADDR** and **PSEL** outputs.

PWRITEInt generation

The signal **PWRITEInt** is a registered version of **BWRITE**. **BWRITE** is captured on the rising edge of **BCLK** when **APBEn** is HIGH with reset clearing it to zero.

Registered BA

The signal **PADDRInt** is a rising edge registered version of **BA**(PADDRWidth-1:0). It is captured on the rising edge of **BCLK** when **APBEn** is HIGH, with reset clearing **PADDRInt** to all zeros.

Internal PENABLE generation

PENABLEInt is generated from one of the Current State registers. This should be changed if the state encoding of the state machine is altered.

BD tristate enable

The BD tristate enable is used to enable **BD** to be driven with the peripheral output data during a read cycle. It is set when **PWRITEInt** is LOW and **PENABLEInt** is HIGH, indicating a peripheral read.

Slave response tristate enable

The slave response signals **BWAIT**, **BERROR** and **BLAST** are only driven by the slave when it is selected (that is **DSELPeri** is HIGH), and during the LOW phase of **BCLK**, so these two inputs are used to generate the slave response enable signal.

PWDATA generation

The output signal **PWDATA** is a registered version of **BD** for a write cycle. In order to minimize the number of signal changes on **PWDATA**, the enable is only set when **NextState** is **ST_WRITE**.

APB output signals

In this part all the APB output ports are driven with their internal signals.

PADDR, PWRITE and **PENABLE** are driven directly by **PADDRInt** and **PWRITEInt** and **PENABLEInt** respectively.

The **PSEL** ports are registered versions of the **PSELInt** signals set on the rising edge of **BCLK**. The signals are registered when **APBEn** is HIGH (read or write cycle), and reset to zero when:

- **BnRES** is LOW
- at the end of the last APB transfer
- in the **ST_WWAIT** state.

Tristate output drivers

The **BD** and slave response signals are all tristated, so can only be driven at the correct times to avoid any drive clashes by using the enable signals created earlier.

BD is driven with the current value of **PRDATA** when **BDEn** is HIGH.

BWAIT is driven with the current value of **BWAITInt** when **BWELEn** is HIGH. The module does not use the other two slave response signals **BERROR** and **BLAST,** which are driven LOW.

3.2 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 can request the bus, 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 during reset, or when no other bus master requires the bus.

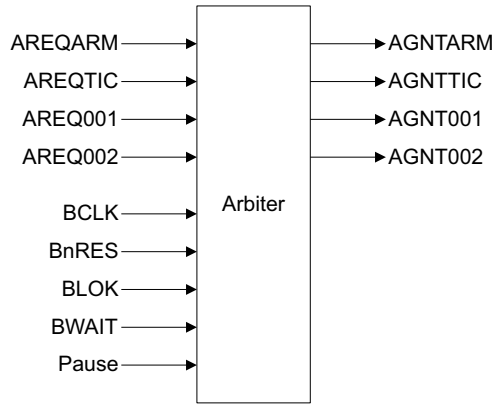


Figure 3-9 Arbiter block diagram

The arbiter included in the EASY design can support up to four bus masters, although only two are used. The operation of the arbiter is described under the following headings:

- *Signal descriptions*.
- *Abitration process* on page 3-18.
- *Signal timing* on page 3-19.
- *Arbitration priorities* on page 3-21.
- *System description* on page 3-22.

3.2.1 Signal descriptions

The signals are described in Table 3-2 on page 3-18.

Note

In systems that only have the ARM and the Test Interface Controller as potential bus masters, the unused **AREQxxx** lines must be tied LOW.

Table 3-2 Signal descriptions

Name	Type	Description
BCLK	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers on the bus.
BnRES	Input	The bus reset signal is active LOW and is used to reset the system and the bus.
BWAIT	Input	This signal is driven by the selected bus slave to indicate whether the current transfer may complete. If **BWAIT** is HIGH a further bus cycle is required. If **BWAIT** is LOW then the transfer may complete in the current bus cycle. **BWAIT** is used by the arbiter to determine when a turnaround cycle is happening on the bus.
BLOK	Input	A shared bus lock signal driven by the currently granted bus master. When HIGH this signal indicates that the current transfer and the next transfer are to be indivisible and no other bus master should be granted the bus.
AREQarm	Input	Request from the ARM processor indicating that it requires the bus. This signal must be set up to the falling edge of **BCLK**.
AREQtic	Input	Request from the test interface controller.
AREQ001	Input	Request from the bus master 001.
AREQ002	Input	Request from the bus master 002.
Pause	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.
AGNTarm	Output	Grant signal to the ARM processor. When HIGH, this signal indicates that the ARM bus master is granted the bus. This signal changes during the LOW phase of **BCLK** and remains valid through the HIGH phase.
AGNTtic	Output	Grant signal to the test interface controller.
AGNT001	Output	Grant signal to bus master 001.
AGNT002	Output	Grant signal to bus master 002.

3.2.2 Abitration process

The ASB arbitration is controlled by the **AREQ**, **AGNT**, **BLOK** and **BWAIT** signals.

When an ASB master requires use of the bus, it sets its **AREQ** output line HIGH. This is sampled by the arbiter, on the falling edge of **BCLK**, and the **AGNT** outputs change according to the arbitration priority scheme used by the system.

The **BLOK** and **BWAIT** signals are used to extend the granted period to allow masters to finish transfers before bus master handover begins. If **BLOK** is set HIGH by the current master, and a higher priority master requests the bus, handover will not start until **BLOK** is set LOW, showing that the locked transfer has finished.

If **BLOK** is set HIGH after handover has begun, it is ignored, as it is too late to have an effect on the handover process.

If **BWAIT** is set HIGH after the handover process has begun, then the **AGNT** lines change as normal, but the new ASB master must allow the current master to finish the bus transfer (**BWAIT** LOW) before taking control of the bus.

3.2.3 Signal timing

The arbitration signal timing depends on:

- the priorities of the masters requesting the bus
- the status of **BLOK** when the request lines change
- the status of **BWAIT** during the handover cycle.

This is shown in Figure 3-10 on page 3-20, and Figure 3-11 on page 3-21, where master A has a higher priority than master B. The masters read their **AGNT** inputs and gain control of the bus on the rising edge of **BCLK**.

Figure 3-10 on page 3-20 shows a normal grant handover where the higher priority master is granted the bus for two cycles before handing it back to the lower priority master. During bus master handover **BLOK** is not driven and therefore the arbiter ignores this signal until the new master has control of the bus.

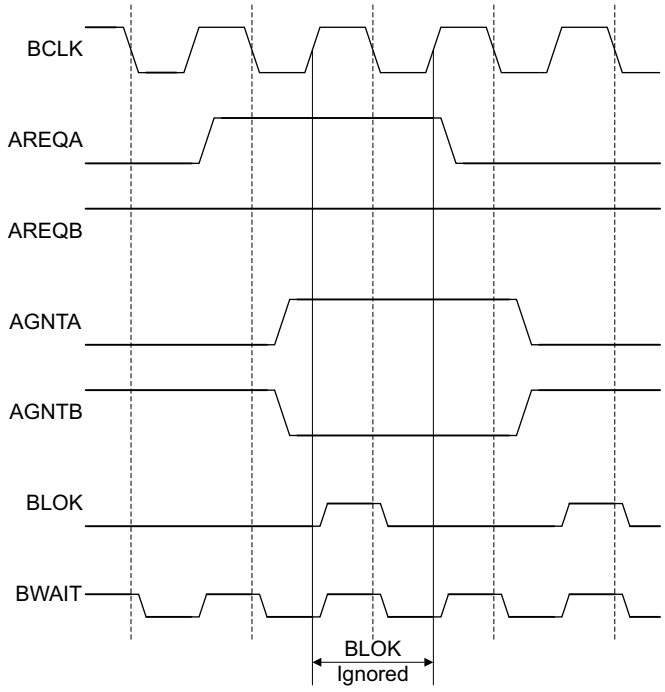


Figure 3-10 Arbitration timing

Figure 3-11 on page 3-21 shows the effect of the **BLOK** signal, as the grant to the lower priority master is held until the **BLOK** signal is released. If one current transfer is extended (**BWAIT** is HIGH), then the **AGNT** lines are set to their new values, but the masters in the system must monitor the **BWAIT** signal and wait until it is LOW before gaining control of the bus.

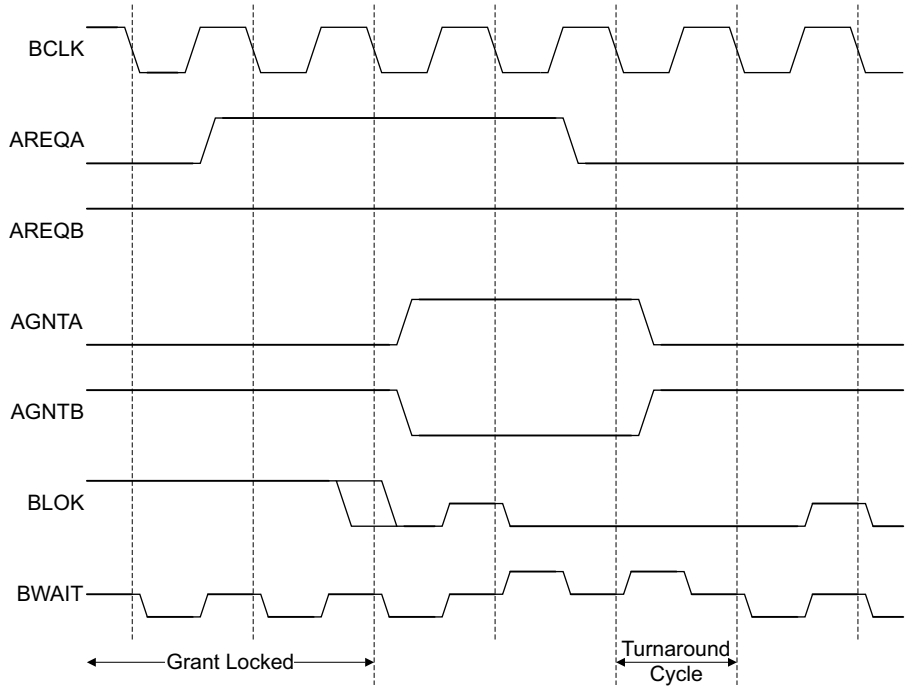


Figure 3-11 Arbitration timing with BLOK set and turnaround cycle

3.2.4 Arbitration priorities

During reset, when **BnRES** is LOW, the arbiter grants use of the bus to the default bus master, and holds all other grant signals inactive.

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

- Test interface controller (highest)
- Bus master1
- Bus master 2
- ARM processor (lowest).

There are four different methods for a bus master to be granted the bus at that time:

- if it is the highest priority master requesting the bus at that time
- if it is the standard master selected during reset
- if it is the standard master selected during pause mode (when **Pause** is set HIGH)
- if it is the default master selected when no masters are requesting the bus.

3.2.5 System description

The following paragraphs give a detailed description of 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 all the registers, and signals used in the system. This part should be read together with the HDL code.

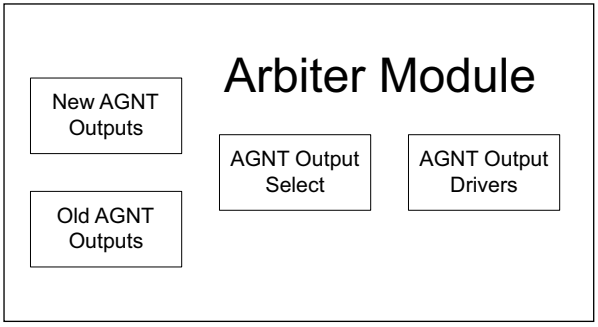


Figure 3-12 Arbiter module block diagram

The arbiter comprises the two versions of the **AGNT** output signals (new and old) and the output select that drives the **AGNT** lines with one of the internal **AGNT** values.

All registers used in the system are clocked from the same signal, the system clock **BCLK**, from either the rising or the falling edge. Enable signals are used to control the loading of the registers. All registers use the asynchronous reset **BnRES**.

A diagram of the arbiter HDL file is shown in Figure 3-13 on page 3-23.

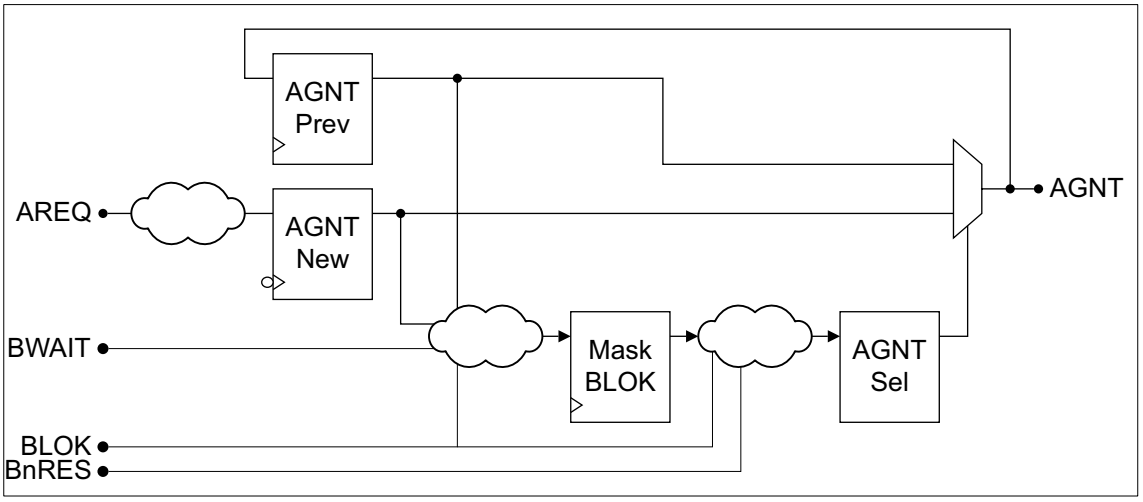


Figure 3-13 Arbiter module system diagram

The parts and processes in the code are described in the following sections:

- *Arbitration scheme*.
- *Stored previous AGNT outputs* on page 3-24.
- *AGNT comparator* on page 3-24.
- *MaskBLOK generation* on page 3-24.
- *AGNT output selection control* on page 3-24.
- *AGNT output select* on page 3-24.
- *AGNT output drivers* on page 3-25.

Arbitration scheme

This part 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 is the priority order of the system. **AREQtic** is the first input that is checked, then **Pause** (also the TIC), **AREQ001**, and through to the lowest priority. As the TIC is the highest priority master, if the **AREQtic** is HIGH, **AGNTticNew** will be driven HIGH no matter what values the other **AREQ** inputs have. If **AREQarm** is set HIGH, **AGNTarmNew** will only be set if all other **AREQ** inputs are LOW, as the **ARM** is the lowest priority master.

The **AGNTarmNew** output is set during reset when **BnRES** is LOW and when no other **AREQ** lines are set HIGH.

Falling edge registers are used to hold the generated **AGNTNew** values.

Stored previous AGNT outputs

The current **AGNT** outputs are not a direct reflection of the **AREQ**, **BnRES** and **Pause** inputs to the Arbiter. This is due to delays during the bus master handover, when **BLOK**, or **BWAIT** are set.

This means that the **AGNT** outputs have to be held at their current values during the handover process, and this is done by the **AGNTPrev** registers, which sample the **AGNT** outputs on the rising edge of the system clock.

AGNT comparator

The AGNT 4-bit comparator is used to compare the **AGNTNew** and the **AGNTPrev** signals to detect a bus master handover, and it sets **AGNTChange** HIGH when they are different.

MaskBLOK generation

The **MaskBLOK** register is used to mask out the **BLOK** signal to the **AGNTSel** latch, so that the **AGNTNew** signals are held on the outputs during handover no matter what the status of **BLOK** is during the handover cycle.

AGNT output selection control

The **AGNTSelNext** signal is used to feed the **AGNTSel** latch, which is transparent when **BCLK** is LOW. A latch is used due to the timing of **BLOK**. **AGNTSel** is used to select either the **AGNTNew** or **AGNTPrev** signals to be driven onto the **AGNT** outputs.

During bus master handover **BLOK** must be ignored (as shown in Figure 3-10 on page 3-20). This is done using **MaskBLOK** in the generation of **AGNTSelNext**. If **BLOK** is set HIGH, **AGNTSelNext** will be driven LOW, but during handover **MaskBLOK** will also be HIGH, keeping **AGNTSelNext** driven HIGH.

During reset (when **BnRES** is HIGH), **AGNTSel** will be driven LOW, which selects the **AGNTPrev** signals to be driven onto the AGNT outputs.

AGNT output select

A 4-bit multiplexor is used to select either the **AGNTNew** or **AGNTPrev** signals to drive the i**AGNT** signals with, depending on the value of **AGNTSel**. If the value of **AGNTSel** is HIGH, **AGNTNew** is selected. If it is LOW, **AGNTPrev** is selected.

AGNT output drivers

The external **AGNT** ports are continuously driven with their internal i**AGNT** signals.

3.3 Decoder

3.3.1 Overview

The decoder performs three functions:

- it generates the slave select signals (**DSELx**) for each of the bus slaves, indicating that a read or write access to that slave is required
- it generates the slave response signals (**BWAIT**, **BLAST** and **BERROR**) during Address-only transfers, when no slave is selected
- it can act as a simple protection unit which prevents attempts to access a protected area of the memory map.

Two implementations of the decoder are provided in one HDL file, using the value of **DECEN** to select between them:

- Decoder with decode cycles. This is the default mode, and is used in fast systems where the decoder might not have enough time to decode the address and assert the corresponding **DSELx** signal in a single clock HIGH phase. This implementation automatically inserts a decode cycle:
  - — at the start of a nonsequential transfer
  - — on a sequential transfer when **BLAST** has been asserted
  - — when 1KB memory boundaries are reached.
- Decoder without decode cycles. This implementation requires that all accesses can be decoded within a single bus cycle, and therefore will only be suitable for slow systems where this can be safely achieved.

A block diagram of the decoder is shown in Figure 3-14 on page 3-27.

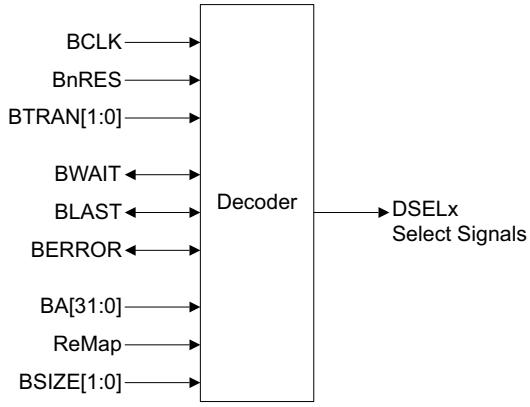


Figure 3-14 Decoder block diagram

3.3.2 Signal description

This section describes the signals that interface to the decoder.

Table 3-3 Signal descriptions

Name	Type	Description
BCLK	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers on the bus.
BnRES	Input	The bus reset signal is active LOW, and is used to reset the system.
BSIZE[1:0]	Input	The transfer size signals indicate the size of the transfer, which may be byte, halfword or word.
BTRAN[1:0]	Input	These signals indicate the type of the next transaction, which may be address-only, nonsequential or sequential. These signals are driven by a bus master when its respective **AGNTx** signal is asserted.
BA[31:0]	Input	The system address bus, which is driven by the active bus master.
Remap	Input	When LOW, the internal memory is not part of the system memory map, and external memory is mapped from address `0x00000000` which normally contains the system startup code. In normal operation this signal is HIGH, allowing the use of the internal memory.
BWAIT	Input/Output	This signal is driven by the selected bus slave to indicate if the current transfer may complete. If **BWAIT** is HIGH a further bus cycle is required, if **BWAIT** is LOW then the transfer may complete in the current bus cycle. When no slave is selected this signal is driven by the bus decoder.

Table 3-3 Signal descriptions (continued)

Name	Type	Description
BERROR	Input/Output	A transfer error is indicated by the selected bus slave using the **BERROR** signal. When **BERROR** is HIGH a bus error has occurred. When **BERROR** is LOW, the transfer is successful. When no slave is selected, this signal is driven by this module.
BLAST	Input/Output	This signal is driven by the selected bus slave to indicate whether the current transfer should be the last of a burst sequence. When **BLAST** is HIGH, the decoder must allow sufficient time for address decoding. When **BLAST** is LOW, the next transfer may continue a burst sequence. When no slave is selected this signal is driven by the bus decoder.
DSELx	Output	These are the select signals from the bus decoder to each individual bus slave, which indicate that the slave device is selected, and a data transfer is required. There is a **DSELx** signal for each ASB bus slave. This signal becomes valid during the HIGH phase before the data transfer is required, and remains active until the last HIGH phase of the transfer.

3.3.3 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 0, and during normal operation, when RAM may be used at address 0.

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 3-15 on page 3-29 shows both the normal and reset memory maps.

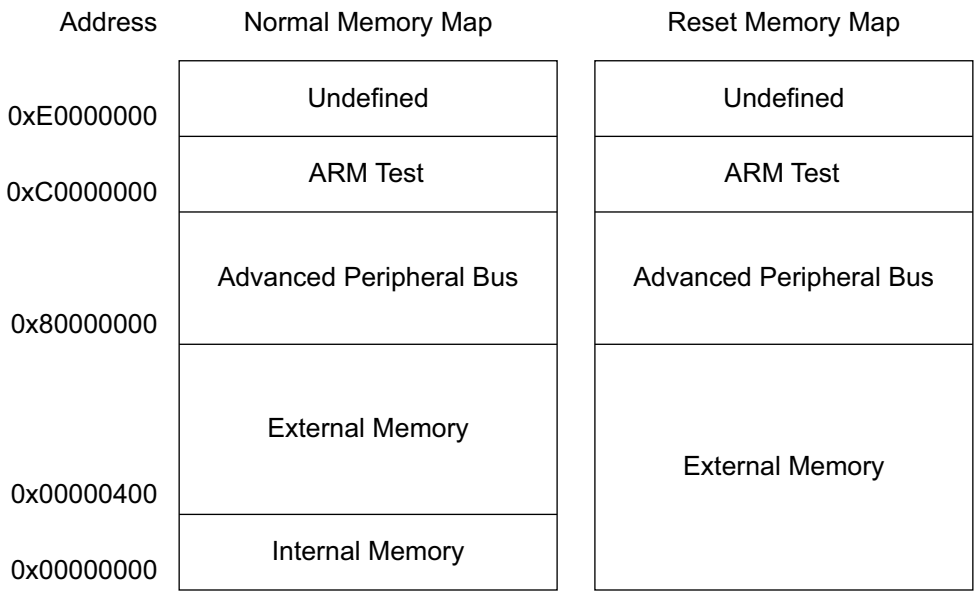


Figure 3-15 Memory map

3.3.4 Function and operation of block

At the start of every transfer on the bus, the decoder can perform a number of actions. The decoder is able to determine when a transfer is about to start by examining the **BWAIT** signal, which will be LOW when the previous transfer is completing.

The actions the decoder may take depend on the type of transfer as shown in Table 3-4.

Table 3-4 Types of transfer

Transfer type	Decoder action
Nonsequential transfer	For a decoder that implements decode cycles, the decoder inserts a single decode cycle to allow the address bus to stabilize and the new address to be decoded. For the first cycle, the decoder drives **BWAIT** HIGH and negates all **DSELx** signals. In the second cycle, the decoder asserts the appropriate **DSELx**, and the selected slave becomes responsible for driving the slave transfer response. For a decoder that does not implement decode cycles, the decoder asserts the **DSELx** signal during the first cycle (see Figure 3-17 on page 3-32).
Sequential transfer with **BLAST** LOW	The decoder drives the appropriate **DSELx** signal, and the selected slave is responsible for driving the slave transfer response.
Sequential transfer with **BLAST** HIGH.	For a decoder that implements the decode cycles, the decoder inserts a single decode cycle to allow the address bus to stabilize and the new address to be decoded. For the first cycle, the decoder drives **BWAIT** HIGH and negates all **DSELx** signals. In the second cycle, the decoder asserts the appropriate **DSELx**, and the selected slave becomes responsible for driving the slave transfer response. For a decoder that does not implement decode cycles, the decoder asserts the **DSELx** signal during the first cycle (see Figure 3-17 on page 3-32).
Address-only transfer	The decoder does not generate any **DSELx** signals, and drives a slave transfer response of **BWAIT** LOW.

Figure 3-16 on page 3-31 shows the timing of a nonsequential access using a decoder that inserts wait states. The wait state is necessary because the address arrives after the falling edge of **BCLK**.

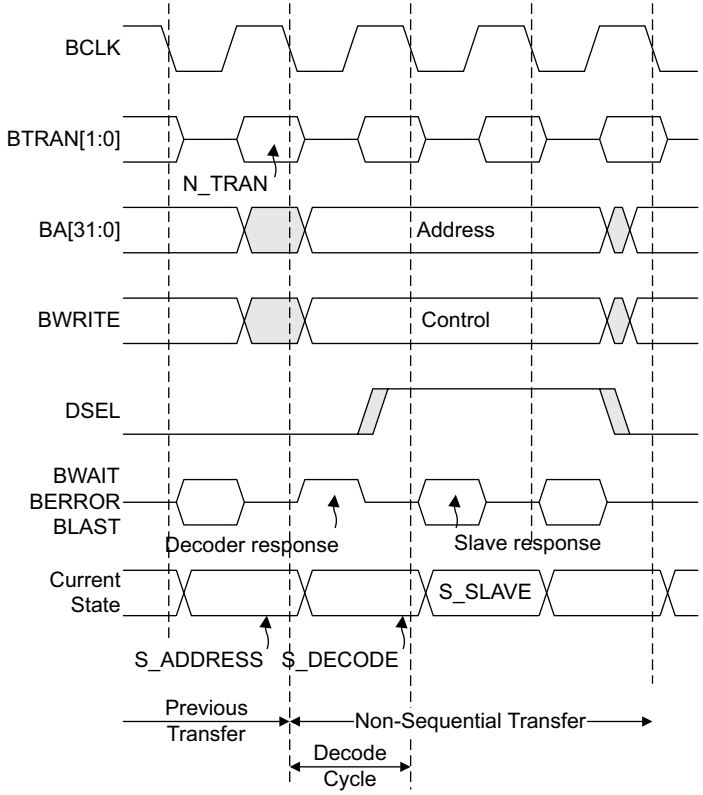


Figure 3-16 Decoder with decode cycles

Figure 3-17 on page 3-32 shows the timing of a nonsequential access using a decoder that does not insert wait states. The wait state is not necessary because the address is valid before the falling edge of **BCLK**.

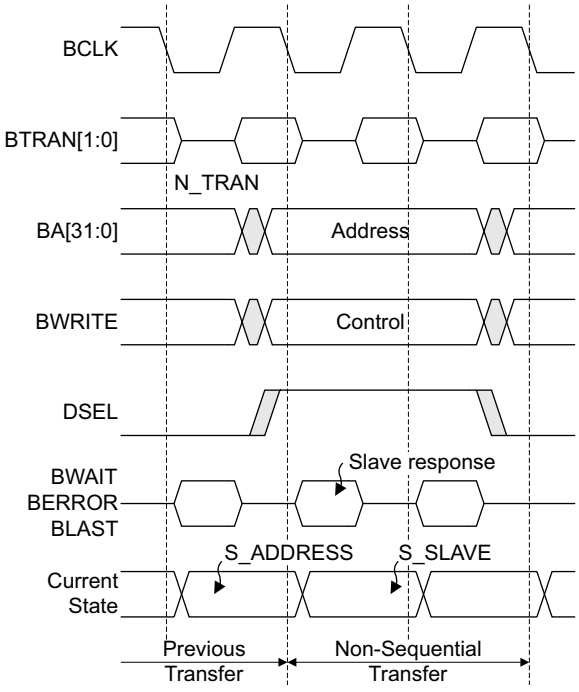


Figure 3-17 Decoder without decode cycles

Slave response signals

The decoder block monitors bus activity, and determines when a transfer to a slave device is required. When a transfer is required, a slave select signal is generated from the address bus decode.

When no slave is selected, the decoder must provide the slave response signals (**BWAIT**, **BLAST** and **BERROR**) in order for the bus to remain synchronized. The decoder will drive these signals during the address-only cycle (as indicated by the **BTRAN[1:0]** signals which are driven by the bus master), and the decode cycle, which occurs at the start of every nonsequential transfer and may also be requested by a slave device using the **BLAST** signal.

The decoder with decode cycles will provide the following responses:

Table 3-5 Decoder with decode cycle response combinations

Condition	BWAIT	BLAST	BERROR
Address-only cycle	0	0	0
Error	0	0	1
Decode cycle	1	0	0

The decoder without decode cycles does not provide the decode cycle response.

When a slave is selected it must provide a response on the **BWAIT**, **BERROR** and **BLAST** signals.

Internal decoder signals

The three internal decoder signals **DecError**, **DecLast** and **DecBlast** are required by the state machines:

- **DecError** is used by both state machines, with and without decode cycles. It is asserted when an access is made to an undefined region in the memory map.
- **DecLast** is only needed by the state machine with decode cycles. It is asserted when the last address in a 1KB boundary is accessed, and is dependent on the size of the transfer (byte, halfword or word).
- **DecBlast** is only needed by the state machine with decode cycles. It is asserted when the slave response signals indicate the last transfer in a burst, when both **BWAIT** and **BERROR** are LOW, and **BLAST** is HIGH.

Decoder state machine

There are two possible implementations of the decoder, depending on the performance requirements of the system design. One implementation inserts decode cycles, and the other does not. The state machine for the decoder with decode cycles has an extra state, **ST_DECODE**.

The decoder state machine is clocked off the falling edge of the bus clock, **BCLK**. Therefore, it is necessary for the decoder to use latched versions of the **BWAIT** and **BLAST** signals (only valid during LOW phase of **BCLK**) during the next state generation.

The decoder will drive:

- an address-only cycle response when in the **ST_ADDRESS** state
- a decode cycle response when in the **ST_DECODE** state
- an error transfer response when in the **ST_ERROR** state.

The decoder will not drive the slave response signals in the **ST_SLAVE** state as this is the responsibility of the selected slave.

Figure 3-18 shows the state machine of the decoder with decode cycles.

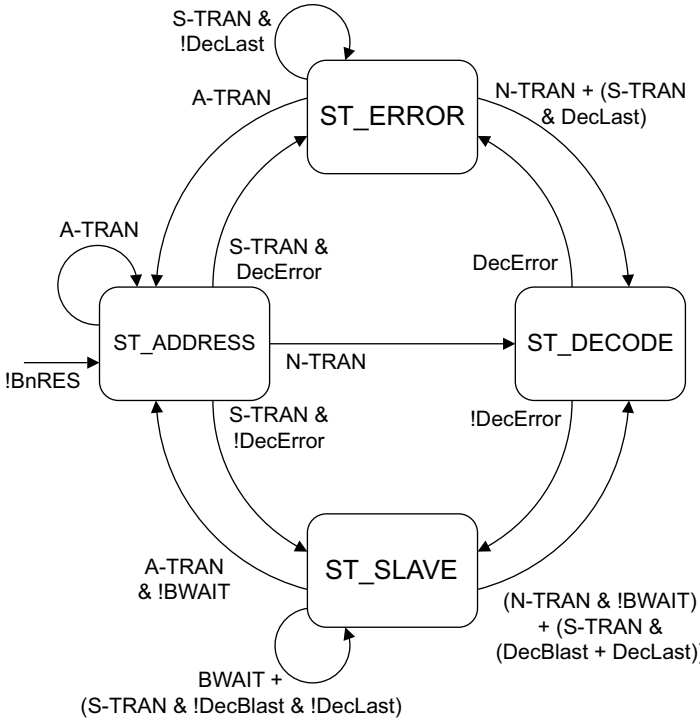


Figure 3-18 Decoder with decode cycles state machine

ST_ADDRESS

The no transfer state is entered when no data transfer is occurring on the bus, or during a reset condition. The **ST_ADDRESS** state is only exited when a transfer is about to occur (N-TRAN or S-TRAN), as indicated by the **BTRAN** signals. If the transfer is nonsequential, the next state will be **ST_DECODE** to allow time to decode the address, but if the transfer is sequential, and **DecError** is not asserted, the next state is **ST_SLAVE**.

ST_DECODE

The **ST_DECODE** state is only entered for a single cycle when address decode is occurring. During normal operation, the next state will always be **ST-SLAVE**. If however, the current address gives an error then the next state will be **ST_ERROR**. **BWAIT** is driven HIGH during this state.

ST_SLAVE

The **ST_SLAVE** state is used when a transfer is occurring to a slave and may be entered from either the **ST_DECODE** or the **ST_ADDRESS** state. The state machine will remain in the **ST_SLAVE** state when **BWAIT** is driven HIGH by the slave, giving the slave time to finish the transfer.

When **BWAIT** driven is LOW by the slave, the next state is:

- **ST_ADDRESS** if the transfer type is address-only
- **ST_DECODE if the transfer type is nonsequential or if DecBlast or DecLast have been asserted. However, if the transfer type is sequential and BLAST or DecLast have not been asserted, the next state remains as ST_SLAVE**.

ST_ERROR

The **ST_ERROR** state is used when an error occurs and is entered when **DecError** is asserted. The state machine remains in the **ST_ERROR** state when the transfer type is sequential, and **DecLast** is not asserted, as the current address will still be to an invalid location.

The next state becomes **ST_ADDRESS** if the transfer type is address-only, or **ST_DECODE** if the transfer type is nonsequential, or if it is sequential and **DecLast** is asserted.

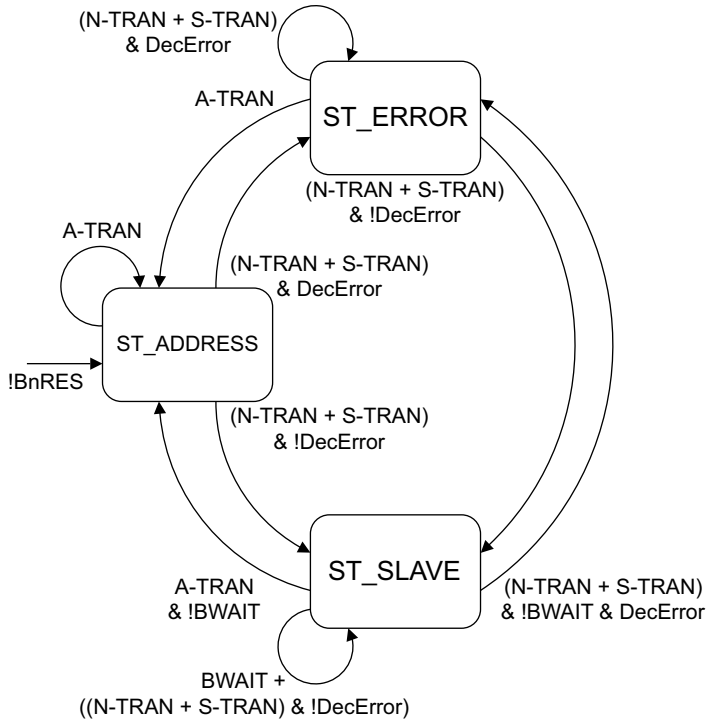


Figure 3-19 Decoder without decode cycles state machine

ST_ADDRESS

The no transfer state is entered when no data transfer is occurring on the bus, or during a reset condition. The **ST_ADDRESS** state is only exited when a transfer is about to occur (N-TRAN or S-TRAN), as indicated by the **BTRAN** signals. If **DecError** is asserted, the next state is **ST_ERROR**, if it is not asserted then the next state is **ST_SLAVE**.

ST_SLAVE

The **ST_SLAVE** state is used when a transfer is occurring to a slave and may be entered from either the **ST_ERROR** or the **ST_ADDRESS** state.

The state machine will remain in the **ST_SLAVE** state when **BWAIT** is HIGH. When **BWAIT** is LOW the next state becomes:

- **ST_ADDRESS** if the transfer type is address-only
- **ST_ERROR** if **DecError** has been asserted

- **ST_SLAVE** if **DecError** has not been asserted.

ST_ERROR

The **ST_ERROR** state is entered when **DecError** is asserted. The state machine remains in the **ST_ERROR** state when the transfer type is sequential or nonsequential, and **DecError** is asserted. The next state is **ST_ADDRESS** if the transfer type is address-only, or **ST_SLAVE** if the transfer type is sequential or nonsequential and **DecError** is not asserted.

Address decoder

The address decoder section must be changed to implement a different memory map. The default system memory maps are shown in Figure 3-15 on page 3-29.

Signals other than the address bus may be used to affect the address map. For example **Remap** creates two different memory maps depending on the current state of the system.

Reset operation

During a reset condition, when **BnRES** is LOW, the decoder block asynchronously removes all slave select signals, and initializes to the **ST_ADDRESS** state.

3.3.5 System description

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

A simple block diagram of the whole system is shown in Figure 3-20 on page 3-38.

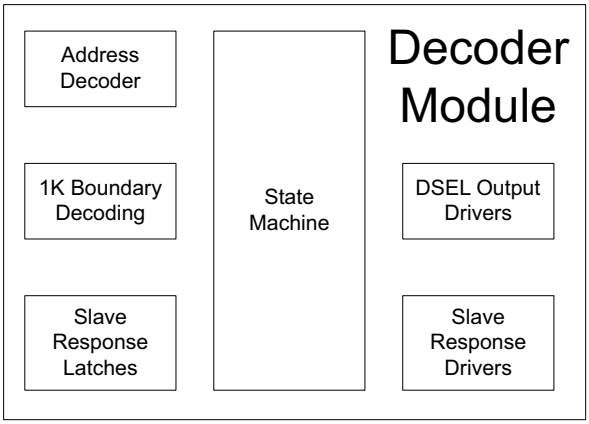


Figure 3-20 Decoder module block diagram

The decoder comprises the state machine which is used to control the generation of the output signals, and the address decoding which is used to determine which ASB slave is to be selected for the current transfer.

All registers used in the system are clocked from the same signal, the system clock **BCLK**, from either the rising or the falling edge. All registers use the asynchronous reset **BnRES**.

A diagram of the decoder HDL file is shown in Figure 3-21 on page 3-39.

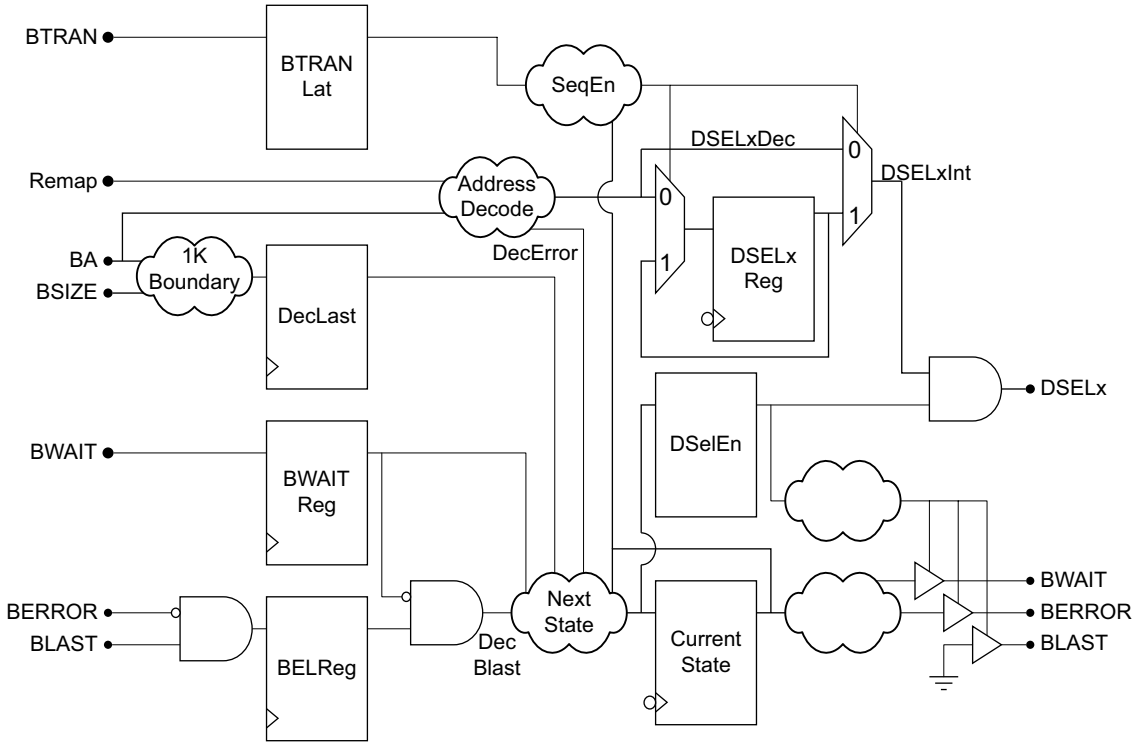


Figure 3-21 Decoder module system diagram

The main sections and processes in the code are:

- *Constant and signal definitions* on page 3-40.
- *ASB address decoding* on page 3-40.
- *Address decoding of 1K boundaries* on page 3-40.
- *Internal DSEL generation* on page 3-40.
- *DecLast generation register* on page 3-41.
- *Input slave response registers* on page 3-41.
- *Next state logic* on page 3-41.
- *State machine* on page 3-42.
- *Current state decoding* on page 3-42.
- *DSelEn generation* on page 3-42.
- *Slave response tristate enable* on page 3-42.
- *DSELx output port drivers* on page 3-42.
- *Tristate slave response output drivers* on page 3-42.

Each of these is now explained in more detail in the following paragraphs.

Constant and signal definitions

The constant **DECEN** is used to define the mode of the decoder, with or without decode states. This is the only part of the HDL file that needs to be altered to switch between modes, and should be set to the correct value before synthesis or simulation.

Note

The simulation buswatcher must be aware of the decoder type being used in the system, which enables it to ignore late arriving ASB signals during the decode cycles. The HDL contains a BUS _DEC_EN constant, which should be set to the same value as the decoder's constant.

The next four constants are the state machine states. The state bits are explicitly defined to minimize the number of pin changes during normal system operation, so only one bit changes between **ST_SLAVE** and **ST_ADDRESS**, and **ST_SLAVE** and **ST_DECODE**.

The signals that are used inside the module are then defined.

ASB address decoding

Address decoding of **BA** is performed on every change of **BA** and **Remap**, setting the internal signals **DSELxInt** with the relevant values. **DecError** is also generated in this section if the current address is not defined in the system memory map.

To change the system memory map, the generation of the **DSELxInt** signals should be changed, using the required number of **BA** pins for each address area.

Address decoding of 1K boundaries

This part is used to determine if the current transfer should be set to the last of a burst when the next sequential transfer would cross a 1K boundary. **BSIZE** is used to set **DecLastMux** when the current transfer is the last word, halfword or byte before a 1K boundary.

Internal DSEL generation

A late arriving, or changing address input will make the decoded **DSELx** lines invalid for a short time. Registers are used to store these values during sequential transfers to create cleaner **DSELx** outputs. The **DSELxReg** registers hold the previous **DSELxDec** values. **SeqEn** is used to generate the **DSELxInt** output drives from the **DSELxDec**, or **DSELxReg** signals, according to the transfer type, and **CurrentState**.

DecLast generation register

This register is used to hold the value of **DecLast** while the current address changes, so that the decode cycle is inserted after the current transfer has finished, and before the first transfer over the 1K boundary.

Input slave response registers

Both versions of the decoder use the **BWAIT** input to determine the next state of the state machine. This is captured on the rising edge of **BCLK** and is used in the generation of **NextState** and **DecBlast**.

All three slave response inputs are used to generate the **DecBlast** signal for the decoder with decode cycles, so the **BERROR** and **BLAST** inputs need to be registered. As they are only used together, a single register can be used on the output of combinational logic of the **BERROR** and **BLAST** inputs, and this can be combined with the output of the **BWAIT** register. This is preferable to using three registers and then combining the outputs of all three of them.

The **DECEN** constant is used on the **DecBlast** signal to hold its value LOW when the decoder without decode cycles is being used. This allows the **DecBlast** generation logic to be removed during synthesis.

The **BTRAN** input is latched for use in the generation of **SeqEn**.

Next state logic

This is the combinational part of the state machine, the next state generation. The next state of the state machine is calculated on the basis of:

- the current state of the state machine
- the **BTRAN** inputs
- the **DecError** and **DecBlast** signals.

The **DECEN** constant is used to change the operation of the state machine for the two different decoders, enabling and disabling different parts of the next state logic depending on the value of **DECEN**.

The system resets into the **ST_ADDRESS** state, and stays in this state until a transfer occurs. The next state becomes **ST_SLAVE,** if the current transfer is to a valid address in the memory map, and **ST_ERROR** if not to a valid location. The state machine without decode cycles stays in **ST_SLAVE** for the duration of the transfer, with the decoder with decode cycles alternating between **ST_DECODE,** and **ST_SLAVE** to allow time for the address to be decoded.

State machine

With these registers **NextState** is loaded into **CurrentState** on the falling edge of **BCLK**.

Current state decoding

The slave response output signals are generated in this part of the code, depending on the current state of the state machine. Internal versions of the signals are used to drive the tristate outputs.

If the current state is **ST_ADDRESS** the **BWAIT**, **BERROR** and **BLAST** signals are driven LOW. **BWAITInt** is set HIGH during **ST_DECODE** (if used), and **BERRORInt** is set HIGH during **ST_ERROR**. The outputs are high impedance while the current state is **ST_SLAVE** to allow the slave to drive them.

DSelEn generation

DSelEn is used to enable the internal **DSELxInt** signals to be driven onto the **DSELx** output ports. Because the value of **NextState** can change on either side of the rising edge of **BCLK**, and the **DSELx** outputs have to become valid during the HIGH phase of **BCLK**. A latch has to be used to set **DSelEn** during the HIGH phase of **BCLK**, and to hold its value during the LOW phase of **BCLK**.

Slave response tristate enable

The **BWELEn** signal is used to enable the tristate outputs of the slave response signals. It is driven HIGH during the LOW phase of **BCLK,** when **DSelEn** is also LOW.

DSELx output port drivers

In this part the **DSELx** output ports are driven with a combination of the internally generated **DSELxInt** signals and the **DSELEn** enable signal.

Tristate slave response output drivers

The three slave response signals are driven with their internal versions when the **BWELEn** enable signal is HIGH, and are tristated at all other times.

3.4 Reset controller

The AMBA specification defines a single reset signal, **BnRES**, which indicates the current reset status of the system.

This section describes the AMBA Reset Controller, which drives the **BnRES** signal. The AMBA reset controller is shown in Figure 3-22.

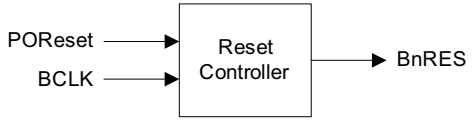


Figure 3-22 Reset controller block diagram

Table 3-6 Signal descriptions

Name	Type	Description
BCLK	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers on the bus.
POReset	Input	Power on reset input. This signal causes a cold reset when HIGH. May be asserted asynchronously to **BCLK**.
BnRES	Output	Reset output. This signal indicates the current reset status.

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

3.4.1 Signal timing

Assertion (the falling edge) of **BnRES** is asynchronous to **BCLK**. De-assertion (the rising edge) of **BnRES** is synchronous to the falling edge of **BCLK**. **BnRES** is only asserted during a Power-On Reset condition, caused by the assertion of the **POReset** signal. The **POReset** input is an asynchronous input, and hence a synchronizing d-type is required to eliminate propagation of metastable values. Figure 3-23 on page 3-44 shows the operation of the **BnRES** signal with respect to the state machine states and an example **POReset** input signal.

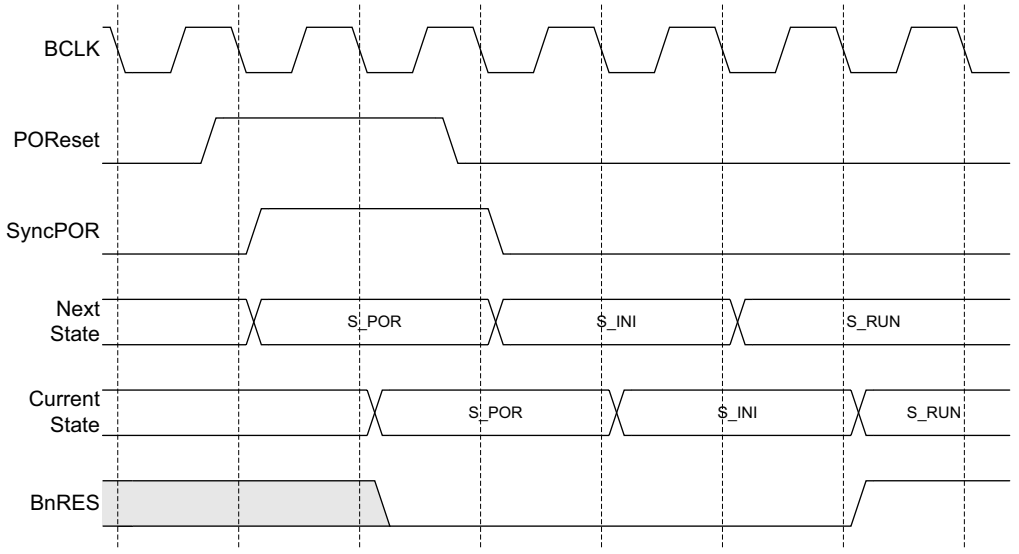


Figure 3-23 BnRES timing

3.4.2 Use of BnRES

BnRES is used to indicate a reset condition. **BnRES** is asserted LOW and is used to indicate that 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 register states that requires initialization.

During reset, the arbiter grants the bus to the default bus master and holds all other grant signals inactive. The decoder negates all select signals, and drives the slave response signals LOW.

3.4.3 Bus reset state machine

The reset controller comprises a state machine running off the falling edge of **BCLK**. The **BnRES** signal directly reflects the bit 0 of the state number shown in Figure 3-24 on page 3-45, which shows the state machine used in the system.

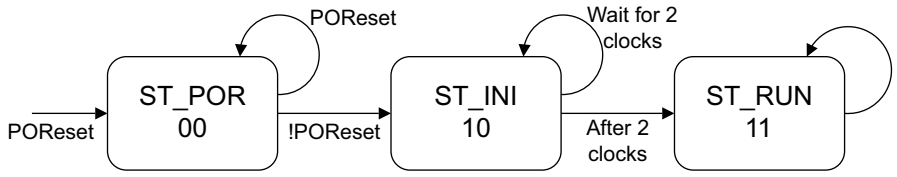


Figure 3-24 State machine for reset controller

ST_POR

This power on reset state initializes all of the system state and ensures that one tristate driver is enabled on the AMBA system bus. Any peripheral state that is initialized on reset is initialized in this state.

The **ST_POR** 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). This major reset is forced as an asynchronous start-up condition and must be recognized by all master and slave devices on the system bus. This state is exited synchronously to the system clock **BCLK**. If there is a clock valid signal in the system, this should be used in the reset controller to prevent the **ST_POR** state from being exited until the clock is valid.

ST_INI

The **ST_INI** state is used to hold the **BnRES** signal asserted (LOW) for some extra clock cycles after the **POReset** signal is de-asserted. In the current implementation this state is maintained for at least two clock cycles, but this period can be increased if necessary. This state, and the **ST_RUN** below, are all entered and exited synchronously to the bus clock.

ST_RUN

ST_RUN is the normal system operation mode: the bus arbiter allocates resources, normal transactions are allowed and the bus clock runs at the normal speed.

3.4.4 System description

The following paragraphs give a detailed description of how the HDL code for the system 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 all of the registers, and signals used in the system. This part should be read together with the HDL code.

A simple block diagram of the whole system is shown in Figure 3-25 on page 3-46.

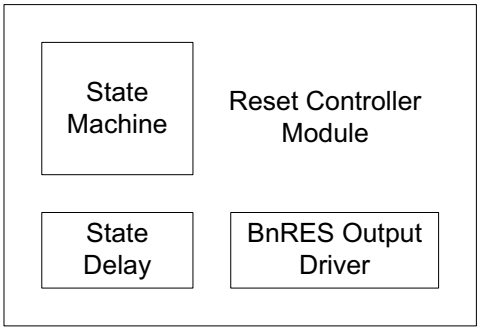


Figure 3-25 Reset controller module block diagram

The reset controller comprises the state machine, which is used to control the generation of the reset output signal, and the counting register that is used to hold the system in the **ST_INI** state for two clock cycles.

All registers used in the system are clocked from the same signal, the falling edge of the system clock **BCLK**.

A diagram of the reset controller HDL file is shown in Figure 3-26.

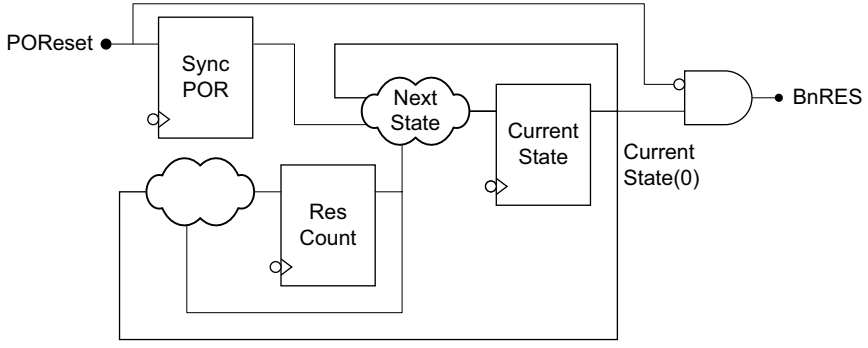


Figure 3-26 Reset controller module system diagram

The main parts and processes in the code are described in the following sections:

- *Constant and signal definitions* on page 3-47
- *Asynchronous input reset synchronization* on page 3-47
- *Next state logic* on page 3-47
- *State machine* on page 3-47
- *ResCount register* on page 3-47
- *Output driver* on page 3-48.

Constant and signal definitions

The three constants are the state machine states. The state bits are explicitly defined to allow one bit of the state machine to be used as the **BnRES** signal (bit 0), reducing the complexity of the module, and avoiding the chance of glitches on the **BnRES** output.

Asynchronous input reset synchronization

POReset is first passed through the falling edge triggered register, to avoid metastability, due to the arrival time of the input. It generates the **SyncPOR** signal, which is then used as the input to **NextState** and the reset for the **ResCount** register.

Next state logic

This is the combinational part of the state machine, the next state generation. Based on the current state of the state machine, the **SyncPOR** input, and the **ResCount** register output, the next state of the state machine is calculated.

The system resets into the **ST_POR** state, and stays in this state until the external **POReset** input signal becomes LOW. The next state then becomes **ST_INI**, and this is held until the two cycle delay produced by **ResCount** is over. Then, the system moves into the **ST_RUN** state, which is held until the system power is removed, or the external **POReset** signal is set again.

State machine

With these two registers, **NextState** is loaded into **CurrentState** on the falling edge of **BCLK**. There is no reset used as the signal **NextState** already includes a reset, and **CurrentState** will get set on the very first falling edge of **BCLK**.

ResCount register

This falling edge register is used to hold the state machine in the **ST_INI** state for two cycles. The **ResCountMux** signal is used to set the register depending on the values of **CurrentState** and **ResCount**. When the current state is **ST_INI**, and **ResCount** is LOW, then the mux term is set HIGH, also setting the register HIGH on the next falling edge of **BCLK**. This then allows **NextState** to be set to **ST_RUN**, and the **BnRES** output becomes HIGH on the next falling edge of the clock. **SyncPOR** is used as the reset for the **ResCount** register. Figure 3-27 on page 3-48 shows the timing of these signals:

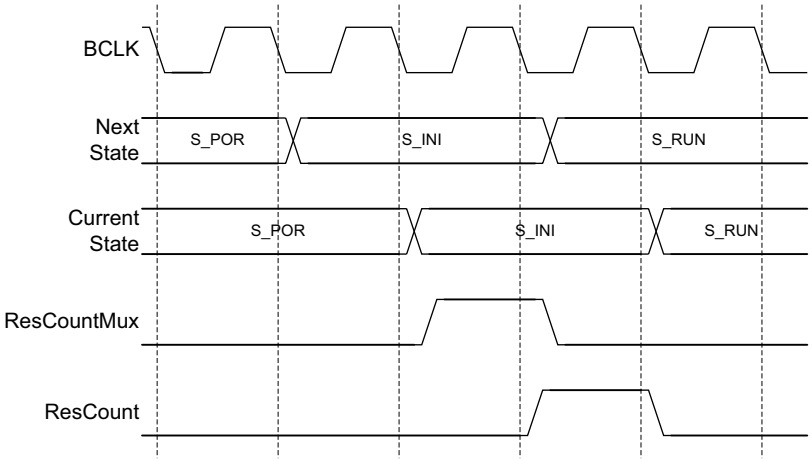


Figure 3-27 Delayed state change timing diagram

Output driver

This section drives the **BnRES** output port with bits 0 of **CurrentState** ANDed together with the inverse of the active HIGH **POReSet** input.

3.5 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 an off-the-shelf 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 ASB to the external memory bus of an AMBA Microcontroller. This allows the connection of up to eight 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. This design comprises four functional blocks:

- the bus latches and drivers
- the clock and enable logic
- the bank select logic
- the memory wait state counter.

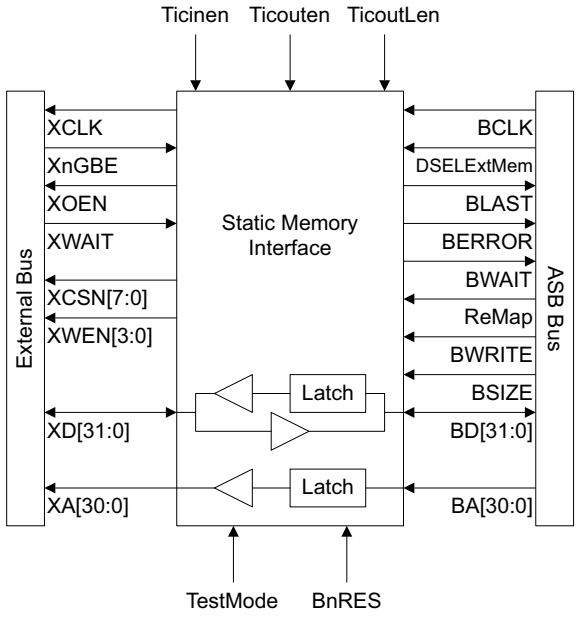


Figure 3-28 Static memory interface block diagram

Table 3-7 Signal descriptions

Name	Type	Description
BCLK	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers on both the ASB and APB.
BnRES	Input	The bus reset signal is active LOW and is used to reset the system and the bus.
BA[30:0]	Input	The system address bus (excluding the most significant bit). The addresses change during the **BCLK** HIGH phase before the transfer to which they refer and remain valid until the last **BCLK** HIGH phase of the transfer.
BWRITE	Input	When HIGH this signal indicates a write transfer and when LOW a read transfer. This signal has the same timing as the address bus.
BSIZE[1:0]	Input	These signals indicate the size of the transfer which may be byte, halfword or word. These signals have the same timing as the address bus.
DSELExtMem	Input	When HIGH, this signal indicates that the SMI is selected. This signal has the same timing as the address bus.
Remap	Input	When LOW this forces the memory in bank 7 to be mapped to all locations in the SMI's address range. This bank should contain the system's start-up program (boot ROM/BIOS). In normal operation this signal is HIGH.
BD	Input/Output	This is the bidirectional system data bus. The data bus is driven by this block during read transfers from external memory, or during system test.
BWAIT	Input/Output	This slave response is driven HIGH by this block during the LOW phase of **BCLK**, when the SMI is selected and the external memory has not completed its current transfer.
BERROR	Output	The SMI does not support this slave response (this is not a general rule for EBIs). It is driven LOW when the SMI is selected during the LOW phase of **BCLK**.
BLAST	Output	The SMI does not support this slave response (this is not a general rule for EBIs). It is driven LOW when the SMI is selected during the LOW phase of **BCLK**.
TestMode	Output	Indicates that the test controller has taken control of the bus. Used to enable the external data bus XD.
Ticinen	Output	When LOW, this indicates that the SMI should enable the test bus (**TBUS**) to drive BD. In the case of the SMI, the test bus is identical to the data bus XD.
Ticouten	Input	When this signal is LOW, the SMI's latched version of **BD** drives **TBUS**.
TicoutLen	Input	When this signal is LOW, the **BD** latch is transparent during the BCLK HIGH phase.

Table 3-7 Signal descriptions (continued)

Name	Type	Description
XWAIT	Input	When HIGH during the **BCLK** LOW phase of a transfer, this signal will force the AMBA bus to wait. The current transfer will complete once **XWAIT** is LOW, and the wait states for the transfer are complete. This signal will normally be tied LOW in the default system.
XnGBE	Input	External global bus enable. When this signal is driven HIGH, the external address and data bus will be driven to high impedance. This should be used in conjunction with **XWAIT**. Under normal conditions this signal should be driven LOW.
XD[31:0]	Input/Out	This is the bidirectional external data bus. In normal operation it is driven by the external bus when **XOEN** is LOW, and by this block when **XOEN** is HIGH. During system test this becomes the test bus **TBUS** and its direction is controlled by the TIC signals.
XA[30:0]	Output	The external address bus is driven when **XnGBE** is LOW. The bus becomes valid during the **BCLK** LOW phase of the transfer and remains valid throughout the rest of the transfer.
XCLK	Output	External clock for peripherals off-chip.
XCSN[7:0]	Output	These signals are active LOW chip enables for each of the eight banks (0-7) of static memory. **XCSN[7]** should be connected to the memory containing the startup program (boot ROM/BIOS) for the system.
XOEN	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]	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 one wait state when writing to memory. **XWEN** becomes valid in the **BCLK** HIGH phase of the first cycle of the transfer and remains valid during the **BCLK** LOW phase of the second cycle.

3.5.1 Functional description

The SMI has five functions in the example system that are described in the following sections:

- *External bus control* on page 3-52
- *Memory bank select* on page 3-53
- *Memory write control* on page 3-54
- *Configurable memory access wait states* on page 3-55
- *System test access* on page 3-55.

External bus control

In normal operation (not in system test mode), to perform a read from the external memory, the **XnGBE** signal must be LOW, to enable driving of the address and data buses (**XA** and **XD**). The latched address (taken from **BA**) is driven onto **XA**. If **XOEN** is LOW (indicating a read) **XD** is driven to a high impedance, to allow the read data to be applied.

Figure 3-29 shows the timing of a read from memory with zero wait states. Note that the data must be valid on the **XD** bus in time for the signal to propagate on-chip so that the **BD** bus becomes valid before the next falling edge of **BCLK**. If this setup time cannot be achieved, the access will require wait states.

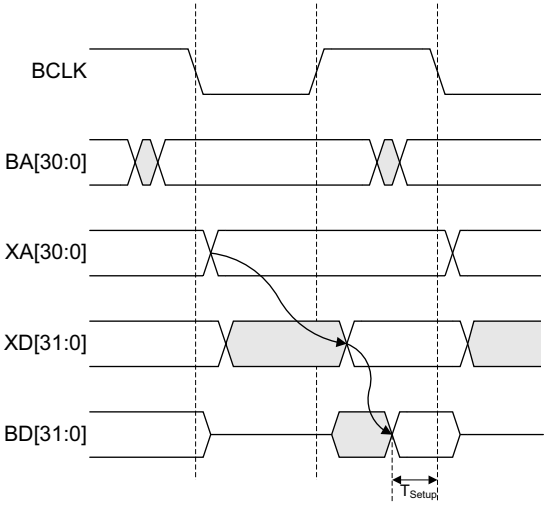


Figure 3-29 Zero wait memory read

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

The SMI requires at least one wait state 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 3-30 on page 3-53, shows the timing of a write to memory with a single wait state.

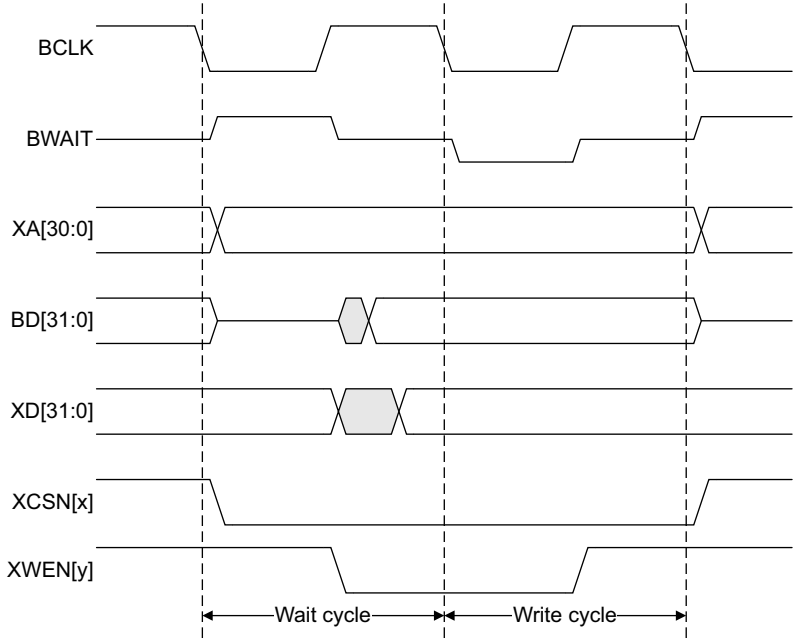


Figure 3-30 Memory write with one wait state

Memory bank select

The **XCSN** chip select lines are controlled by the values of **BA**, **Remap,** and **DSELExtMem.** A falling edge registered version of **DSELExtMem** is used, and a latched version of **BA** is used (transparent when **BCLK** is LOW), so **XCSN** is effectively generated on the falling edge of **BCLK**.

Table 3-8 on page 3-53 shows the relationship between the three inputs and the generated value of **XCSN**.

Table 3-8 XCSN coding

Inputs			Output
DSELExtMem	**Remap**	**BA[30:28]**	**XCSN[7:0]**
0	X	XXX	11111111
1	0	XXX	01111111
1	1	000	11111110

Table 3-8 XCSN coding (continued)

Inputs			Output
DSELExtMem	**Remap**	**BA[30:28]**	**XCSN[7:0]**
1	1	001	11111101
1	1	010	11111011
1	1	011	11110111
1	1	100	11101111
1	1	101	11011111
1	1	110	10111111
1	1	111	01111111

XCSN[7:0] is also held in the "11111111" 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[0] controls XD[31:24].

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

Table 3-9 on page 3-54 shows the bytes selected according to the **BSIZE** and **BA[1:0]** inputs.

Table 3-9 XWEN coding

BSIZE[1:0]	BA[1:0]	XWEN[3:0]
10 (word)	XX	0000
01 (half word)	0X	1100
01 (half word)	1X	0011

Table 3-9 XWEN coding (continued)

BSIZE[1:0]	BA[1:0]	XWEN[3:0]
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 one and three for writes. Figure 3-30 on page 3-53 shows a memory transfer with one wait state. A transfer with more wait states causes further wait cycles. The address and data information remains valid until the 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 selected by altering the 2-bit wide constants **READWAIT** and **WRITEWAIT**. **WRITEWAIT** must be value 01 or greater. The SMI also allows transfer wait to be extended indefinitely. This is done by asserting **XWAIT** HIGH. To wait the current transfer, **XWAIT** must be asserted before the rising edge of **BCLK** in the last waited cycle of the access.

Note

If the transfer is zero wait state, **XWAIT** should be asserted before the transfer commences to allow a **BWAIT** HIGH cycle to be inserted.

The transfer cannot complete until **XWAIT** is LOW for at least one cycle.

System test access

During system test the SMI is controlled by three active LOW signals from the TIC:

- **Ticinen** (test data in enable)
- **Ticouten** (test data out enable)
- **TicoutLen** (test data out latch).

For more information on system test, refer to the *AMBA Specification*. For the SMI, **BCLK** is used as **TCLK**, and **XD** as **TBUS**. The TIC signals control the data bus drivers and the latch directly. It is necessary to override the normal operation of the interface when in test mode. This is done with the **TestMode** signal from the TIC.

3.5.2 System description

The following paragraphs give a detailed description of how the HDL code for the 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, and signals used in the system. This section should be read together with the HDL code.

A simple block diagram of the whole system is shown below in Figure 3-31.

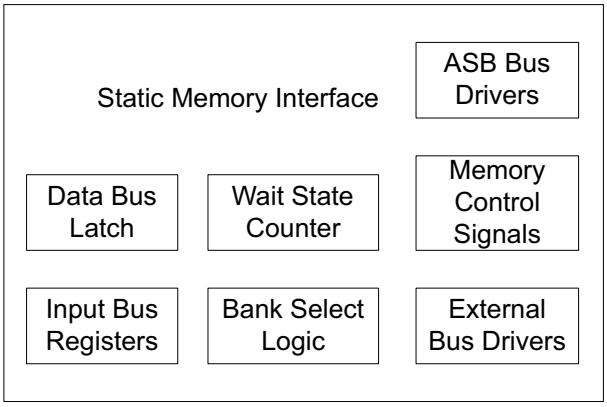


Figure 3-31 Static memory interface module block diagram

The static memory interface module comprises the input latches and 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 system clock **BCLK**, from either the rising or the falling edge. Enable signals are used to control the loading of the registers. All registers use the asynchronous reset **BnRES**.

A diagram of the static memory interface HDL file is shown in Figure 3-32 on page 3-57.

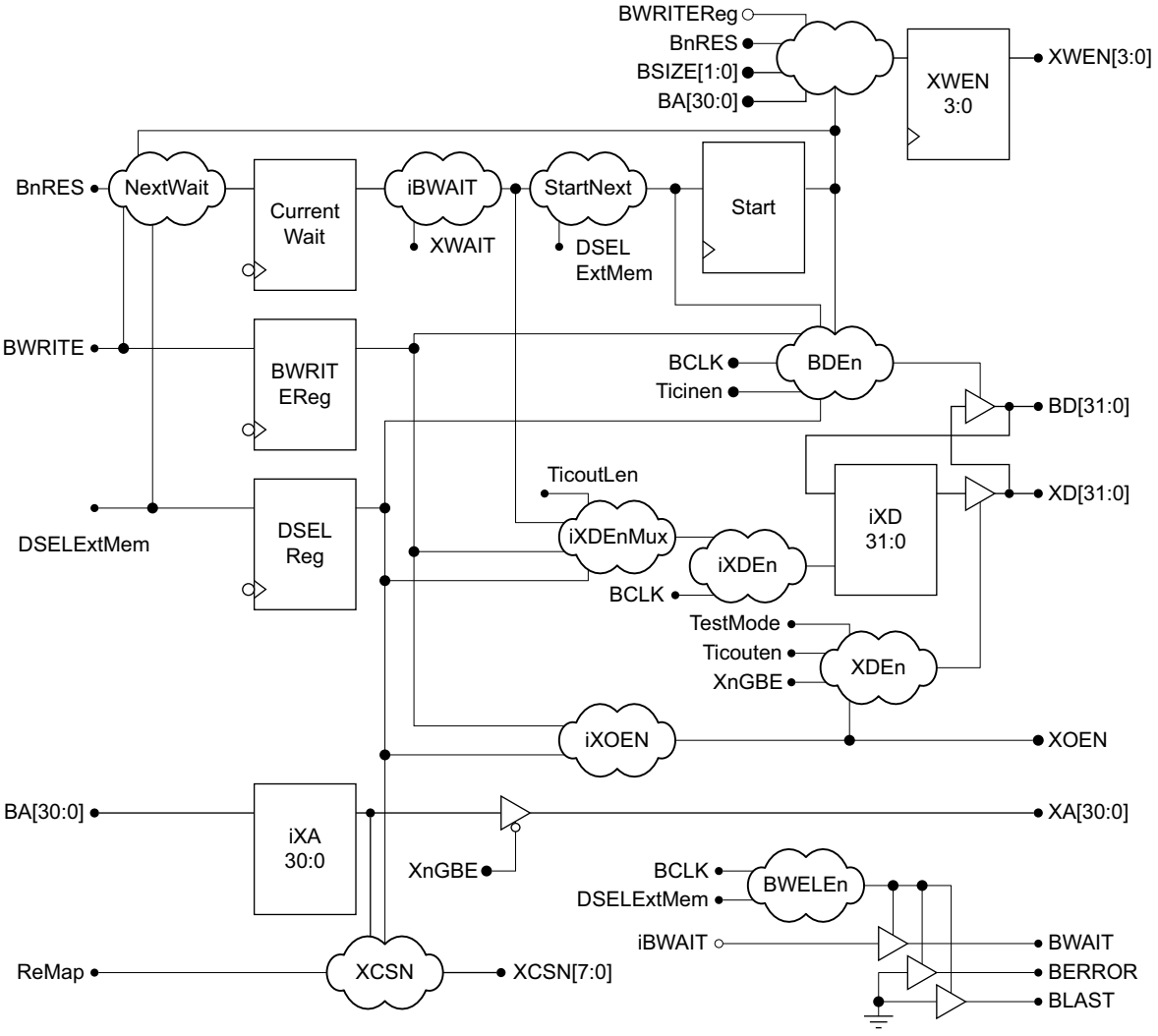


Figure 3-32 Static memory interface module system diagram

The main sections and processes in the code are described in the following sections:

- *Constant and signal definitions* on page 3-58
- *Registered input signals* on page 3-58
- *Wait state counter* on page 3-58
- *Start generation* on page 3-58
- *iXA generation* on page 3-59
- *XWENNext generation* on page 3-59.

Constant and signal 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 (one 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.

The next set of constants are used in the generation of the write enable (**XWEN**) output signals, and are chosen depending on the current values of **BA** and **BSIZE**. **XWENX** is used when there is an error on one of the **BSIZE** or **BA** inputs. The signals that are used inside the module are then defined.

Registered input signals

This section is used to generate registered versions of the input signals **BWRITE** and **DSELExtMem**, as some parts of the module need to use the registered versions to generate the correct timing on the outputs.

Wait state counter

The counter state machine is made up of two parts, the **NextWait** value generation and the register to hold the current wait count value. As soon as the external memory is selected with **DSELExtMem** being set HIGH, **NextWait** is loaded with the value of **READWAIT** or **WRITEWAIT**, depending on the type of access that is being performed. This is then loaded into the register on the next falling edge of **BCLK**, and is decremented every falling clock edge until it reaches zero.

Start generation

The **Start** signal is used to indicate the start of a transfer to, or from the external memory, and is generated by a register. The input signal to the register is split up into **iBWAIT** and **StartNext**. The **iBWAIT** signal is used to generate the **BWAIT** output and is used by other sections. **iBWAIT** is set HIGH when **CurrentWait** is not equal to zero, and is set to **XWAIT** when **CurrentWait** equals zero. This allows waits generated by the memory to be passed onto the ASB. This signal is then passed through to the register when the external memory has been selected, generating the **StartNext** signal.

The **Start** signal is fed back into the wait state counter to make it decrement the wait count value while the system is waiting for the transfer, and is also used in the generation of the enables for **XWEN** and **BD**.

iXA generation

The external memory address bus is a direct copy of the lowest 30 bits of the ASB address bus, but delayed by half a clock cycle. A latched version of **BA** is used to generate an internal version of **XA,** which is fed to the input of a tristate buffer connected to the **XA** bus. On reset **iXA** is LOW. Latches are used to generate **XA** with the correct timing, due to the possibility of **BA** becoming valid after the falling edge of **BCLK**. For systems with a slow clock, if **BA** is always valid before the falling edge of the clock, falling edge triggered registers can be used instead of latches.

XWENNext generation

The memory write enable signals are generated when the SMI is selected for a write transfer, and it is the start of transfer. **BA** and **BSIZE** are then decoded, as in Table 3-9 on page 3-54, to generate the **XWENNext** signals, which are passed onto the **XWEN** registers.

XDInt generation

Due to the timing requirements between the ASB data bus and the external memory data bus, transparent latches are used to connect **BD** to **XD**. During a memory write cycle, **XDInt** is a latched version of **BD**. It is held until the start of the next transfer, when the new data value is latched before use. On reset **XDInt** is set LOW.

iXOEN generation

The internal version of the output enable signal is generated from the registered **BWRITE** and **DSELExtMem** inputs to create the correct timing for the external bus. It is set LOW during reads, and HIGH at all other times.

Output enables

This section contains the output enables that are used to enable the internal versions of the output signals.

The **XDEn** and **BDEn** signals are generated from internal signals and the system test signals generated by the TIC module. This allows the external bus to be used either as a test interface or an external memory connection.

BWELEn is set HIGH when the external memory is selected, **BCLK** is LOW and **BnRES** is HIGH.

External bus output port drivers

This section contains the signals that are directly driven onto the outputs without being tristated. **XCLK** is just a direct copy of the system clock **BCLK**, with the same timing.

XCSN is generated from **DESLReg**, **Remap** and **iXA**. The top three bits of the address are decoded as shown in Table 3-8 on page 3-53. The boot ROM is selected on any external memory access, when **Remap** is LOW. On reset, or when the external memory is not selected, all **XCSN** bits are HIGH.

A register is used to generate **XWEN** from the internal signal, with all bits set HIGH on reset.

XOEN is directly driven by the internal signal **iXOEN**.

Tristate output drivers

The tristate outputs used in the system are driven in this section. These include:

- the ASB data bus
- the three slave response signals
- the external address and data buses.

3.6 Example system external memory

The external memory devices and their configurations are described below.

3.6.1 Memory devices

Two types of memory devices are used for external memory in the example system:

- a simple model of a SRAM, arranged 32KBx8, with active LOW write enable, output enable and chip select inputs
- a simple model of an EPROM arranged as 16KBx8, with active LOW output enable and chip select inputs.

  The demonstration memory models used in the example system are behavioral only, and include no data timing checks.

3.6.2 Memory connection

The memory is configured as:

- 7 banks of SRAM, accessed by chip selects **XCSN[6:0]**
- 1 bank of boot ROM, accessed by chip select **XCSN[7]**.

All SRAM and ROM banks are four bytes wide. Byte, halfword and word access is provided by the write enable XWEN[3:0] lines.

3.7 Test interface controller

Test Interface Controller (TIC) is a state machine that provides an AMBA bus master for system test. It controls the *External Bus Interface* (EBI) to sample or drive the ASB data bus **BD**.

The AMBA TIC is an ASB bus master that accepts test vectors from the external test bus (the 32-bit external data bus, if available) and initiates bus transfers. The TIC latches address vectors from the test bus and drives the ASB address bus.

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

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

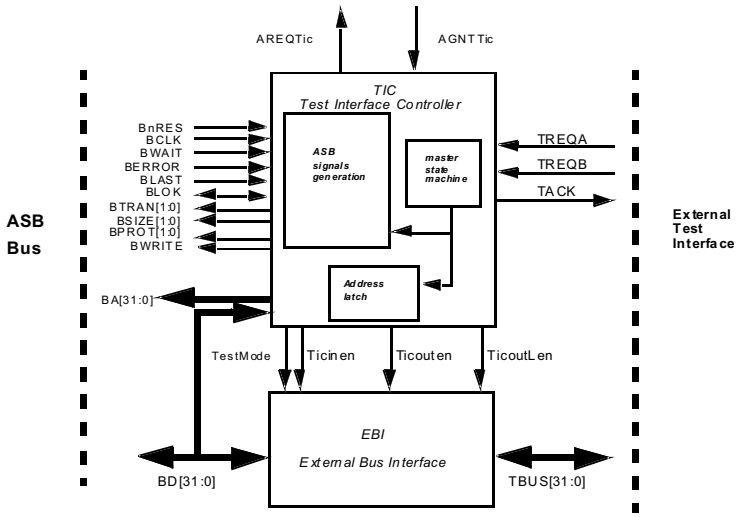


Figure 3-33 Test interface controller block diagram

Figure 3-33 on page 3-62 represents a TIC block in a system where the external 32-bit 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 test mode access. Such systems would use the **TestMode** signal to force the EBI into this state.

Table 3-10 Signal descriptions

Name	Type	Description
BCLK	Input	System (bus) clock. This clock times all bus transfers. In the example system this clock also operates in test mode as **TCLK**.
BnRES	Input	This active LOW signal indicates the reset status of the bus and is driven by the reset controller.
BWAIT	Input	This signal indicates when current transfer will complete. It is valid on the rising edge of **BCLK**. It must be used together with the other slave response lines **BERROR** and **BLAST**.
BERROR	Input	This signal is used with **BWAIT** and **BLAST** to form the retract term in the bus master state machine.
BLAST	Input	This signal is used with **BERROR** and **BWAIT** to form the retract term.
BD[31:0]	Input	In test mode, the data bus is used to load values into the TICs address latch during address vectors. The TIC then drives the system address bus **BA** with this value during subsequent single/burst read-write vectors.
AGNTTic	Input	Grant signal that grants the bus to the test controller.
AREQTic	Output	Request from the TIC, indicating that this master requires the bus. This signal must be set up to the falling edge of **BCLK**. The arbiter should treat this signal as the highest priority request line (over and above any complex arbitration scheme it might support).
BSIZE[1:0]	Output	These signals indicate the size of the transfer. By default the transfer size is always a word (32 bits), but this may be changed using the control vector. These signals have the same timing as the address bus.
BTRAN[1:0]	Output	These signals indicate the type of the next transaction, which in this master may be Address-only or Sequential. They are valid during the HIGH phase before the transfer to which they refer.
BLOK	Input	Bus lock signal. By default the TIC does not perform locked transfers. The control vector may be used to force the TIC to perform locked transfers.
BPROT[1:0]	Output	These signals deal with address location access protection control. By default the TIC signals indicate supervisor mode transfers, but this may be changed using the control vector. They have the same timing as the address bus.
BA[31:0]	Output	System address bus. The addresses become valid during the HIGH phase before the transfer to which they refer, and remain valid until the last HIGH phase of the transfer.

Table 3-10 Signal descriptions (continued)

Name	Type	Description
BWRITE	Output	When HIGH, this signal indicates a write transfer, when LOW, a read. This signal has the same timing as the address bus.
TREQA	Input	Test request A. This signal is used, in combination with **TREQB**, to control access to the system bus from the test bus.
TREQB	Input	Test request B. This signal is used, in combination with **TREQA**, to control access to the system bus from the test bus.
TACK	Output	Test acknowledge. This signal is used to indicate that the test interface has been granted access to the system bus. It is also used to indicate transfer delays (transfers with wait cycles).
TestMode	Output	Indicates that the test controller has taken control of the bus. It should be used to enable the external 32-bit test bus (for EBIs that need the **TBUS** to be specifically enabled) and to select the system clock source for test (**TCLK**).
Ticinen	Output	This active LOW signal indicates that the EBI should drive **TBUS** onto **BD**.
TicoutLen	Output	When LOW, the **BD** latch in the EBI should be transparent.
Ticouten	Output	This active LOW signal indicates that the EBI should drive its latched version of **BD** onto the external **TBUS**.

3.7.1 Functional description

The TIC has two fundamental modes of operation:

- ASB bus master
- system test.

The ASB bus master is selected when an AMBA system is in low power mode (if supported). When granted as the low power master, the TIC performs no test functions but keeps the AMBA bus in a state such that:

- no data transfers occur (**BTRAN** = A-TRAN)
- the bus is available to be granted to another master when requested (**BLOK** = LOW).

For system test, an external tester is used to drive the external pins **TREQA** and **TREQB.**

Table 3-11 Basic TIC operation

Inputs		Outputs	
TREQA	**TREQB**	**TACK**	**TIC mode**
0	0	0	Acting as ASB bus master
1	0	0	Entering test mode
1	1	1	Test mode entered, ADDRESS vector.
X	X	X	During test *(see note)*
0	0		Leaving test mode

Note

During test at least one of **TREQA** and **TREQB** must be HIGH. If both are LOW this indicates end of test.

When entering test mode, the TIC requests the bus. Once it is granted as the current bus master test mode is entered, and the **TACK** signal is pulled HIGH.

Clocking issues

When not in test mode an AMBA system is clocked by **BCLK**. The source of **BCLK** is not defined in the AMBA standard and may be from an off-chip source or more likely an on-chip PLL for low power consumption. In both cases **TREQA** is synchronized in the TIC by the rising edge of **BCLK**.

When test mode is entered the AMBA system will be clocked by **TCLK**, generated from an external source. Then **TREQA** and **TREQB** signals should be set up to the rising edge of **TCLK**.

The switch over from an internal **BCLK** source to an external **TCLK** source is not handled by the TIC. If an external **BCLK** source is used **BCLK** and **TCLK** can be identical.

Note

The Example AMBA System uses an externally generated **BCLK** (**BCLK=TCLK**).

This data sheet details one form of TIC that assumes an external clock can be used to drive the system during test. If this is not feasible it is possible for a TIC to run vectors at a much slower rate than **BCLK**. This possibility is not covered in this document.

Test mode

In test mode the signal **TestMode** is driven HIGH. This forces **BCLK** to be driven from an external source (**TCLK**) and the EBI to provide a 32-bit bidirectional channel through which values can be read and written to **BD**. This 32-bit channel is referred to as **TBUS**, though in systems with a 32-bit external data bus **TBUS** will be identical to the external data bus.

In AMBA systems which do not have a full 32-bit external data bus, address pins may have special test functionality (connecting bidirectional pads that can act as inputs during system test) to provide a **TBUS** connection.

The **TREQA** and **TREQB** signals should both be high on entering test mode. They are used to control the TIC which allows values to be read or written to any address location inside the microcontroller (it cannot perform read or writes to external memory as the external bus is occupied by the test bus, **TBUS**). This is done by applying TIC vectors, of which there are three basic types:

- READ data
- WRITE data
- ADDRESS vector.

Before a READ or WRITE vector can access a location the appropriate address must have been loaded. Therefore, the first vector should be an ADDRESS type at the beginning of testing.

Table 3-12 TIC vectors

TREQA	TREQB	Vector
1	1	ADDRESS vector. This must be the first and last vector in a test sequence. It is also used as a turnaround cycle after a write vector.
1	0	WRITE data. This must be followed by a turnaround cycle.
0	1	READ data.
0	0	End of test. This must be proceeded by an ADDRESS vector.

The state machine diagram, shown in Figure 3-34 on page 3-67, illustrates one relationship between TREQA, TREQB and the test vector type.

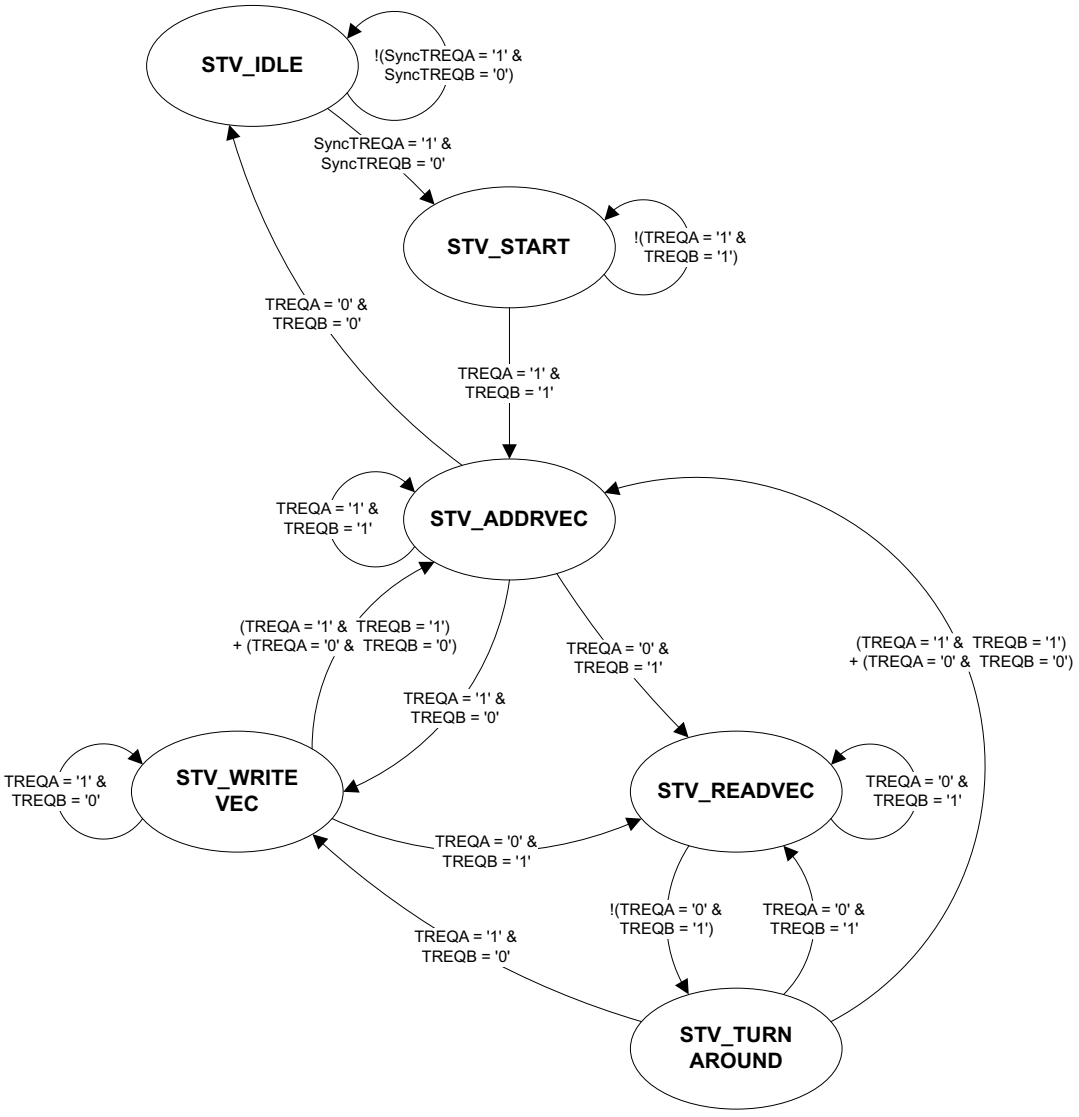


Figure 3-34 State machine

When the vectors are applied the **TACK** signal is monitored. This is normally HIGH. However, if the transfer initiated by the previous vector is not complete, **TACK** will go LOW off the falling edge of **TCLK**. When this occurs, the next vector is held until **TACK** goes HIGH. This is shown inFigure 3-36 on page 3-68. Vector 1 starts a transfer with one wait cycle, vector 2 is held while the transfer completes.

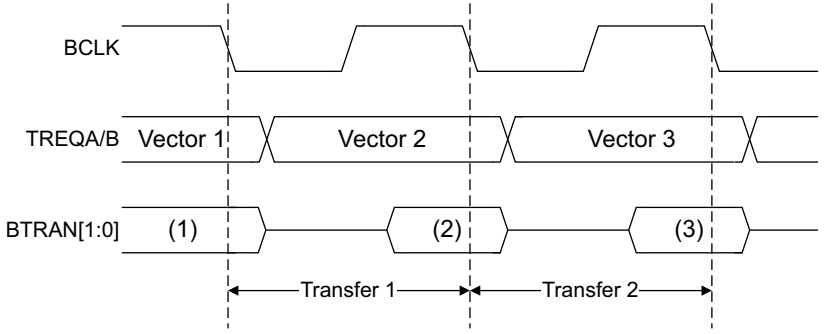


Figure 3-35 TIC vectors and AMBA transfers

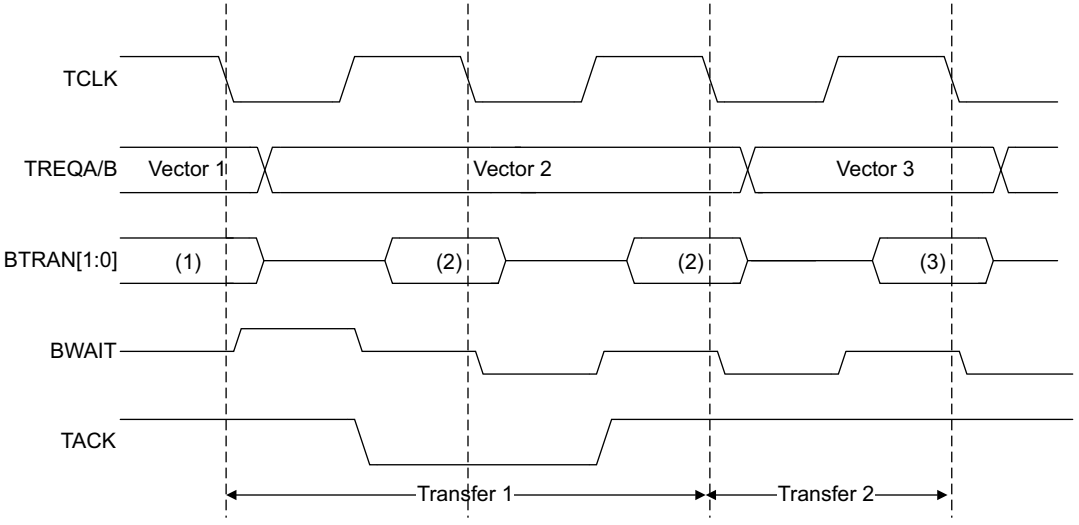


Figure 3-36 Vectors and waited transfers

Although all vector types are applied as above, the timing characteristics for **TBUS** and the number of each vector type applied is vector dependent. The types of vectors are:

- ADDRESS
- WRITE
- READ.

ADDRESS vector

Only one ADDRESS vector is required for an address transfer. The **TBUS** must be driven with the address value required during the ADDRESS vector transfer (that is transfer 2 shown in Figure 3-36 on page 3-68). This value is latched from **BD** by the TIC, driven onto **BA** by the end of the ADDRESS vector transfer, and is held for subsequent READ or WRITE vectors.

During an ADDRESS vector, the TIC drives **BTRAN** as A-TRAN, and no locations are accessed in the microcontroller.

WRITE vector

Only one WRITE vector is required for each write transfer. As with the ADDRESS vector, the value to be written should be driven on **TBUS** during the transfer for the WRITE vector. The TIC drives **BTRAN** as S-TRAN, and asserts **BWRITE**, causing a write transfer to the address location set up by the last ADDRESS vector.

READ vector

Unlike ADDRESS or WRITE vectors, the READ vector **TBUS** activity does not take place during its corresponding AMBA transfer. During the read transfer period, the **TBUS** should be undriven. The value read by the READ vector is driven out on **TBUS** in the cycle following the transfer. The read vectors are shown in Figure 3-37.

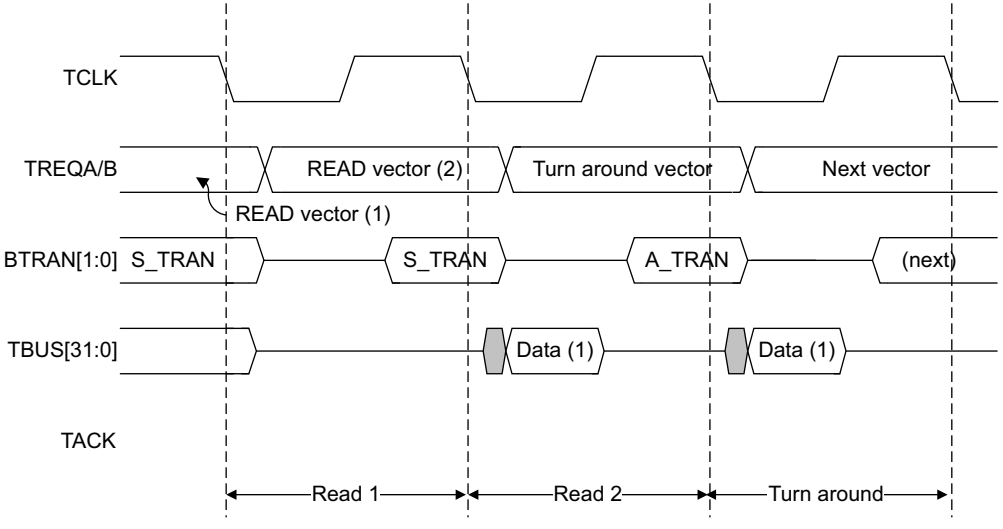


Figure 3-37 Read vectors and turnaround

Here two reads are done from the same address location. Since **TBUS** is driven in the cycle following the read 2 transfer, the READ vector 2 can not be followed by a WRITE or an ADDRESS vector (this would require **TBUS** to be driven by the tester in the cycle following the read 2 transfer). Thus a turnaround vector is needed. Turnaround is indicated by**TREQA=**1, **TREQB=**1, which is identical to an ADDRESS vector. However, no address change occurs since **TBUS** is not driven and **BD** is not latched onto **BA**. The turnaround vector may be followed by ADDRESS vectors (or any other type of vector) in which case the address will change in subsequent cycles.

3.8 AMBA ARM7TDMI interface

The AMBA ARM7TDMImodule interfaces between the ARM7TDMI and the ASB bus, allowing the ARM7TDMI to become an ASB bus master. The module also includes a test interface, allowing the ARM7TDMI to be selected as a bus slave and test via the TIC interface. If, however, an alternative test approach is to be used, the test logic can be removed from the AMBA interface.

The top level block diagram is shown in Figure 3-38 on page 3-72, which shows how the wrapper interfaces to the ARM7TDMI. A number of the ASB input signals are routed through the wrapper before becoming inputs to the ARM7TDMI, and the ARM7TDMI outputs are also routed through the wrapper before being driven onto the ASB.

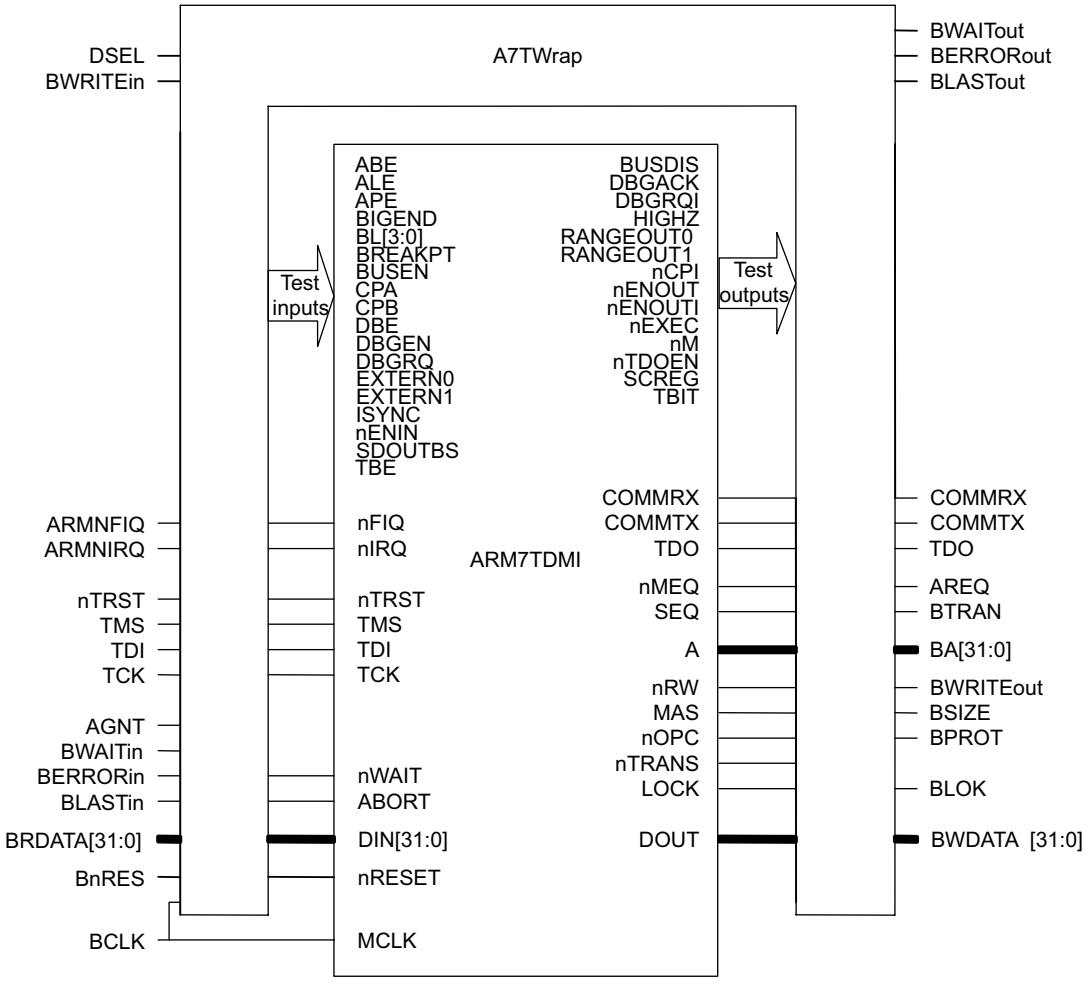


Figure 3-38 Block diagram for ARM7TDMI AMBA master logic

3.8.1 Signal description

Table 3-13 ASB signal descriptions

Name	Type	Description
ARMNFIQ	Input	**ARMNFIQ** is the ARM fast interrupt request, and is routed to the **nFIQ** input on the ARM CPU.
ARMNIRQ	Input	**ARMNIRQ** is the ARM interrupt request, which is routed to the **nIRQ** input on the ARM CPU.
COMMRX	Output	Communications channel receive. 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	Output	Communications channel transmit. When HIGH, this signal denotes that the communications channel transmit buffer is empty.
AGNT	Input	**AGNT** is a signal from the bus arbiter that indicates that the bus master will be granted the bus when **BWAIT** is LOW. This signal changes during the LOW phase of **BCLK** and remains valid through the HIGH phase.
BCLK	Input	System (bus) clock. This clock times all bus transfers. The clock has two distinct phases. Phase 1 in which **BCLK** is LOW, and phase 2 in which **BCLK** is HIGH.
BnRES	Input	This active LOW signal indicates the reset status of the bus and is driven by the reset controller.
DSELARM	Input	**DSELARM** is a signal from the bus decoder to a bus slave, indicating that the slave device is selected and that a data transfer is required. For this module, the signal is used to put the ARM core into a test mode so that vectors can be written in and out of the core. This signal becomes valid during the **BCLK** HIGH phase before the data transfer is required, and remains active until the last **BCLK** HIGH phase of the transfer.
BA	Output	**BA** is the system address bus, which is driven by the current bus master. The addresses become valid during the **BCLK** HIGH phase before the transfer to which they refer, and remain valid until the last **BCLK** HIGH phase of the transfer.
BD[31:0]	Input/Output	**BD[31:0]** is the bidirectional system data bus.

Table 3-13 ASB signal descriptions (continued)

Name	Type	Description
BERROR	Input	A transfer error is indicated by the selected bus slave using the **BERROR** signal. When **BERROR** is HIGH, a transfer error has occurred. When **BERROR** is LOW, the transfer is successful. This signal is also used in combination with the **BLAST** signal to indicate a bus retract operation. When no bus transfer is taking place, this signal is driven by the system decoder. The selected bus slave drives this signal in the LOW phase of **BCLK** and is valid set up to the rising edge of **BCLK**.
BLAST	Input	**BLAST** is driven by the selected bus slave to indicate whether the current transfer should be the last of a burst sequence. When **BLAST** is HIGH the next bus transfer must allow for sufficient time for address decoding. When **BLAST** is LOW, the next transfer may continue a burst sequence. This signal is also used in combination with the **BERROR** signal to indicate a bus retract operation. When no bus transfer is taking place, this signal is driven by the bus decoder. The selected bus slave drives this signal in the LOW phase of **BCLK** and is valid set up to the rising edge of **BCLK**.
BWAIT	Input	**BWAIT** is driven by the selected bus slave to indicate whether the current transfer may complete. If **BWAIT** is HIGH, a further bus cycle is required. If **BWAIT** is LOW the transfer may complete in the current bus cycle. When no bus transfer is taking place this signal is driven by the system decoder. The selected bus slave drives this signal in the LOW phase of **BCLK** and is valid set up to the rising edge of **BCLK**.
BWRITE	Output	When HIGH this signal indicates a write cycle, and when LOW a read cycle. This signal has the same timing as the address bus.
AREQ	Output	**AREQ** indicates to the arbiter that the block requires the bus. In this module, this signal is permanently tied HIGH by default, indicating that the ARM CPU requires the bus at all times. **AREQ** must be set up to the falling edge of **BCLK**.
BTRAN[1:0]	Output	These signals indicate the type of the next transaction, which may be address-only, nonsequential or sequential. These signals are driven by this block when **AGNT** is asserted, and are valid during the **BCLK** HIGH phase before the transfer to which they refer.
BLOK	Output	When HIGH, this signal indicates that the following transfer is to be indivisible, and that no other bus master should be given access to the bus. The signal is driven by this block when granted, and becomes valid during the **BCLK** HIGH phase before the transfer to which it refers. It remains valid until the last **BCLK** HIGH phase of the last locked transfer.

Table 3-13 ASB signal descriptions (continued)

Name	Type	Description
BSIZE[1:0]	Output	These signals indicate the size of the transfer, which may be byte, halfword or word. These signals have the same timing as the address bus.
BPROT[1:0]	Output	These signals provide additional information about a bus access, and are primarily intended for use by a bus protection unit or by the decoder. The signals indicate whether the transfer is an opcode fetch or data access, as well as whether the transfer is a supervisor mode access, or user mode access. These signals have the same timing as the address bus.
nTRST	Input	**nTRST** is part of the IEEE 1149.1 JTAG standard. When LOW, it resets the JTAG interface.
TCK	Input	The JTAG clock (Test clock). This is part of the IEEE 1149.1 JTAG standard.
TMS	Input	Test mode select. This is part of the IEEE 1149.1 JTAG standard.
TDI	Input	Test data in. This is part of the IEEE 1149.1 JTAG standard.
TDO	Output	Test data out. This is part of the IEEE 1149.1 JTAG standard.

3.8.2 Overview of the wrapper blocks

Figure 3-39 on page 3-76 shows the modules that make up the ARM7TDMI AMBA interface.

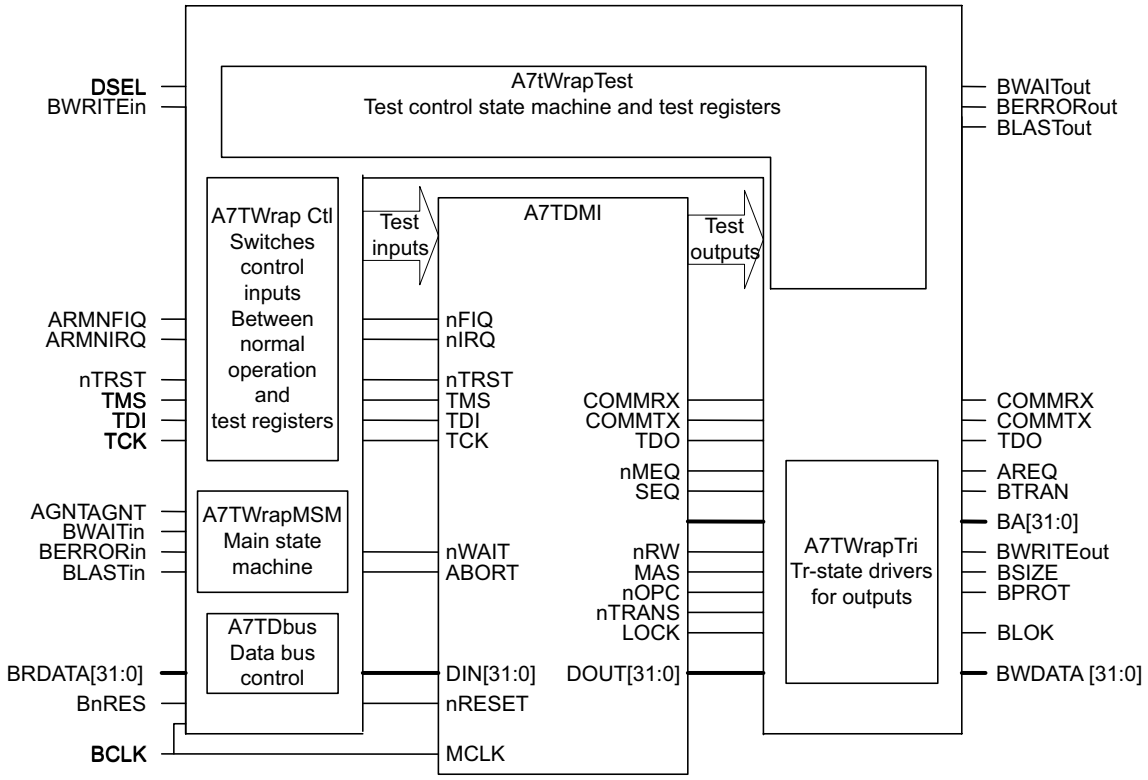


Figure 3-39 Main modules of the ARM7TDMI AMBA interface

The wrapper is made up of the following blocks:

- A7TWrapMSM
- A7TDBus
- A7TWrapTri
- A7TWrapTest
- A7TWrapCtl.

A7TWrapMSM The main state machine block is responsible for determining when the core is granted access to the ASB bus. It uses the transfer response signals (**BWAIT**, **BLAST** and **BERROR**) as well as the grant signal (**AGNT**) to determine when the core is granted and controls the both the core, using the nWAIT signal, and also controls the tristate buffers on the output to ensure the output signals are only driven when the ARM is granted the bus.

A7TDBus The data bus control block is used to route both the input and output data from the ARM core. The ARM core unidirectional **DIN** and **DOUT** buses are used and are combined together to form the bidirectional **BD[31:0]**.

A7TWrapTri The tristate buffer module is used to control the driving of all the ASB output signals. If the bus master is to be used in a design which does not use tristate buses then this module can be modified to generate the appropriate signals.

A7TWrapTest The test module includes the following functions:

- The main test state machine to control the application of the test vectors.
- A 28-bit test register to store the value of the control inputs during test.
- An output mux to select either the control outputs or the address to be driven onto the data bus.

A7TWrapCtl The control inputs block switches a number of control signals between the values required for normal operation and the values applied during test. If the test logic is removed then this module is not required.

Table 3-14 shows how the ARM7TDMI macrocell signals are connected.

Table 3-14 Connections of ARM7TDMI signals

Name	Description	Type	Connected to
A[31:0]	Addresses	Output	**BA[31:0]** via a tristate buffer.
ABE	Address bus enable	Input	Tied HIGH to drive the address and control signals at all times. The tristate control of the address and control signals is provided using a tristate buffer external to the ARM macrocell.
ABORT	Memory abort	Input	Latched version of **BERROR**.
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.

Table 3-14 Connections of ARM7TDMI signals (continued)

Name	Description	Type	Connected to
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 LOW 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	Inout	Unconnected as the unidirectional data buses are used.
DBE	Data bus enable	Input	Tied HIGH to drive the data buses at all times. Tristate control of the data bus is provided using a tristate buffer external to the ARM macrocell.
DBGACK	Debug acknowledge	Output	Only used for test.
DBGEN	Debug enable	Input	Tied HIGH to allow use of JTAG debug.
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 **BD**.
DOUT[31:0]	Data output bus	Output	Used to drive **BD**.
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 output	Output	Unconnected output.
EXTERN[1:0]	External input	Input	Tied LOW as there is no external debug logic.
HIGHZ		Output	Only used for test.

Table 3-14 Connections of ARM7TDMI signals (continued)

Name	Description	Type	Connected to
ICAPCLKBS	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	Used to form **BLOK**.
MAS[1:0]	Memory access size	Output	Used to form **BSIZE**.
MCLK	Memory clock input	Input	Main clock input connected to **BCLK**.
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.
nIRQ	Not interrupt request	Input	Connected to system **ARMNIRQ**.
nM[4:0]	Not processor mode	Output	Only used for test.
nMREQ	Not memory request	Output	Used to form **BTRAN[1]**.
nOPC	Not opcode fetch	Output	Used to form **BPROT**.
nRESET	Not reset	Input	From system **BnRES**.
nRW	Not read/write	Output	Used to form **BWRITE**.
nTDOEN	Not**TDO** enable	Output	Only used for test.
nTRANS	Not memory translate	Output	Used to from **BPROT**.
nTRST	Not test reset	Input	From system **nTRST**.
nWAIT	Not wait	Input	Generated from the main state machine in combination with the system **BWAIT** signal.
PCLKBS	Boundary scan update clock	Output	Unconnected output.

Table 3-14 Connections of ARM7TDMI signals (continued)

Name	Description	Type	Connected to
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 form **BTRAN[1:0]**.
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.
TBIT	Thumb state	Output	Only used for test.
TCK	Test clock	Input	From system **TCK**.
TCK1		Output	Unconnected output.
TCK2		Output	Unconnected output.
TDI		Input	From system **TDI**.
TDO	Test data output	Output	To system **TDO**.
TMS		Input	From system **TMS**.

3.8.3 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. To implement additional debug logic external to the core, these signals may be used.
- 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.

3.8.4 Description of the wrapper blocks

This section contains a detailed description of each of the following blocks:

- *ARM7TWrap*
- *ARM7TWrapMSM*
- *A7TDBus* on page 3-82
- *A7TWrapTri* on page 3-82
- *A7TWrapTest* on page 3-83.

ARM7TWrap

This top-level module is purely structural, and connects together all the other modules with the wrapper.

The default signal configurations described above are tied off at this level and if they are required externally to the wrapper, they can be routed to the ports of this block.

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.

ARM7TWrapMSM

The main state machine is used to determine when the core is clocked and when the various output signals can be driven onto the ASB bus. The state machine has eight states, which are described below:

IDLE When in the IDLE state, the ARM does not require the ASB, so can be clocked freely. The ARM is not granted, so drive onto the ASB is disabled.

BUSIDLE When in the BUSIDLE state, the ARM is granted ownership of the ASB but does not require it. The ARM is clocked freely and **nMREQ/SEQ** are used to generate the **BTRAN** signals.

HOLD In the HOLD state, the ARM wishes to use the ASB but is not granted its ownership. The ARM is waited until it regains bus ownership. **BTRAN** is forced to ATRAN to provide the ATRAN-STRAN combination required during master ownership changeover.

HANDOVER In the HANDOVER state, the ARM has just gained ownership of the bus. The ARM is still waited as no data will be returned this cycle. **BTRAN** is forced to STRAN to form the second half of the ATRAN-STRAN cycle sequence.

ACTIVE In the ACTIVE state, the ARM is actively performing bus transactions. The ARM's wait input is derived from the ASB **BWAIT** signal. The output data is driven if the transfer is a write cycle.

RETRACT In the RETRACT state, the core is not clocked as the transfer cannot complete. The state machine will insert an ATRAN cycle as the first part of the ATRAN-STRAN sequence required to retry the transfer.

CPRTIDLE In the CPRTIDLE state, the ARM is performing a CPRT while not granted ASB ownership. The core is clocked, but none of the output signals are driven onto the ASB.

CPRTBUSIDLE In the CPRTBUSIDLE state, the ARM is performing a CPRT while granted the bus.

A7TDBus

The data bus control block is used to route both the input and output data from the ARM core.

The ARM core unidirectional DIN and DOUT buses are used. On the input side this block provides a transparent latch that is used to reduce the input hold time required by the core. On the output side, this block selects the output data either from the ARM core or, in test mode, from the test module.

A7TWrapTri

The tristate buffer module is used to control the driving of all the ASB output signals. If the bus master is to be used in a design that does not use tristate buses, this module can be modified to generate the appropriate signals.

The data bus is driven either during a write transfer (as indicated by the main state machine), or when the core is being tested (and either the address or the control signal outputs are being read).

The address and control signals are driven from this module, as controlled by the main state machine.

The tristate control for the BTRAN signals is simply derived from the grant signal, AGNT.

This block also contains the tristate drivers for the transfer response signals (**BWAIT**, **BLAST** and **BERROR**), which are only used when the core is being tested. When selected for test, these signals have a fixed zero, wait-state, no-error response. If the test logic is removed, these tristate drivers may also be removed.

A7TWrapTest

The test module includes the following functions:

- The main test state machine to control the application of the test vectors.
- A 28-bit test register to store the value of the control inputs during test.
- An output mux to select either the control outputs or the address to be driven onto the data bus.

The state diagram for the main test state machine is shown in Figure 3-40 on page 3-84.

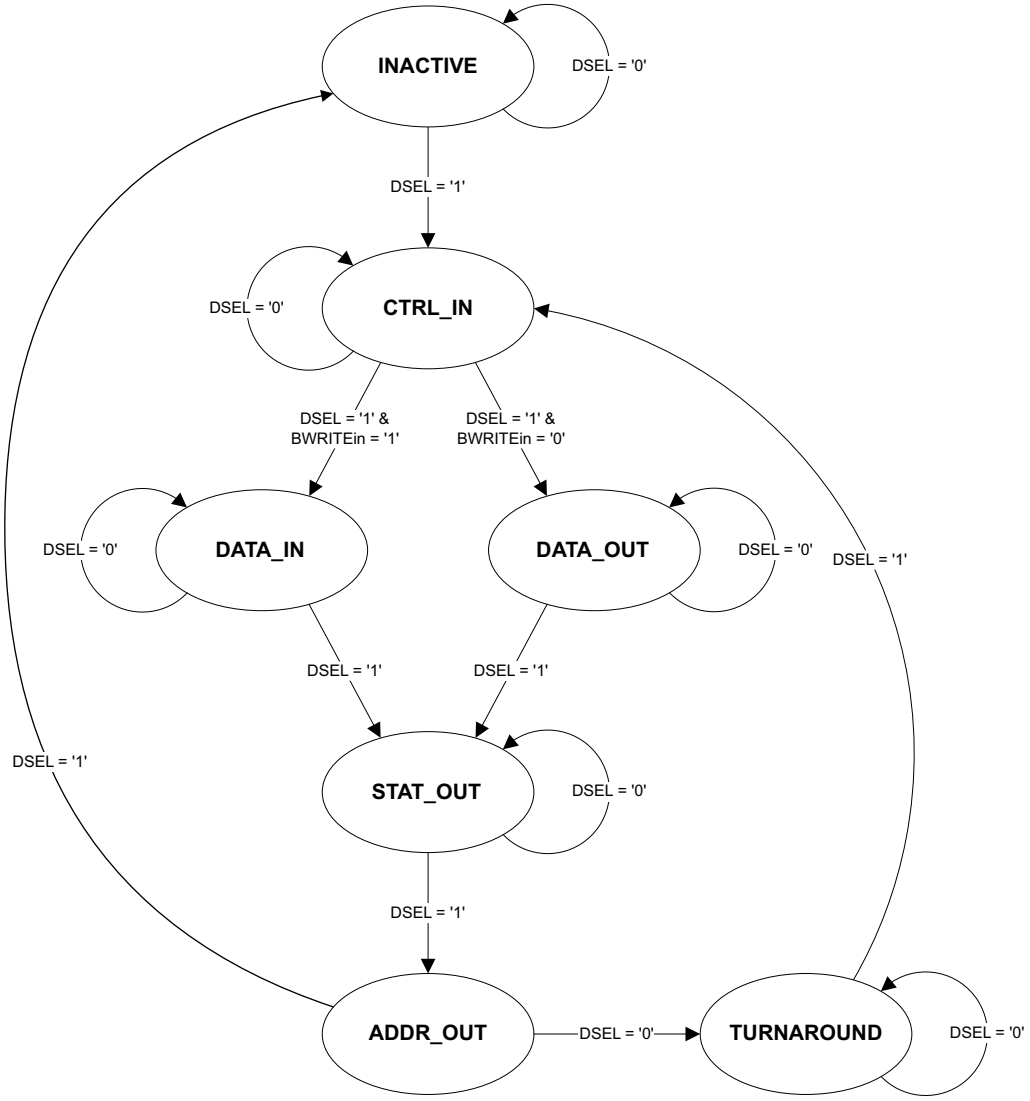


Figure 3-40 Main test state machine state diagram

The **DSEL** signal is used to control when test vectors are applied, and therefore the transitions through the test state machine.

CTRL_IN In the **CTRL_IN** state, the write vector on the data bus is used to load a test register, which determines the values of the control signal that will be applied to the core when it is clocked,

DATA_IN The **DATA_IN** state is entered when the core is performing a read transfer. The data for the read is supplied from **BD**. The core is clocked in this state.

DATA_OUT The **DATA_OUT** state is used for write transfers, and the output data from the core is driven onto **BD**. The core is also clocked in this state.

During the **STAT_OUT** and the **ADDR_OUT** states, the output status signals and then the address bus are driven onto **BD**, so that they can be read externally. The address and status signals from the core are checked after each transfer.

The core will then pass via the TURNAROUND state back to the **CTRL_IN** state.

To exit from test, an extra read vector is required after the **ADDR_OUT** state, and this will force the test state machine back to the INACTIVE state. This in turn will cause all the core input and output signals to revert to their normal mode of operation.

The test module also includes a 28-bit test register that is loaded during the **CTRL_IN** state, and determines the value on the control inputs when the core is clocked. This occurs during either **DATA_IN** or **DATA_OUT**. Table 3-15 shows the control input bit positions.

Table 3-15 ARM7TDMI control input bit positions

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 **MclkEnable**, 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 **MclkEnable** and **BCLK**.
nTRST	Not test reset.	17	
EXTERN1	External input 1.	16	
EXTERN0	External input 0.	15	
DBGRQ	Debug request	14	

Table 3-15 ARM7TDMI control input bit positions (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	ANDed with **MclkEnable**.
nFIQ	Not fast interrupt request.	4	
nIRQ	Not interrupt request.	3	
ABORT	Memory abort	2	AMBA **BERROR** signal must be latched, as it is only valid at the end of phase 1.
nWAIT	Not wait	1	ANDed with **MclkEnable**, so that the core state can only change during the data access cycle.
nRESET	Not reset	0	

The final part of the test control logic is the output mux. This determines which signals are driven onto the **BD** bus:

- During **DATA_OUT** the core DOUT bus is driven onto **BD**.
- During **ADDR_OUT** the address bus is driven onto **BD**.
- During **STAT_OUT** the status outputs are driven onto **BD**.

Table 3-16 shows the bit positions of the status output signals when driven onto the data bus.

Table 3-16 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	
DBGRQI	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 **MclkEnable** is used to clock a transparent latch that will capture the correct state.
nENOUTI	Not enable output	16	**nENOUT** is only valid during the data access cycle, so **MclkEnable** is used to clock a transparent latch 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	

Table 3-16 ARM7TDMI status bit positions (continued)

Signal	Description	Bit position	Comment
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	

A7TWrapCtl

The A7TWrapCtl control block is part of the AMBA test harness and switches a number of control signals between the values required for normal operation and the values applied during test. If the test logic is removed then this module is not required.

3.8.5 Removal of the test interface

The test interface logic may be removed by removing the A7TWrapTest module from the top level of the wrapper design. It is then necessary to tie the outputs which were originally generated from this block to fixed values and this is described in the HDL code.

The removal of the test interface means that the **TestStatOut** signal should be static. This signal is used in the A7TDbus block to switch the output data between the core, and the test logic. Therefore it should be confirmed that this mux has either been removed automatically during synthesis, or has been removed from the RTL code.

It is also possible to remove the three tristate drivers for **BWAIT**, **BLAST** and **BERROR** in the A7TWrapTri block.

Chapter 4
APB Modules

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

4.1 Interrupt controller

The interrupt controller 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 available.

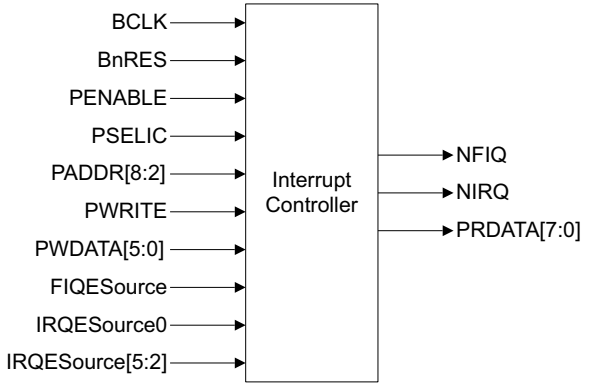


Figure 4-1 Interrupt controller module block diagram

4.1.1 Hardware interface and signal description

The interrupt controller module is connected to the APB bus. Table 4-1 shows the signal descriptions for the interrupt controller.

Table 4-1 APB signal descriptions for interrupt controller

Name	Type	Description
BCLK	In	This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers.
BnRES	In	The bus reset signal is active LOW and is used to reset the system.
PENABLE	In	This enable signal is used to time all accesses on the peripheral bus.
PSELIC	In	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 **BA**.

Table 4-1 APB signal descriptions for interrupt controller (continued)

Name	Type	Description
PADDR[8:2]	In	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	In	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]	In	The write peripheral data bus is driven by the bridge at all times.
PRDATA[7:0]	Out	The read peripheral data bus is driven by this block during read cycles (when **PWRITE** is LOW and **PSELIC** is HIGH).
FIQESource	In	**FIQ** interrupt signal into the interrupt module. This active HIGH signal indicates that a fast interrupt request has been generated.
IRQESource0 **IRQESource[5:2]**	In	**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	Out	Active LOW fast interrupt request input to the ARM core.
NIRQ	Out	Active LOW interrupt request input to the ARM core.

4.1.2 Interrupt controller

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 would be 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 six interrupt request lines available. This can be extended to up to 32 sources by altering the IRQSize constant 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 a FIQ interrupt to an IRQ interrupt.

Interrupt control

The interrupt controller provides interrupt status, raw interrupt status and 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 4-2 on page 4-5 shows the structure for a single segment of the interrupt controller.

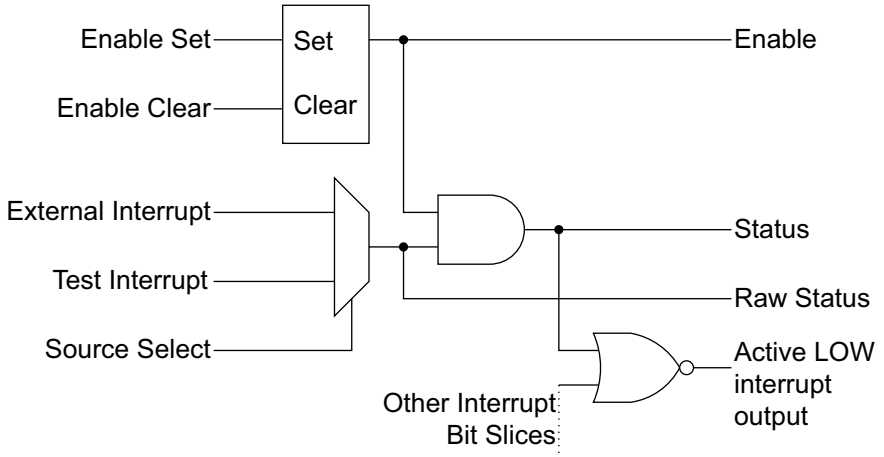


Figure 4-2 Single bit slice of the interrupt controller

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.

4.1.3 Interrupt controller 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 4-2 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

Table 4-2 Memory map of the interrupt controller APB peripheral (continued)

Address	Read Location	Write Location
IntBase + 0x10C		FIQEnableClear
IntBase + 0x014	IRQTestSource	IRQTestSource
IntBase + 0x018	IRQSourceSel	IRQSourceSel
IntBase + 0x114	FIQTestSource	FIQTestSource
IntBase + 0x118	FIQSourceSel	FIQSourceSel

4.1.4 Interrupt controller 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.

The following register is also provided:

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**.

4.1.5 Interrupt registers standard configuration

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 4-3 Example of IRQ sources

Bit	Interrupt Source
0	
1	Programmed Interrupt
2	Comms Rx

Table 4-3 Example of IRQ sources (continued)

Bit	Interrupt Source
3	Comms Tx
4	Timer 1
5	Timer 2

4.1.6 System description

This part gives a detailed description of how the HDL code for the system 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, and signals used in the system. This section should be read together with the HDL code.

Figure 4-3 shows the interrupt controller module block diagram.

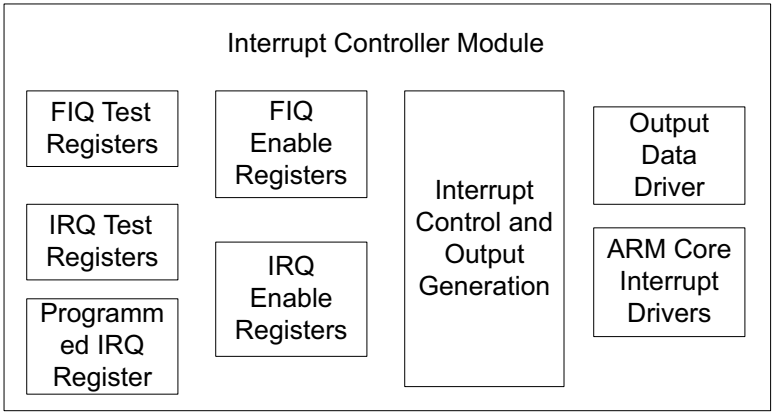


Figure 4-3 Interrupt controller module block diagram

The main parts of the interrupt controller are the interrupt registers and test registers that are used to control the use of the interrupt input signals to generate the interrupt outputs to the ARM core.

All registers used in the system are clocked from the rising edge of the system clock **BCLK**. Enable signals are used to control the loading of the registers, and multiplexor signals are used to remove the combinational logic from the register statement. All registers use the asynchronous reset **BnRES**.

Two diagrams are used to show the interrupt controller HDL file. Figure 4-2 on page 4-5 shows the layout of the bit slices that are used for bit 0 of the FIQ and bits 0 and 2:5 of the IRQ, while Figure 4-3 on page 4-8 shows the layout of the whole system:

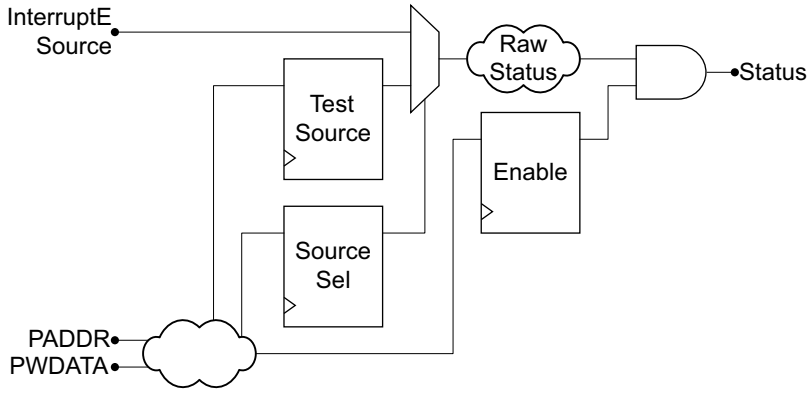


Figure 4-4 Interrupt controller slice system diagram

Figure 4-4 shows a diagram of a single interrupt controller slice. Figure 4-5 on page 4-10 shows the use of each interrupt slice within the whole system. Interrupt bit 1 (IRQSoft) is implemented differently to the other interrupt slices (bits 0, and 2 to 5).

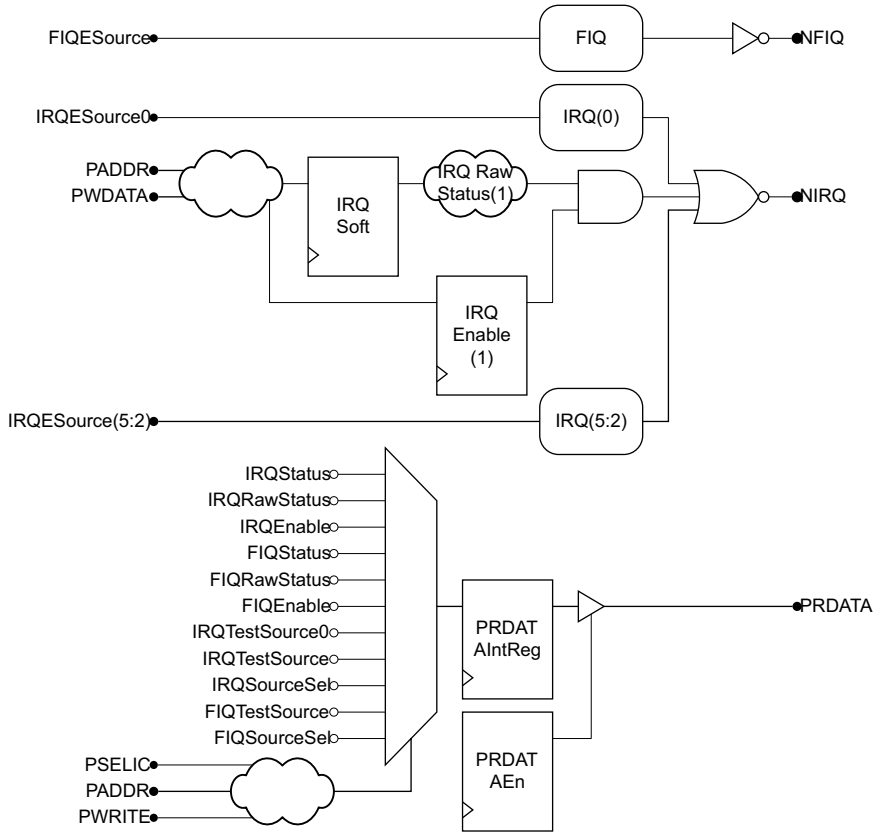


Figure 4-5 Interrupt controller module system diagram

The main sections and processes in the code are:

- *Constant and signal definitions* on page 4-11.
- *General signals* on page 4-11.
- *IRQTestSource register* on page 4-11.
- *IRQSourceSel register* on page 4-12.
- *IRQSource multiplexer* on page 4-12.
- *IRQSoft register* on page 4-12.
- *IRQRawStatus signal* on page 4-12.
- *IRQEnable register* on page 4-12.
- *IRQStatus signal* on page 4-13.
- *FIQTestSource register* on page 4-13.
- *IRQSourceSel register* on page 4-12.

- *FIQSource multiplexer* on page 4-13.
- *FIQRawStatus signal* on page 4-13.
- *FIQEnable register* on page 4-13.
- *FIQStatus signal* on page 4-13.
- *Output data generation* on page 4-13.
- *Interrupt outputs to ARM core* on page 4-14.
- *Tri-state driver for data bus* on page 4-14.

The above sections and processes are now explained in more detail in the following paragraphs.

Constant and signal 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 six 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.

The signals that are used inside the module are then defined.

General signals

The signal **Addr** is used to simplify the address checking logic that is synthesized, by setting the unused address bits LOW, so that they are ignored during address checking. This signal is then used instead of **PADDR** for all address checking.

SetClear is used to control the setting or clearing of the enable registers. This variable is just a direct copy of bit 2 of the address bus **PADDR** (LOW when address is **EnableSet**, HIGH when address is **EnableClear** for both IRQ and FIQ addresses).

Enable is used in the generation of the enable signals for all of the registers used in the system. It is a combination of the APB signals **PENABLE**, **PSELIC** and **PWRITE**.

IRQTestSource register

The **IRQTestSource** register is used to test the interrupt controller. It consists of registers for all bits of the IRQ system that are used (currently bits 0 and 2:5), and are set when a value is written to the **IRQTestSource** address. They are all reset to LOW.

IRQSourceSel register

The **IRQSourceSel** register is used to control the **IRQRawStatus** selection. It stores the value that was written to the **IRQSourceSel** address. It is reset to LOW.

IRQSource multiplexer

The **IRQSource** multiplexer is used to generate the intermediate signal **IRQSource** (which is used to generate **IRQRawStatus**). It uses the current value of **IRQSourceSel** to select between the external interrupt input signals **IRQESource**, and the current value of the test signals **IRQTestSource**.

IRQSoft register

IRQSoft is the software programmable interrupt register. Writes to bit 1 of the **IRQSoft** address are used to set and clear this interrupt source, and the default reset state is LOW.

IRQRawStatus signal

The **IRQRawStatus** signal is used to combine the **IRQSource** and **IRQSoft** signals, and is the value obtained when the **IRQRawStatus** address is read.

IRQEnable register

This section is used to generate the **Enable** signals that are used to control how the interrupt signals are used in the interrupt controller.

The **IRQEnableEn** signal is used to enable the generation of **IRQEnableNext** and **IRQEnable**, and is set HIGH when a value is written to the **IRQEnableSet** or **IRQEnableClear** locations.

The **IRQEnableNext** term is used to control the setting and clearing of the **IRQEnable** register, using the **SetClear** signal to determine how to use the value that has been written. To set bits in the **IRQEnable** register, the new next value becomes a logical OR of the written value and the current value of **IRQEnable**, ensuring that only the required bits are changed. To clear bits in the register, the new next value becomes a logical AND of the inverted written value and the current value of **IRQEnable**, therefore clearing the enables for the required bits. The next value is only generated when the **IRQEnableEn** signal is HIGH, as this is the only time that the signal is required to be loaded into the **IRQEnable** register. The next value becomes valid after the rising edge of **BCLK**, due to the timing of the **Enable** signal.

The **IRQEnable** register is cleared on reset, and set to **IRQEnableNext** on the rising edge of **BCLK** and when **IRQEnableEn** is set HIGH.

IRQStatus signal

IRQStatus is a combination of the enable lines and the interrupt sources, and is used in the generation of the **NIRQ** output to the ARM core. It is also the value that is read from the **IRQStatus** address.

FIQTestSource register

This section is the same as for the IRQ, but is only one bit wide in the default system.

FIQSourceSel register

This section is the same as for the IRQ.

FIQSource multiplexer

This section is the same as for the IRQ, but is only one bit wide in the default system.

FIQRawStatus signal

The **FIQRawStatus** signal is a direct copy of the **FIQSource** multiplexer output in the default single bit system, and is the value obtained when the **FIQRawStatus** address is read.

FIQEnable register

This section is the same as for the IRQ, but is only one bit wide in the default system.

FIQStatus signal

This section is the same as for the IRQ, but is only one bit wide in the default system.

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 value of **Addr** is compared with all of the register addresses, and the value of **PRDATAInt** is set accordingly. This is then stored in the **PADDRIntReg** register to help decrease the output propagation time by using a registered output, rather than an output with the combinational delay of the large multiplexer. This register also synchronizes the reading of all raw interrupt inputs to the rising edge of the clock.

Interrupt outputs to ARM core

This section drives the **NIRQ** and **NFIQ** outputs according to the values of the **Status** signals. As the outputs are active LOW, then they are set LOW when any of the interrupt lines are HIGH. For the **NIRQ** line this is done by comparing the current value of **IRQStatus** with zero, and setting **NIRQ** HIGH if they are equal. As there is only one FIQ interrupt line, then this is inverted to generate **NFIQ**.

Tri-state driver for data bus

PRDATAEnNext is used as the input to the **PRDATAEn** register, and is set during the second clock cycle of a read when **PENABLE** is HIGH. The **enable** register is used to reduce the read output setup time, by removing the combinational logic delay of the **PRDATAEnNext** generation and replacing it with the D to Q setup time of a rising edge register.

The APB data bus is driven with the current internal version when the enable signal is HIGH, and tristated at all other times.

4.2 Remap and pause controller

The reset and pause module provides:

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

A block diagram of the remap and pause module is shown in Figure 4-6.

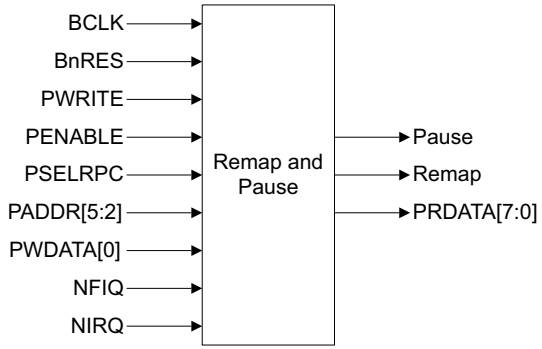


Figure 4-6 Remap and pause module block diagram

4.2.1 Hardware interface and signal description

The remap and pause module is connected to the APB bus. Table 4-4 describes the APB signals used and produced.

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

Name	Type	Description
BCLK	Input	This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers.
BnRES	Input	The bus reset signal is active LOW and is used to reset the system.
PENABLE	Input	This enable signal is used to time all accesses on the peripheral bus.
PSELRPC	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 **BA**.
PADDR[5:2]	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.

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

Name	Type	Description
PWRITE	Input	This signal indicates a write when HIGH and a read when LOW. It has the same timing as the peripheral address bus.
PWDATA[0]	Input	The write peripheral data bus is driven by the bridge at all times.
PRDATA[7:0]	Output	The read peripheral data bus is driven by this block during read cycles (when **PWRITE** is LOW and **PSELRPC** is HIGH).
NFIQ	Input	NFIQ interrupt input from the interrupt controller.
NIRQ	Input	NIRQ interrupt input from the interrupt controller.
Pause	Output	The **Pause** signal is HIGH when in the wait for interrupt **Pause** mode, and LOW at all other times.
Remap	Output	The **Remap** signal is LOW while the reset memory map is in use, and is HIGH when the normal memory map is in use.

4.2.2 Remap and pause

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.

4.2.3 Remap and pause 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 4-5 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

4.2.4 Remap and pause register descriptions

Pause

Pause register 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

ID register Read-only. The **ID** register 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.

ID bit 0 flag Identification bit.

0 - no further ID information

1 - further ID information is available

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

Reset memory map

ClearResetMap register

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 0 at reset, but to change the memory map after reset, such that RAM is located at location 0 for normal operation. In a system where such remapping does not occur, writing to this register has no effect.

Reset status

ResetStatus register Read-only. The **ResetStatus** register 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 **Status** 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.

POR bit 0 flag Power on reset bit.

0 - no POR since flag was last cleared

1- POR

ResetStatusClear register

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

ResetStatusSet register

Write-only. This location is used to set reset status flags. When writing to this register each data bit which is HIGH causes the corresponding bit in the **ResetStatus** register to be set. Data bits which are LOW have no effect on the corresponding bit in the **ResetStatus** register. This register has no function in the minimal reference microcontroller specification, because the power on reset status bit cannot be set by software. This register is included in the specification to ensure the reset status functionality can be expanded.

4.2.5 System description

This part gives a detailed description of how the HDL code for the 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 and signals used in the system. This section should be read together with the HDL code.

A simple block diagram of the whole system is shown in Figure 4-6 on page 4-15.

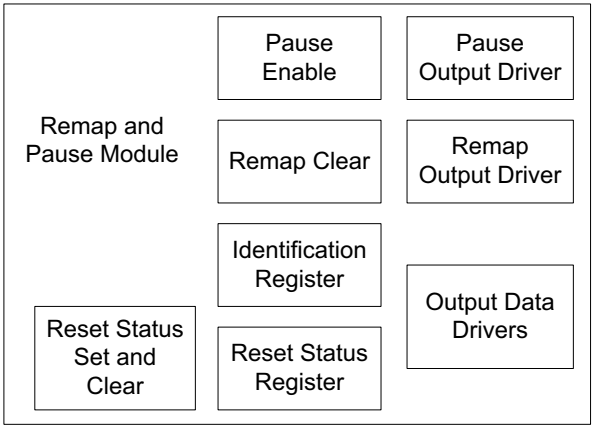


Figure 4-7 Remap and pause module block diagram

The main parts of the remap and pause module are the **Remap** and **Pause** registers that are used to control the operation of the system, and the **ResetStatus** register,

All registers used in the system are clocked from the rising edge of the system clock **BCLK**. Enable signals are used to control the loading of the registers, and a multiplexor signal is used to remove the combinational logic from the **Pause** register statement. All registers use the asynchronous reset **BnRES**.

A diagram of the remap and pause HDL file is shown in Figure 4-8.

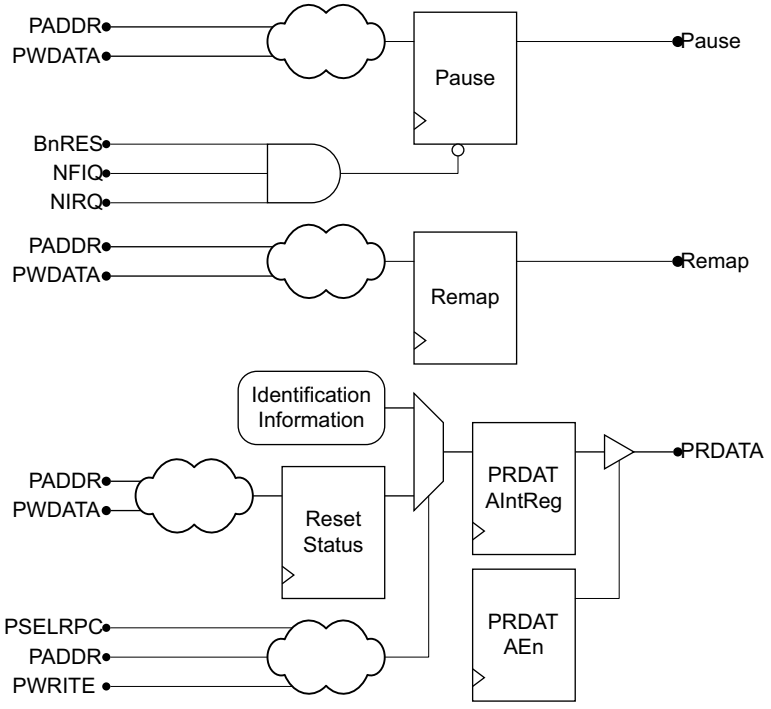


Figure 4-8 Remap and pause module system diagram

The main sections and processes in the code are:

- *Constant and signal definitions* on page 4-21.
- *General signals* on page 4-21.
- *ResetStatus register* on page 4-21.
- *Remap output register* on page 4-21.
- *Pause output register* on page 4-22.
- *Output data generation* on page 4-22.
- *Tri-state driver for data bus* on page 4-23.

The above sections are now explained in more detail in the following paragraphs.

Constant and signal definitions

The constant **REGSIZE** is used to set the maximum size of the registers used in the system. The only two registers that can be read and are multi-bit are the **Identification** and **ResetStatus** registers, so **REGSIZE** is set the same as the larger of these two registers. The default system only uses a single bit for **Identification** and **ResetStatus**, so **REGSIZE** is set to 1. The extra unused bits in these two registers (if they are of different sizes) must be set to LOW.

The constant **IDENTIFICATION** holds the identification information about the system.

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

The signals that are used inside the module are then defined.

General signals

The signal **Addr** is used to simplify the address checking logic that is synthesized, by setting the unused address bits LOW, so that they are ignored during address checking. This signal is then used instead of **PADDR** for all address checking.

ResetStatus register

This section contains the **ResetStatus** register, that is used to show power on reset status only in the default system. The single bit gets set HIGH on reset, and set LOW if the **ResetStatusClear** address is written to with bit 0 set HIGH (bits that are LOW will have no effect on the value of **ResetStatus**). The default system does not allow the register to be set HIGH again through the **ResetStatusSet** address.

If **Identification** is larger than **ResetStatus**, then the **ResetStatus** register should only be used to set the bits needed, and an extra statement should be used to set the unused bits to LOW.

Remap output register

The **Remap Output** register is used to hold the value of **Remap**, which is used to determine the memory map that is used by the system. It is set LOW on reset, and is set HIGH when the **ClearResetMap** address is written to with any value. Once set HIGH, it can only be set LOW by a system reset.

Pause output register

The **Pause Output** register is used to hold the value of **Pause**, which is used to make the system enter a wait for interrupt state. It is synchronously set HIGH (on the rising edge of **BCLK**) when the **Pause** address is written to, and is asynchronously set LOW by **BnRES**, **NFIQ** or **NIRQ**. Once set HIGH, it can only be set LOW with a reset or an interrupt.

Figure 4-9 shows the operation of setting and clearing the **Pause** registered output:

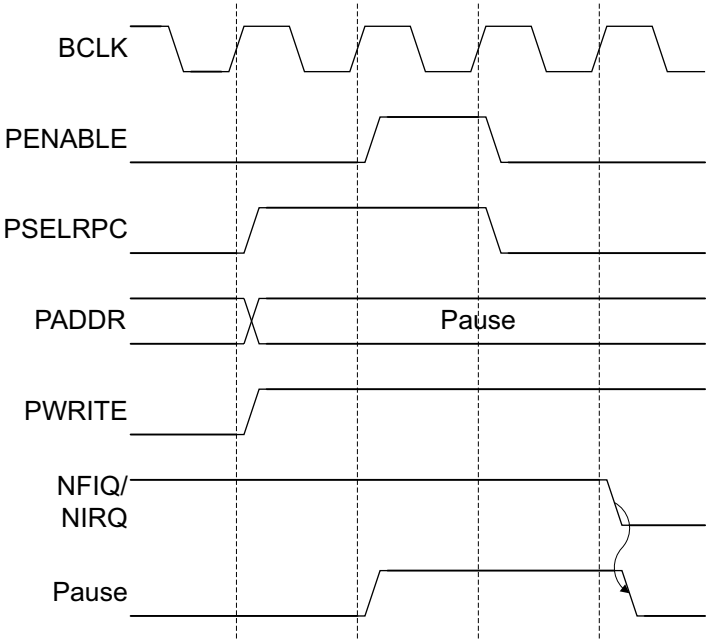


Figure 4-9 Pause signal timing

An asynchronous clear is used on the register. This means there is a totally asynchronous path from the interrupt source to this register, which allows the system clock to be stopped during a low power mode when pause mode is entered.

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 value of **Addr** is compared with all of the register addresses, and the value of **PRDATAInt** is set accordingly. This is then stored in the **PADDRIntReg** register to help decrease the output propagation time by using a

registered output, rather than an output with the combinational delay of the large multiplexer. This register also synchronizes the reading of all raw interrupt inputs to the rising edge of the clock.

Tri-state driver for data bus

PRDATAEnNext is used as the input to the **PRDATAEn** register, and is set during the second clock cycle of a read when **PENABLE** is HIGH. The **enable** register is used to reduce the read output setup time, by removing the combinational logic delay of the **PRDATAEnNext** generation and replacing it with the D to Q setup time of a rising edge register.

The APB data bus is driven with the current internal version when the **enable** signal is HIGH, and tristated at all other times.

4.3 Timer

The timer module consists of:

- two instantiations of the free-running counters (FRCs)
- system clock prescale and test clock generation.

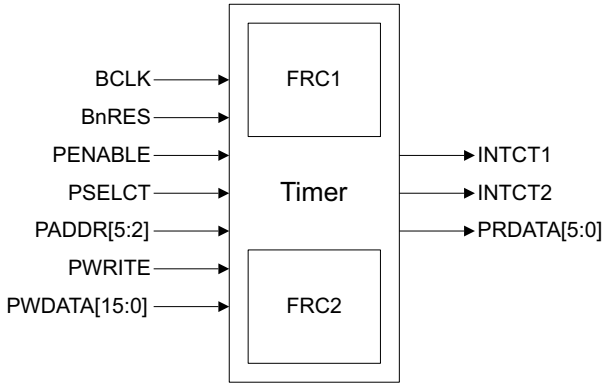


Figure 4-10 Timer module block diagram

4.3.1 Hardware interface and signal description

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

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

The signal descriptions for the timer are listed in Table 4-6.

Table 4-6 APB signal descriptions for timer

Name	Type	Description
BCLK	In	This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers.
BnRES	In	The bus reset signal is active LOW and is used to reset the system.
PENABLE	In	This strobe signal is used to time all accesses on the peripheral bus.
PSELCT	In	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 **BA**.
PADDR[5:2]	In	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.

Table 4-6 APB signal descriptions for timer (continued)

Name	Type	Description
PWRITE	In	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]	In	This signal indicates a write when HIGH and a read when LOW. It has the same timing as the peripheral address bus. It becomes valid before **PENABLE** goes HIGH and remains valid after **PENABLE** goes LOW.
PRDATA[15:0]	Out	The read peripheral data bus is driven by this block during read cycles (when **PWRITE** is LOW and **PSELCT** is HIGH).
INTCT1	Out	Active HIGH interrupt signal to the interrupt controller module. This signal indicates an interrupt has been generated by timer 1, having been decremented to zero.
INTCT2	Out	Active HIGH interrupt signal to the interrupt controller module. This signal indicates an interrupt has been generated by timer 2, having been decremented to zero.

Table 4-7 Signal descriptions for FRC

Name	Type	Description
BCLK	In	Direct connection from timers module
BnRES	In	Direct connection from timers module
PENABLE	In	Direct connection from timers module
PSELCT	In	Direct connection from timers module
PADDR[4:2]	In	Direct connection from timers module
PWRITE	In	Direct connection from timers module
PWDATA[15:0]	In	Direct connection from timers module
Frcsel	In	FRC register select, driven high when a register in this FRC is addressed (one of two different outputs from timer)
Enable0	In	Counter clock enable, divide by 1
Enable1	In	Counter clock enable, divide by 16

Table 4-7 Signal descriptions for FRC (continued)

Name	Type	Description
Enable2	Out	Counter clock enable, divide by 256
Intfrc	Out	Interrupt output from the counter, generated when 16 bit counter reaches zero (one of two different inputs to timer)
Dataout	Out	Read data output used to generate PRDATA for register reads (one of two different inputs to Timer)

4.3.2 Timer introduction

Two timers are defined as the minimum provided within a system, although this may be expanded easily. The same principle of simple expansion has been applied to the register configuration, allowing more complex timers to be used. These are, from the programmer's model, compatible with those already defined.

Two modes of operation are available:

- Free-running mode. The timer 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.

4.3.3 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 from the **Load** register and continues to decrement. In this mode the timer effectively generates a periodic interrupt. The mode is selected by a bit in the **Control** register.

At any point, the current timer value may be read from the **Value** register.

The timer is enabled by a bit in the **Control** register. At reset, the timer is disabled, the interrupt is cleared, and the **Load** register is set to zero. The mode and prescale value are set to free running, and clock divide of 1 respectively.

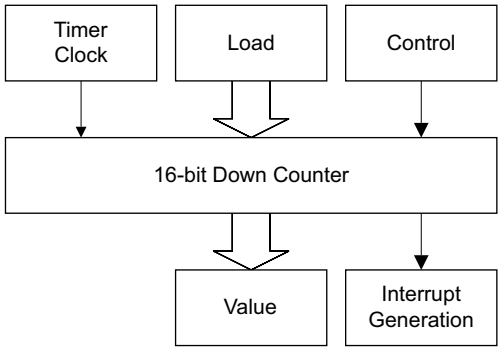


Figure 4-11 Timer operation

e timer clock is generated by a prescale unit. The timer clock may be one of the following:

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

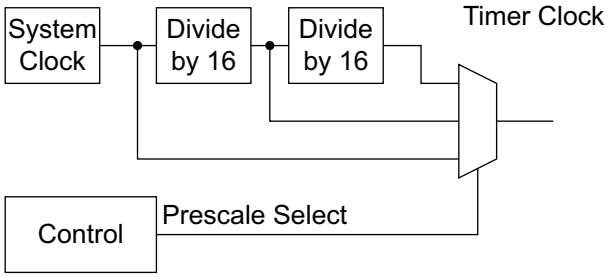


Figure 4-12 Prescale unit

4.3.4 Timer memory map

The base address of the timers 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 4-8 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

4.3.5 Timer register descriptions

Load Read-write register. (**Timer1Load**, **Timer2Load**). The **Load** register contains the initial value to be loaded into the timer and is also used as the reload value in periodic timer mode. This register is the same width as the counter (default is 16 bits).

Value Read-only register. (**Timer1Value**, **Timer2Value**). The **Value** location gives the current value of the timer.

Clear Write-only register. (**Timer1Clear**, **Timer2Clear**). Writing to the **Clear** location clears an interrupt generated by the counter timer.

Control Read-write register. (**Timer1Control**, **Timer2Control**). The **Control** register provides enable/disable, mode and prescale configurations for the timer.

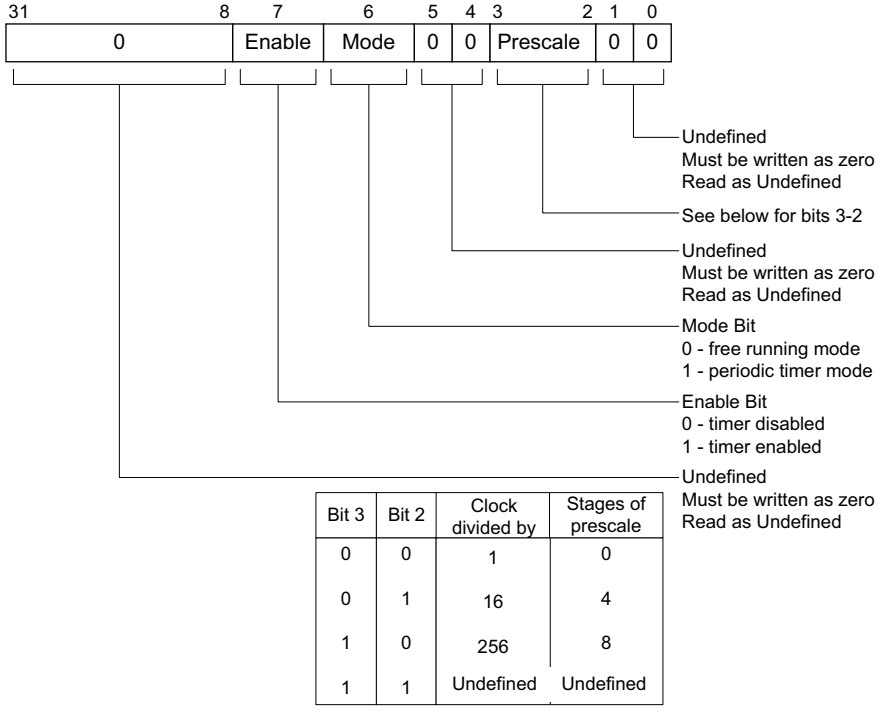


Figure 4-13 The control register

4.3.6 Test register

wo special registers are provided for validation purposes: **Timer1Test** and **Timer2Test**. These locations should not be accessed during normal operation.

Both registers are read-write and are 2 bits wide:

Table 4-9 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, but the test clock select bit is stored in a single register in the timer, but can be accessed from either test address.

When the counter test mode bit is set, the 16-bit counter of the selected timer is divided into four separate 4-bit counters that continually loop round from 15 to 0. This ensures 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, normal or test). Clearing this bit (default) selects the system clock as the prescale clock input (normal operation).

4.3.7 System description

This part gives a detailed description of how the HDL code for the timers block 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, and signals used in the system. This section should be read together with the HDL code.

A simple block diagram of the whole system is shown in Figure 4-15 on page 4-32.

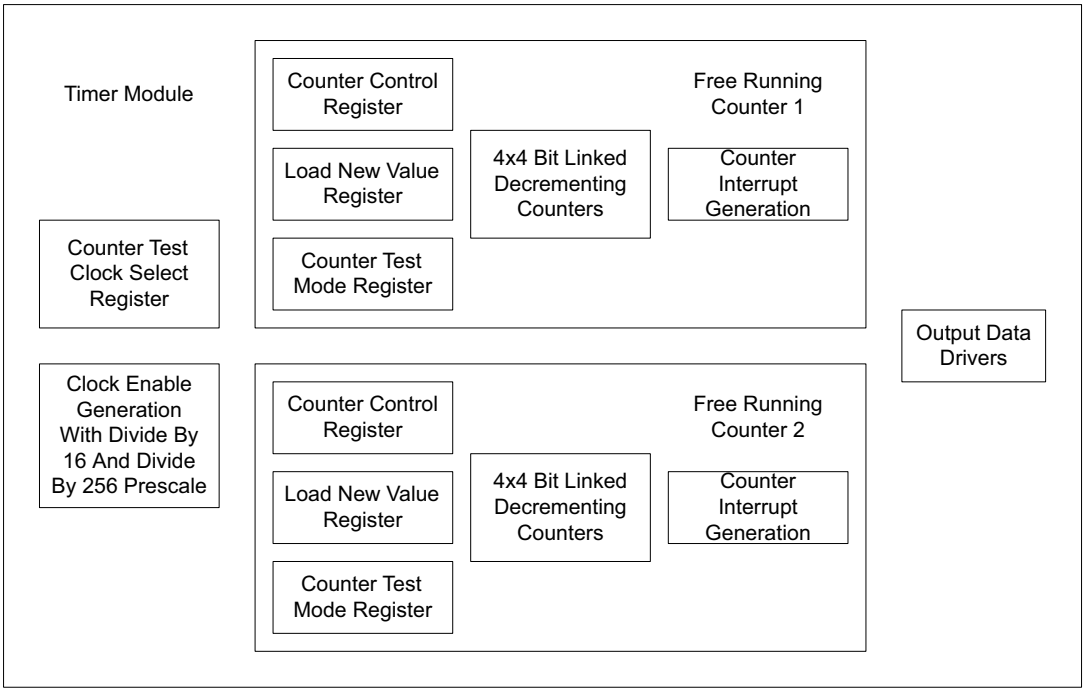


Figure 4-14 Timer module block diagram

Most of the system is contained in the two instances of the FRCs (the registers, counters and interrupt generation), but the three different speed clock enables that are used by the counters are generated in the timer module and passed to the FRCs.

All registers used in the system are clocked from the rising edge of the system clock **BCLK**. Enable signals are used to control the loading of the registers, and multiplexor signals are used to remove the combinational logic from the register statement. Most registers use the asynchronous reset **BnRES**, although is some cases a synchronous reset is used.

Timer system description

A diagram of the timer's HDL file is shown in Figure 4-15 on page 4-32

.

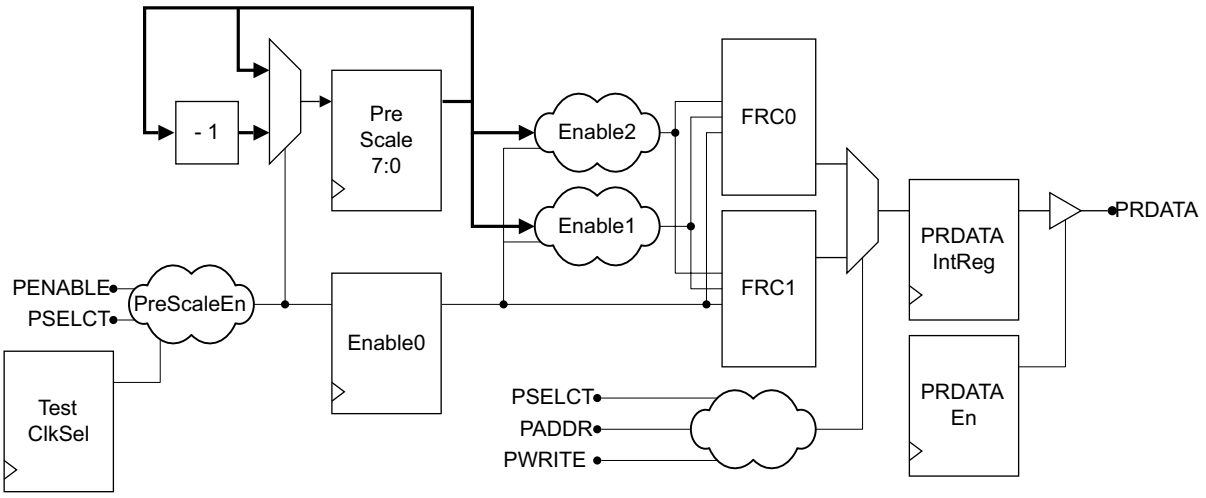


Figure 4-15 Timer module system diagram

The main sections and processes in the code are:

- *Address decoder*.
- *TestClkSel register*.
- *Clock prescaler* on page 4-33.
- *Output enable generation* on page 4-33.
- *Output data generation* on page 4-34.
- *Tri-state driver for data bus* on page 4-34.
- *Free running counter blocks* on page 4-34.

The above sections are now explained in more detail in the following paragraphs.

Address decoder

This section is used to generate select lines (**FrcSel1** and **2**) to the FRCs based on the current address on **PADDR** (as there are two instantiations of an identical FRC module, then part of the address decoding has to be done at the previous system level), and the **TestSel** signal which is used to indicate an access to either of the **Test** registers.

TestClkSel register

The **TestClkSel** register is used to store the current value of bit 1 of both test registers (the test clock select bit). A read or write to either test register address 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 any of the timers registers.

Output enable generation

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

Enable0 is a registered version of the **PreScaleEn** signal, which allows alignment of **Enable0** with **PENABLE** while the test clock is in use.

Enable1 and **2** are generated from the outputs of the **PreScale** registers and the current state of **Enable0**, using **PreScale(3)** to generate the divide by 16 **Enable1** signal, and **PreScale(7)** for the divide by 256 **Enable2** signal.

Figure 4-16 and Figure 4-17 on page 4-34 show the timing of these enable signals:

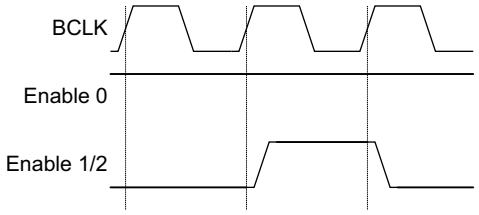


Figure 4-16 Timer module counter enable timing - system clock selected

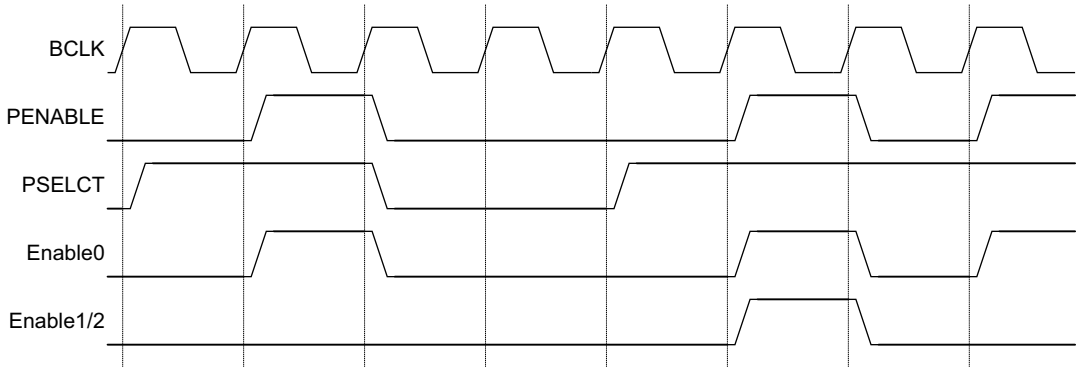


Figure 4-17 Timer module counter enable timing - test clock selected

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 data bus. The value of **PADDR** is compared with all of the register addresses, and the value of **PRDATAInt** is set accordingly. This is then stored in the **PADDRIntReg** register to help decrease the output propagation time by using a registered output rather than an output with the combinational delay of the large multiplexer. This register also synchronizes the reading of all raw interrupt inputs to the rising edge of the clock.

Tri-state driver for data bus

PRDATAEnNext is used as the input to the **PRDATAEn** register, and is set during the second clock cycle of a read when **PENABLE** is HIGH. The **enable** register is used to reduce the read output setup time, by removing the combinational logic delay of the **PRDATAEnNext** generation and replacing it with the D to Q setup time of a rising edge register.

The APB data bus is driven with the current internal version when the enable signal is HIGH, and tristated at all other times.

Free running counter blocks

Two instances of the identical counter modules are included in the timers module.

4.3.8 FRC system description

A diagram of the FRC HDL file is shown in Figure 4-18 on page 4-35.

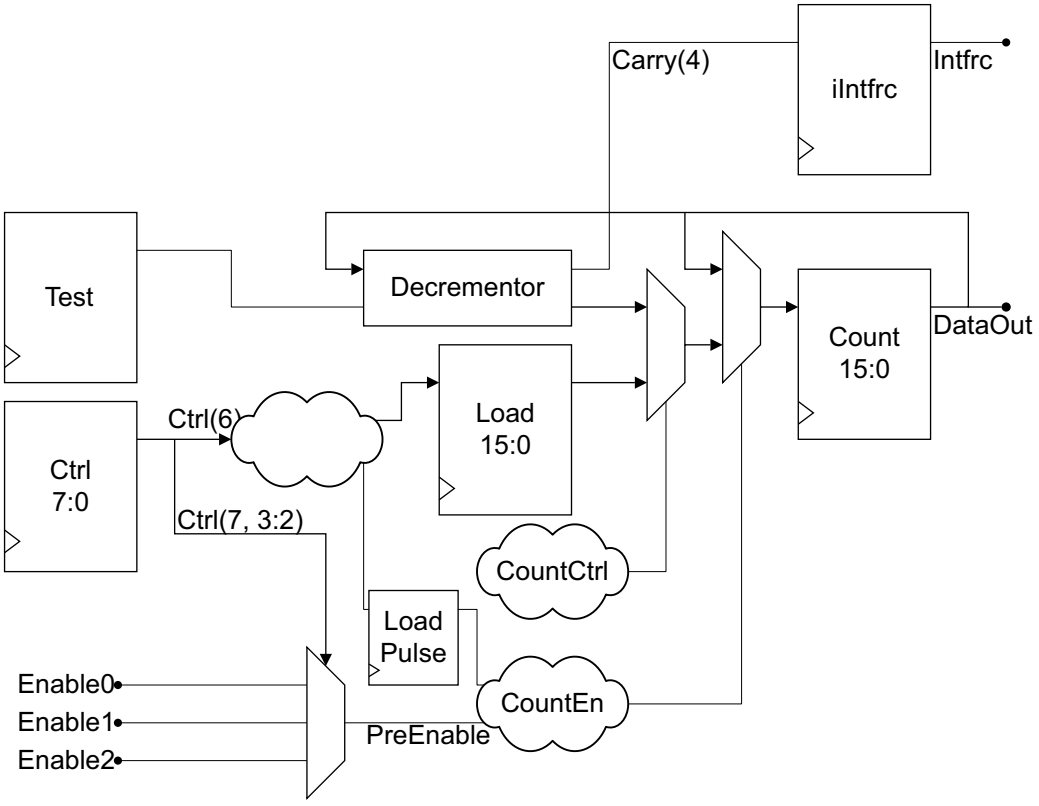


Figure 4-18 FRC module system diagram

The main sections and processes in the code are:

- *Register select decoder* on page 4-36.
- *Control register* on page 4-36.
- *Test register* on page 4-36.
- *Load register* on page 4-36.
- *Counter load pulse generation* on page 4-36.
- *Counter enable* on page 4-36.
- *16-bit count down* on page 4-36.
- *Counter register* on page 4-37.
- *Interrupt generation* on page 4-37.
- *Output data generation* on page 4-37.

The above sections are now explained in more detail in the following paragraphs.

Register select decoder

This section generates the **Testsel** and **Regsel** signals from the current address and state of the input port **Frcsel**, which are then used in the generation of new values for the **Load**, **Control** and **Test** registers, and the **ReadSel** signal.

Control register

The **Control** register is split into two parts, as only bits 7:6 and 3:2 are used.

Test register

This is bit 0 (counter test mode) of the **Test** register, refer to *Test register* on page 4-29, which splits the counter up into four 4-bit counters when enabled.

Load register

The **Load** register contains the current load value that has been written to the timers module, which is passed straight through to the **Count** registers when written to, and is also loaded into the **Count** registers when periodic mode is enabled.

Counter load pulse generation

When a new value is written to the **Load** register, this must be passed to the **Count** register. An extra pulse is added to **CountEn**, when a load operation is performed, to clock the load data in.

Counter enable

The **Enable0/1/2** input is selected according to the values of the prescale bits in the **Control** register (bits 2 and 3), and this is then combined with the **Enable** signal (**Ctrl(7)**) and the additional **Load** register pulse to generate the final enable line to the counter registers, **CountEn**.

16-bit count down

The counter is split up into four 4-bit parts (nibbles) to allow efficient testing. Eight separate processes are used in this section to decrement the counter, and to generate the carry signals from one nibble to the next. The operation of this is shown in Figure 4-19 on page 4-38.

The lowest nibble of the counter must always count down, so it has no carry in signal. The other three nibbles only count down when all of the previous nibbles have reached zero, so they each have a separate carry in signal that is generated by the previous nibble. When test mode is selected (**Test** HIGH), all four nibbles decrement together.

Carry(4) from the most significant nibble is used to generate the interrupt output, and to make the counter read from the **Load** register when periodic mode is set.

Counter register

The **Counter** register uses two control signals. **CountEn** is used to enable the register around the rising edge of **BCLK**, and is generated from the input enable lines. **CountCtrl** is used as a select on the input multiplexor value, to set it to **NextCount** for normal operation, or to **Load** when the counter has reached zero and is in periodic mode or if the **Load** register has just been written to.

Interrupt generation

This generates the final interrupt output for the timer, which is set when the counter reaches zero (equivalent to **Carry(4)** being set), and the interrupt is then held until it is cleared by a write to the **TimerClear** address. The **iIntfrcNext** signal is used to hold this value, and is also used to clear the interrupt.

The internal interrupt **iIntfrc** signal is driven onto the output port **Intfrc**, which is then converted to **INTCT1/2** at the timer module outputs.

Output data generation

This uses the value of **ReadSel** to drive **DataOut** with a combination of the **Count**, **Load**, **Control** and **Test** registers, which is then passed on to **PRDATA** in the timer module when any of the FRCs registers are read from.

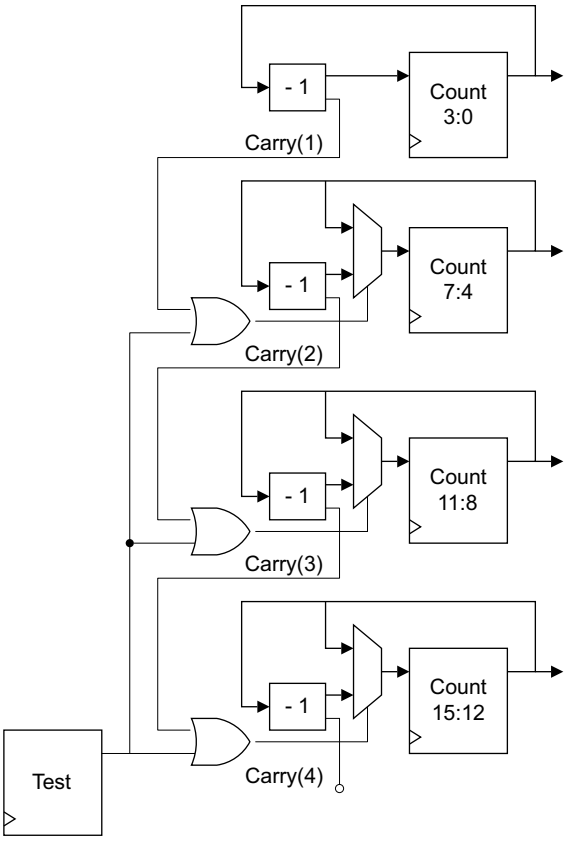


Figure 4-19 FRC module decrement diagram

Chapter 5
Test Interface Driver

This chapter describes the use of the external AMBA Test Interface Driver module (the *TICBOX*). It includes a description of the *TICTalk*command language. For more information about AMBA and the Test Interface protocol please refer to the *AMBA Specification*. The chapter contains the following:

5.1 Introduction

You should be familiar with AMBA and its test interface protocol. If not, please refer to the *AMBA Specification* for further information.

ICBOX is a module external to an AMBA system that drives the test interface lines to gain access to the ASB bus and then applies test vectors from a test input file (see Figure 5-1). This test input file is the output from running a C program written with the *TICTalk* command language.

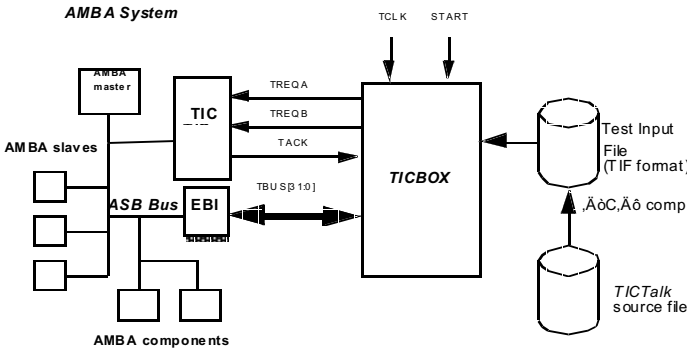


Figure 5-1 TICBOX connection to an AMBA system

In order to gain access to the ASB bus, the TICBOX will signal a test request (driving **TREQA** HIGH and **TREQB** LOW). Once the request is granted (**TACK** driven HIGH), the test input file is read and translated by the TICBOX into AMBA test interface transactions.

The TICBOX applies test vectors to the system every time the **TACK** line indicates the system is ready. On read cycles the value is masked and then compared with an expected value (also masked), and an error message is asserted if the comparison fails.

5.2 TICBOX usage

The TICBOX model communicates with the rest of the system through the following connections:

Table 5-1 TICBOX connections

Signal Name	Type	Function
TCLK	Input	This is the system clock **BCLK** in test mode. All the test interface transactions are timed using this signal.
START	Input	When this signal is active (HIGH), it indicates that the TICBOX should request mastership of the AMBA system by driving **TREQA** and **TREQB** both HIGH after the next falling edge of **TCLK**.
TREQA	Output	Test Bus Request A. Indicates test vector mode. Please 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 **TCLK** and held to the falling edge of **TCLK**.
TREQB	Output	Test Bus Request B. Indicates test vector mode. Please refer to the Test Interface chapter on the *AMBA Specification* for further information about the test protocol. It is driven early in the LOW phase of **TCLK** and held to the falling edge of **TCLK**.
TACK	Input	Test Acknowledge. Indicates that the test bus has been granted and also that a test access has been completed.
TBUS[31:0]	Input/Output	32-bit bidirectional test port.

The TICBOX also requires two variables to be defined through generics:

Table 5-2 TICBOX variables

Variable	VHDL Type	Description
FileName	string	This is the name of the test input file (in TIF format) to be read by the TICBOX. For VHDL the default name is `infile.tif`, and for Verilog is `infile.sim`.
HaltOnMismatch	boolean	If set (TRUE for VHDL, variable defined for Verilog), the simulation will stop if, on a read vector sequence, the expected value and the read value don't match. If cleared (FALSE for VHDL, variable undefined for Verilog), the TICBOX will issue a warning message reporting that a mismatch has occurred, but simulation will continue running until the end.

The use of the TICBOX is very simple: once the test interface signals **TCLK**, **TREQA**, **TREQB**, **TACK** and **TBUS[31:0]** are connected to the AMBA system and the Test Input File (in TIF format) is created (see *TICTalk command language* on page 5-6), the simulation environment (test bench) should assert the **START** signal to indicate that the TICBOX should request mastership of the bus.

At this point the TICBOX will generate TIC vectors as specified on the TIF until the end of the file is reached. The TICBOX will then signal end of test by driving **TREQA** and **TREQB** both LOW. The test bench should at this point drive the **START** signal LOW to halt the simulation.

Example 5-1 A typical simulation output while running a TIC program

```
#    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 the above example you will note that an error read has occurred, but the error message is broadcast later in the simulation. This is because there is an elapsed time between when the read is requested, and when the information arrives to the TICBOX to be compared with the expected value. The simulation has been run with the HaltOnMismatch variable set to FALSE, and therefore the program does not stop after the error has been detected.

5.3 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:

- write address vector
- write test vector
- burst write of test vectors
- read test vector
- burst read of test vectors
- change from writes to reads and vice-versa.

The *TICTalk* language performs these actions by combining together a number of basic commands. These commands are described in the following section.

5.3.1 TICTalk commands

The basic *TICTalk* commands are:

- *Write address vector (A)*
- *Write test vector (W)* on page 5-7
- *Read test vector (R)* on page 5-7
- *Burst read vector (B)* on page 5-7
- *Repeat last command (L)* on page 5-7
- *Include the string message into the TIF (C)* on page 5-7
- *Exit test mode (E)* on page 5-7.

The above commands are described in the following paragraphs.

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), a write test vector (change from writes to reads) and to indicate the last read on a burst, but not after another read test vector. To signal a burst sequence of reads, the burst read vector command should be used instead.

Burst read 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 end of test.

5.3.2 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 Reads to Writes:

This change can only be made after a R command (R-W), and not after a B command.

Change from Writes to Reads:

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).

5.3.3 The TICTalk file

Example 5-2 Example 5-2 C program using the *TICTalk* commands

```
#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 MaskControl0x000000CC
#define MaskValue 0x0000FFFF
```



```
#define DUMMY0x12345678

#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()
}
```

It can be seen that 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. Here is where the Counter_Base constant should be defined. This ensures portability of the test program to other systems with different peripheral address mapping.

5.3.4 Generating a test input file format

To generate a TIF file, the *TICTalk* program should be C compiled (using gcc for example) in the following manner:

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 on the TICBOX, for example:

object_file > infile.tif

This output file should then be copied or linked to the directory where the TICBOX model exists.

5.3.5 TIF format

The TIF is very similar to the *TICTalk* file, 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 5-3 Example 5-3 TIF format

```
; 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
```


Chapter 6
Designer's Guide

This chapter takes a basic look at 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:

6.1 Adding bus masters

For bus masters, only two blocks require changes:

- the arbiter
- the decoder.

The arbiter currently has facilities for up to two more masters without any modification. A new master needs to be connected to the appropriate **AREQx** and **AGNTx** signals. This can be done by altering the top level HDL file, which connects all ASB modules together.

Note

If a system requires more than four masters, the HDL file arbiter will also have to be modified.

Modifications to the decoder are described in *Choosing a decoder implementation* on page 6-5.

6.1.1 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 **AREQx** permanently HIGH.

Currently the ARM bus master always asserts **AREQx**, 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.

6.1.2 Bus master requirements

New designs of bus master must drive all the relevant signals at appropriate times. For more information consult the *AMBA Specification*and the *AMBA ARM7TDMI Interface Data Sheet*.

6.2 Adding ASB slaves

When a slave is added, only the decoder needs to be modified. This will add a **DSELx** signal for the new slave, which must also be added to the ASB top level HDL file.

6.2.1 Decoder modifications

This block is quite easily modified. When adding new **DSELx** lines, 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 decoders gate count low)
- ensure that all areas of address space have one and only one slave selected
- comment memory map changes in the HDL.

The decoder may be extended to allow for more complex memory map handling. This is not considered further in this document.

6.2.2 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 ASB to APB bridge interface have more complex requirements. For more information consult the *AMBA Specification*.

6.3 Adding APB peripherals

When adding a peripheral, the HDL file `APBif` needs to be modified. The changes will add a **PSELx** signal, which will also have to be added to the APB top level HDL file where the new peripheral should be instantiated. If the peripheral requires connections to ASB components or pads, these signals will also have to be added to the ASB top level HDL file.

6.3.1 APB bridge modifications

When adding new **PSELx** lines similar steps should be taken to those outlined in *Decoder modifications* on page 6-3 although reset memory map will not be an issue for peripherals.

6.3.2 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 **BCLK** in peripherals unless absolutely necessary (its use will dramatically increase power consumption).
- Ensure that peripherals cannot drive **PRDATA[31:0]** during reset (by including a **BnRES** 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.

6.4 Choosing a decoder implementation

The EASY microcontroller is provided with two different decoder implementations. The choice between one of the two implementations will be system-dependent, as both models share the same port names.

6.4.1 Decoder with decode cycles

This is the default model. This implementation automatically inserts a decode cycle under the following circumstances:

- at the start of a non-sequential transfer
- on a sequential transfer when BLAST has been asserted
- when 1K memory boundaries are reached.

This decoder is used on fast systems where the decoder might not have enough time to decode the address and assert the corresponding DSEL signal in a single clock HIGH phase.

6.4.2 Decoder without decode cycles

This implementation attempts to decode the address bus on every cycle and therefore will only be suitable for slow systems where this can be safely achieved.

