

Introduction

Understanding the embedded stripe interface to the PLD is key to implementing a system efficiently using the ARM®-based embedded-processor PLD. The embedded stripe interface comprises the embedded stripe bridge interface and the dual-port RAM interface. The embedded stripe bridges provide a fast interface to the embedded stripe, although the interface to the dual-port RAM is arguably faster. This application note focuses on the embedded stripe bridges.



For details on the dual-port RAM interface, see *Application Note 173: Excalibur Solutions—DPRAM Reference Design*.

The embedded stripe bridges facilitate on-chip peripheral communication; their feature set complements the embedded stripe bus architecture. The read/write options of the embedded stripe bridges accelerate throughput, thereby increasing system performance. The embedded stripe bridges also manage clock domains, allowing logic in the PLD that interfaces to the embedded stripe to be fully asynchronous to the embedded stripe bus clock.



For details of the architecture of the ARM-based devices, see the *ARM-based Embedded Processor PLD Hardware Reference Manual*. See the *AMBA Specification* for more about the AHB.



Refer to “[Revision History](#)” on page 20 to see a summary of changes made for this version of the document.



EXCALIBUR™

Embedded Stripe Bus Architecture Overview

The embedded stripe of an ARM-based device uses two AHB buses as its communication medium: AHB1 and AHB2. AHB1 and AHB2 are pipelined 32-bit implementations with separate read and write data buses. The embedded stripe supports all AHB transactions, including the following:

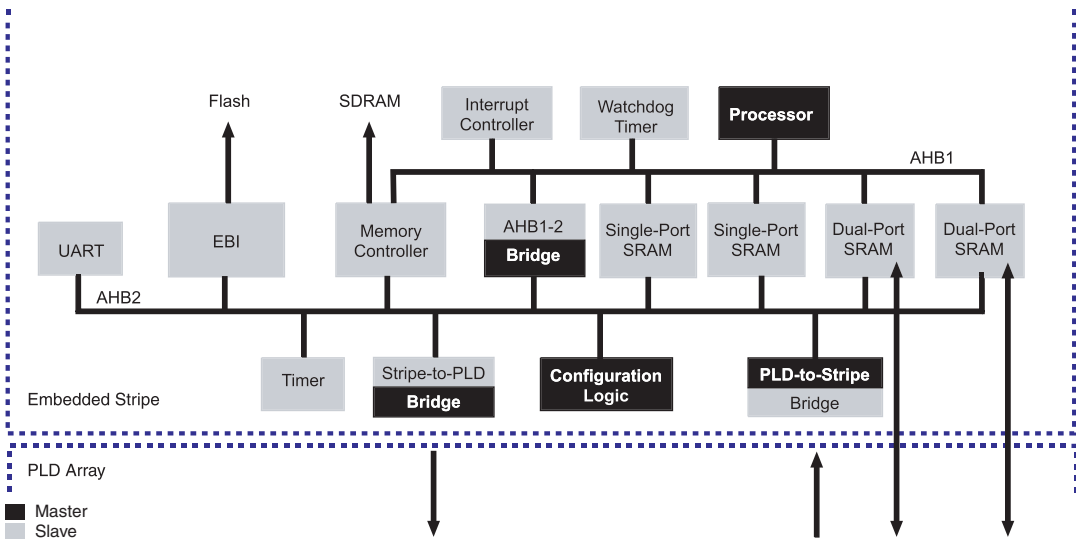
- Burst transactions
 - INCR4
 - INCR8
 - INCR16
 - Unspecified length INCR



- WRAP4
- WRAP8
- WRAP16
- Split transactions on the EBI
- Locked transactions

Figure 1 shows a simplified block diagram of the embedded stripe bus architecture.

Figure 1. Embedded Stripe Bus Architecture

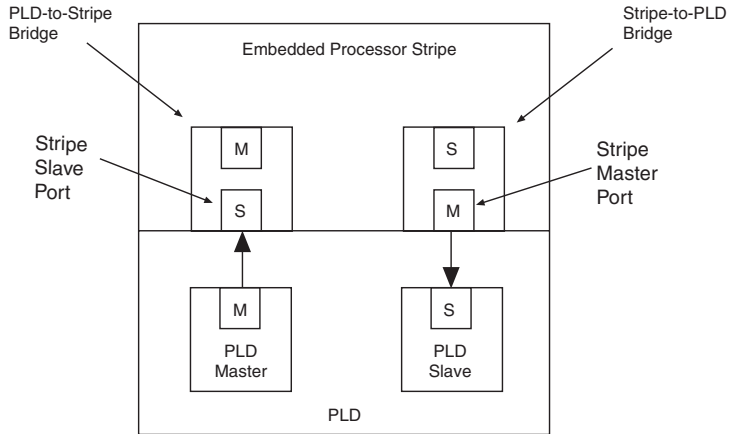


The embedded processor is the sole bus master on AHB1 and has fast access to the slaves on AHB1. The AHB1-2 bridge gives the embedded processor access to AHB2 slaves and PLD slaves via the stripe-to-PLD bridge. The PLD-to-stripe bridge gives masters in the PLD access to slaves on AHB2. The clock in AHB2 runs at half the speed of the AHB1 clock.

Embedded Stripe Bridge Architecture

The embedded stripe bridges function as both masters and slaves on their respective buses. Each bridge has both an AHB master interface and an AHB slave interface, referred to as the master port and slave port respectively. Each bridge receives transactions through its slave port and drives transactions to their destination through its master port.

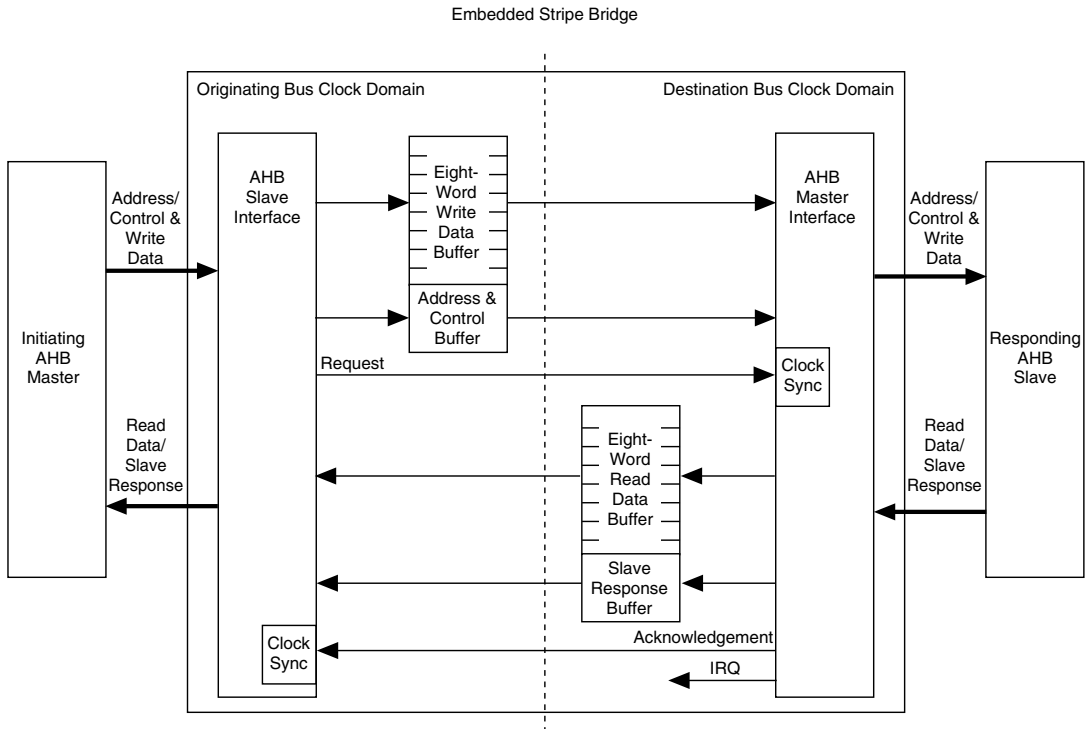
Figure 2 on page 3 shows the interface between the PLD and the embedded stripe.

Figure 2. Interface Between the PLD and Embedded Stripe

M Master Port - initiates transactions
 S Slave Port - responds to transactions

Figure 3 on page 4 shows the internal architecture of the embedded stripe bridges.

Figure 3. Embedded Stripe Bridge



The embedded stripe bridges function in the same way, and they all have the following functional blocks:

- AHB slave interface
- AHB master interface
- Write buffer
- Read buffer

AHB Slave Interface

The AHB slave interface on the embedded stripe bridges accepts transactions from initiating masters, and drives the transaction information to write buffer. The AHB slave interface is also responsible for synchronizing response information from targeted slaves and driving it to the initiating master.

AHB Master Interface

The AHB master interface on the embedded stripe bridges synchronizes transactions from the write buffer and regenerates the address and control information for the transaction. The AHB master interface is also responsible for passing slave responses to the read buffer.

Write Buffer

The write buffer is used to store write transactions data, address information, and control information. One location holds the address and control information for the transaction, and eight words are used to buffer the write data.

Read Buffer

The read buffer stores read data and responses from the slaves that interface to the embedded stripe bridges. One location holds the slave response information, and eight words are used to buffer the read data.

The embedded stripe bridges provide an effective means of communication between the device peripherals. To realize the full potential of the embedded stripe bridges, it is important to be familiar with their feature set and how the embedded stripe bridges process transactions. This section explains how the embedded stripe bridges function.

Bridge Operation

Bridge Write Transactions

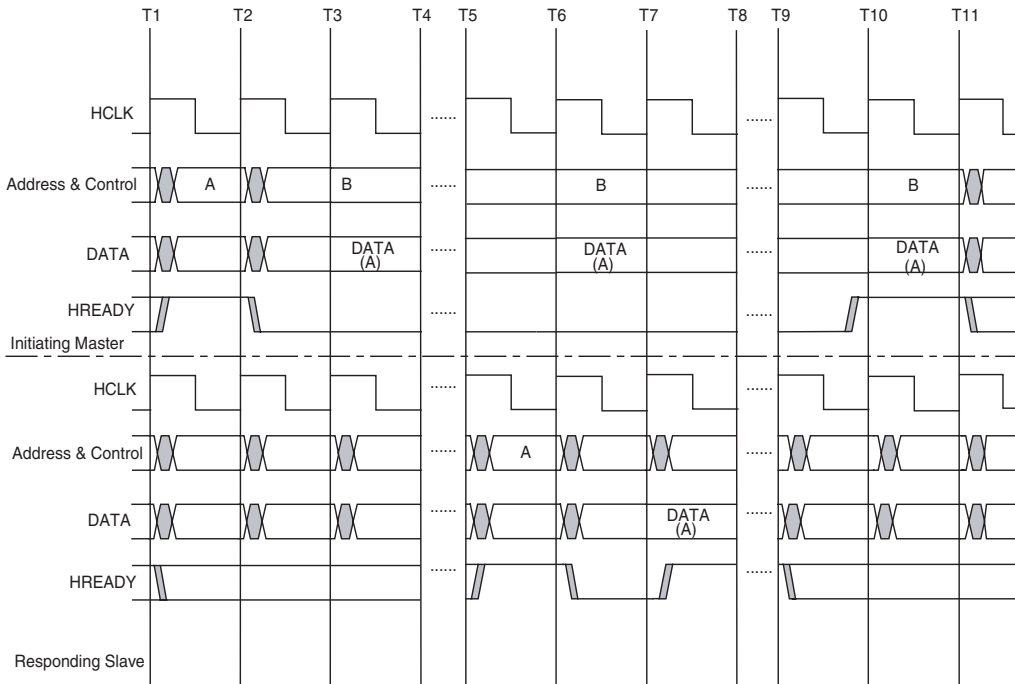
The embedded stripe bridges perform two types of write transaction:

- Non-posted writes
- Posted writes

For a non-posted write transaction, the initiating master passes the transaction to the slave interface on the bridge. The slave interface inserts wait states to the initiating master and sends the transaction to the write buffer, with a request, to the master interface. On receiving the transaction, the master interface requests the destination bus and synchronizes the transaction to the destination bus clock domain. Once the master interface is granted the bus, it drives the transaction to the destination slave. The master interface then samples the response from the slave and sends the response back to the read buffer, together with an acknowledgement indicating that it is ready to process another transaction. The slave interface reads the read buffer, synchronizes the response, relinquishes the wait states, and passes the response to the initiating master.

Figure 4 shows a timing diagram of a non-posted write transaction traversing an embedded stripe bridge.

Figure 4. Non-Posted Write Timing Diagram



The advantage of non-posted write transactions is that the initiating master can determine whether the data that was sent reached its destination before issuing another transaction. One example of a situation where non-posted writes are useful is when a master is writing to the configuration register of a slave to change how the slave processes transactions. The master should delay issuing another write transaction to the slave until the changes take affect.

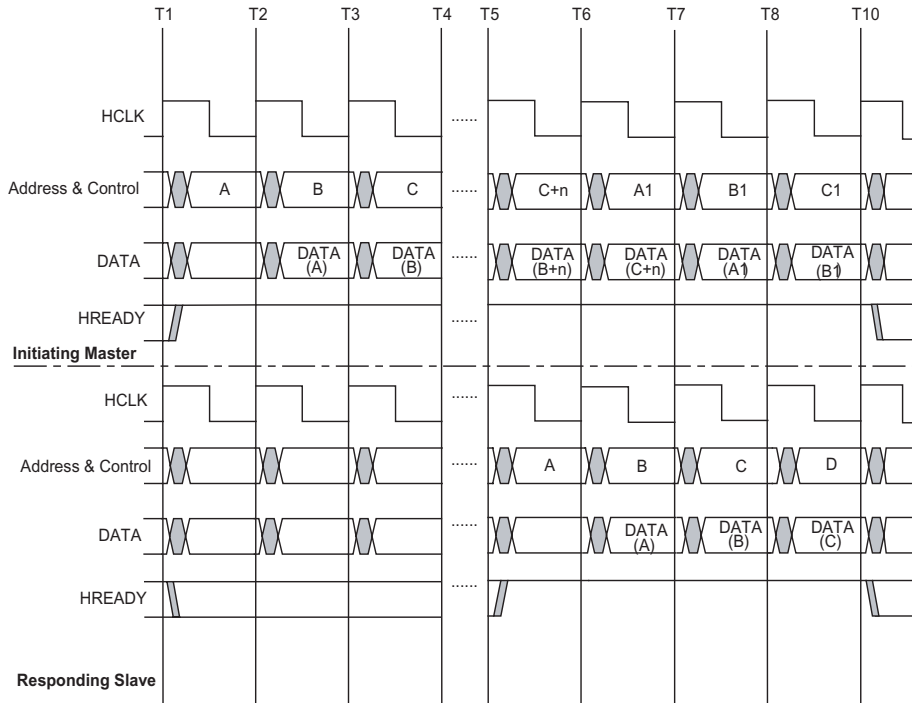
Although non-posted writes allow a master to know when a transaction has finished, they are quite slow. The initiating master could potentially see two arbitration delays, because it has to arbitrate for the bus it is on, and the master side of the bridge has to arbitrate for the destination bus. In addition, the transaction is synchronized twice before the transaction is complete. Both the address and control information and the responses from the slaves have to be synchronized. Figure 4 shows the synchronizations at times T4 to T5 and T8 to T9 respectively. Finally, the initiating master also sees any wait states inserted by the slave on the destination bus. In Figure 4, this is shown at T7.

To minimize the delays seen by the initiating master the embedded stripe bridges support posted-write transactions. Posted-write transactions allow bus masters to burst write data to the embedded stripe bridges; after the data is posted, the bus master can continue to process other transactions before the transaction posted to the bridge reaches its destination.

For a posted-write transaction, the initiating master passes the address and control information for the burst to the slave interface on the bridge. The slave interface accepts the address and control information, passes it to the write buffer and sends a request to the master interface. The slave interface accepts data for the burst until the write buffer is full, and then inserts wait states. Once the initiating master has finished writing the data for the burst, the slave interface responds and the initiating master can start the next transaction. The master interface reads the address and control information from the write buffer and synchronizes the destination bus clock domain before arbitrating for the bus. When the bus is granted, the master interface regenerates the address and control information for the transaction and reads the response to the transaction from the destination slave. If there is an erred transaction, the bridge generates an interrupt and the bridge status registers preserve information about the transaction that caused the interrupt.

[Figure 5 on page 8](#) is a timing diagram of a posted-write transaction traversing an embedded stripe bridge.

Figure 5. Posted Write Timing Diagram



In the example above the initiating master starts a burst at address A. The final location of the burst is represented by address $C+n$. At time T7, the initiating master starts a new burst transaction to location A1, which does not entail crossing the bridge.

Posted-write transactions can have a dramatic effect on the system performance. See [“System Performance” on page 11](#) for details of how posted writes affect system performance.

By default, write posting is enabled in the embedded stripe bridges. Setting the `NW` bit of the appropriate bridge control register disables write posting.

See [Appendix A](#) for details of how to disable/enable write posting.

Bridge Read Transactions

The embedded stripe bridges are able to perform two different types of read transaction:

- Non pre-fetched reads
- Pre-fetched reads

For either type of read, because the initiating master requires the data, it must wait for a response from the slave, which means that the sequence of events in the first beat of a read transaction is similar to a non-posted write. Subsequent beats differ, depending on the type of read transaction. Like non-posted writes, read transactions can incur a significant delay because of arbitration and synchronization. To reduce the delay, the embedded stripe bridges support pre-fetched read transactions.

When the amount of data requested by the master cannot be determined (i.e., an unspecified length burst), a pre-fetched read transaction anticipates the need for additional data by filling the read data buffer.

When the slave interface receives a transaction, it passes it to the master interface. For an undefined length burst the master interface fills the read buffer with the successive address locations from the slave until the read buffer is full or the burst is terminated. The slave interface passes the data from the read buffer to the initiating master. When a read buffer location becomes empty, the master interface pre-fetches the next address location from the slave. When the slave interface detects a new transaction from the master interface, any pre-fetched data in the read buffer is no longer valid.

Read pre-fetching increases performance. The improvement in performance depends on several variables. See [“System Performance” on page 11](#) for details.

By default, read pre-fetching is enabled in the embedded stripe bridges. Setting the NP bit of the appropriate bridge control register disables read pre-fetching for the bridge globally. You can also enable and disable read pre-fetching on a memory-region basis by modifying the NP bit of the memory map register for the region. See [Appendix A](#) for details of how to disable and enable read pre-fetching.



Do not use read pre-fetching on registers where reads have side effects.

Clock Synchronization

The PLD-to-stripe bridge and the stripe-to-PLD bridge synchronize transactions between the PLD and the AHB2 clock, because they are asynchronous. The AHB1-2 bridge has no clock synchronization circuitry, because the AHB2 clock is derived from the AHB1 clock.

The PLD-to-stripe and stripe-to-PLD bridges take between two and three clock cycles of the destination bus to synchronize a transaction with the initiating bus clock domain. Non-posted write transactions, non pre-fetched transactions, and pre-fetched read transactions incur two separate synchronization delays, because responses from the slave must be passed back to the initiating master. Posted-write transactions only incur one synchronization delay, because the slave interface of the bridge provides the response for the transaction, which is on the same clock domain.

Bridge Interrupts

The embedded stripe bridges can generate interrupts based on the status of transactions passing through the embedded stripe bridges. This is important in the case of posted-write transactions, because a master initiating a transaction does not know whether it reaches its final destination. The interrupt signals that there was a bus error.

When an embedded stripe bridge receives an error response from a slave, the bridge asserts the interrupt at the beginning of the second cycle of the mandatory two-cycle error response generated by the slave. For additional details on slave error responses, see the *AMBA Specification*.

The embedded processor can read the `INT_SOURCE_STATUS` register to determine whether the source of an interrupt was either the AHB1-2 bridge or the stripe-to-PLD bridge. See *Application Note nnn: Excalibur Solutions—Using the Excalibur Stripe Interrupt Controller* and the *ARM-Based Embedded Processor PLDs Hardware Reference Manual* for details on the `INT_SOURCE_STATUS` register.

For the PLD-to-stripe bridge, the `SLAVE_BUSERRINT` signal on the slave interface of the PLD-to-stripe bridge is asserted to alert the PLD master. Once the PLD master detects the assertion of `SLAVE_BUSERRINT`, it can then sample the PLD-to-stripe bridge registers by asserting `SLAVE_HSELREG`. The PLD-to-stripe registers contain the address and control information for the transaction that caused the interrupt. See [Figure 3 on page 24](#) for details on these registers.



The embedded processor does not have direct access to the registers PLD-to-Stripe registers. However, logic in the PLD can be produced so that the embedded processor can access these registers indirectly.

System Performance

There are many variables to consider when you are trying to estimate the system performance you can achieve using the embedded stripe bridges. Variables such as the relative clock speeds of the bridge interfaces and the access options of the peripherals have an effect on bridge throughput, bridge latency, and bus utilization. The rest of this section uses example transactions to highlight how system performance relates to interfacing with the embedded stripe bridges.

Bridge Throughput

The amount of data moved across the embedded stripe bridges in a given time varies, depending on variables such as write/read options for the embedded stripe bridges, relative clock speed, and the interface of the slave involved in the transaction.

The Effect of Posted Writes

Posted-write transactions can have a dramatic effect on system performance. [Table 1](#) shows bridge throughput numbers for a transaction from the embedded processor to a slave in the PLD. For this example, the master side of the AHB1-2 bridge is running at 200 MHz and the slave side is running at 100 MHz. Both the master and slave side of the stripe-to-PLD bridge are running at 100 MHz. [Figure 6](#) shows the process.

Figure 6. Transaction Path through the Device

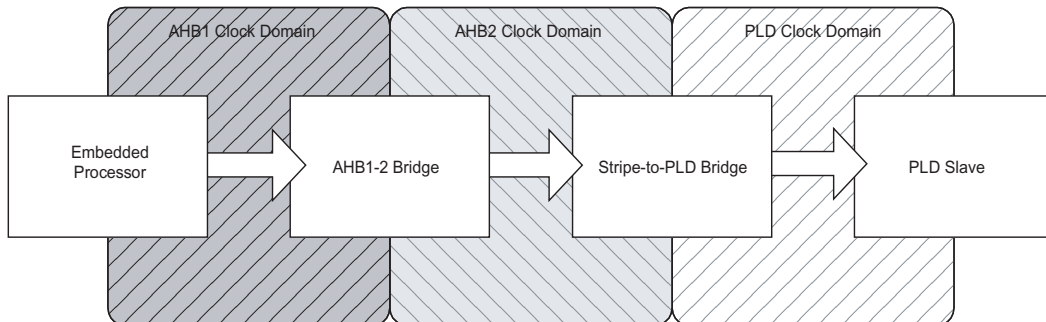


Table 1. Bridge Write Transaction Throughput from the Embedded Processor to a PLD Slave

Transaction Number	AHB1-2 Bridge	Stripe-to-PLD Bridge	Throughput (Mbytes per Second)
1	Write-posting enabled	Write-posting enabled	194.9
2	Write-posting enabled	Write-posting disabled	44.4
3	Write-posting disabled	Write-posting enabled	75.6
4	Write-posting disabled	Write-posting disabled	33

As shown in [Table 1](#), posted writes can have a dramatic effect on system performance, but their effect is greatest when the clock frequency of the initiating master exceeds the destination slave clock frequency.

Transactions 2 and 3 demonstrate the effect of relative clock frequency on throughput. For transaction 3, write posting is disabled in the AHB1-2 bridge and enabled in the stripe-to-PLD bridge. The initiating master is the embedded processor running at 200 MHz, and the destination slave is in the PLD, running at 100 MHz. The throughput for this transaction is 75.6 Mbytes per second, which is a significant increase on the throughput achieved when write posting is disabled for both embedded stripe bridges.

For transaction 2, write posting is enabled for the AHB1-2 bridge and disabled for the stripe-to-PLD bridge. The embedded processor posts the write transaction to the AHB1-2 bridge and, when the transaction is posted, it can issue another transaction immediately. Because the transaction is posted to the bridge, the master side of the AHB1-2 bridge becomes the initiating master, running at the same speed as the destination slave. The resulting throughput is 44.4 Mbytes per second, which is a smaller improvement over transaction 4 than that achieved by transaction 3.

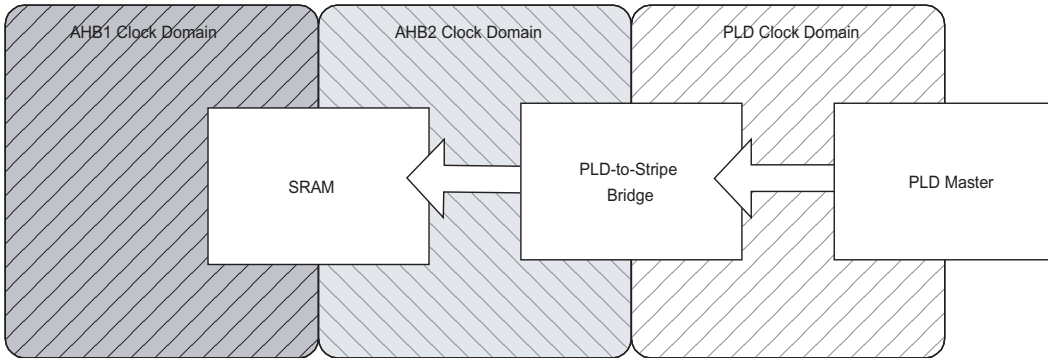
In general, write-posting has the greatest effect on throughput when the clock of the initiating master is faster than the destination slave clock.

The Effect of Read Pre-Fetching

Read pre-fetching can have a dramatic effect on system performance.

[Table 2](#) shows bridge throughput numbers for a transaction from a master in the PLD to SRAM in the embedded stripe. For this example, the master side of the PLD-to-stripe bridge was running at 100 MHz, and the slave side was varied to see the effect on throughput.

[Figure 7 on page 13](#) shows the read pre-fetching process for a transaction.

Figure 7. Transaction Path through the Device**Table 2. Bridge Read Transaction Throughput from SRAM to a PLD Master**

Transaction Number	Slave_hclk Frequency (PLD Clock Domain)	AHB2 Frequency	PLD-to-Stripe Bridge	Throughput (Mbytes per Second)
1	100 MHz	100 MHz	Pre-fetching enabled	100.4
2	100 MHz	100 MHz	Pre-fetching disabled	41.4
3	50 MHz	100 MHz	Pre-fetching enabled	98.5
4	50 MHz	100 MHz	Pre-fetching disabled	33.1

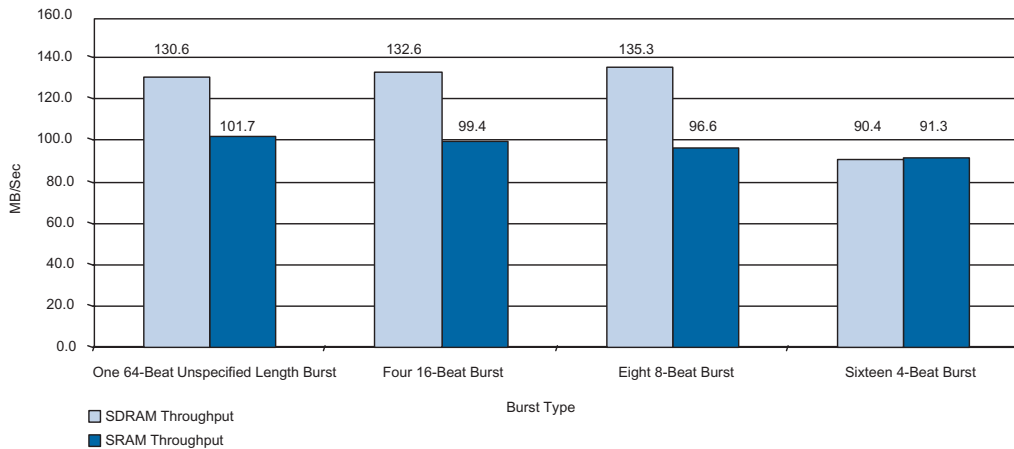
As shown in [Table 2](#), pre-fetching has a greater effect on throughput if the clock frequency of the initiating master is slower than the destination slave clock frequency. For transactions 1 and 2, the clock frequency of both the PLD master and AHB2 is 100 MHz. For transactions 3 and 4, the clock in the PLD master is running at half the speed of the AHB2 clock. The difference in throughput is higher in the case of transactions 3 and 4, because of the relative clock speed. Read pre-fetching allows the read buffer to fill at the speed of AHB2 instead of encountering BUSY cycles which have been inserted because the data from the slave has not reached the master interface of the bridge.

The Effect of the Slave AHB Interface Buffer

The sustained throughput of the system also depends on the ability of the destination slaves to accept data or generate data. The depth of the interface buffer of a slave and how it processes data affect the amount of information that can traverse the embedded stripe bridges between initiating masters and destination slaves within a given time frame.

Figure 8 charts the throughput for a master in the PLD performing different types of burst transaction to both SRAM and SDRAM in the embedded stripe. Both the master clock and the AHB2 clock are running at 100 MHz. The SDRAM is single data rate RAM running at 100 MHz. The master in the PLD transfers 64 words of data to SDRAM and SRAM. The 64 words of data are transferred using different burst lengths with one IDLE cycle between bursts: one 64-beat, four 16-beat, eight 8-beat and sixteen 4-beat bursts of data.

Figure 8. Throughput Versus Burst Type



The results of the transactions show that SDRAM is usually faster than SRAM. Because the interfaces to the stripe peripherals differ, so does the throughput.

When the SRAM interface is optimized for access from the AHB1 bus, the interface allows fast access for the embedded processor, but inserts wait states for every beat in a burst when a master in the PLD accesses the SRAM. Because wait states accompany every SRAM access, the greatest data throughput results when all the data is sent in one burst.

The SDRAM controller is asynchronous to the AHB2 clock, so transactions to SDRAM have to be synchronized. The synchronization for SDRAM is done in the SDRAM controller's FIFO instead of at the interface, which means that a transaction fills the SDRAM FIFO at constant speed instead of incurring a synchronizer delay in every beat of the burst.

In addition, when the SDRAM controller's FIFO is full, it inserts wait states to hold off any additional data presented by the PLD-to-stripe bridge. Prudent choices of burst lengths and allowing the FIFO time to empty both help to increase throughput. In the example in [Figure 8](#), an eight 8-beat burst yielded a higher throughput than one 64-beat burst. Although the difference in throughput is minor in the example, changes to relative clock speeds, buffer depth, and burst sequence magnify the difference.

Prudent choices of burst length can also help with the bus utilization. Incurring a persistent wait state in each transaction, which holds up the bus, is pointless when a transaction to another slave could occur while the FIFO in the SDRAM controller is emptying. This is an especially important consideration for the PLD-to-stripe bridge, which always has priority on AHB2. An imprudent burst length can stall AHB2 and prevent the processor from accessing AHB2.

Because of the many variations in systems, the best way to determine the optimum throughput for a given application is to simulate the transactions with the full stripe model. Refer to *Application Note 192: Excalibur Solutions-Embedded Stripe Performance Designs* or *Application Note 181: Excalibur Solutions—Multi-Master Reference Design* for details.

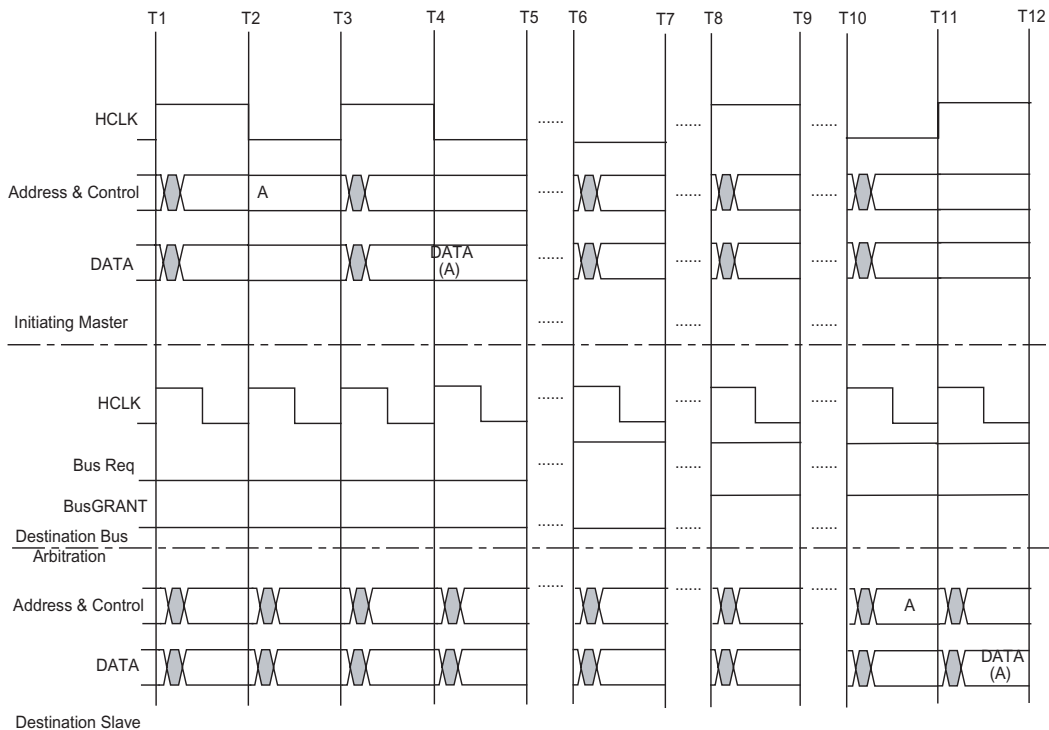
Bridge Latency

The time taken by first beat in a transaction is important for system performance. The latency of the embedded stripe bridges affects system efficiency and many variables contribute the overall performance. Clock relationships, the point of reference, and arbitration delay are variables that affect the latency through the embedded stripe bridges.

When calculating latency, it is important to establish a point of reference. The point of reference is the clock used to measure the latency cycles. In this document, the point of reference is the clock of the master that is initiating the transaction.

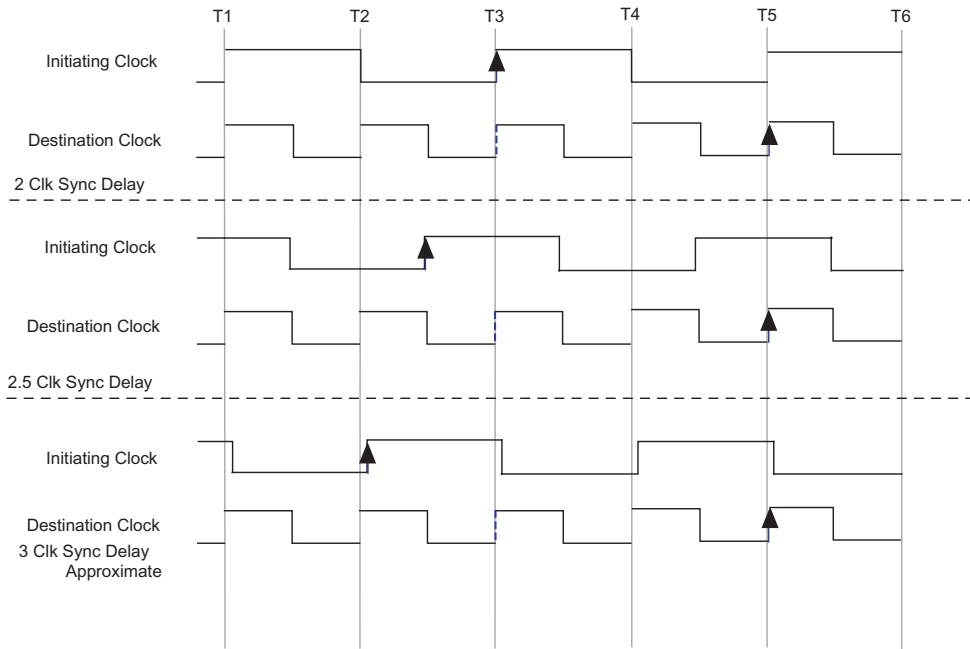
The components of the latency through the embedded stripe bridges are the synchronizer delay, the arbitration delay, the delay caused by slave wait states, and clock ratios. [Figure 9 on page 16](#) illustrates the delays.

Figure 9. Latency Timing



Synchronization Delay

The synchronizer delay is two to three clock cycles on the destination bus, depending on the skew of the clocks. The timing diagram in [Figure 10 on page 17](#) shows the number of synchronization clock delays based on clock skew. The arrow on the initiating clock shows the clock edge on which the transaction starts. The dotted line on the destination clock shows when the transaction is clocked in the synchronization circuitry. The arrow on the destination clock shows when data on the destination bus is valid.

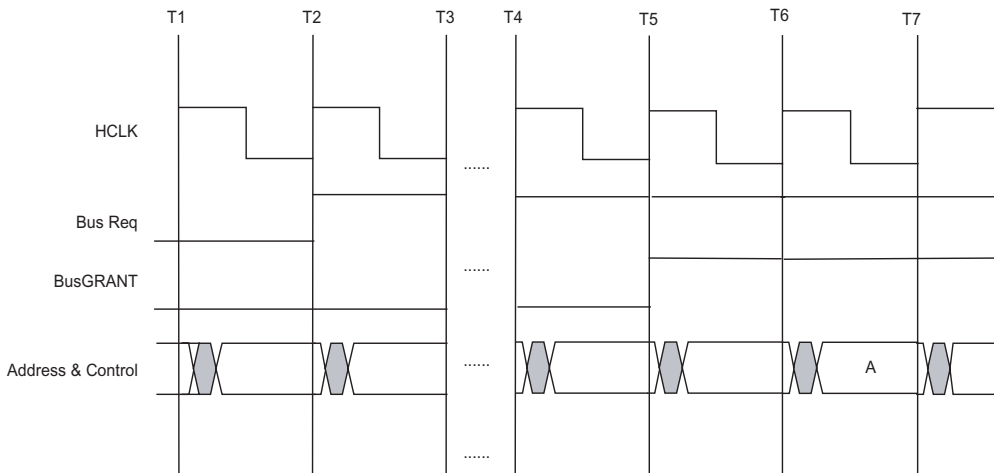
Figure 10. Synchronization Delay Based on Clock Skew

Posted writes incur a synchronizer delay and all other transaction types incur two. The second synchronizer delay for reads and non-posted writes is counted on the initiating bus. There is no synchronizer delay for transactions that travel across the AHB1-2 bridge. The AHB2 clock is derived from the AHB1 clock and therefore they are synchronous.

Arbitration Delay

Transactions that travel across the bridge incur an arbitration delay, because the master interface must gain access to the destination bus.

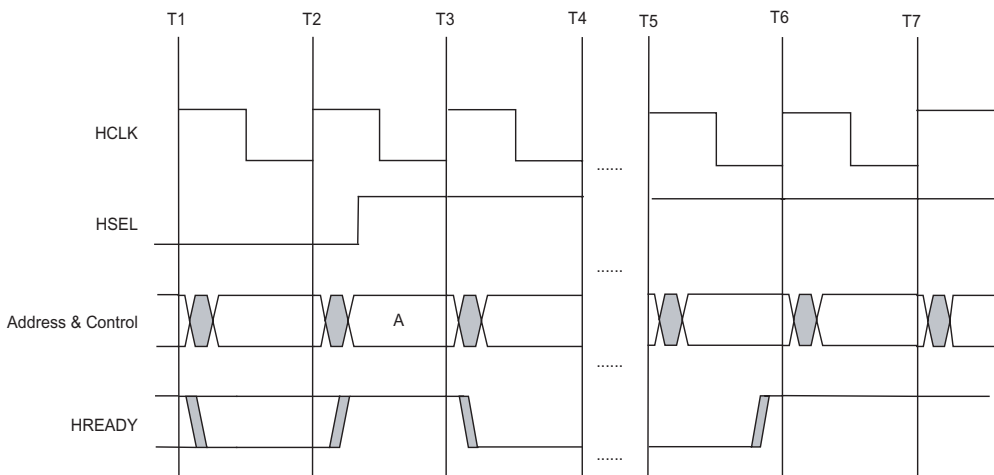
[Figure 11 on page 18](#) shows the arbitration sequence across the master interface of the embedded stripe bridges.

Figure 11. Arbitration Delay

At time T2 in [Figure 11](#), the master interface requests ownership of the destination bus. The time between T3 and T4 is delayed by other activity on the destination bus, but if the bus is idle this delay is omitted. T4 to T5 is the bus handover cycle. T5 to T6 is the extra clock required for the master interface to drive the bus after it has obtained ownership.

Wait State Delay

The interface of the destination slave inserts wait states, which are the final component of the bus latency. [Figure 12 on page 19](#) is a timing diagram shows the wait state delay inserted by the slave.

Figure 12. Wait State Delay

The time from T3 to T6 represents the wait states that are inserted by the slave. One additional clock cycle is needed for the transition from the address phase to the data phase of the transaction.

Clock Ratios

The sum of the synchronization delay, arbitration delay, and the wait state delay on the destination bus must be translated to the number of clocks on the initiating bus. The calculation involves multiplying the sum of the delays by the ratio of the initiating bus clock speed to the destination bus clock speed. The resulting number, plus any additional clock cycles on the initiating bus due to resynchronization, is the final number of clocks it takes for the transaction to travel across the bridge.

Summary

The Excalibur embedded stripe is a powerful platform that meets the needs of current embedded applications. There are two buses in the embedded stripe, AHB1 and AHB2, which facilitate on-chip communications and also allow the segregation of peripherals. Separating slow peripherals from fast peripherals allows fast peripherals to maximize the time for which they can function at sustained high speed. The embedded stripe bridges provide an effective means of communication between the two embedded stripe buses and the PLD, with minimal latency. The write posting and read pre-fetching features of the embedded stripe bridges increase system performance.

Because of the many variants in systems, the best way to determine the optimum latency for a given application is to simulate the transaction using the full stripe model. See *Application Note 192: Excalibur Solutions-Embedded Stripe Performance Designs* or *Application Note 181: Excalibur Solutions - Multi-Master Reference Design* for details.

Revision History

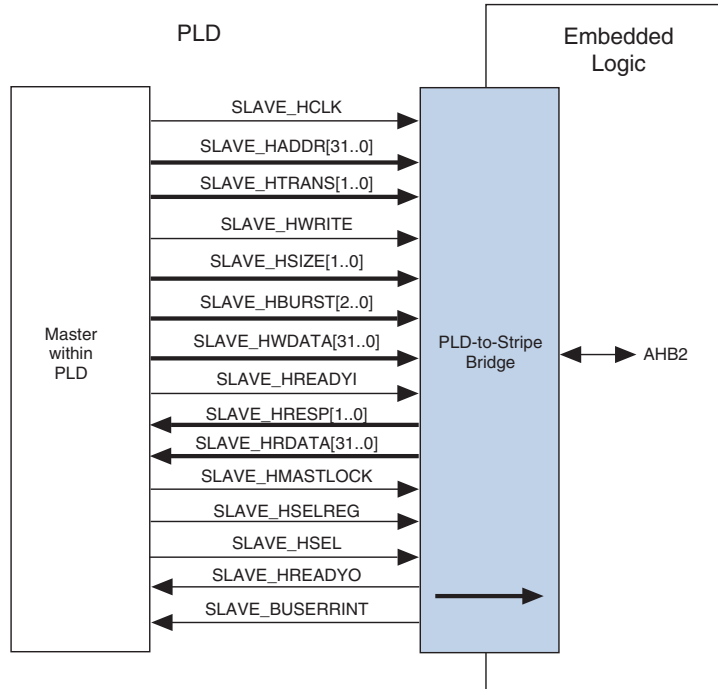
Table 3 shows the document revision history.

Table 3. Revision History	
Date	Description
March 2001	Initial publication
January 2002 (2.0)	Major document revision
June 2002 (2.1)	Minor cross-reference updates

PLD-to-Stripe Bridge Interface

This section describes the PLD-to-stripe bridge interface. [Figure 1](#) shows the architecture of the PLD-to-stripe bridge.

Figure 1. PLD-to-Stripe Bridge



[Table 1 on page 22](#) lists the signals on the PLD-to-stripe bridge and describes them.

Table 1. PLD-to-Stripe Bridge Signals *Note (1)*

Signal	Source	Description
SLAVE_HCLK	PLD	Times all bus transfers. Signal timings are related to its rising edge clock
SLAVE_HADDR[31..0]	PLD	32-bit system address bus
SLAVE_HTRANS[1..0]	PLD	The type of the current transfer
SLAVE_HWRITE	PLD	When high, this signal indicates a write transfer; when low, a read transfer
SLAVE_HSIZE[1..0]	PLD	Indicates the size of transfer
SLAVE_HBURST[2..0]	PLD	Indicates whether the transfer forms part of a burst
SLAVE_HWDATA[31..0]	PLD	Used to transfer data from the master to the bus slaves during writes
SLAVE_HREADYI	PLD	When high, this signal indicates that a transfer has finished on the bus. Slaves on the bus need SLAVE_HREADY as both an input and output signal
SLAVE_HREADYO	Stripe	When high, this signal indicates that a transfer has finished on the bus. Slaves on the bus need SLAVE_HREADY as both an input and output signal
SLAVE_HRESP[1..0]	Stripe	Additional information on the status of a transfer
SLAVE_HRDATA[31..0]	Stripe	Used to transfer data from bus slaves to the master during reads
SLAVE_HMASTLOCK	PLD	When high, indicates that the master requires locked access to the bus
SLAVE_BUSERRINT	Stripe	Interrupt signifying a bus error
SLAVE_HSELREG	PLD	Register selection signal
SLAVE_HSEL	PLD	Interface selection signal

Note:

(1) For further details, refer to the *AMBA Specification, Revision 2.0*.

Stripe-to-PLD Bridge Interface

This section describes the stripe-to-PLD bridge interface. [Figure 1 on page 21](#) shows the architecture of the stripe-to-PLD bridge.

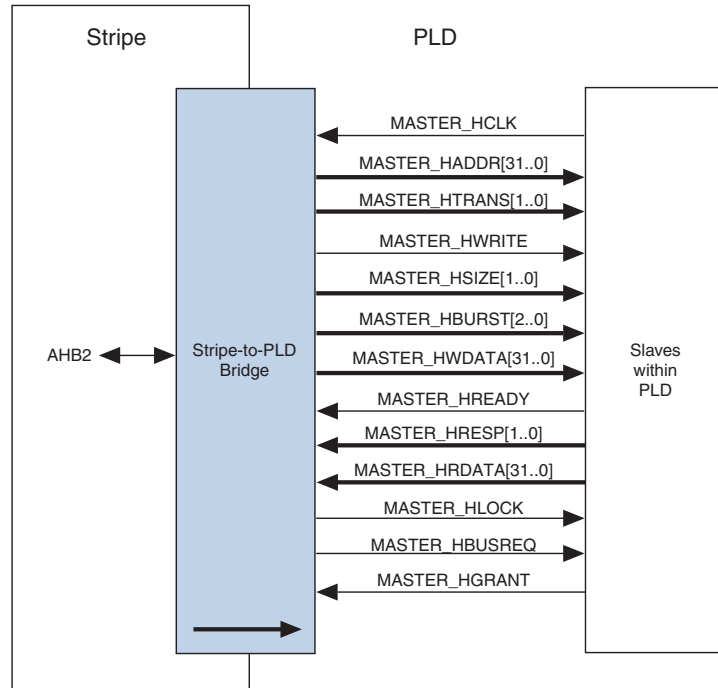
Figure 2. Stripe-to-PLD Bridge

Table 2 lists the signals on the stripe-to-PLD bridge and describes them..

Table 2. Stripe-to-PLD Bridge Signals (Part 1 of 2) Note (1)

Signal	Source	Description
MASTER_HCLK	PLD	Times all bus transfers. Signal timings are related to its rising edge clock
MASTER_HADDR[31..0]	Stripe	32-bit system address bus
MASTER_HTRANS[1..0]	Stripe	Type of the current transfer
MASTER_HWRITE	Stripe	When high, this signal indicates a write transfer; when low, a read transfer
MASTER_HSIZE[1..0]	Stripe	Indicates the size of transfer
MASTER_HBURST[2..0]	Stripe	Indicates whether the transfer forms part of a burst
MASTER_HWDATA[31..0]	Stripe	Used to transfer data from the master to the bus slaves during writes
MASTER_HREADY	PLD	When high, this signal indicates that a transfer has finished on the bus
MASTER_HRESP[1..0]	PLD	Additional information on the status of a transfer
MASTER_HRDATA[31..0]	PLD	Used to transfer data from bus slaves to the master during reads
MASTER_HLOCK	Stripe	When high, indicates that the master requires locked access to the bus
MASTER_HBUSREQ	Stripe	A signal from the master to the arbiter, requesting the bus

Table 2. Stripe-to-PLD Bridge Signals (Part 2 of 2) Note (1)

Signal	Source	Description
MASTER_HGRANT	PLD	In conjunction with MASTER_HREADY, indicates that the bus master has been granted the bus

Note:

(1) For further details, refer to the *AMBA Specification, Revision 2.0*.

Embedded Stripe Bridge Registers

This section describes the control registers in the embedded stripe and explains how to enable and disable read-prefetching and write-posting.

Figure 3 shows the format of the bridge registers in the embedded stripe.

Figure 3. Embedded Stripe Bridge Registers

Register Name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Address
AHB12B_CR																	NW	NP	Base + 100H														
AHB12B_SR																	HBURST	HSIZE	HRTN	WF	Base + 800H												
AHB12B_ADDRSR	HADDR																		Base + 804H														
PLDSB_CR																	NW	NP	Base + 110H														
PLDSB_SR																	HBURST	HSIZE	HRTN	WF	Base + 114H												
PLDSB_ADDRSR	HADDR																		Base + 118H														
PLDMB_CR																	NW	NP	Base + 120H														
PLDMB_SR																	HBURST	HSIZE	HRTN	WF	inaccessible												
PLDMB_ADDRSR	HADDR																		inaccessible														

Figure 4 shows the control register format for the memory regions. Each memory region is governed by a dedicated control register.

Figure 4. Format of the Memory Map Registers

Register Name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Address
MMAP_xxx																	NW	NP	Base + nnn														



For specific details of the register fields, see the *ARM-Based Embedded Processor PLDs Hardware Reference Manual*.

Enabling Write-Posting

You can enable write-posting for a specific bridge. To do so, you need to clear the NW bit in the appropriate control register: AHB12B_CR, PLDSB_CR, or PLDMB_CR (see Figure 3).

Enabling Read Pre-Fetching

You can either enable read-prefetching for a specific bridge or for a memory region:

- To enable read-prefetching for a specific bridge, you need to set the NP bit in the control register for the appropriate bridge: AHB12B_CR, PLDSB_CR, or PLDMB_CR (see [Figure 3](#))
- To enable read-prefetching for a memory region, you need to set the NP bit in the control register for the appropriate memory region (see [Figure 4](#))



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>
Applications Hotline:
(800) 800-EPLD
Literature Services:
lit_req@altera.com

Copyright © 2002 Altera Corporation. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services. All rights reserved.



I.S. EN ISO 9001