ARM966E-S

(Rev 1)

Technical Reference Manual

ARM DDI 0186A

ARM966E-S
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
31st July 2000	A	First Release

Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

`http://www.arm.com`

Contents

ARM966E-S Technical Reference Manual

List of Tables
ARM966E-S Technical Reference Manual

List of Figures
ARM966E-S Technical Reference Manual

Preface

This preface introduces the ARM966E-S and its reference documentation. It contains the following sections:

- *About this document* on page xii
- *Further reading* on page xv
- *Feedback* on page xvi.

About this document

This document is a reference manual for the ARM966E-S.

Intended audience

This document has been written for experienced hardware and software engineers who might or might not have any experience of ARM products.

Using this manual

This document is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the ARM966E-S.

Chapter 2 ***Programmer's Model***

Read this chapter for a description of the programmer's model including a summary of the ARM966E-S coprocessor registers.

Chapter 3 ***Memory Map***

Read this chapter for a description of the ARM966E-S fixed memory map implementation.

Chapter 4 ***Tightly-coupled SRAM***

Read this chapter for a description of the requirements and operation of the tightly-coupled SRAM.

Chapter 5 ***Direct Memory Access (DMA)*****)**

Read this chapter for a description of the optional DMA interface in the ARM966E-S.

Chapter 6 ***Bus Interface Unit***

Read this chapter for a description of the operation of the Bus Interface Unit and write buffer.

Chapter 7 ***Coprocessor Interface***

Read this chapter for a description of the coprocessor interface and the operation of common coprocessor instructions.

Chapter 8 ***Debug Support***

Read this chapter for a description of the debug support for the ARM966E-S and the EmbeddedICE-RT logic.

Chapter 9 ***Embedded Trace Macrocell Interface***

Read this chapter for a description of the ETM interface, including details of how to enable the interface.

Chapter 10 ***Test Support***

Read this chapter for a description of the test methodology used for the ARM966E-S synthesized logic and tightly-coupled SRAM.

Appendix A ***Signal Descriptions***

Read this appendix for a description of the ARM966E-S signals.

Appendix B ***AC Parameters***

Read this appendix for a description of the timing parameters applicable to the ARM966E-S.

Appendix C ***SRAM Stall Cycles***

Read this appendix for a description of the tightly-coupled SRAM stall cycle mechanism in the ARM966E-S.

Typographical

The typographical conventions are:

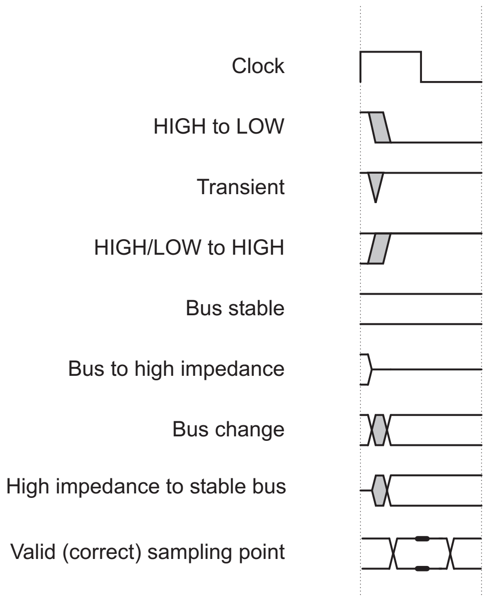
italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at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

< and > Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example:

- `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
- The Opcode_2 value selects which register is accessed.

Timing diagram conventions

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



Key to timing diagram conventions

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

Further reading

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

If you would like further information on ARM products, or if you have questions not answered by this document, please contact `info@arm.com` or visit our web site at `http://www.arm.com`.

ARM publications

ARM Architecture Reference Manual (ARM DDI 0100).

ARM9E-S Technical Reference Manual (ARM DDI 0165).

AMBA Specification Rev 2.0 (ARM IHI 0011).

AHB Example AMBA System Technical Reference Manual (ARM DDI 0170).

Other publications

IEEE Std. 1149.1- 1990, *Standard Test Access Port and Boundary-Scan Architecture*.

Feedback

ARM Limited welcomes feedback both on the ARM966E-S, and on the documentation.

Feedback on the ARM966E-S

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

- the product name
- a concise explanation of your comments

Feedback on the ARM966E-S

If you have any comments about this document, please send e-mail to `errata@arm.com` giving:

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

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM966E-S processor. It contains the following sections:

1.1 About the ARM966E-S

The ARM966E-S is a *synthesizable* macrocell combining an ARM processor with tightly-coupled SRAM memory. It is a member of the ARM9 Thumb family of high-performance, 32-bit *System-on-Chip* (SoC) processor solutions and is targeted at a wide range of embedded applications where high performance, low system cost, small die size, and low power are all important.

The ARM966E-S processor macrocell provides a complete high-performance processor subsystem, including an ARM9E-S RISC integer CPU, tightly-coupled SRAM for each of the instruction and data CPU interfaces, write buffer and an AMBA AHB bus interface. Providing this complete high-frequency subsystem frees the SoC designer to concentrate on design issues unique to their system. The synthesizable nature of the device eases integration into ASIC technologies.

The tightly-coupled SRAMs within the ARM966E-S macrocell allow high-speed operation without incurring the performance and power penalties of accessing the system bus, while having a lower area overhead than a cached memory system. The size of both the instruction and data SRAM are implementor-configurable to allow tailoring of the hardware to the embedded application. Additionally, You can configure the data SRAM interface to allow *Direct Memory Access* (DMA) to this RAM.

The ARM9E-S core within the ARM966E-S macrocell executes both the 32-bit ARM and 16-bit Thumb instruction sets, allowing trade off between high performance and high code density. Additionally the ARM9E-S features:

- ARMv5T 32-bit instruction set with improved ARM/Thumb code interworking and enhanced multiplier designed for improved DSP performance
- ARM debug architecture with additional support for real-time debug, which allows critical exception handlers to execute while debugging the system.

The ARM966E-S includes support for external coprocessors allowing floating point or other application-specific hardware acceleration to be added.

To minimize die size and power consumption the ARM966E-S does not provide virtual to physical address mapping as this is not required by most embedded systems. A simple fixed memory map is implemented for the close-coupled local RAM, ideally suited to small, fast, real-time embedded control applications.

The ARM966E-S synthesizable implementation supports the use of a scan test methodology for the standard cell logic and *Built-In-Self-Test* (BIST) for the tightly-coupled SRAM.

1.2 Microprocessor block diagram

The ARM966E-S block diagram is shown in Figure 1-1.

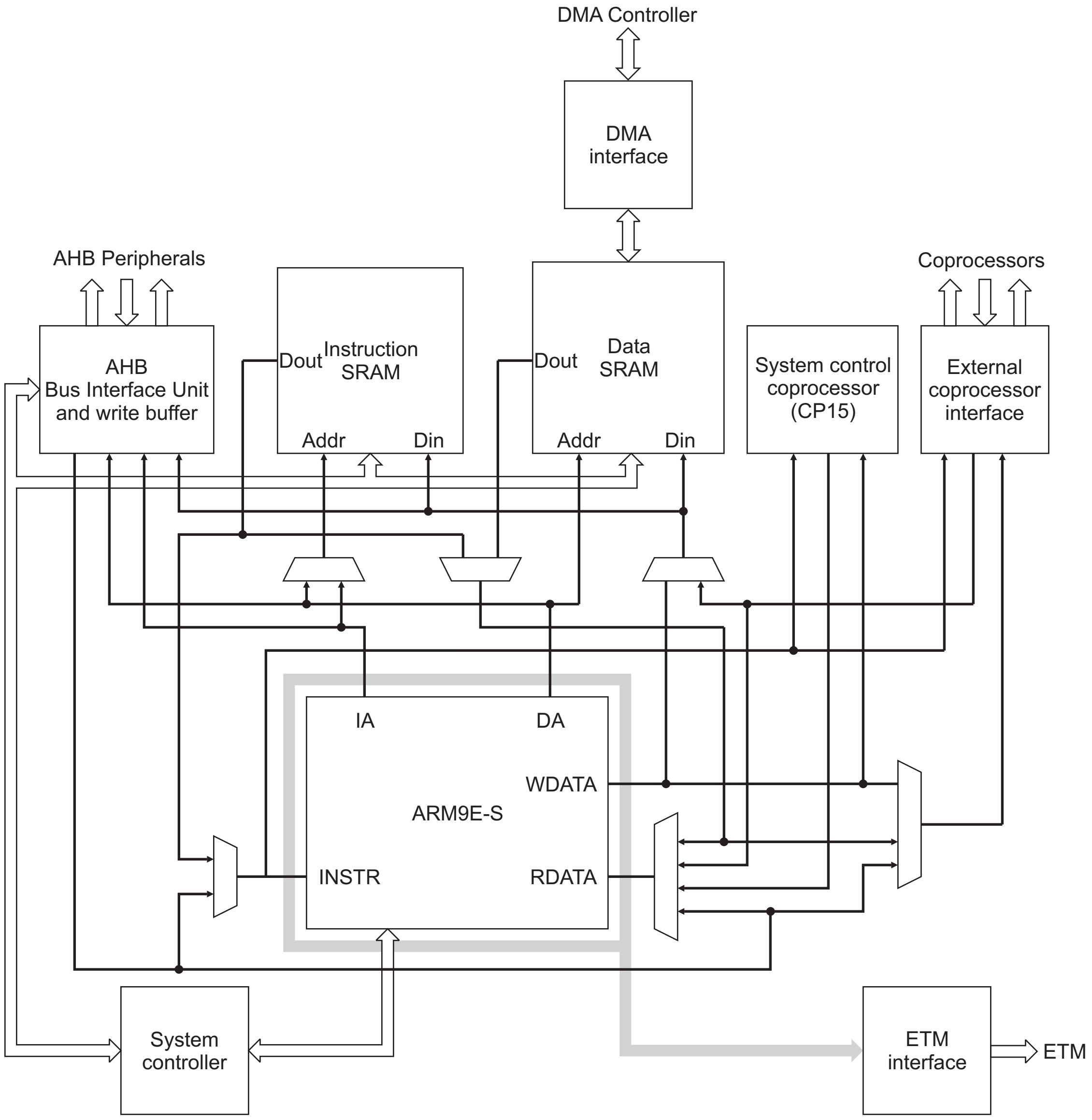


Figure 1-1 ARM966E-S block diagram

Chapter 2
Programmer's Model

This chapter describes the programmer's model for the ARM966E-S. It contains the following sections:

2.1 About the programmer's model

The programmer's model for the ARM966E-S macrocell primarily consists of the ARM9E-S core programmer's model (see *About the ARM9E-S programmer's model* on page 2-3). Additions to this model are required to control the operation of the ARM966E-S internal coprocessors, and any coprocessor connected to the external coprocessor interface.

There are two internal coprocessors within the ARM966E-S:

- CP14 within the ARM9E-S core allows software access to the debug communications channel
- CP15 allows configuration of the tightly-coupled SRAM and write buffer and other ARM966E-S system options such as big or little-endian operation.

The registers defined in CP14 are accessible with `MCR` and `MRC` instructions. These are described in *The debug communications channel* on page 8-19.

The registers defined in CP15 are accessible with `MCR` and `MRC` instructions. These are described in *ARM966E-S CP15 registers* on page 2-4.

Any coprocessors registers and operations, attached to the external coprocessor interface, are accessible with appropriate coprocessor instructions.

2.2 About the ARM9E-S programmer's model

The ARM9E-S processor core implements the ARM architecture v5T, that includes the 32-bit ARM instruction set and the 16-bit Thumb instruction set. For a description of both instruction sets, see the *ARM Architecture Reference Manual*. Contact ARM for complete descriptions of both instruction sets.

2.2.1 Data Abort model

The ARM9E-S implements the *base restored data abort model*, that differs from the *base updated data abort model* implemented by ARM7TDMI.

The difference in the Data Abort model affects only a very small section of operating system code, the Data Abort handler. It does not affect user code. With the *base restored data abort model*, when a Data Abort exception occurs during the execution of a memory access instruction, the base register is always restored by the processor hardware to the value the register contained *before* the instruction was executed. This removes the requirement for the Data Abort handler to *unwind* any base register update that might have been specified by the aborted instruction.

The *base restored data abort model* significantly simplifies the software Data Abort handler.

2.3 ARM966E-S CP15 registers

CP15 allows configuration of the tightly-coupled SRAM and write buffer and other ARM966E-S system options such as big or little-endian operation.

The ARM966E-S coprocessor 15 registers are described in the following sections:

- *CP15 register map summary*
- *Register 0, ID code* on page 2-5
- *Register 1, Control register* on page 2-5
- *Register 7, Core control* on page 2-7
- *Register 15, Test* on page 2-9.

2.3.1 CP15 register map summary

The ARM966E-S incorporates CP15 for system control. The register map for CP15 is shown in Table 2-1.

Table 2-1 CP15 register map

Register	Function	Access
0	ID code	Read-only
1	Control	Read/write
2-6	Reserved	Undefined
7	Core control	Write-only
13	Trace process identifier	Read/write
8-14	Reserved	Undefined
15	Test	Read/write

Note

Register 15 provides access to more than one register. The register access depends on the value of the opcode_2 field. See the register descriptions in this section for more information.

2.3.2 Register 0, ID code

This is a read-only register that returns a 32-bit device ID code. The ID code register is accessed by reading CP15 register 0 with the opcode_2 field set to any value. For example:

```
MRC p15, 0, rd, c0, c0, 0; returns ID register
```

The contents of the ID code are shown in Table 2-2.

Table 2-2 Register 0, ID code

Register bits	Function	Value
31:24	Implementor	`0x41`
23:20	Variant	`0x0`
19:16	ARM architecture v5T	`0x05`
15:4	Part number	`0x966`
3:0	Version	Version specific

2.3.3 Register 1, Control register

This register contains the global control bits of the ARM966E-S (see Table 2-3). All reserved bits must either be written with zero or one, as indicated, or written using read-modify-write. The reserved bits have an unpredictable value when read. To read and write this register:

```
MRC p15, 0, rd, c1, c0, 0; read Control register
MCR p15, 0, rd, c1, c0, 0; write Control register
```

Table 2-3 Register 1, Control register

Register bit	Function
31:16	Reserved (should be zero)
15	Configure disable loading **TBIT**
14	Reserved (should be zero)
13	Alternate vector select
12	Instruction SRAM enable

Table 2-3 Register 1, Control register (continued)

Register bit	Function
11:8	Reserved (should be one)
7	Endian
6:4	Reserved (should be one)
3	Write buffer enable
2	Data SRAM enable
1:0	Reserved (should be zero)

Bit 15, Configure disable loading TBIT

When HIGH the ARM9E-S core disables certain ARMv5T defined behavior involving loading data to the PC. This bit is cleared LOW during reset to provide ARMv5T compatibility.

Bit 13, Alternate vectors select

This bit controls the base address used for the exception vectors. When LOW, the base address for the exception vectors is `0x0000 0000`. When HIGH, the base address is `0xFFFF 0000`.

Note

Bit 13 is initialized either HIGH or LOW during system reset, depending on the value of the input pin, **VINITHI**. This allows the exception vector location to be defined during reset to suit the boot mechanism of the application. You can then reprogram as required following system reset.

Bit 12, Instruction SRAM enable

This bit controls the behavior of the tightly-coupled instruction SRAM. When HIGH, all accesses to the fixed instruction memory space as shown in Figure 3-1 on page 3-2, access the instruction SRAM. When LOW, all accesses to the instruction memory space access the AMBA AHB.

Note

Bit 12 is initialized either HIGH or LOW during system reset depending on the value of the input pin **INITRAM**.

Bit 7, Endian

Selects the endian configuration of the ARM966E-S. When this bit is HIGH, big-endian configuration is selected. When LOW, little-endian configuration is selected. This bit is cleared LOW during reset.

Bit 3, Write buffer enable

This bit controls the use of the write buffer. When HIGH, all stores to the fixed bufferable space of the AMBA AHB (as shown in Figure 3-1 on page 3-2) are treated as buffered writes. When LOW, all stores to the AMBA AHB are treated as nonbufferable.

If the write buffer is disabled having previously been enabled, any writes already in the write buffer FIFO complete as buffered writes.

This bit is cleared LOW during reset.

Bit 2, Data SRAM enable

This bit controls the behavior of the tightly-coupled Data SRAM. When HIGH, all data interface accesses to the fixed data memory space as shown in Figure 3-1 on page 3-2, access the Data SRAM. When LOW, all accesses to the data memory space access the AMBA AHB.

Note

Bit 2 is initialized either HIGH or LOW during system reset depending on the value of the input pin **INITRAM**.

2.3.4 Register 7, Core control

You can use a write to this register, to perform *wait for interrupt* and *drain write buffer* operations.

Wait for interrupt

This operation allows the ARM966E-S to enter a low-power standby mode. When the operation is invoked, the clock enable to the processor core is negated until either an interrupt or a debug request occurs. This function is invoked by a write to Register 7. The following ARM instruction causes this to occur:

```
MCR p15, 0, rd, c7, c0, 4; wait for interrupt
```

This is the preferred encoding that must be used by new software. For compatibility with existing software, ARM966E-S also supports the following ARM instruction that has the same affect:

```
MCR p15, 0, rd, c15, c8, 2; wait for interrupt
```

This stalls the processor from the time that the instruction is executed until **nFIQ**, **nIRQ**, or **EDBGRQ** are asserted. Also, if the debugger sets the debug request bit in the EmbeddedICE-RT control register then this causes the wait-for-interrupt condition to terminate.

In the case of **nFIQ** and **nIRQ**, the processor core is *woken up* regardless of whether the interrupts are enabled or disabled (that is, independent of the I and F bits in the processor CPSR). The debug-related *waking* only occurs if **DBGEN** is HIGH, that is, only when debug is enabled.

If interrupts are enabled, the ARM9E-S core is guaranteed to take the interrupt before executing the instruction after the wait for interrupt. If debug request is used to wake up the system, the processor enters debug-state before executing any more instructions.

Wait for interrupt does not prevent the write buffer from emptying.

Drain write buffer

This CP15 operation causes instruction execution to be stalled until the write buffer is emptied. This operation is useful in real-time applications where the processor has to be sure that a write to a peripheral has completed before program execution continues. An example is where a peripheral in a bufferable region is the source of an interrupt. When the interrupt has been serviced, the request must be removed before interrupts can be re-enabled. This can be ensured if a drain write buffer operation separates the store to the peripheral and the enable interrupt functions.

The drain write buffer operation is invoked by a write to Register 7 using the following ARM instruction:

```
MCR cp15, 0, rd, c7, c10, 4; drain write buffer
```

This stalls the processor core until any outstanding accesses in the write buffer have been completed, that is, until all data has been written to external memory.

2.3.5 Register 13, Trace process identifier

This register provides a mechanism to allow the Real-time Trace tools to identify the currently executing process in multi-tasking environments.

The contents of this register are replicated on the **ETMPROCID** pins of the ARM966E-S. The **ETMPROCIDWR** signal is set HIGH for a single clock cycle whenever this register is written to. Table 2-4 shows the trace process identifier for read and write.

Table 2-4 Register 13, Trace process identifier

Register	Read	Write
Trace Process Identifier	MRC p15,0,Rd,c13,c1,1	MCR p15,0,Rd,c13,c1,1

2.3.6 Register 15, Test

This register provides access to:

- the tightly-coupled Instruction and Data SRAM test features
- the trace control features.

Both features are supported by the ARM966E-S.

The register map for CP15 register 15 is shown in Table 2-5.

Table 2-5 Register 15, Test register map

Register	Read	Write
Trace Control Register	`MRC p15, 1, Rd, c15, c1, 0`	`MCR p15, 1, Rd, c15, c1, 0`
BIST control register	`MRC p15, 1, Rd, c15, c0, 1`	`MCR p15, 1, Rd, c15, c0, 1`
Instruction BIST address register	`MRC p15, 1, Rd, c15, c0, 2`	`MCR p15, 1, Rd, c15, c0, 2`
Instruction BIST general register	`MRC p15, 1, Rd, c15, c0, 3`	`MCR p15, 1, Rd, c15, c0, 3`
Data BIST address register	`MRC p15, 1, Rd, c15, c0, 6`	`MCR p15, 1, Rd, c15, c0, 6`
Data BIST general register	`MRC p15, 1, Rd, c15, c0, 7`	`MCR p15, 1, Rd, c15, c0, 7`

Note

Opcode_1 is set HIGH when accessing Register 15. Opcode_2 is used to index registers within the Register 15 register map.

Trace control register

The trace control register allows the masking of interrupts during trace. This register allows **nIRQ** and **nFIQ** interrupt priority over **FIFOFULL** to be programmed. Table 2-6 shows the bit assignments within the Trace control register.

Table 2-6 Trace control register

Register bit	Content
0	Reserved (should be zero)
1	1 = Mask **nIRQ** interrupts during trace 0= Do not mask **nIRQ** interrupts during trace
2	1 = Mask **nFIQ** interrupts during trace 0 = Do not mask **nFIQ** interrupts during trace
31:3	Reserved (should be zero)

BIST control register

Table 2-7 BIST control register

Register bit	Meaning when written	Meaning when read
31:21	Instruction SRAM BIST size	Instruction SRAM BIST size
20	Reserved (should be zero)	Instruction SRAM BIST complete flag
19	Reserved (should be zero)	Instruction SRAM BIST fail flag
18	Instruction SRAM BIST enable	Instruction SRAM BIST enable
17	Instruction SRAM BIST pause	Instruction SRAM BIST pause
16	Instruction SRAM BIST start strobe	Instruction SRAM BIST running flag
15:5	Data SRAM BIST size	Data SRAM BIST size
4	Reserved (should be zero)	Data SRAM BIST complete flag
3	Reserved (should be zero)	Data SRAM BIST fail flag

Table 2-7 BIST control register (continued)

Register bit	Meaning when written	Meaning when read
2	Data SRAM BIST enable	Data SRAM BIST enable
1	Data SRAM BIST pause	Data SRAM BIST pause
0	Data SRAM BIST start strobe	Data SRAM BIST running flag

Table 2-7 on page 2-10 shows the bit assignments within the BIST control register.

At reset, all bits are cleared LOW. BIST must be enabled before a BIST operation is started. When BIST is enabled to test one or both tightly-coupled SRAMs, the SRAM being tested is automatically disabled by clearing its enable bit in CP15 Register 1. This is to prevent the programmer inadvertently using the SRAM following a BIST operation, because the BIST algorithm corrupts the SRAM contents.

The BIST size field determines the size of the BIST operation. The value written to this field N, is decoded as follows:

BIST size in bytes = 2^{N+2}

Some examples are shown in Table 2-8.

Table 2-8 BIST size encoding examples

Instruction RAM BIST size [31:21]	N	Size of test
000000 00001 (minimum)	1	8 bytes
000000 00100	4	64 bytes
000000 00111	7	512 bytes
000000 01000	8	1 KB
000000 01010	10	4 KB
000000 01111	15	128 KB
000000 11000 (maximum)	24	64 MB

Note

BIST size bits [31:26] should be zero.

Writing to the BIST control register with Bit[0] set initiates a Data SRAM BIST operation.

Writing to the BIST control register with Bit[16] set initiates an Instruction SRAM BIST operation.

You can run Instruction and Data BIST operations individually or concurrently. You must set up the Size, Pause and Enable bits within the BIST control register prior to initiating a BIST operation.

Reading the BIST control register returns the status of the BIST operations. See *BIST of tightly-coupled SRAM* on page 10-4 for a detailed description of the BIST support and the additional register 15 BIST registers.

Chapter 3
Memory Map

This chapter describes the ARM966E-S fixed memory map implementation.It contains the following sections:

3.1 About the ARM966E-S memory map

The ARM966E-S couples Instruction and Data SRAM memories of configurable size to the ARM9E-S core. This allows high-speed operation without incurring the performance and power penalties of accessing the system bus. A write buffer is used to minimize traffic on the AHB bus.

To provide simple control over the SRAM and write buffer, a fixed memory map is implemented within the ARM966E-S. Figure 3-1 illustrates this map.

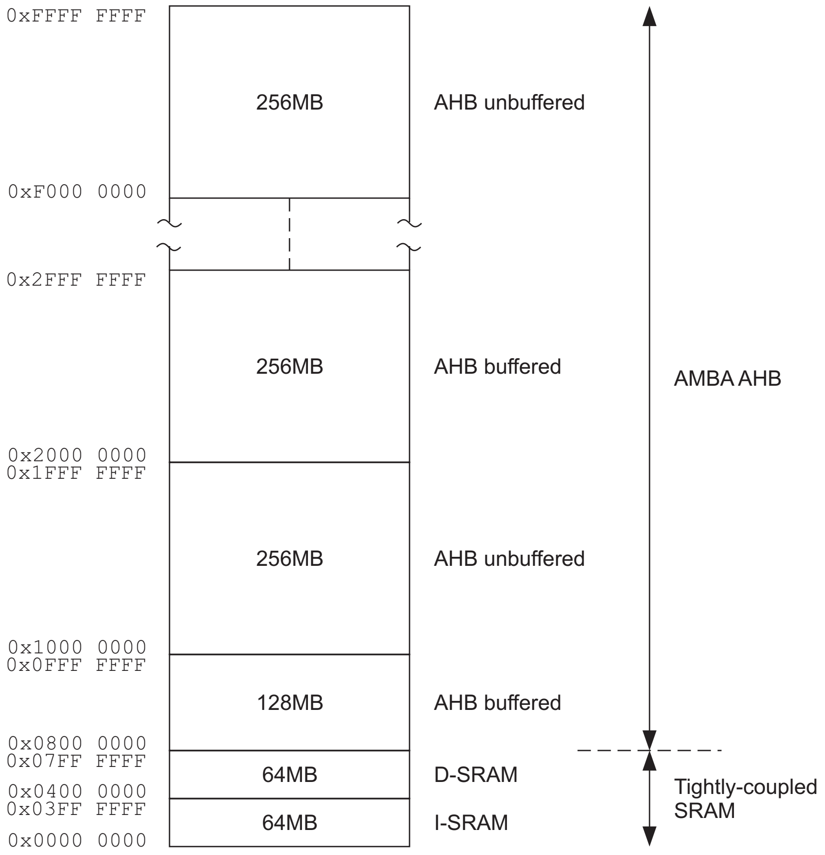


Figure 3-1 ARM966E-S memory map

3.2 Tightly-coupled SRAM address space

The tightly-coupled *Instruction SRAM* (I-SRAM) and *Data SRAM* (D-SRAM) are located at the bottom of the memory map. Each SRAM is allocated a 64MB address space, the bottom 64MB space mapping to I-SRAM and the next 64MB range mapping to D-SRAM.

In practice, each SRAM is likely to be much smaller than the 64MB allowable and the address decode is implemented so that each memory is aliased throughout its 64MB range. See Figure 3-2 for an example of a 16KB I-SRAM aliased through the 64MB address space.

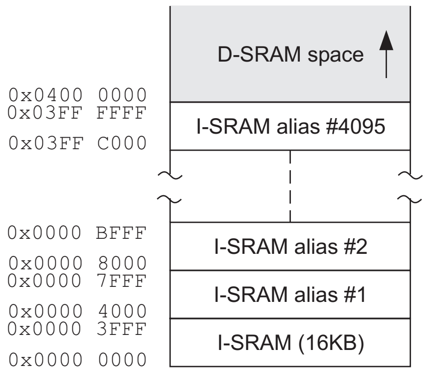


Figure 3-2 I-SRAM aliasing example

All accesses to addresses above the 128MB combined SRAM address space result in AMBA AHB transfers controlled by the *Bus Interface Unit* (BIU).

An instruction fetch from the ARM9E-S core to the D-SRAM address space goes to the AHB, regardless of whether the D-SRAM is enabled. A data interface access from the ARM9E-S core can access both the D-SRAM *and* the I-SRAM. The ability to additionally access the I-SRAM is required to allow the fetching of inline literals within code, for programming of the instruction I-SRAM, and for debugging purposes.

When an SRAM is disabled, all accesses to its address space go to the AHB. When enabled, the SRAM must be programmed before use. The tightly-coupled SRAMs can be enabled or disabled during reset depending on the value of the input pin **INITRAM**. Several boot options are available using **INITRAM** and the exception vectors location pin **VINITHI**. These are discussed in *Using INITRAM input pin* on page 4-4.

3.3 Bufferable write address space

The use of the ARM966E-S write buffer is controlled by both the CP15 control register and the fixed address map.

When the ARM966E-S comes out of reset, the write buffer is disabled by default. All data writes to the AHB are performed as unbuffered. The ARM9E-S is stalled until the BIU has performed the write on the AHB interface.

When the write buffer is enabled by writing to CP15 control register bit 3 (see *ARM966E-S CP15 registers* on page 2-4), the data address (**DA[31:0]**) from the ARM9E-S core controls whether the write buffer is used. If bit 28 of **DA** is set, the write is treated as un-buffered. If bit 28 is clear however, the write is treated as a buffered write and the BIU write buffer FIFO is used. Buffered writes allow the core to continue program execution while the write is performed on the AHB. If the write buffer is full the core is stalled until space becomes available in the FIFO. See *Write buffer operation* on page 6-3 for details of the BIU and write buffer behavior.

Note

Writes to tightly-coupled SRAM address space do not get sent to the AHB if the SRAM being accessed is enabled (the SRAMs do *not* write-through). If either SRAM is disabled and a write is performed to its address space, the write is performed as a buffered AHB write if the write buffer is enabled. If not, the write is un-buffered.

Chapter 4
Tightly-coupled SRAM

This chapter describes the tightly-coupled SRAM in the ARM966E-S. It contains the following sections:

For details of the ARM9E-S interface signals referenced in this section, refer to the *ARM9E-S Technical Reference Manual*.

4.1 ARM966E-S SRAM requirements

The ARM966E-S tightly-coupled SRAM is built from blocks of ASIC library compiled SRAM. The *Instruction SRAM* (I-SRAM) and *Data SRAM* (D-SRAM) can each be any size from 0 bytes to 64MB, although to ease implementation the size must be an integer power of two. The I-SRAM and D-SRAM can have different sizes.

To allow the I-SRAM to be initialized and for access to literal tables during execution, the data interface of the ARM9E-S core must be able to access the I-SRAM. This requires that the instruction and data addresses are multiplexed before entering the I-SRAM and the instruction data is routed both to the instruction and data interfaces of the core. See Figure 1-1 on page 1-3 for details of this data and address multiplexing.

ARM966E-S supports the use of synchronous SRAM. The SRAM control has been implemented in a way that expects the compiled SRAM memory cells to return read data to ARM9E-S in a single-cycle. This requirement applies to both the I-SRAM and D-SRAMs. See Figure 4-1 for a typical read cycle (I-SRAM shown).

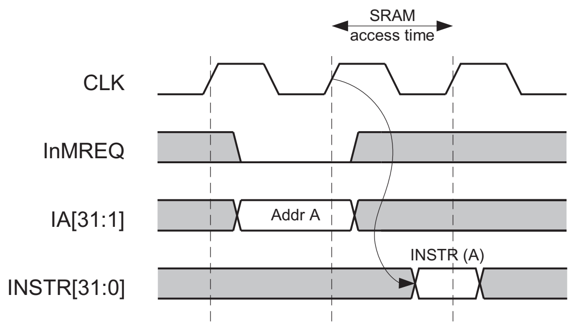


Figure 4-1 SRAM read cycle

During normal program execution, the instruction and data interfaces of the ARM9E-S can be active simultaneously. In this case both SRAMs can be simultaneously accessed allowing the core to continue execution without any stall cycles. There are cases however, where stall cycles are encountered when accessing the SRAM.

4.2 SRAM stall cycles

Stall cycles can occur in both the I-SRAM and D-SRAMs. The two RAMs share a common stall mechanism. Because memory write in an ARM9E-S system is a two-cycle operation, CPU memory access during the second cycle must be stalled. The I-SRAM, has additional stall cycles as it can be accessed by both the instruction and data interfaces of the ARM9E-S. In order to maximize memory interface frequency performance, data read requests to the I-SRAM are pipelined by one clock cycle. Any stall requirement is detected by the SRAM control and factored into its response to the ARM966E-S system controller. The ARM9E-S **SYSCLKEN** input is then de-asserted until the SRAM has performed the access.

Table 4-1 shows the number of stall cycles added for different stall mechanisms for the I-SRAM.

Table 4-1 I-SRAM stall cycles

Number of added cycles	Stall mechanism
1	Data read.
1	Data read followed by write.
1	Data write followed by instruction fetch or data read.
1	Data read followed by instruction fetch.
1	Simultaneous instruction fetch and data read.
2	Simultaneous instruction fetch and data write.
2	Data read or write followed by simultaneous instruction fetch and data read or write.

Note

Data reads from the I-SRAM incur a single-cycle stall for each read instruction and not each separate RAM read. LDM and LDR operations both incur a single stall cycle.

The D-SRAM stall mechanism is write followed by read, and the number of stall cycles added is one.

For a detailed description of SRAM stall cycles, see Appendix C *SRAM Stall Cycles*.

4.3 Enabling the SRAM

There are two mechanisms for controlling the enable of the SRAM:

- both I-SRAM and D-SRAM can be enabled or disabled during reset by the input pin **INITRAM**
- the I-SRAM and D-SRAM can be individually enabled or disabled through software `MCR` instructions to CP15.

4.3.1 Using INITRAM input pin

Two resets are described in the following sections:

- *Reset with INITRAM LOW*
- *Reset with INITRAM HIGH*.

Reset with INITRAM LOW

The **INITRAM** pin is provided to allow the ARM966E-S to boot with both SRAM blocks either enabled or disabled. If **INITRAM** is held LOW during reset, the ARM966E-S comes out of reset with both SRAMs disabled. All accesses to I-SRAM and D-SRAM space go to the AHB. The SRAM can then be individually or jointly enabled by writing to the CP15 control register (register 1).

Reset with INITRAM HIGH

If however, **INITRAM** is held HIGH during reset, both SRAM blocks are enabled when the ARM966E-S comes out of reset. This is normally used for a warm reset where the SRAM has already been programmed before the application of **nRESET** to the ARM966E-S. In this case, the SRAM contents are preserved and the ARM966E-S can run directly from the tightly-coupled SRAM following reset. Either one or both SRAM can be further disabled or enabled by writing to the CP15 control register.

Note

If **INITRAM** is held HIGH during a cold reset (the SRAM has not previously been initialized), the **VINITHI** pin must be set HIGH to ensure that the ARM966E-S boots from `0xFFFF 0000`, that is in AHB address space and is substantially outside the SRAM address space. This is necessary because if **VINITHI** is LOW, the ARM966E-S attempts to boot from `0x0000 0000`, and this selects the uninitialized I-SRAM.

4.3.2 Using CP15 control register

When out of reset, the behavior of the tightly-coupled SRAM is controlled by the state of CP15 control register.

Enabling the I-SRAM

You can enable the I-SRAM by setting bit 12 of the CP15 control register. This register must be accessed in a read-modify-write fashion, to preserve the contents of the bits not being modified. See *ARM966E-S CP15 registers* on page 2-4 for details of how to read and write the CP15 control register. When the I-SRAM has been enabled, all future ARM9E-S instruction fetches and data accesses to the I-SRAM address space as shown in Figure 3-1 on page 3-2 causes the I-SRAM to be accessed.

Enabling the I-SRAM greatly increases the performance of the ARM966E-S as the majority of accesses to it can be performed with no stall cycles, whereas accessing the AHB might cause several stall cycles for *each* access.

Caution

Care must be taken to ensure that the I-SRAM is appropriately initialized before it is enabled and used to supply instructions to the ARM9E-S core. If the core tries to execute instructions from uninitialized I-SRAM, the behavior is unpredictable.

Disabling the I-SRAM

You can disable the I-SRAM by clearing bit 12 of the CP15 control register. When the I-SRAM has been disabled, all further ARM9E-S instruction fetches access the AHB. If the core performs a data access to the I-SRAM address space as shown in Figure 3-1 on page 3-2, an AHB access is performed.

Note

The contents of the SRAM are preserved when it is disabled. If it is re-enabled, accesses to previously initialized SRAM locations returns the preserved data.

Enabling the D-SRAM

You can enable the D-SRAM by setting bit 2 of the CP15 control register. See *ARM966E-S CP15 registers* on page 2-4 for details of how to read and write this register. When the D-SRAM has been enabled, all future read and write accesses to the D-SRAM address space, as shown in Figure 3-1 on page 3-2, cause the D-SRAM to be accessed.

Disabling the D-SRAM

You can disable the D-SRAM by clearing bit 2 of the CP15 control register. When the D-SRAM is disabled, all further reads and writes to the D-SRAM address space, as shown in Figure 3-1 on page 3-2, access the AHB. Read and write accesses to I-SRAM address space uses the I-SRAM or accesses the AHB depending on if it is enabled.

4.4 ARM966E-S SRAM wrapper

The ARM966E-S allows you to have control over the size of the I-SRAM and D-SRAM (up to a maximum of 64MBytes each). It is not possible to have a single generic interface between the ARM966E-S and the SRAM, due to the large number of differing compiled SRAM that can be integrated into an ARM966E-S system, potentially each with a unique interface.

To ease the task of integrating differing SRAM into the ARM966E-S, an interface *wrapper* block has been developed to ensure that when wrapped, the SRAM provides a standard interface to the ARM966E-S SRAM control. ARM provides an example SRAM wrapper containing three example interfaces, see *Example SRAM interfaces* on page 4-8. You must study these examples and decide which is most appropriate for the type of SRAM available. A script is provided which automates any required changes.

The RAM interface RTL allows you to trade off speed against power performance so that you can tailor the ARM966E-S to suit a particular requirement.

There are five SRAM modules instantiated at the top-level of the ARM966E-S. Figure 4-2 shows the structure of these three modules.

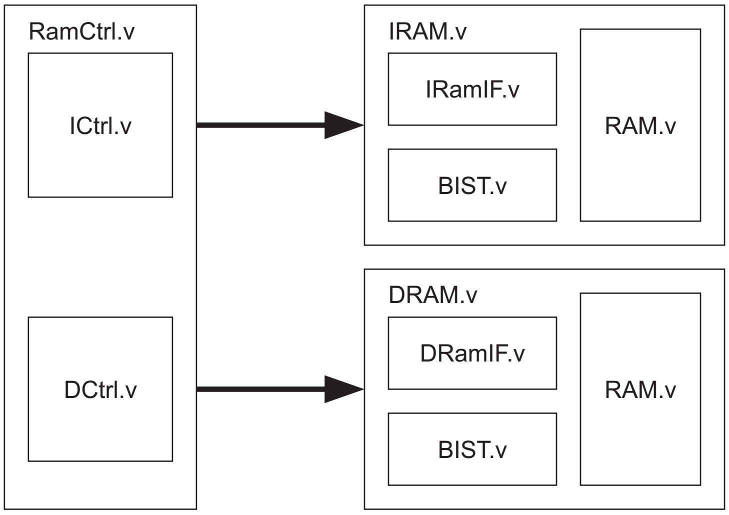


Figure 4-2 ARM966E-S SRAM hierarchy

`RamCtrl.v` contains the RAM control logic that is partner-independent. This logic is fixed.

`IRamIF.v` and `DRamIF.v` generate the SRAM specific ChipSelect, WriteEnable, and ByteWrite signals. Your own library RAMs are instantiated inside `InstrRAM.v` and `DataRAM.v` .

4.4.1 Example SRAM interfaces

The example wrapper supplied by ARM contains three RAM interface examples. All of the interface modifications are done in the `IRamIF.v` and the `DRamIF.v` blocks for the I-SRAM and D-SRAM respectively. The example SRAM interfaces are:

- *ONESEGX32*
- *FOURSEGX32* on page 4-9
- *FOURSEGX8* on page 4-10.

Note

The examples shown here are for 32KByte I-SRAM (8K words x 4bytes). The interface for D-SRAM is identical.

ONESEGX32

Figure 4-3 shows the simplest interface I-SRAM. To use this, the SRAM must consist of a single word-wide RAM that has byte-write control.

Only single ChipSelect and WriteEnable signals are required.

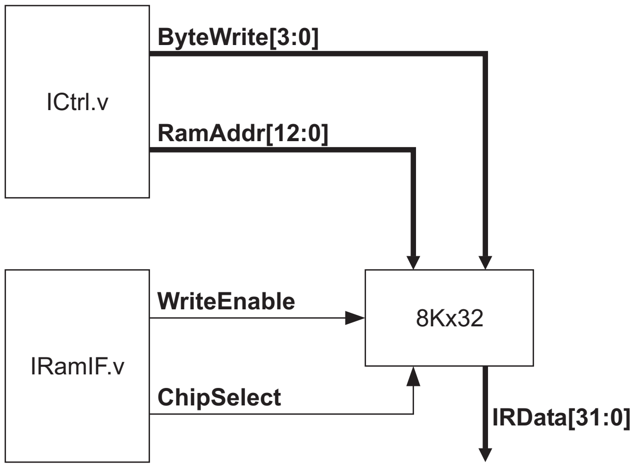


Figure 4-3 ONESEGX32 interface

FOURSEGX32

You can use the example shown in Figure 4-4 when it is not possible to construct the SRAM from a single physical block due to either layout constraints or generator constraints, or because a single SRAM segment does not meet timing constraints.

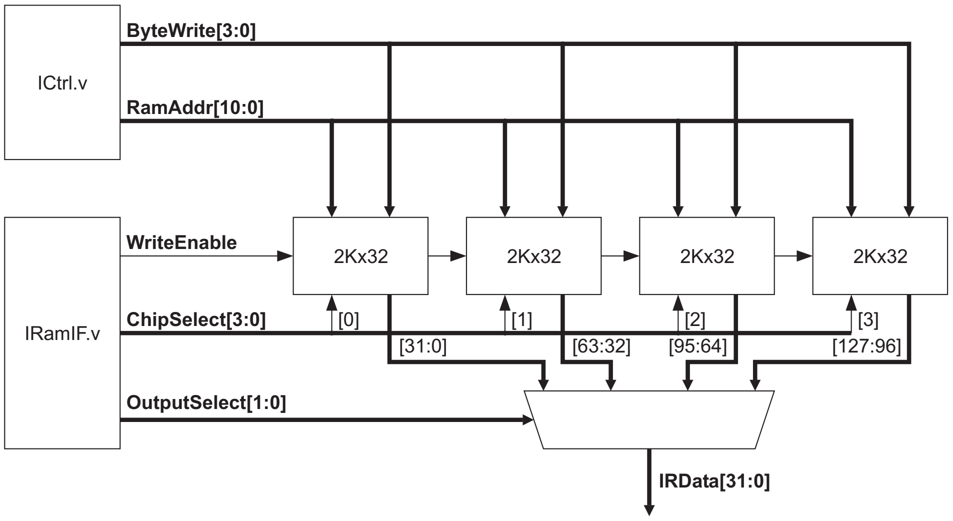


Figure 4-4 FOURSEGX32 interface

Separate chip select signals are required for each SRAM block.

——— **Note** ———

- The generation of separate chip select signals for each SRAM block ensures good power performance, because only the segment being accessed is enabled.
- The SRAM address is 11 bits in this example (compared with the 13 bit address in *ONESEGX32* on page 4-8). **RamAddr[12:11]** are used to generate separate chip selects for each segment.

If it is not possible to have separate chip select signals for each block of RAM, for example if the RAM is asynchronous, then separate write enable signals are required for each segment. The use of asynchronous RAMs is not recommended due to the increased power consumption of this solution.

——— **Note** ———

The wrapper RTL does not support asynchronous RAMs.

FOURSEGX8

Figure 4-5 on page 4-11 shows that the SRAM needs to be split into four-byte wide segments where an SRAM does not support byte-writes. In order to give an example of the most complex interface possible, Figure 4-5 on page 4-11 assumes that each byte-wide SRAM needs to be split into four blocks (see word-wide SRAM in *FOURSEGX32* on page 4-9).

In *FOURSEGX32* on page 4-9 the SRAM Address is 11 bits. Bits [12:11] of the address are used to decode which of the four word-wide RAMs is selected.

In Figure 4-5 on page 4-11 **ByteWrite[3:0]** is used (inside IRamIF.v) to decode each word-wide chip select into four separate chip select signals, one for each byte of the word.

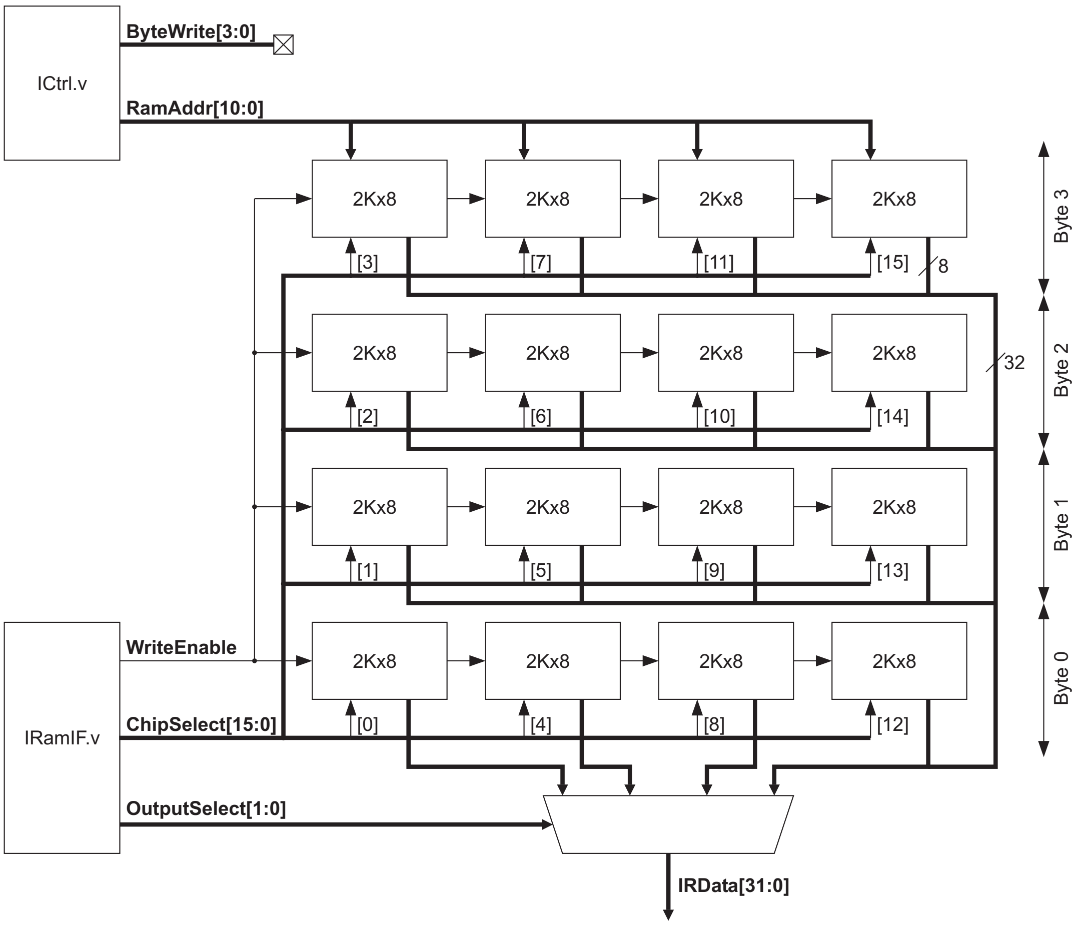


Figure 4-5 FOURSEGX8 interface

Chapter 5
Direct Memory Access (DMA)

This chapter describes the optional DMA interface in the ARM966E-S. It contains the following sections:

5.1 About the DMA interface

A DMA port is provided on the ARM966E-S. You can connect this port to the D-SRAM in the ARM966E-S. This allows direct access to the D-SRAM from outside the ARM966E-S boundary. If this feature is not required the DMA port is tied off in the RTL and made redundant. You have the option of interfacing the DMA port to a dual-port RAM or a single-port RAM, providing the ability to choose the solution that best meets area, performance, and software requirements.

The DMA port enables direct access to the data RAM, bypassing the CPU core. The ARM966E-S provides the control logic to access the RAM. The implementation of a DMA controller is application-specific and so any DMA control logic is instantiated outside of the ARM966E-S macrocell boundary.

Figure 3-1 on page 3-2 shows DMA addresses directly map to the RAM location in the data RAM 64MB address space. The RAM controller in the ARM966E-S uses bits [31:26] of the CPU data address to decode Data RAM address space access. Bits [31:26], however, are not required to be driven by the DMA controller because DMA access is always to this address space. RAM aliasing occurs for DMA access in the same way as aliasing occurs for CPU accesses. See *Tightly-coupled SRAM address space* on page 3-3 for more information.

——— **Note** ———

The decision to connect to the DMA port, and to a particular type of RAM, is made prior to synthesis.

———————————

5.1.1 Single-port RAM DMA solution

DMA accesses to a single-port RAM must be done through the same interface that the CPU uses to access the RAM. CPU accesses to the RAM must be prevented while DMA transfers are taking place. This is done by stalling the core for the duration of the DMA transfer. The DMA controller requests access to the D-RAM by asserting **DMAWait**. When the CPU has been stalled on the next instruction boundary, the ARM966E-S asserts **DMAReady** to notify to the DMA controller that it now has ownership of the RAM and can proceed with the transfer.

The single-port RAM DMA solution must be used where the die area of a dual-port RAM is not acceptable and the performance impact of stalling the core during DMA transfers is acceptable.

Figure 5-1 on page 5-3 shows how the ARM966E-S DMA port interfaces to a single-port RAM.

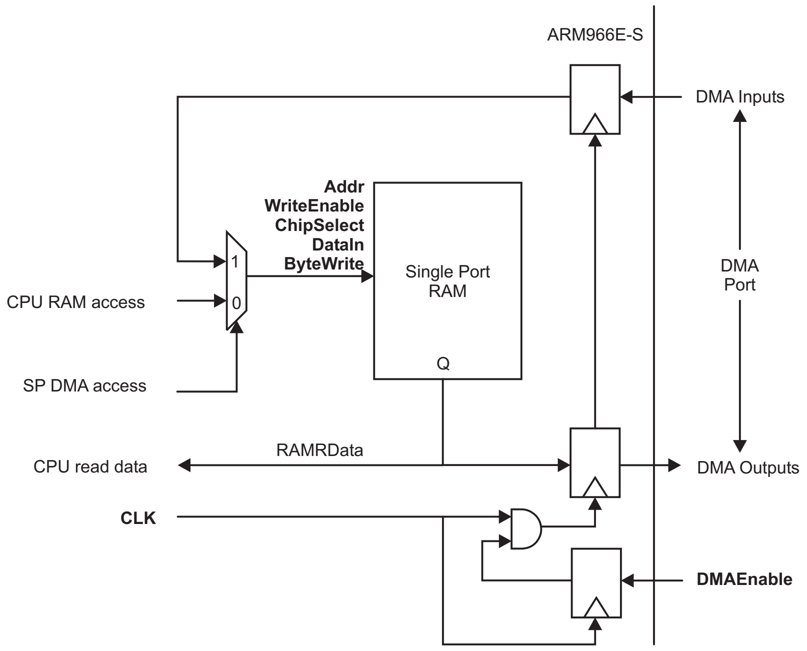


Figure 5-1 Single-port RAM DMA interface

5.1.2 Dual-port RAM DMA solution

If the data RAM is implemented using dual-port RAM, the second port is used exclusively for DMA. The CPU and DMA can access the data RAM simultaneously so the core does not need to be stalled. A dual-port RAM DMA solution provides higher performance than the single-port solution, but uses a larger die area. The programmer must ensure that DMA and CPU do not access the same memory locations simultaneously. The behavior of accessing the same memory locations simultaneously is either undefined or illegal. Simultaneous access behavior is summarized in Table 5-1.

Table 5-1 Simultaneous access behavior

Core access	DMA access	behavior
Read	Read	Valid

Table 5-1 Simultaneous access behavior (continued)

Core access	DMA access	behavior
Read	Write	Undefined
Write	Read	Undefined
Write	Write	Illegal

Figure 5-2 shows how the ARM966E-S DMA port interfaces to a dual-port RAM. For modelling purposes, the dual-port DMA solution also supports the single-port access route. Single-port access reduces performance in the dual-port solution and is unlikely to be used, so to prevent the core from being stalled, **DMAWait** must be tied LOW.

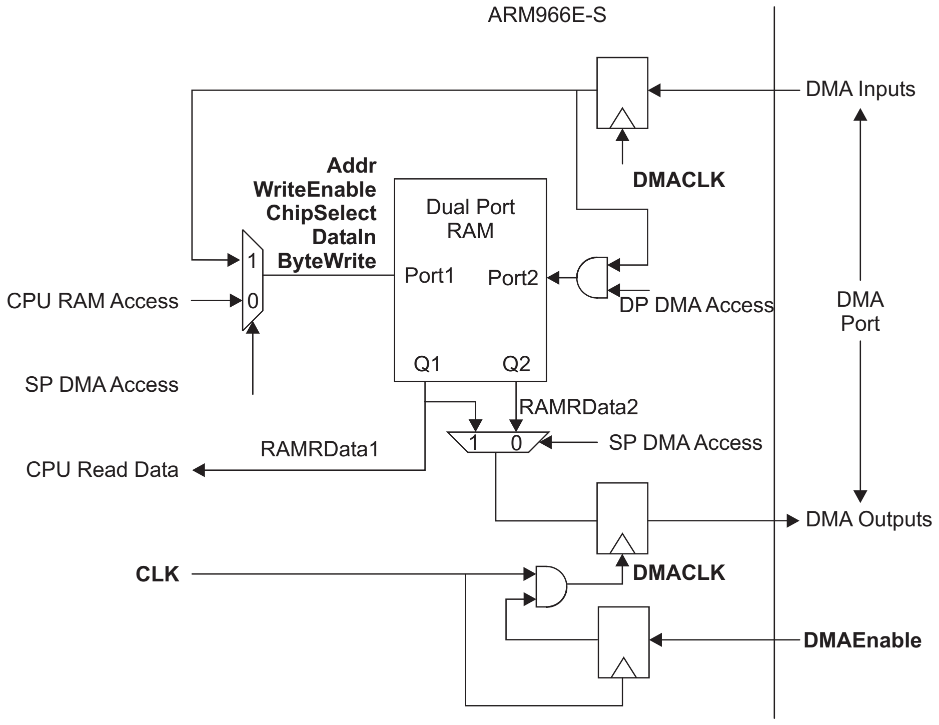


Figure 5-2 Dual-port RAM DMA interface

5.2 Timing interface

To ease the system integration task and to provide RAM independent timings, the ARM966E-S registers all DMA inputs and outputs. This section details the behavior of the ARM966E-S for DMA read and writes to single and dual-port RAMs.

Note

The dual-port RAM DMA solution also supports the single-port operation and so the single-port diagrams are also applicable to dual-port RAMs.

5.2.1 Single-port RAM reads

Figure 5-3 shows DMA read operation from a single-port RAM.

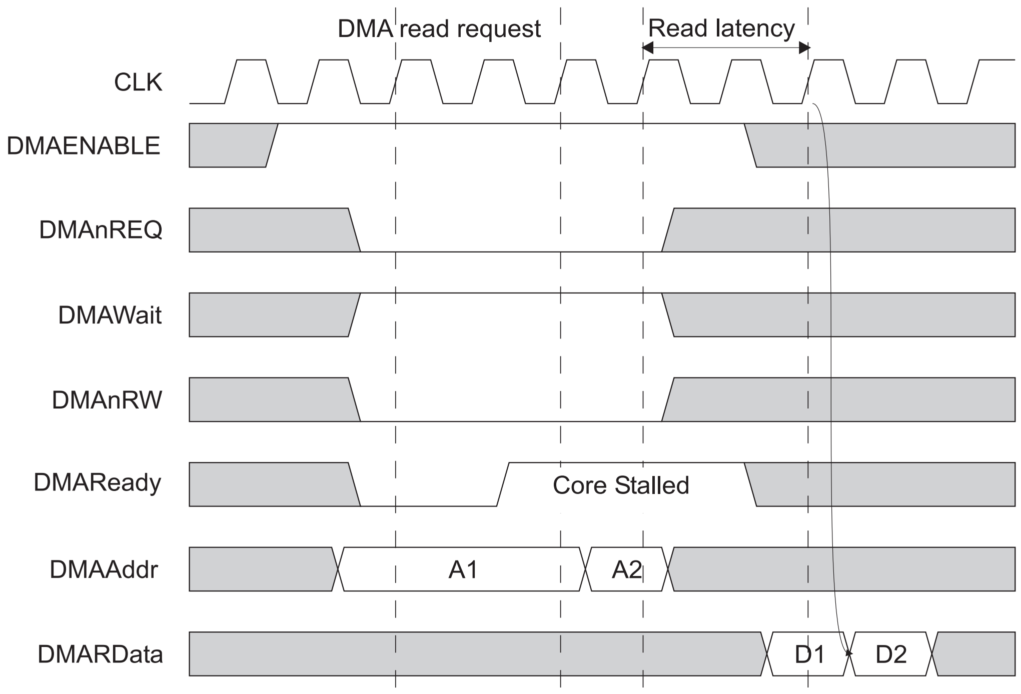


Figure 5-3 Single-port RAM DMA reads

The DMA controller makes a read request by taking **DMAnREQ** LOW and asserting **DMAWait**. **DMAReady** is asserted by the ARM966E-S when the core has been stalled on the next instruction boundary and informs the DMA controller that it can proceed with its transfer.

Figure 5-3 on page 5-5 also shows the minimum latency between **DMAWait** being registered HIGH and **DMAReady** being asserted is two clock cycles, when **DMAWait** is registered on an instruction boundary. The maximum latency occurs when **DMAWait** is asserted on the first cycle of an `LDM` or unbuffered `STM` AHB access. The latency in this case can be calculated from the information in *Introduction to instruction cycle timings* on page 11-2.

The DMA controller can increment the read address on the next rising clock edge after **DMAReady** is asserted. Read data is driven on **DMARData** in the third cycle after the read address is sampled by the ARM966E-S (one cycle to register the address, one cycle for the RAM read and one cycle for registering the RAM read data). The first read address, **DMAAddr**, is registered by the ARM966E-S on the next rising clock edge after **DMAReady** is asserted.

The DMA controller has *ownership* of the RAM from **DMAReady** being asserted until it takes **DMAWait** LOW. When **DMAWait** has been taken LOW, the DMA controller loses ownership of the RAM. **DMAWait** *must* be taken LOW at the end of a DMA access to allow CPU flow to continue.

DMAENABLE must be asserted one cycle prior to a request being made and can be deasserted one cycle prior to the last read data being returned.

Note

If **DMAWait** is not asserted, the ARM966E-S does not respond to single-port RAM DMA requests.

5.2.2 Single-port RAM writes

Figure 5-4 on page 5-7 shows DMA write operation to a single-port RAM.

The DMA controller requests write access to the RAM in the same way as single-port RAM reads except that **DMAnRW** is HIGH. Because data writes are single cycle operations, data to be written must be present in the same cycle as the address. The first write address, **DMAAddr**, is registered by the ARM966E-S on the next rising clock edge after **DMAReady** is asserted. The write to the RAM happens in the following cycle, due to the single cycle latency of the input registers. The first write address, **DMAAddr**, and data, **DMAWData**, is registered by the ARM966E-S on the next rising clock edge after **DMAReady** is asserted.

The behavior of **DMAWait** is as for single-port RAM reads.

DMAENABLE must be asserted one cycle prior to a request being made and can be deasserted when **DMAnREQ** is taken HIGH after the last request.

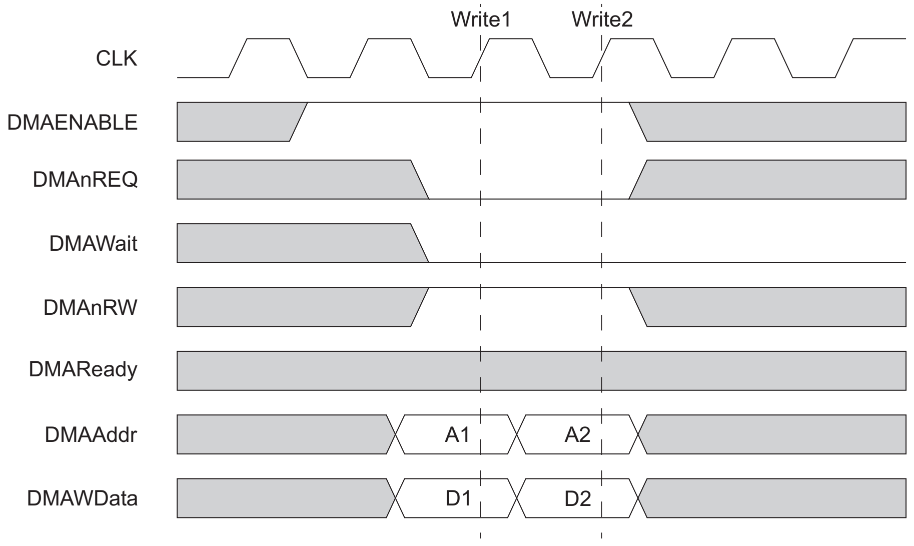


Figure 5-4 Single-port RAM DMA writes

5.2.3 Dual-port RAM reads

Figure 5-5 on page 5-8 shows DMA read operations to a dual-port RAM.

A read request is initiated by taking **DMAnREQ** and **DMAnRW** both LOW. The address, **DMAAddr**, must be valid in the same cycle. The read data, **DMARData**, is returned in the third cycle after the request is registered by the ARM966E-S (one cycle to register the request, one cycle to read the RAM, and one cycle to register the output data).

Note

Because the ARM966E-S core does not need to be stalled for dual-port DMA accesses, the DMA controller can access the data RAM continuously. **DMAWait** must be tied LOW otherwise the DMA access is by the first port of the RAM and the interface behaves as described in *Single-port RAM reads* on page 5-5.

DMAReady is redundant for dual-port RAM accesses and does not need to be sampled by the DMA controller.

DMAENABLE must be asserted one cycle prior to a request being made and can be deasserted one cycle prior to the last read data being returned

.

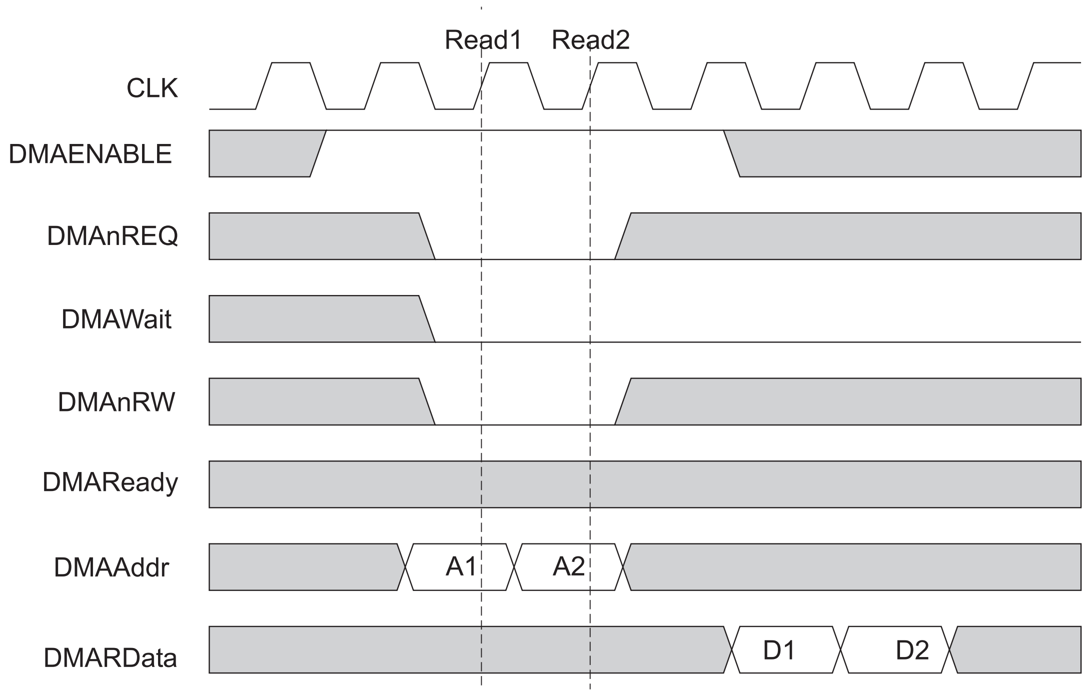


Figure 5-5 Dual-port DMA reads

5.2.4 Dual-port RAM writes

Figure 5-6 on page 5-9 shows dual-port write operations to a dual-port RAM.

A write request is initiated by taking **DMAnREQ** LOW and **DMAnRW** HIGH. The address, **DMAAddr**, and write data, **DMAWData**, must be valid in the same cycle. The write to the RAM happens in the following cycle, due to the one cycle latency of the input registers.

Note

Because the ARM966E-S core does not need to be stalled for dual-port DMA accesses, the DMA controller can access the data RAM continuously. **DMAWait** must be tied LOW otherwise the DMA access is by the first port of the RAM and the interface behaves as described in *Single-port RAM writes* on page 5-6.

DMAReady is redundant for dual-port RAM accesses and does not need to be sampled by the DMA controller.

DMAENABLE must be asserted one cycle prior to a request being made and can be deasserted when **DMAnREQ** is taken HIGH after the last request.

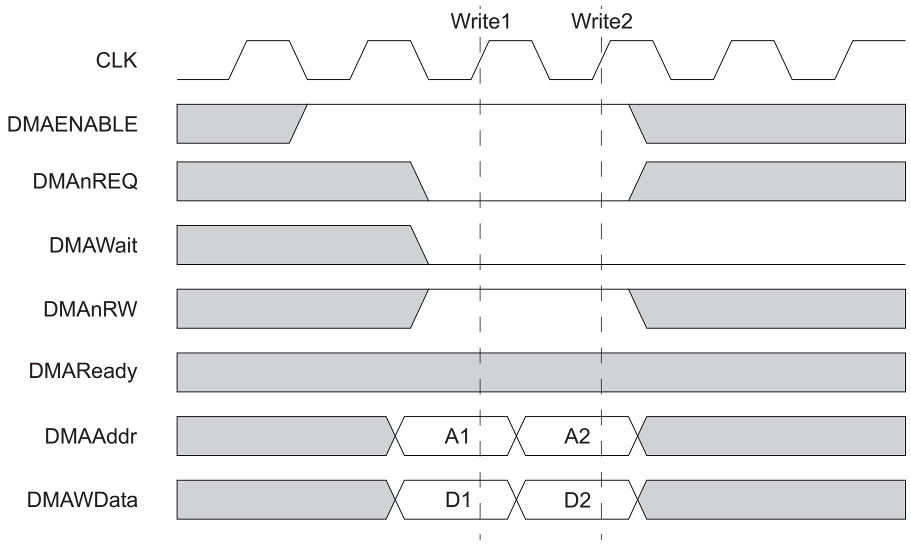


Figure 5-6 Dual-port RAM DMA writes

5.2.5 Mixed read and writes

Figure 5-7 on page 5-10 shows:

- an example of intermingled DMA read and write operations
- that reads and writes can be performed back-to-back.

The behavior is the same for both single and dual-port RAMs. Depending on whether the RAM was single or dual-port, the behavior of **DMAENABLE**, **DMAWait**, and **DMAReady** is described in sections *Single-port RAM reads* on page 5-5 to *Dual-port RAM writes* on page 5-8.

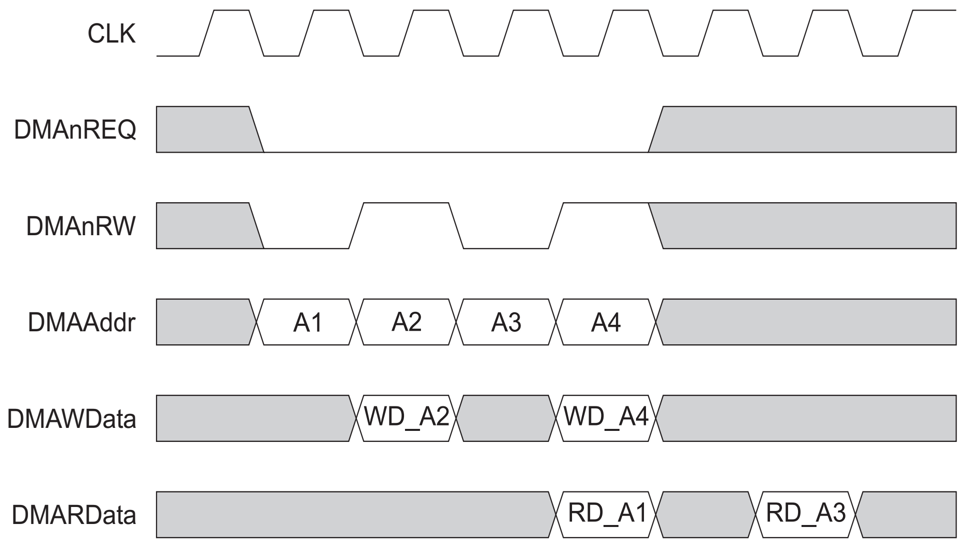


Figure 5-7 Mixed DMA read and write

5.3 DMAENABLE setup and hold cycles

Table 5-2 shows the minimum number of setup cycles and hold cycles for **DMAENABLE** with respect to **DMAnREQ** for both single and dual-port RAMs.

Table 5-2 DMAENABLE setup and hold cycles with respect to DMAnREQ

Operation	Setup	Hold
Dual-port RAM DMA read	1	1
Dual-port RAM DMA write	1	0
Single-port RAM DMA read	1	1
Single-port RAM DMA write	1	0

To reduce power consumption, **DMAENABLE** must be taken LOW when DMA accesses are not taking place or if DMA is not implemented.

5.4 Summary of signal behavior

Table 5-3 summarizes the behavior of **DMAENABLE**, **DMAWait**, **DMAnREQ**, and **DMAReady** for single and dual-port RAM solutions in addition to the required connections of these signals if no DMA is implemented.

Table 5-3 DMA signal behavior

Signal	Dual-port RAM DMA	Single-port RAM DMA	No DMA
DMAENABLE (Input)	See Table 5-2 on page 5-11	See Table 5-2 on page 5-11	Must be tied LOW external to the ARM966E-S.
DMAWait (Input)	The DMA controller does not need to stall the ARM966E-S and so this signal must be tied LOW external to the ARM966E-S.	The DMA controller must drive this signal HIGH whenever it requires access to the data RAM.	Must be tied LOW external to the ARM966E-S.
DMAnREQ (Input)	Must be driven LOW by the DMA controller whenever it requires access to the data RAM	Must be driven LOW by the DMA controller whenever it requires access to the data RAM.	Must be tied HIGH external to the ARM966E-S.
DMAReady (Output)	Can be ignored by the DMA controller because it always has access to the RAM.	Must be registered by the DMA controller so that it knows when the ARM966E-S has been stalled.	Do not care.

Chapter 6
Bus Interface Unit

This chapter describes the ARM966E-S *Bus Interface Unit* (BIU) and write buffer. It contains the following sections:

6.1 About the BIU and write buffer

The ARM966E-S supports an Advanced Microprocessor Bus Architecture (AMBA) Advanced High-performance Bus (AHB) interface. The AHB is a new generation of AMBA interface that addresses the requirements of high-performance synthesizable designs, including:

- single clock edge operation (rising edge)
- unidirectional (nontristate) buses
- burst transfers
- split transactions
- single-cycle bus master handover.

See the *AMBA Rev 2.0 AHB specification* for full details of this bus architecture.

The ARM966E-S BIU implements a fully-compliant AHB bus master interface and incorporates a write buffer to increase system performance. The BIU is the link between the ARM9E-S core with its tightly-coupled SRAM and the external AHB memory. The AHB memory must be accessed to initialize the tightly-coupled SRAM. The AHB memory must also be accessed to access code and data that are not assigned to the tightly-coupled SRAM address space (or if the SRAM is disabled).

When an external AHB access is performed, the BIU and the system controller handshake to ensure that the ARM9E-S core is stalled. If the write buffer is used, it might be possible to allow the core to continue program execution. The BIU is responsible for controlling the write buffer and related stall behavior (see *Write buffer operation* on page 6-3).

6.2 Write buffer operation

The ARM966E-S implements a 12-entry write buffer, where the entries can be address or data depending on the nature of the writes being executed by the ARM9E-S core. The write buffer helps to decouple the core from the wait cycles incurred when accessing the AHB. If a write is sent to the write buffer, the core is able to continue program execution without having to wait for the write to complete on the AHB. More writes can be committed to the write buffer without stalling if spare entries are available.

If the write buffer becomes full, the ARM9E-S core must be stalled until an AHB access occurs and some write data is written, therefore freeing up the necessary FIFO entries.

Alternatively, if the core performs a read from or unbuffered write to the AHB address space, the core is stalled until all write buffer entries have been written (the write buffer is *drained*). The write buffer is drained to ensure data coherency, in that the core might try to read from a location that it has recently modified and is still in the write buffer awaiting AHB access.

6.2.1 Committing write data to the write buffer

The write buffer is used when the following conditions are met:

- the write buffer is enabled
- the address is in a bufferable region
- the address is in AHB external memory, or the address selects a tightly-coupled SRAM that is disabled.

For details on write buffer enable and the ARM966E-S fixed address map, see

- *Register 1, Control register* on page 2-5
- *About the ARM966E-S memory map* on page 3-2.

When a write is performed by the core and conforms to the above conditions, the address for the write is put into the first available entry of the write buffer FIFO. The next available entry is used for the write data. If the write is a store multiple (`STM`), subsequent entries are used for each word of the `STM`. It is therefore possible for the FIFO to contain 11 words of a `STM` where the first entry contains the address and the remaining 11 entries contain the write data.

Alternatively, if several shorter bufferable `STM` or single writes (`STR`) instructions are performed, one address entry is used for each write instruction. The worst case is that only six data words fill the FIFO caused by six `STR` writes. In this case the FIFO holds six address entries and six data entries.

Figure 6-1 on page 6-4 shows an example where the BIU FIFO is being filled by the following write instructions:

```
STMIA r13!,{r2-r4} ; store three registers to the stack
STRB      r5,[r6]  ; store byte
STMIA r13!,{r3-r4} ; store two registers to the stack
STR     r7,[r2]    ; single store
```

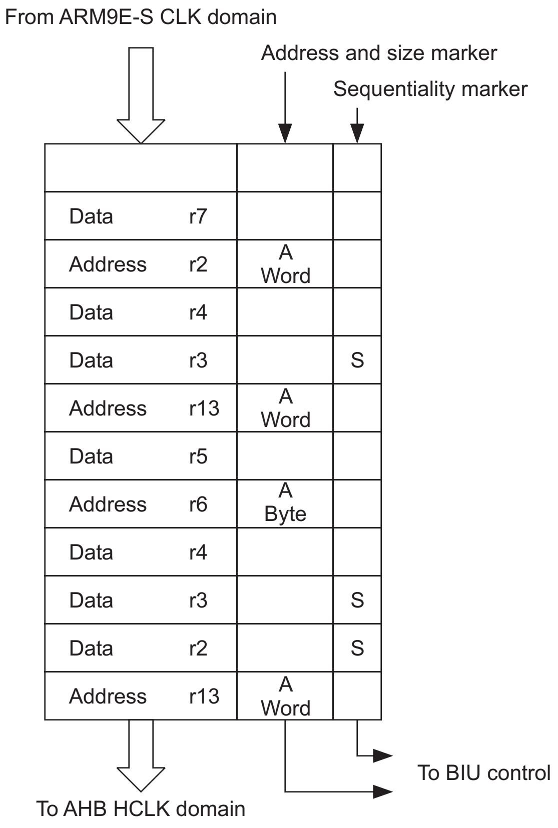


Figure 6-1 Write buffer FIFO content example

6.2.2 Draining write data from the write buffer

The write buffer can drain *naturally* where AHB writes occur whenever data is committed to the FIFO. The core is only stalled, if the write buffer overflows. However, there are times when a complete drain of the write buffer is *enforced*.

Natural write buffer drain

When a write is being committed to the write buffer FIFO, a signal is sent to the BIU to initiate an AHB write. The BIU then pops the address for the write from the FIFO followed by the data and starts an AHB transfer (assuming the ARM966E-S is the granted bus master). This process might take several cycles because the slave being accessed for the write might have a multi-wait cycle response. Additionally, the AHB can be run at a lower rate than the ARM966E-S system introducing extra delay to the buffered write process. This can lead to the core trying to commit data at a higher rate than the FIFO can be drained, resulting in the FIFO becoming full. The ARM9E-S core is stalled until an entry becomes available.

When an address is placed in the write buffer, a marker is also stored to indicate if the size of the write is, byte, halfword or word. If a `STM` is performed, a sequentiality marker is stored with the data, to indicate to the BIU that the address incrementer must be used to produce the AHB address for the second and following writes of the `STM`. This mechanism allows only one FIFO entry to be used for the address, leaving more room for data (see Figure 6-1 on page 6-4).

If a `STM` crosses a 1KB boundary, the AHB specification requires that the first access in the new 1KB region is a nonsequential access. This allows the BIU to have a small 1KB incrementer, because the ARM9E-S data address can be resampled during the nonsequential cycle. For this reason, the write buffer must also break up accesses that cross a 1KB region, by forcing the sequentiality marker LOW for the preceding data location and committing an extra address entry at the start of the new region.

Note

Because the ARM9E-S core is free to continue program execution following a buffered write, without having to wait for the write to complete on the AHB, external Data Aborts can *not* be returned by buffered writes.

Enforced write buffer drain

There are two situations where the core is stalled and the write buffer is forced to drain completely before program execution can continue:

- an instruction fetch, data load, or unbuffered write to the AHB is being requested
- a drain write buffer instruction is being executed.

AHB read access requested

To ensure data coherency, you must prevent the core from reading data from a location that has recently been modified (by the core or an external coprocessor `STC` instruction) and is still in the write buffer awaiting AHB access. If the AHB read access is allowed to occur before the write buffer is drained, the old version of data at that location is fetched causing a data coherency failure.

For this reason, whenever an AHB read is requested, as an ARM9E-S instruction fetch or a data load or load multiple, the core must be stalled until the write buffer is drained. No special logic is used to force a write buffer drain as this process is occurring whenever data is present within the buffer. However, special logic is required to stall the core until the last buffered write has *completed* on the AHB.

Drain write buffer instruction

You can use an `MCR` instruction to CP15 register 7 to force the core to be stalled until the write buffer is empty and the final write is completed on the AHB. This instruction is described in *Register 7, Core control* on page 2-7. This instruction is useful when the software requires that a write is completed before program execution continues.

6.2.3 Enabling the write buffer

The write buffer can be enabled by setting bit 3 of the CP15 control register. When this bit is set, all writes to bufferable address locations use the write buffer. If a slave peripheral in a bufferable region returns an AHB Data Abort, the abort is ignored when the write buffer is enabled.

Note

For debugging purposes, you can disable the write buffer to allow AHB Data Aborts to be returned from bufferable regions.

6.2.4 Disabling the write buffer

When data is committed to the write buffer it is always written to the AHB. If the write buffer is disabled by clearing bit 3 of the CP15 control register, any existing write data in the write buffer is completed. Additionally, if the core is sent to sleep by the `wait for interrupt` command, any writes in the write buffer FIFO are also completed.

If the programmer requires no more buffered writes to occur following write buffer disable or a `wait for interrupt` instruction, the write buffer must first be drained with a `drain write buffer` command.

6.3 AHB bus master interface

The ARM966E-S implements a fully-compliant AHB bus master interface and is defined in the *AMBA Rev 2.0 Specification*. You must refer to this document for a detailed description of the AHB protocol.

6.3.1 Overview of AHB

The AHB architecture is based on separate cycles for address and data (rather than the phase of the clock in the ASB architecture). The address and control for an access are broadcast from the rising edge of **HCLK** in the cycle *before* the data is expected to be read or written. During this data cycle, the address and control for the *next* cycle are driven out. This leads to a fully pipelined address architecture.

When an access is in its data cycle, a slave can wait the access by driving the **HREADY** response LOW. This has the effect of stretching the current data cycle and therefore the pipelined address and control for the next access is also stretched. This creates a system where all AHB masters and slaves sample **HREADY** on the rising edge of the **HCLK** to determine whether an access has completed and a new address can be sampled or driven out.

6.3.2 ARM966E-S transfer descriptions

The ARM966E-S BIU performs a subset of the possible AHB bus transfers available. This section describes the transfers that can be performed and some back-to-back transfer cases:

- *Burst transfers* on page 6-8
- *Bus request* on page 6-8
- *Sequential instruction fetch* on page 6-8
- *Back-to-back LDR or STR accesses* on page 6-9
- *Simultaneous instruction and data request* on page 6-10
- *STM timing* on page 6-11
- *LDM timing* on page 6-11
- *STM followed by instruction fetch* on page 6-12
- *LDM followed by instruction fetch* on page 6-13
- *STM crossing a 1KB boundary* on page 6-14
- *LDM crossing a 1KB boundary* on page 6-15
- *SWP instruction* on page 6-16.

All timing examples assume one-to-one clocking where the ARM966E-S and AHB share the same clock. See *AHB clocking* on page 6-17 for details of AHB clocking modes.

Burst transfers

Because the ARM966E-S does not implement cache memory, burst transfers of fixed length commonly used for cache linefill and data cache writeback, are not supported. All burst accesses are defined to be INCRemental (**HBURST[2:0]** = 001), because the only indication to the ARM966E-S about the sequentiality of the access is the **DMORE** output from the ARM9E-S core. This output indicates that there is at least one more access following the current access, but does not indicate how many more sequential accesses can be expected.

Bus request

At the start of every AHB access, the ARM966E-S requests access to the bus by asserting **HBUSREQ** to the arbiter. It must then wait for an acknowledge signal from the arbiter (**HGRANT**), before beginning the transfer on the next rising edge of **HCLK**. In Figure 6-2, the slave being addressed has a single-cycle response to the read access and therefore the **HREADY** response is driven HIGH and fed to the ARM966E-S BIU.

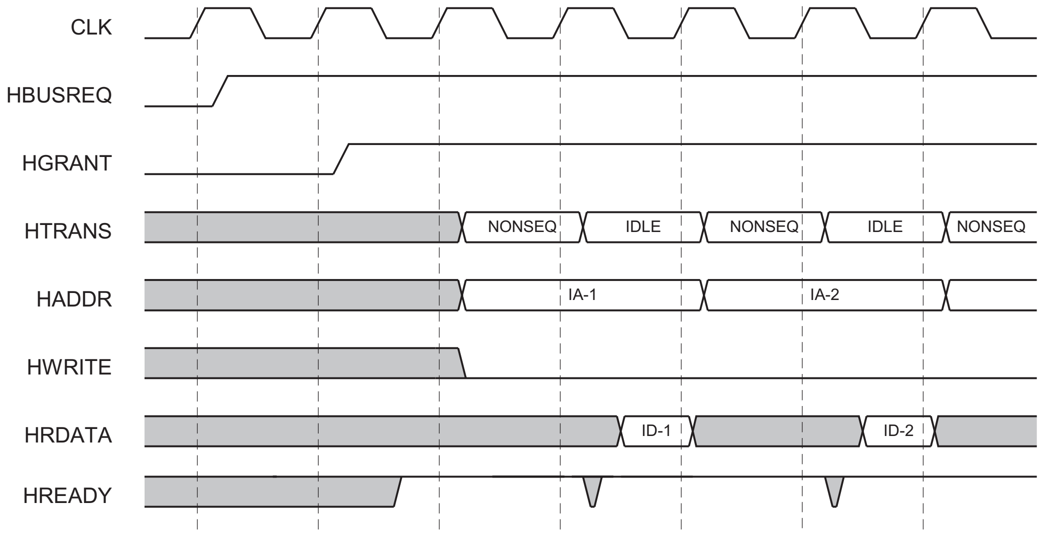


Figure 6-2 Sequential instruction fetches, after being granted the bus

Sequential instruction fetch

When the ARM9E-S fetches instructions from the AHB address space or if the tightly-coupled I-SRAM is disabled, AHB read transfers are initiated by the BIU. The instruction interface does not have the benefit of a pipelined **MORE** signal, so the BIU cannot detect a sequential access and use an address incrementer to perform back-to-back sequential cycles. All instruction fetches are treated as non-sequential accesses.

Figure 6-3 shows a series of sequential instruction fetches where any data access being performed by the ARM9E-S is using the tightly-coupled SRAM. Therefore, data accesses do not interfere with the instruction fetches.

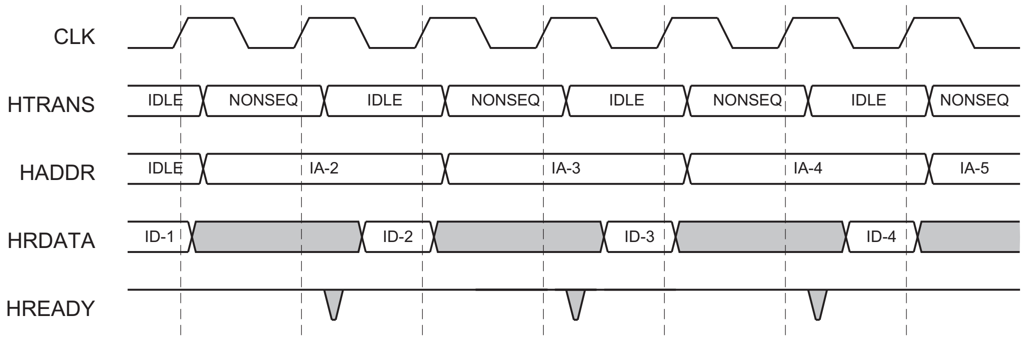


Figure 6-3 Sequential instruction fetches, no AHB data access required

Back-to-back LDR or STR accesses

Figure 6-4 shows ARM966E-S bus activity when a sequence of LDR instructions is executed.

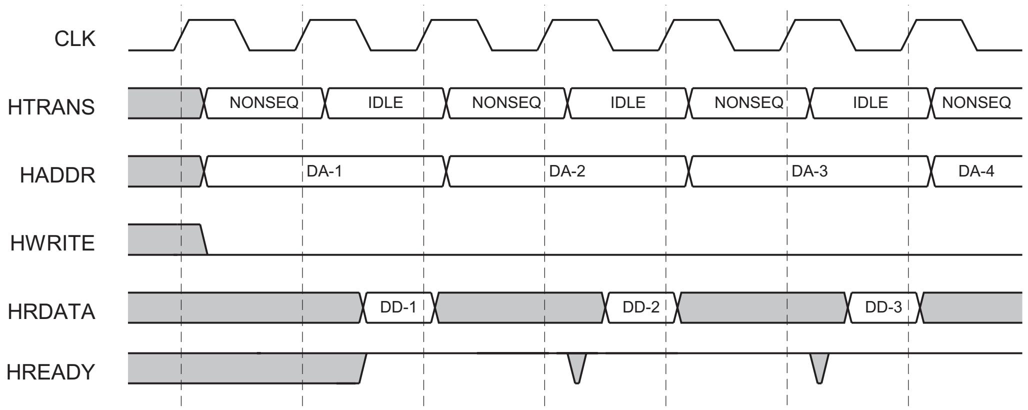


Figure 6-4 Back-to-back LDR, no external instruction access

A series of NONSEQ/IDLE transfers is indicated for each access.

Even though the transfers are to sequential addresses, each access is treated as a separate nonsequential transfer. Figure 6-4 on page 6-9 assumes that all instruction fetches from the ARM9E-S core are being serviced by the I-SRAM.

——— **Note** ———

An identical series of NONSEQ or IDLE transfers is seen if executing a sequence of back-to-back `STR` instructions.

———————————

Simultaneous instruction and data request

When the ARM9E-S makes a simultaneous instruction and data request, both resident in AHB memory, the BIU must arbitrate between the two accesses. The data access is always completed first, stalling the ARM9E-S until the instruction fetch completes.

Figure 6-5 shows an example of an `STR` instruction causing a simultaneous instruction and data request.

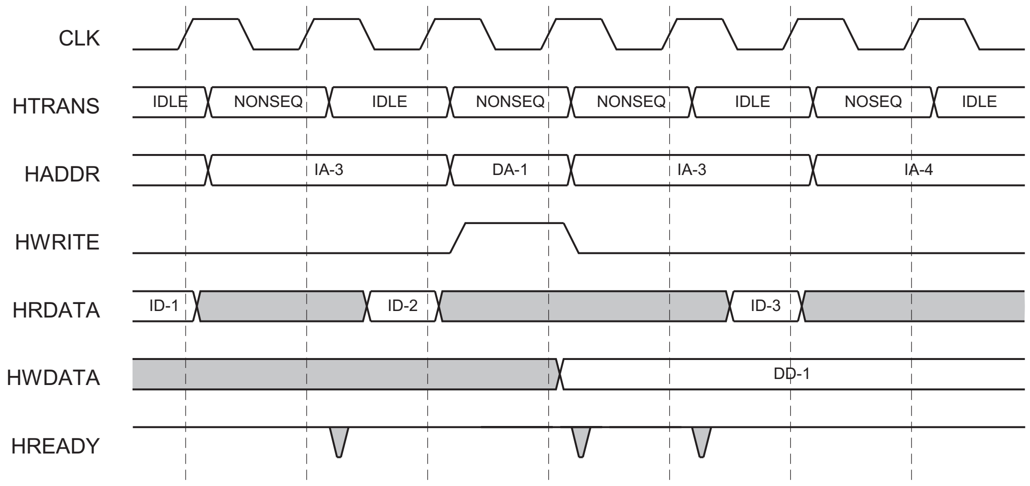


Figure 6-5 Simultaneous instruction and data requests

During the cycle that [IA-3] is first driven onto **HADDR**, the BIU detects a simultaneous data request. [IA-3] fetch is suspended until the data access has completed.

STM timing

Figure 6-6 shows the timing for an `STM` instruction, transferring three words. Outputs to the AHB are not driven during IDLE cycles, and so hold their previous value. This includes the **HBURST** output, continuing to indicate INCRemental until the next nonsequential transfer. This should not cause any confusion to other AHB components as **HTRANS** indicates IDLE cycles.

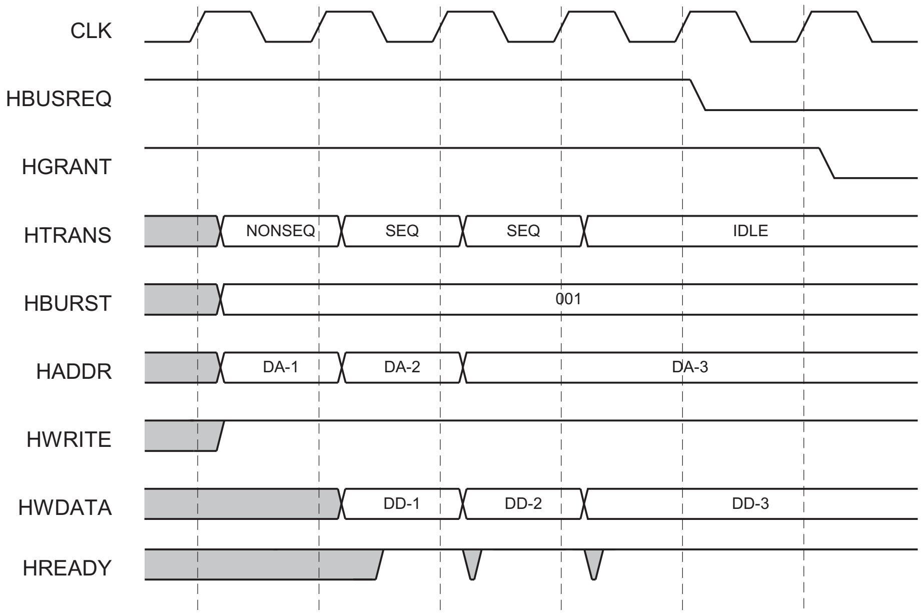


Figure 6-6 Single STM, no instruction fetch

Note

If an `STM` is not immediately followed by an external instruction access one IDLE cycle is inserted, and **HBUSREQ** is driven LOW. An `STM`, immediately followed by any other AHB data access, also results in one IDLE cycle being inserted between the two accesses.

LDM timing

Figure 6-7 on page 6-12 shows the timing for an `LDM` instruction, transferring three words.

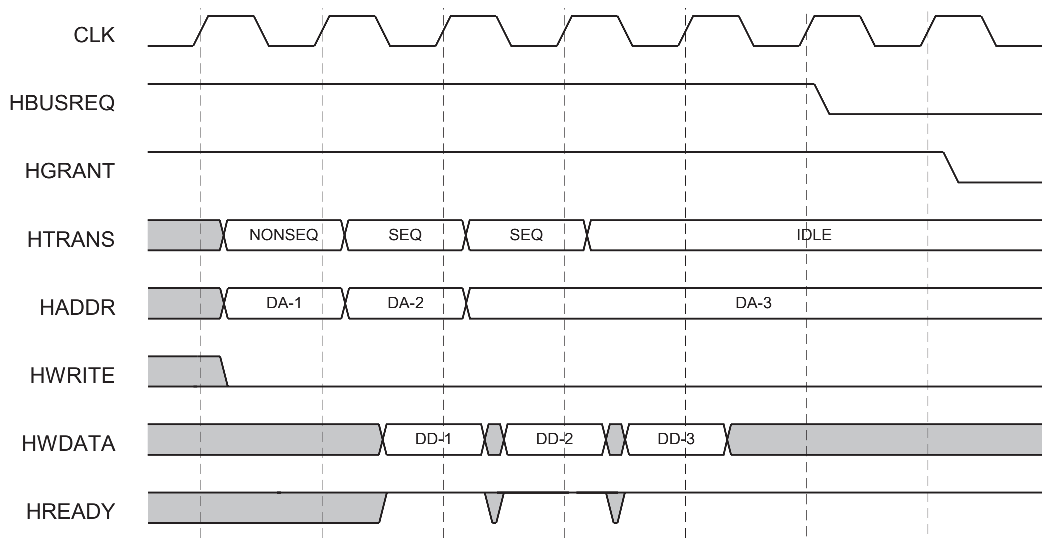


Figure 6-7 Single LDM, no instruction access

Note

HBUSREQ is driven LOW after two IDLE cycles which are inserted after a `LDM` that is immediately followed by an external instruction access. An `LDM`, immediately followed by any other AHB data access, also results in two IDLE cycles being inserted between the two accesses.

STM followed by instruction fetch

Figure 6-8 on page 6-13 shows an example of an `STM` transferring three words, immediately followed by an instruction fetch. The instruction read begins with a NONSEQ/IDLE sequence after the final sequential data access. In this example, subsequent instruction fetches are sequential.

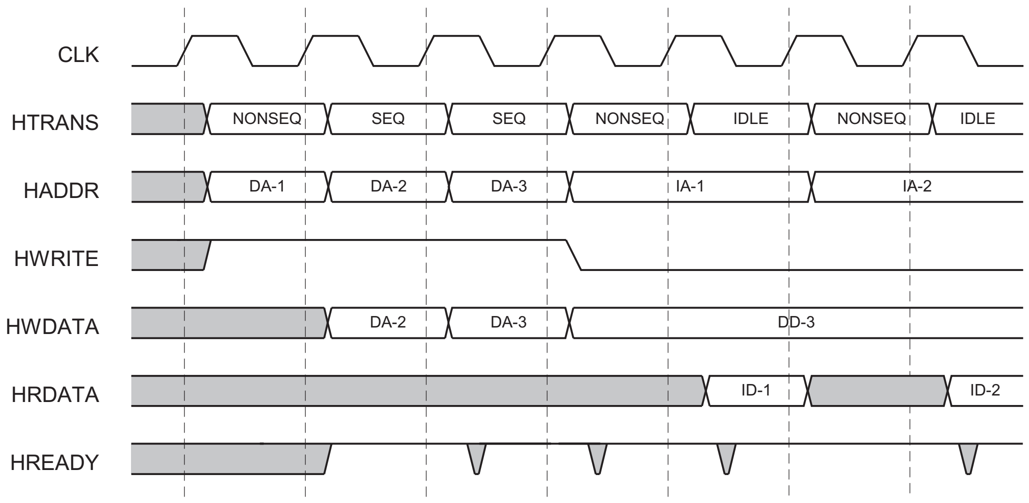


Figure 6-8 Single STM, followed by sequential instruction fetch

Note

The single IDLE cycle that normally occurs at the end of an `STM` is filled by the NONSEQ cycle for the instruction fetch.

LDM followed by instruction fetch

Figure 6-9 on page 6-14 shows an example of a `LDM` transferring three words, immediately followed by an instruction fetch. A single IDLE cycle is inserted after the final sequential data access, and instruction fetch begins with a NONSEQ/IDLE sequence.

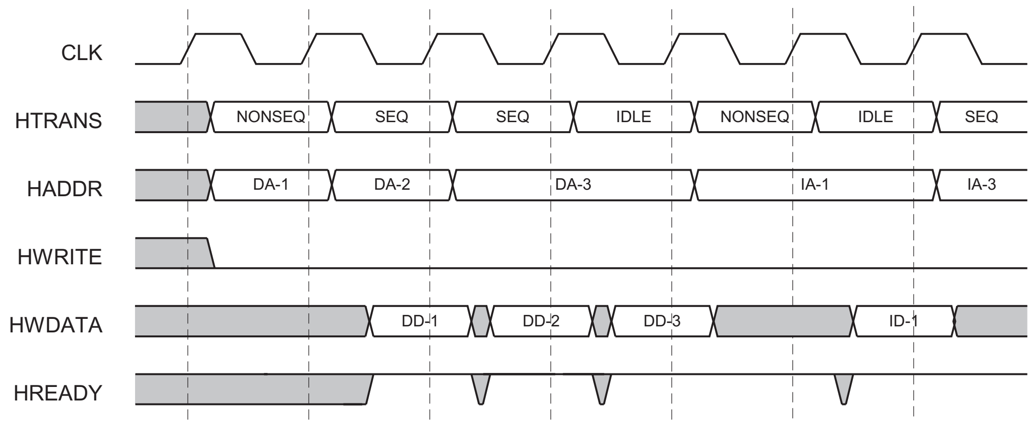


Figure 6-9 Single LDM followed by sequential instruction fetch

Note

The NONSEQ cycle of the instruction fetch replaces the second IDLE cycle that occurs when an AHB data access is required following the `LDM`.

STM crossing a 1KB boundary

AMBA Rev.2 Specification states that sequential accesses must not cross 1KB boundaries. The ARM966E-S splits sequential accesses that cross a 1KB boundary into two sets of separate accesses.

Figure 6-10 on page 6-15 shows bus activity when a `STM` writing four words, crosses a 1KB boundary. DA-3 is the first address in a new 1KB region. The two sets of transfers each begin with a nonsequential access type, and are separated by an IDLE cycle.

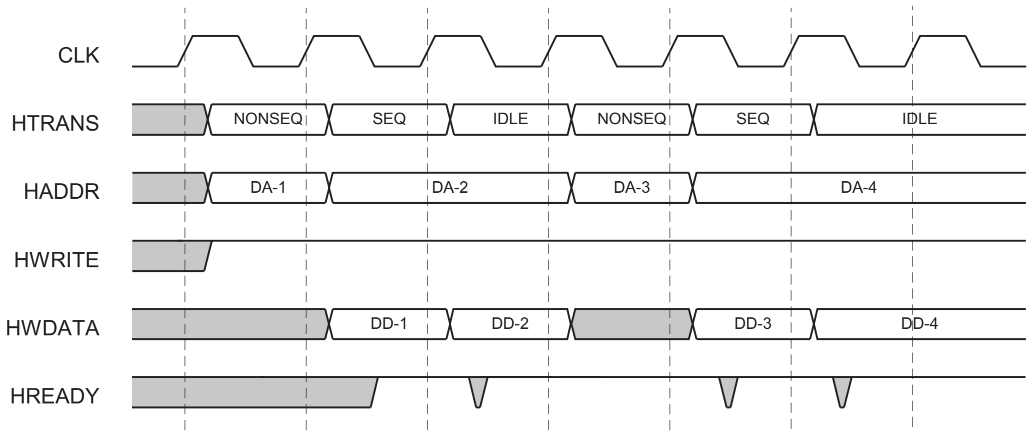


Figure 6-10 Single STM, crossing a 1KB boundary

LDM crossing a 1KB boundary

Figure 6-11 shows bus activity when a `LDM` reading four words, crosses a 1KB boundary. The two sets of transfers each begin with a nonsequential access type, and are separated by two IDLE cycles.

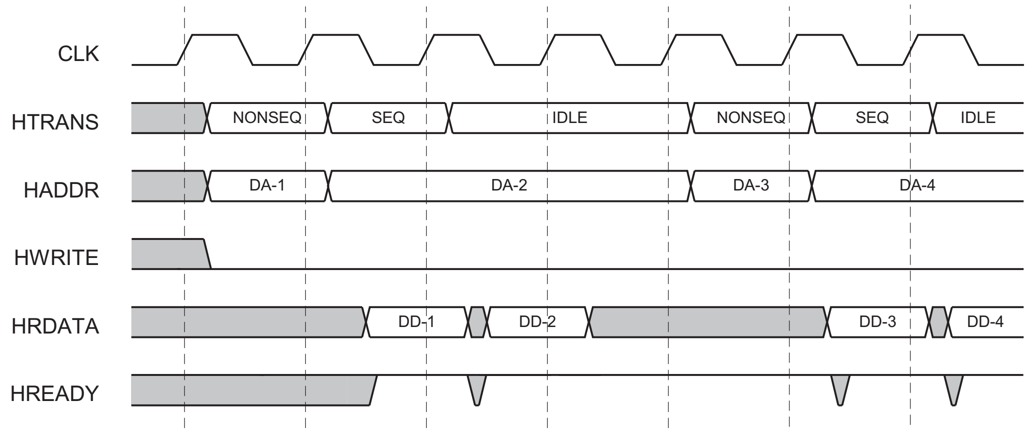


Figure 6-11 Single LDM, crossing a 1KB boundary

SWP instruction

The ARM `SWP` instruction performs an atomic read-modify-write operation. It is commonly used with semaphores to guarantee that another process cannot modify a semaphore when it is being read by the current process.

If the ARM966E-S performs a `SWP` operation to an AHB address location, the access is always unbuffered to ensure that the core is stalled until the write has occurred on the AHB. The BIU asserts the **HLOCK** output to prevent the AHB arbiter from granting a different master, ensuring that the read-modify-write is atomic.

Figure 6-12 shows a `SWP` instruction.

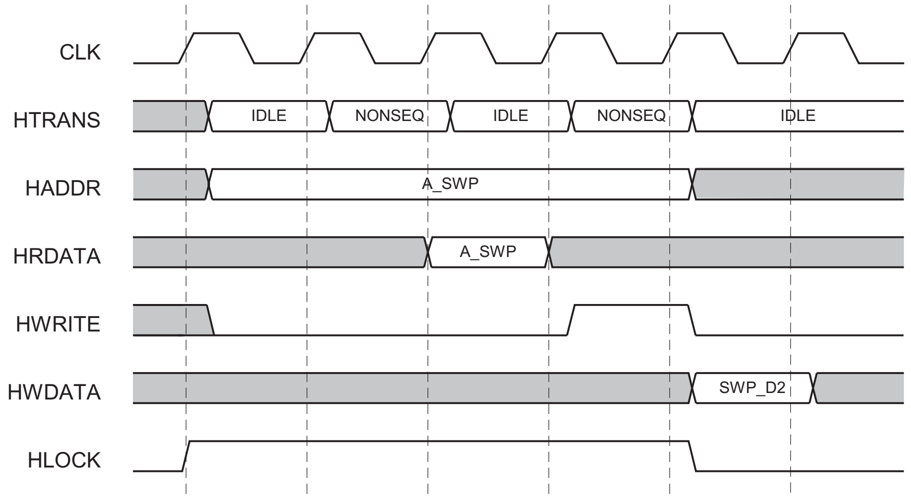


Figure 6-12 SWP instruction

6.4 AHB clocking

The ARM966E-S design uses a single rising edge clock **CLK** to time all internal activity. In many systems where the ARM966E-S is embedded, it is desirable to run the AHB at a lower rate. To support this requirement, the ARM966E-S requires a clock enable, **HCLKEN**, to time AHB transfers.

The **HCLKEN** input is driven HIGH around a rising edge of the ARM966E-S **CLK** to indicate that this rising edge is also a rising edge of **HCLK**. This requires that **HCLK** is synchronous to the ARM966E-S **CLK**.

When the ARM9E-S is running from tightly-coupled SRAM or performing writes using the write buffer, the ARM966E-S **HCLKEN** and **HREADY** inputs are ignored in terms of generating the **SYSCLKEN** core stall signal. The core is only stalled by SRAM stall cycles or if the write buffer overflows. This means that the ARM9E-S is executing instructions at the faster **CLK** rate and is effectively decoupled from the **HCLK** domain AHB system.

If however, an AHB read access or unbuffered write is required, the core is stalled until the AHB transfer has completed. Because the AHB system is being clocked by the lower rate **HCLK**, it is necessary to examine **HCLKEN** to detect when to drive out the AHB address and control to start an AHB transfer. **HCLKEN** is then required to detect the following rising edges of **HCLK** so that the BIU knows the access has completed. Figure 6-13 shows an example of an AHB read access where there is a 3:1 ratio of **CLK** to **HCLK**.

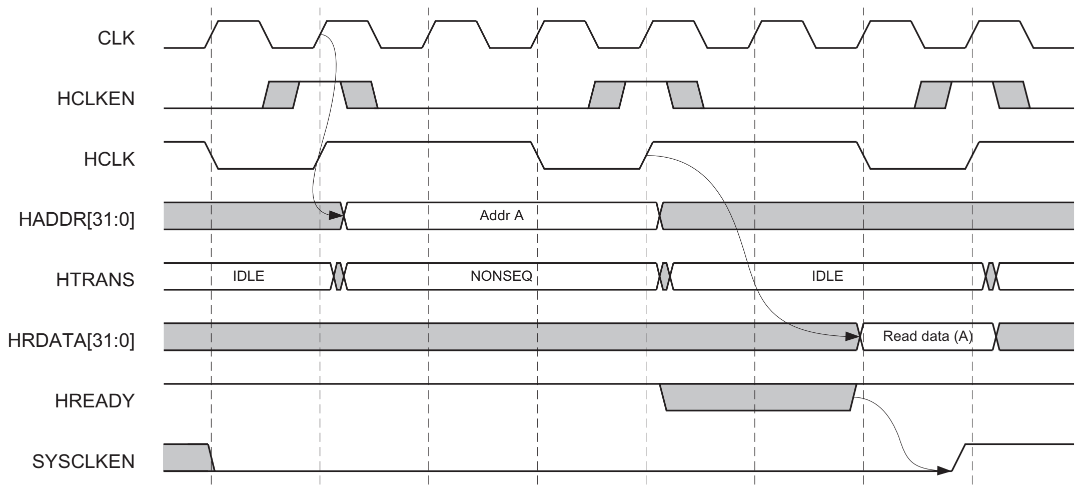


Figure 6-13 AHB 3:1 clocking example

If the slave being accessed at the **HCLK** rate has a multi-cycle response, the **HREADY** input to the ARM966E-S is driven LOW until the data is ready to be returned. The BIU must therefore perform a logical AND on the **HREADY** response with **HCLKEN** to detect that the AHB transfer has completed. When this is the case, the ARM9E-S core can then be enabled by reasserting **SYSCLKEN**.

Note

When an AHB access is required, the core must be stalled until the next **HCLKEN** pulse is received, before it can start the access, and then until the access has completed. This stall before the start of the access is a synchronization penalty and the worst case can be expressed in **CLK** cycles as the **CLK** to **CLK** ratio minus one.

6.4.1 CLK to HCLK skew

The ARM966E-S drives out the AHB address on the rising edge of **CLK** when the **HCLKEN** input is true. The AHB outputs have output hold and delay values relative to **CLK**. However, these outputs are used in the AHB system where **HCLK** is used to time the transfers. Similarly, inputs to the ARM966E-S are timed relative to **HCLK** but are sampled within the ARM966E-S with **CLK**. This leads to hold time issues from **CLK** to **HCLK** on outputs and from **HCLK** to **CLK** on inputs. In order to minimize this effect the skew between **HCLK** and **CLK** must be minimized.

Clock tree insertion at top level

Considering the skew issue in more detail, the ARM966E-S has a clock tree inserted to allow an evenly distributed clock to be driven to all the registers in the design. The registers that drive out AHB outputs and sample AHB inputs are timed off **CLK’** at the bottom of the inserted clock tree and subject to the clock tree insertion delay. To maximize performance, when the ARM966E-S is embedded in an AHB system, the clock generation logic to produce **HCLK** must be constrained so that it matches the insertion delay of the clock tree within the ARM966E-S. This can easily be achieved by a clock tree insertion tool if the clock tree is inserted for the ARM966E-S and the embedded system at the same time (top level insertion).

Figure 6-14 on page 6-19 shows an example of an AHB slave connected to the ARM966E-S.

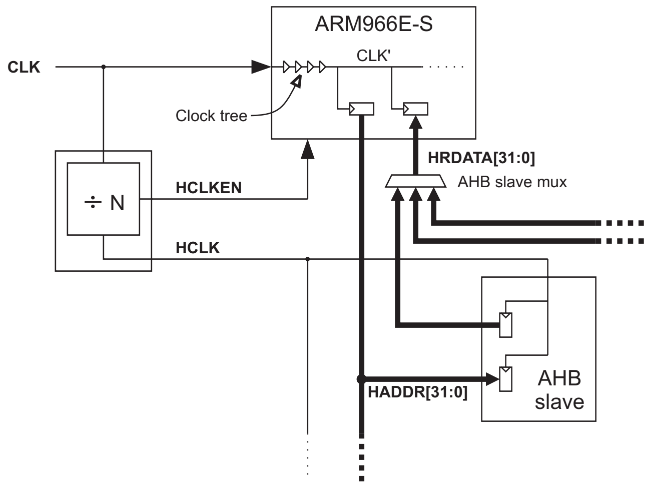


Figure 6-14 ARM966E-S CLK to AHB HCLK sampling

In this example, the slave peripheral has an input setup and hold, and an output hold and valid time relative to **HCLK**. The ARM966E-S has an input setup and hold, and an output hold and valid relative to **CLK'**, the clock at the bottom of the clock tree. Clock tree insertion must be used to position the **HCLK** to match **CLK'** for optimal performance.

Hierarchical clock tree insertion

If the ARM966E-S has clock tree insertion performed before embedding it, buffers are added on input data to match the clock tree so that the setup and hold is relative to the top level **CLK**. This is guaranteed to be safe at the expense of extra buffers in the data input path.

The **HCLK** domain AHB peripherals must still meet the ARM966E-S input setup and hold requirements. Because the ARM966E-S inputs and outputs are now relative to **CLK**, the outputs do appear comparatively later by the value of the insertion delay. This ultimately leads to lower AHB performance.

Chapter 7
Coprocessor Interface

This chapter describes the ARM966E-S pipelined coprocessor interface. It contains the following sections:

7.1 About the coprocessor interface

ARM966E-S fully supports the connection of on-chip coprocessors through the external coprocessor interface and supports all classes of coprocessor instructions.

The interface differs from the basic ARM9E-S coprocessor interface. To ease integration of an external coprocessor, the interface from the ARM966E-S to the coprocessor has been pipelined by a single clock cycle.

This ensures that ARM966E-S interface outputs, which otherwise arrive late in the clock cycle, are driven out directly from registers to the external coprocessor. This significantly eases the implementation task for an external coprocessor.

7.1.1 Synchronizing the external coprocessor pipeline

A coprocessor connected to the ARM966E-S determines which instructions it needs to execute by implementing a pipeline follower in the coprocessor. Because each instruction arrives from instruction memory (either from the I-SRAM or AHB interface) it enters both the ARM9E-S pipeline and the coprocessor pipeline follower. Because the interface is itself pipelined, the coprocessor pipeline follower operates one cycle behind the ARM9E-S, sampling the **CPINSTR[31:0]** output bus from the ARM966E-S interface.

In order to hide the pipeline delay, a mechanism inside the interface block stalls the ARM9E-S for a cycle by internally modifying the coprocessor handshake signals whenever an external coprocessor instruction is decoded. This allows the external coprocessor to **catch up** with the ARM9E-S core.

After this initial stall cycle, the two pipelines can be considered synchronized. The ARM9E-S then informs the coprocessor when instructions move from Decode into Execute, and whether the instruction has passed its condition codes and is to be executed.

Note

Because the ARM966E-S hides the synchronization of the coprocessor pipeline follower, its coprocessor handshake interface is similar to that of the native ARM9E-S. This implies that an ARM9E-S designed *pipeline follower* can interface to the ARM966E-S without modification. The data path of the coprocessor differs however, due to the ARM966E-S pipelined output data **CPDOUT[31:0]**.

7.1.2 External coprocessor clocking

The coprocessor data processing instruction (`CDP`) is used for coprocessor instructions that do not operate on values in ARM registers or in main memory. One example is a floating-point multiply instruction for a floating-point accelerator processor.

To enable coprocessors to continue execution of `CDP` instructions while the ARM9E-S core pipeline is stalled (for instance while waiting for an AHB transfer to complete), the coprocessor receives the free-running system clock **CLK**, and a clock enable signal **CPCLKEN**. If **CPCLKEN** is LOW around the rising edge of **CLK** then the ARM9E-S core pipeline is stalled and the coprocessor pipeline follower must not advance.

This prevents any new instructions entering Execute within the coprocessor but allows a `CDP` instruction in Execute to continue execution. The coprocessor is only stalled when the current instruction leaves Execute and new instructions are required from the ARM966E-S interface.This goes some way towards decoupling the external coprocessor from the ARM9E-S memory interface.

There are three classes of coprocessor instructions:

- `LDC/STC`
- `MCR/MRC`
- `CDP`.

Examples of how a coprocessor executes these instruction classes are given in the following sections:

- *LDC/STC* on page 7-4
- *MCR/MRC* on page 7-8
- *CDP* on page 7-10

7.2 LDC/STC

The `LDC` and `STC` instructions are used respectively to transfer data to and from external coprocessor registers and memory. In the case of the ARM966E-S, the memory can be either tightly-coupled SRAM or AHB depending on the address range of the access and SRAM enable.

The cycle timing for these operations is shown in Figure 7-1.

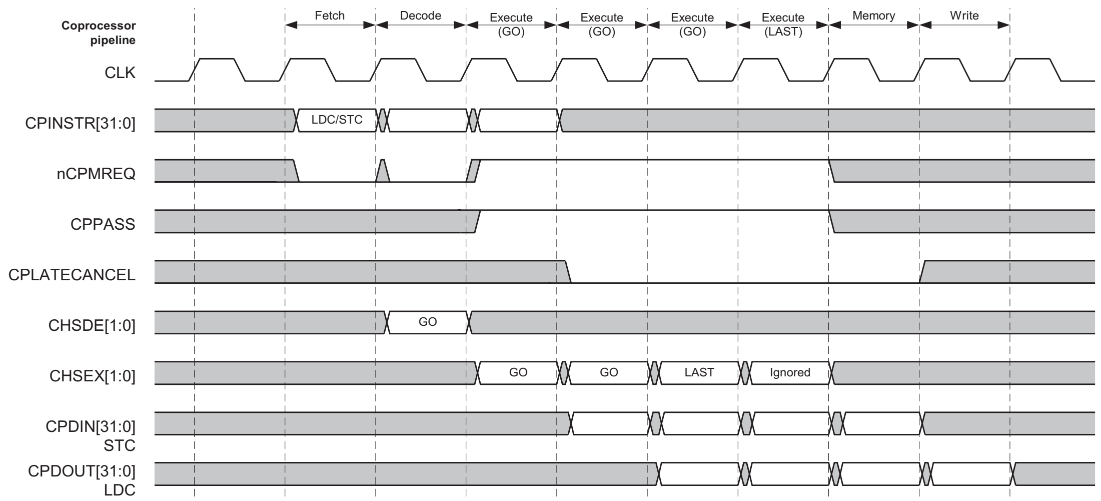


Figure 7-1 LDC/STC cycle timing

In this example, four words of data are transferred. The number of words transferred is determined by how the coprocessor drives the **CHSDE[1:0]** and **CHSEX[1:0]** buses.

As with all other instructions, the ARM9E-S performs the main decode off the rising edge of the clock during the Decode stage. From this, the core commits to executing the instruction and so performs an instruction fetch. The coprocessor instruction pipeline keeps in step with ARM9E-S core by monitoring **nCPMREQ**, which is a registered version of the ARM9E-S core instruction memory request signal **InMREQ**.

At the rising edge of **CLK**, if **CPCLKEN** is HIGH, and **nCPMREQ** is LOW, an instruction fetch is taking place, and **CPINSTR[31:0]** contains the fetched instruction on the next rising edge of the clock, when **CPCLKEN** is HIGH.

This means that:

- the last instruction fetched must enter the Decode stage of the coprocessor pipeline

- the instruction in the Decode stage of the coprocessor pipeline must enter its Execute stage
- the fetched instruction must be sampled.

In all other cases, the ARM9E-S pipeline is stalled, and the coprocessor pipeline must not advance.

During the Execute stage, the condition codes are compared with the flags to determine whether the instruction really executes or not. The output **CPPASS** is asserted, HIGH, if the instruction in the Execute stage of the coprocessor pipeline:

- is a coprocessor instruction
- has passed its condition codes.

If a coprocessor instruction busy-waits, **CPPASS** is asserted on every cycle until the coprocessor instruction is executed. If an interrupt occurs during busy-waiting, **CPPASS** is driven LOW, and the coprocessor stops execution of the coprocessor instruction.

Another output, **CPLATECANCEL**, cancels a coprocessor instruction when the instruction preceding it caused a data abort. This is valid on the rising edge of **CLK** on the cycle that follows the first Execute cycle of the coprocessor instructions. This is the only cycle in which **CPLATECANCEL** can be asserted.

On the rising edge of the clock, the ARM9E-S processor examines the coprocessor handshake signals **CHSDE[1:0]** or **CHSEX[1:0]**:

- If a new instruction is entering the Execute stage in the next cycle, it examines **CHSDE[1:0]**.
- If the currently executing coprocessor instruction requires another Execute cycle, it examines **CHSEX[1:0]**.

7.2.1 Coprocessor handshake states

The handshake signals encode one of four states:

ABSENT	If there is no coprocessor attached that can execute the coprocessor instruction, the handshake signals indicate the ABSENT state. In this case, the ARM9E-S takes the undefined instruction trap.
WAIT	If there is a coprocessor attached that can handle the instruction, but not immediately, the coprocessor handshake signals are driven to indicate that the ARM9E-S processor core must stall until the coprocessor can catch up. This is known as the *busy-wait* condition. In this case, the ARM9E-S processor core loops in an IDLE state waiting for

CHSEX[1:0] to be driven to another state, or for an interrupt to occur. If **CHSEX[1:0]** changes to ABSENT, the undefined instruction trap is taken. If **CHSEX[1:0]** changes to GO or LAST, the instruction proceeds as described here. If an interrupt occurs, the ARM9E-S processor is forced out of the busy-wait state. This is indicated to the coprocessor by the **CPPASS** signal going LOW. The instruction is restarted later and so the coprocessor must not commit to the instruction (it must not change any coprocessor state) until **CPPASS** is asserted HIGH, when the handshake signals indicate the GO or LAST condition.

GO The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires at least another cycle of execution. Both the ARM9E-S processor core and the coprocessor must also consider the state of the **CPPASS** signal before actually committing to the instruction. For an LDC or STC instruction, the coprocessor instruction drives the handshake signals with GO when two or more words still need to be transferred. When only one more word is to be transferred, the coprocessor drives the handshake signals with LAST. During the Execute stage, the ARM9E-S processor core outputs the address for the LDC/STC. Also in this cycle, **DnMREQ** is driven LOW, indicating to the ARM966E-S memory system that a memory access is required at the data end of the device. The timing for the data on **CPDOUT** and **CPDIN** is shown in Figure 7-1 on page 7-4.

LAST An LDC or STC can be used for more than one item of data. If this is the case, possibly after busy waiting, the coprocessor drives the coprocessor handshake signals with a number of GO states, and in the penultimate cycle LAST (LAST indicating that the next transfer is the final one). If there is only one transfer, the sequence is [WAIT,[WAIT,...]],LAST. LAST is also usually driven for CDP instruction.

7.2.2 Coprocessor handshake encoding

Table 7-1 shows how the handshake signals **CHSDE[1:0]** and **CHSEX[1:0]** are encoded.

Table 7-1 Handshake encoding

[1:0]	Meaning
10	ABSENT

Table 7-1 Handshake encoding (continued)

[1:0]	Meaning
00	WAIT
01	GO
11	LAST

Note

If an external coprocessor is not attached in the ARM966E-S embedded system, the **CHSDE[1:0]** and **CHSEX[1:0]** handshake inputs must be tied off to indicate ABSENT.

7.2.3 Multiple external coprocessors

If multiple external coprocessors are to be attached to the ARM966E-S interface, the handshaking signals can be combined by ANDing bit1, and ORing bit0. In the case of two coprocessors which have handshaking signals **CHSDE1**, **CHSEX1** and **CHSDE2**, **CHSEX2** respectively:

CHSDE[1] = **CHSDE1[1]** AND CHSDE2[1]

CHSDE[0] = **CHSDE1[0]** OR CHSDE2[0]

CHSEX[1] = **CHSEX1[1]** AND CHSEX2[1]

CHSEX[0] = **CHSEX1[0]** OR **CHSEX2[0]**.

7.3 MCR/MRC

These cycles look very similar to `STC`/`LDC`. An example, with a busy-wait state, is shown in Figure 7-2. First **nCPMREQ** is driven LOW to denote that the instruction on **CPINSTR[31:0]** is entering the Decode stage of the pipeline. This causes the coprocessor to decode the new instruction and drive **CHSDE[1:0]**. In the next cycle **nCPMREQ** is driven LOW to denote that the instruction has now been issued to the Execute stage. If the condition codes passes, and the instruction is to be executed, the **CPPASS** signal is driven HIGH and the **CHSDE[1:0]** handshake bus is examined (it is ignored in all other cases).

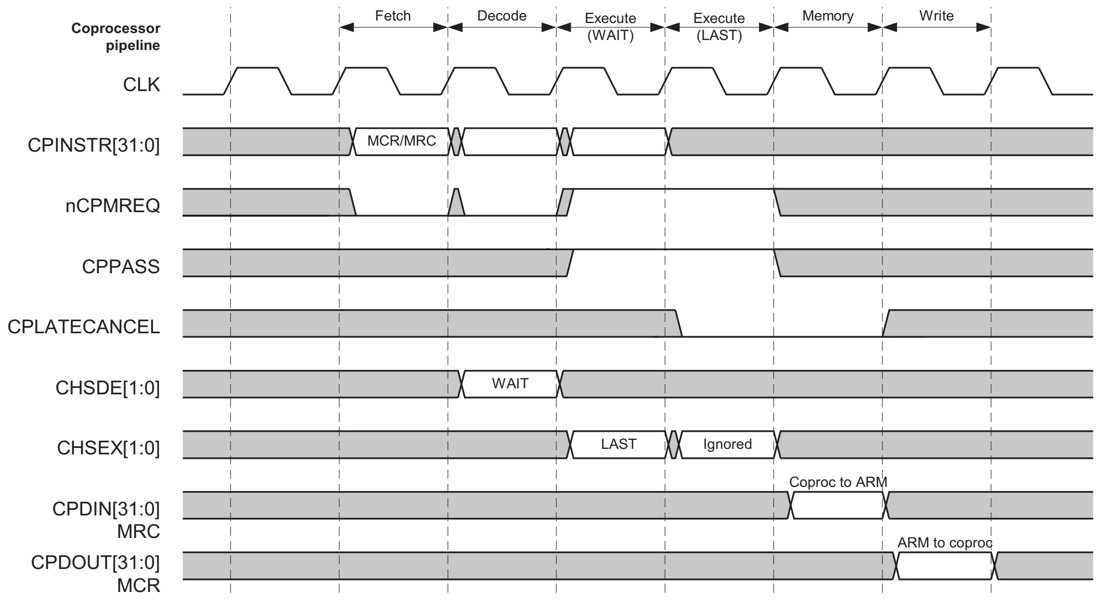


Figure 7-2 MCR/MRC transfer timing with busy-wait

For any successive Execute cycles the **CHSEX[1:0]** handshake bus is examined. When the LAST condition is observed, the instruction is committed. In the case of a `MCR`, the **CPDOUT[31:0]** bus is driven with the registered data. In the case of a `MRC`, **CPDIN[31:0]** is sampled at the end of the ARM9E-S core Memory stage and written to the destination register during the next cycle.

7.4 Interlocked MCR

If the data for a MCR operation is not available inside the ARM9E-S core pipeline during its first Decode cycle, then the ARM9E-S core pipeline interlocks for one or more cycles until the data is available. An example of this is where the register being transferred is the destination from a preceding LDR instruction.

In this situation the MCR instruction enters the Decode stage of the coprocessor pipeline, and then remains there for a number of cycles before entering the Execute stage. Figure 7-3 gives an example of an interlocked MCR that also has a busy-wait state.

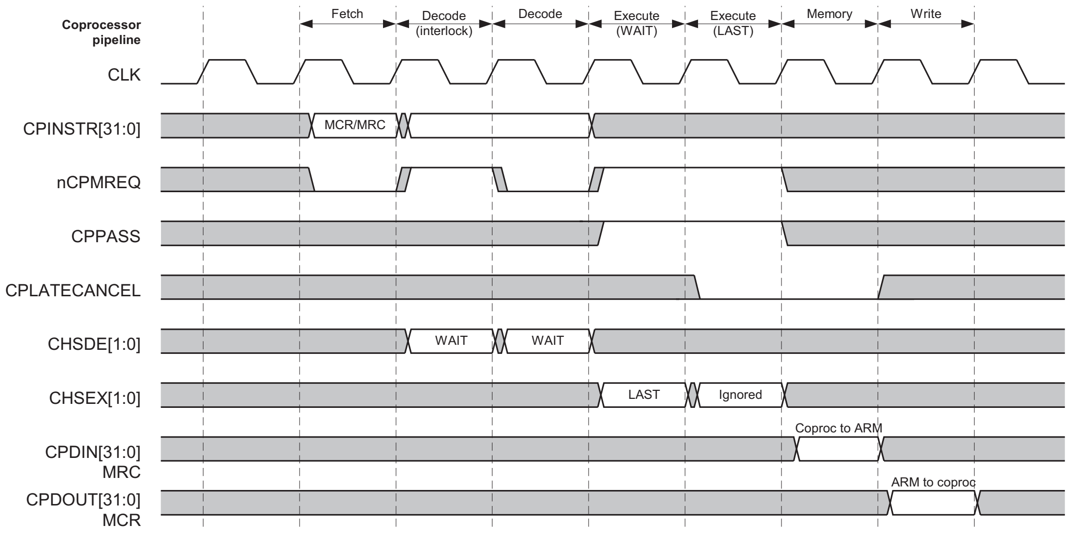


Figure 7-3 Interlocked MCR/MRC timing with busy-wait

7.5 CDP

`CDP` instructions normally execute in a single cycle. Like all the previous cycles, **nCPMREQ** is driven LOW to signal when an instruction is entering the Decode and then the Execute stage of the pipeline:

- if the instruction really is to be executed, the **CPPASS** signal is driven HIGH during the Execute cycle
- if the coprocessor can execute the instruction immediately it drives **CHSDE[1:0]** with LAST
- if the instruction requires a busy-wait cycle, the coprocessor drives **CHSDE[1:0]** with WAIT and then **CHSEX[1:0]** with LAST.

Figure 7-4 shows a cancelled `CDP` due to the previous instruction causing a Data Abort.

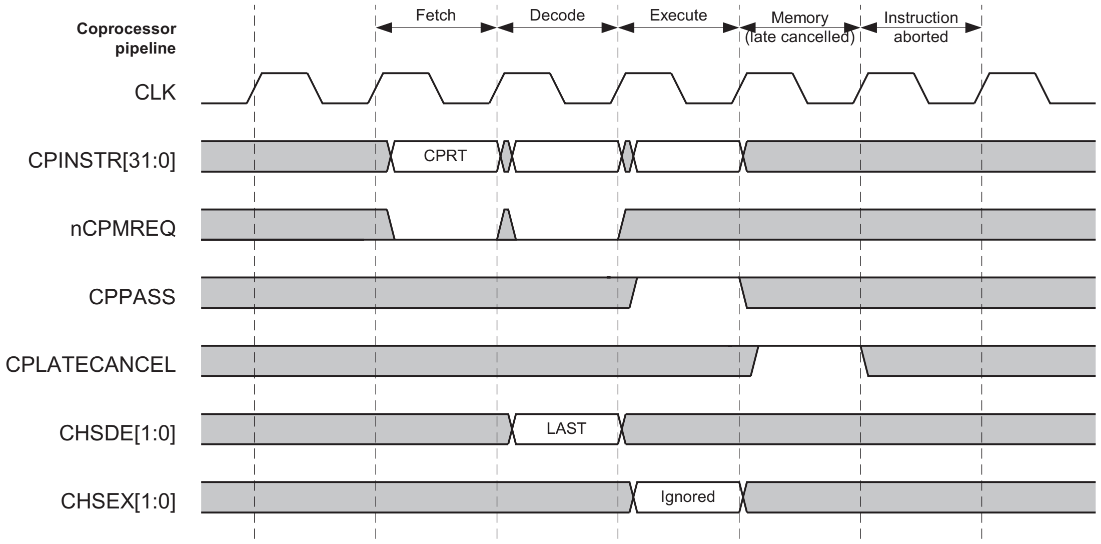


Figure 7-4 Late cancelled CDP

The `CDP` instruction enters the Execute stage of the pipeline and is signaled to execute by **CPASS**. In the following cycle **CPLATECANCEL** is asserted. This causes the coprocessor to terminate execution of the `CDP` instruction and for it to cause no state changes to the coprocessor.

7.6 Privileged instructions

The coprocessor restricts certain instructions for use in privileged modes only. To do this, the coprocessor tracks the **nCPTRANS** output. Figure 7-5 shows how **nCPTRANS** changes after a mode change.

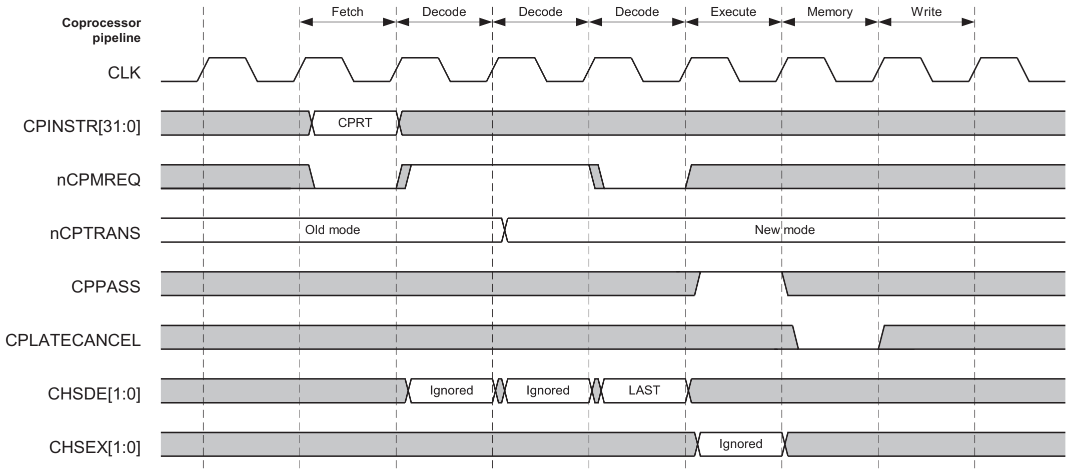


Figure 7-5 Privileged instructions

The first two **CHSDE[1:0]** responses are ignored by the ARM9E-S because it is only the final **CHSDE[1:0]** response, as the instruction moves from Decode into Execute, that counts. This allows the coprocessor to change its response when **nCPTRANS** changes.

7.7 Busy-waiting and interrupts

The coprocessor is permitted to stall, or busy-wait, the processor during the execution of a coprocessor instruction if, for example, it is still busy with an earlier coprocessor instruction. To do so, the coprocessor associated with the Decode stage instruction drives WAIT onto **CHSDE[1:0]**. When the instruction concerned enters the Execute stage of the pipeline, the coprocessor drives WAIT onto **CHSEX[1:0]** for as many cycles as necessary to keep the instruction in the busy-wait loop.

For interrupt latency reasons the coprocessor might be interrupted while busy-waiting, causing the instruction to be abandoned. Abandoning execution is done through **CPPASS**. The coprocessor must monitor the state of **CPPASS** during every busy-wait cycle.

If it is HIGH, the instruction must still be executed. If it is LOW, the instruction must be abandoned.

Figure 7-6 shows a busy-waited coprocessor instruction being abandoned due to an interrupt.

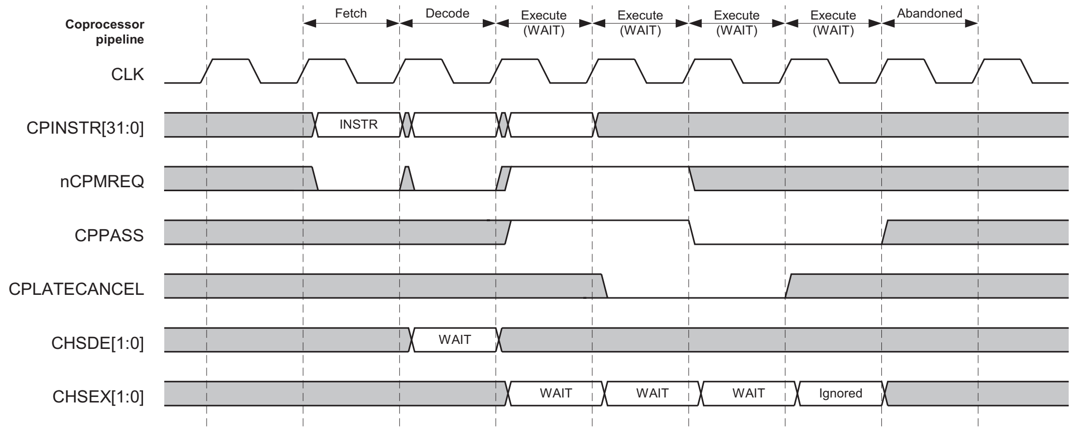


Figure 7-6 Busy-waiting and interrupts

Chapter 8
Debug Support

This chapter describes the ARM966E-S debug interface. It contains the following sections:

The ARM9E-S EmbeddedICE-RT logic is also discussed in this chapter including:

8.1 About the debug interface

The *ARM966E-S* debug interface is based on IEEE Std. 1149.1- 1990, *Standard Test Access Port and Boundary-Scan Architecture*. Refer to this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

The ARM9E-S processor core within the ARM966E-S contains hardware extensions for advanced debugging features. These make it easier to develop application software, operating systems, and the hardware itself.

The debug extensions allow you to force the core into *debug state*. In debug state, the core and ARM966E-S memory system are effectively stopped, and isolated from the rest of the system. This is known as *halt mode* operation and allows the internal state of the ARM9E-S core, ARM966E-S system, and external state of the AHB to be examined while all other system activity continues as normal. When debug is complete, the ARM9E-S restores the core and system state, and resumes program execution.

In addition, the ARM9E-S supports a real-time debug mode, where instead of generating a breakpoint or watchpoint, an internal Instruction Abort or Data Abort is generated. This is known as *monitor mode* operation.

When used in conjunction with a debug monitor program activated by the abort exception entry, You can debug the ARM966E-S while allowing the execution of critical interrupt service routines. The debug monitor program typically communicates with the debug host over the ARM966E-S debug communication channel. Monitor mode debug is described in *Monitor mode debug* on page 8-23.

8.1.1 Stages of debug

A request on one of the external debug interface signals, or on an internal functional unit known as the *EmbeddedICE-RT logic*, forces the ARM9E-S into debug state. The interrupts that activate debug are:

- a breakpoint (a given instruction fetch)
- a watchpoint (a data access)
- an external debug request.

The internal state of the ARM9E-S is examined using a JTAG-style serial interface, allowing instructions to be serially inserted into the core pipeline without using the external data bus. For example, when in debug state, a *STore Multiple* (`STM`) can be inserted into the instruction pipeline, and this exports the contents of the ARM9E-S registers. This data can be serially shifted out without affecting the rest of the system.

8.1.2 Clocks

The system and test clocks must be synchronized externally to the ARM966E-S macrocell. The ARM Multi-ICE debug agent directly supports one or more cores within an ASIC design. To synchronize off-chip debug clocking with the ARM966E-S macrocell requires a three-stage synchronizer. The off-chip device (for example, Multi-ICE) issues a **TCK** signal, and waits for the **RTCK** (**R**eturned **TCK**) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next **TCK** until after **RTCK** is received.

Figure 8-1 shows this synchronization.

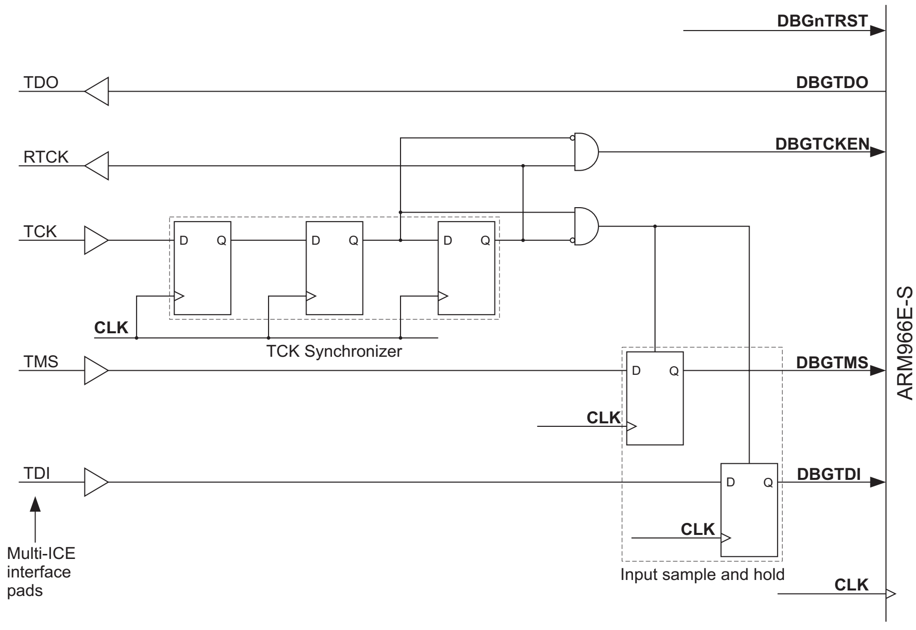


Figure 8-1 Clock synchronization

8.2 Debug systems

The ARM966E-S forms one component of a debug system that interfaces from the high-level debugging performed by you to the low-level interface supported by the ARM966E-S. Figure 8-2 shows a typical debug system.

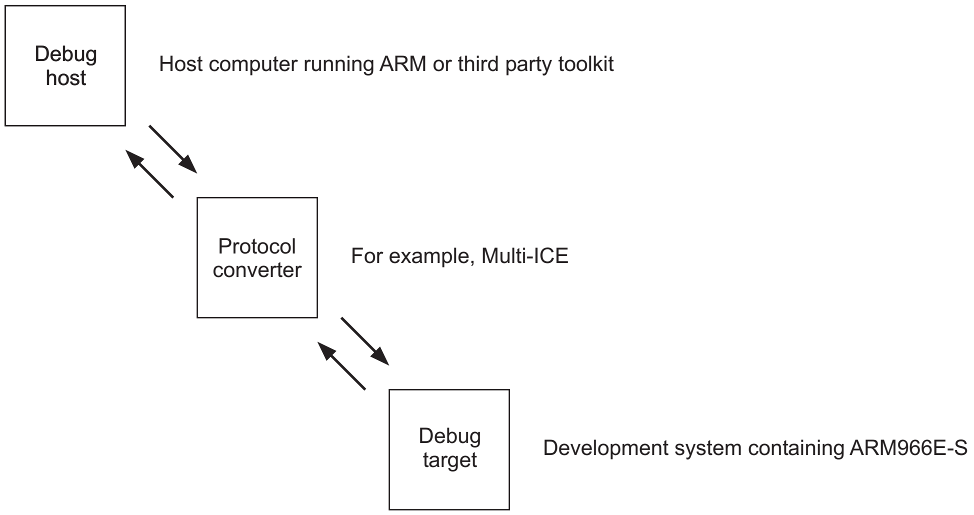

Debug host
Host computer running ARM or third party toolkit
Protocol converter
For example, Multi-ICE
Debug target
Development system containing ARM966E-S

Figure 8-2 Typical debug system

A debug system typically has three parts:

- *The debug host*
- *The protocol converter*
- *ARM966E-S debug target* on page 8-5.

The debug host and the protocol converter are system-dependent.

8.2.1 The debug host

The debug host is a computer that is running a software debugger, such as *armsd*. The debug host allows you to issue high-level commands such as setting breakpoints or examining the contents of memory.

8.2.2 The protocol converter

An interface, such as a parallel port, connects the debug host to the ARM966E-S development system. The messages broadcast over this connection must be converted to the interface signals of the ARM966E-S. The protocol converter performs the conversion.

8.2.3 ARM966E-S debug target

The ARM9E-S core within the ARM966E-S has hardware extensions that ease debugging at the lowest level. The debug extensions:

- allow you to stall the core from program execution
- examine the core internal state
- examine the state of the memory system
- resume program execution.

The following major blocks of the ARM9E-S debug model are shown in Figure 8-3.

ARM9E-S CPU core This includes hardware support for debug.

EmbeddedICE-RT logic This is a set of registers and comparators used to generate debug exceptions (such as breakpoints). This unit is described in *About the EmbeddedICE-RT* on page 8-16.

TAP controller This controls the action of the scan chains using a JTAG serial interface.

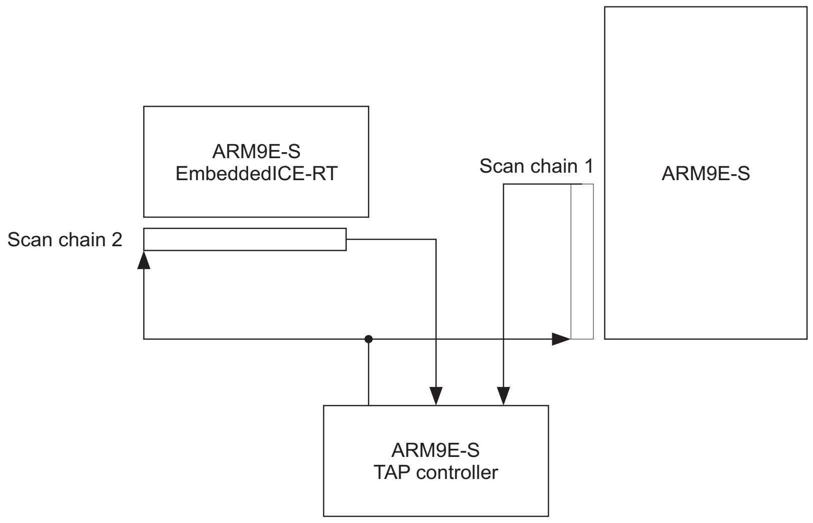


Figure 8-3 ARM9E-S block diagram

The ARM9E-S debug model is extended within the ARM966E-S by the addition of scan chain 15. This is used for debug access to the CP15 register bank, to allow the system state within the ARM966E-S to be configured while in debug state, for instance to enable or disable the SRAM before performing a debug load or store.

The rest of this chapter describes the ARM9E-S and ARM966E-S hardware debug extensions.

8.3 ARM966E-S scan chain 15

Scan chain 15 is provided to allow debug access to the CP15 register bank, to allow the system state within the ARM966E-S to be configured while in debug state.

The order of scan chain 15 from the **DBGTDI** input to the **DBGTDO** output is shown in Table 8-1

Table 8-1 Scan chain 15 addressing mode bit order

Bits	Contents
38	Read = 0, write = 1
37:32	CP15 register address
31:0	CP15 register value

.

The CP15 register address field of scan chain 15 provides debug access to the CP15 registers is shown in Table 8-2.

Table 8-2 Mapping of scan chain 15 address field to CP15 registers

Bit [38]	Bits[37:32]	Bits[31:30]	CP15 reg number	Meaning
0	0 0000 0	xx	C0	Read ID register
0	0 0001 0	xx	C1	Read control register
1	0 0001 0	xx	C1	Write control register
0	1 1111 1	00	C15	Read BIST control register
1	1 1111 1	00	C15	Write BIST control register
0	1 1111 0	01	C15	Read IBIST address
1	1 1111 0	01	C15	Write IBIST address
0	1 1111 1	01	C15	Read IBIST General
1	1 1111 1	01	C15	Write IBIST general
0	1 1111 0	11	C15	Read DBIST address

Table 8-2 Mapping of scan chain 15 address field to CP15 registers (continued)

Bit [38]	Bits[37:32]	Bits[31:30]	CP15 reg number	Meaning
1	1 1111 0	11	C15	Write DBIST address
0	1 1111 1	11	C15	Read DBIST general
1	1 1111 1	11	C15	Write DBIST general

The scan address decode overloads the existing functional decode logic that is used to access the CP15 registers during `MCR` and `MRC` instructions (see *ARM966E-S CP15 registers* on page 2-4.

The decode overload is performed as follows:

Bit [37]	Corresponds to Opcode 1 of an `MCR` or `MRC` instruction.
Bit [36:33]	Correspond to the CRn field of an `MCR` or `MRC` instruction.
Bit [32]	Corresponds to bit 0 of the Opcode 2 field of an `MCR or MRC` instruction.
Bits [2:1]	Of opcode 2 are tied to 00 during debug state.

The debug scan chain, SC15, only allows access to bit[0] of the OpCode2 field by default. To allow access to the Address and General BIST registers within CP15 Register 15, bits [31:30] of SC15 are overloaded as shown in Table 8-2 on page 8-7. There are certain restrictions with the overloading; when writing to the BIST General registers (i.e. writing a new seed), bits[31:30] of the seed are restricted to those values shown in Table 8-2 on page 8-7. These bits are not used in the BIST Address registers and so there are no debug restrictions when accessing these registers.

The ability to control the ARM966E-S system state through scan chain 15 provides extra debug visibility. For example, if the debugger wishes to compare the contents of an address that maps to the I-SRAM or D-SRAM, with the same address in external memory, the debugger can:

1. Load from the address with the SRAM enabled to return the SRAM data.
2. Disable the SRAM.
3. Perform the load again. The second load now accesses the AHB because the SRAM is disabled, returning the value from AHB memory.

8.4 Debug interface signals

There are four primary external signals associated with the debug interface:

- **DBGIEBKPT**, **DBGDEWPT**, and **EDBGRQ** are system requests for the ARM966E-S to enter debug state
- **DBGACK** is used by the ARM966E-S to flag back to the system that it is in debug state.

8.4.1 Entry into debug state on breakpoint

Any instruction being fetched from memory is sampled at the end of a cycle. To apply a breakpoint to that instruction, the breakpoint signal must be asserted by the end of the same cycle. This is shown in Figure 8-4 on page 8-10.

You can build External logic, such as additional breakpoint comparators, to extend the breakpoint functionality of the EmbeddedICE-RT logic. These outputs must be applied to the **DBGIEBKPT** input. This signal is ORed with the internally-generated breakpoint signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external breakpoints are possible.

A breakpointed instruction is allowed to enter the Execute stage of the pipeline, but any state change as a result of the instruction is prevented. All writes from previous instructions complete as normal.

The Decode cycle of the debug entry sequence occurs during the Execute cycle of the breakpointed instruction. The latched breakpoint signal forces the processor to start the debug sequence.

Figure 8-4 on page 8-10 shows breakpoint timing.

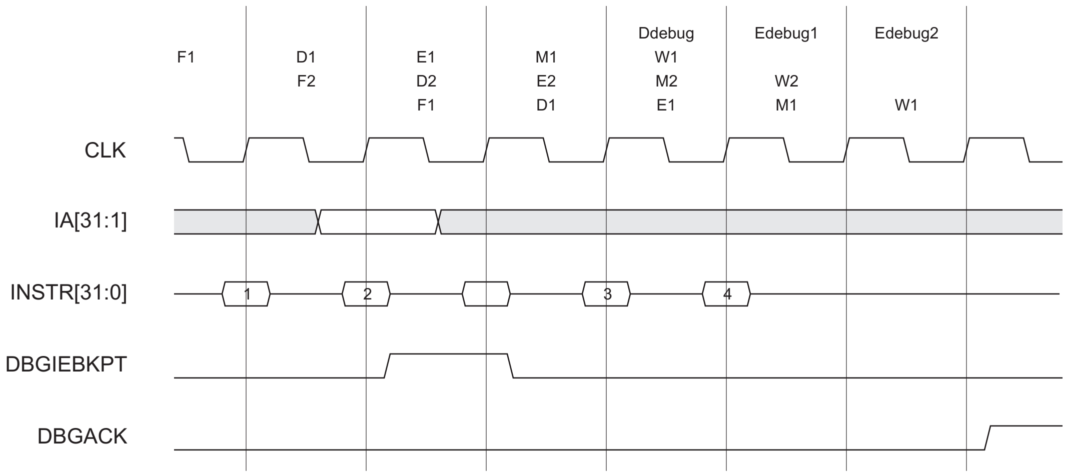


Figure 8-4 Breakpoint timing

8.4.2 Breakpoints and exceptions

A breakpointed instruction can have a Prefetch Abort associated with it. If so, the Prefetch Abort takes priority and the breakpoint is ignored. (If there is a prefetch abort, instruction data might have been invalid, the breakpoint might have been data-dependent, and as the data might be incorrect, the breakpoint might have been triggered incorrectly.)

`SWI` and undefined instructions are treated in the same way as any other instruction that might have a breakpoint set on it. Therefore, the breakpoint takes priority over the `SWI` or undefined instruction.

On an instruction boundary, if there is a breakpointed instruction and an interrupt (**nIRQ** or **nFIQ**), the interrupt is taken and the breakpointed instruction is discarded. When the interrupt is being serviced, the execution flow is returned to the original program. This means that the instruction that was previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters debug state when it reaches the Execute stage of the pipeline.

When the processor enters halt mode debug state, it is important that further interrupts do not affect the instructions executed. For this reason, as soon as the processor enters stop-mode debug state, interrupts are disabled, although the state of the I and F bits in the *Program Status Register (PSR)* are not affected.

8.4.3 Watchpoints

Entry into debug state following a watchpointed memory access is imprecise. This is necessary because of the nature of the pipeline.

External logic, such as external watchpoint comparators, can be built to extend the functionality of the EmbeddedICE-RT logic. Their output must be applied to the **DBGDEWPT** input. This signal is simply ORed with the internally-generated **Watchpoint** signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external watchpoints are possible.

After a watchpointed access, the next instruction in the processor pipeline is always allowed to complete execution. Where this instruction is a single-cycle data-processing instruction, entry into debug state is delayed for one cycle while the instruction completes. The timing of debug entry following a watchpointed load in this case is shown in Figure 8-5.

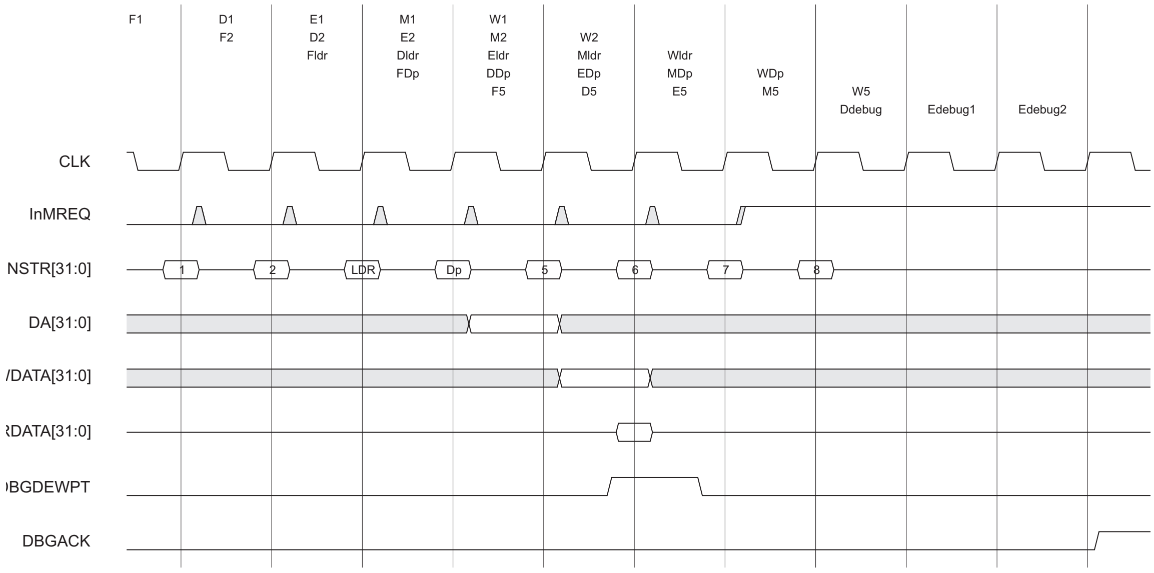


Figure 8-5 Watchpoint entry with data processing instruction

Note

Although instruction 5 enters the Execute stage, it is not executed, and there is no state update as a result of this instruction. When the debugging session is complete, normal continuation involves a return to instruction 5, the next instruction in the code sequence to be executed.

The instruction following the instruction that generated the watchpoint might have modified the *Program Counter (PC)*. If this happens, it is not possible to determine the instruction that caused the watchpoint. A timing diagram showing debug entry after a watchpoint where the next instruction is a branch is shown in Figure 8-6. However, it is always possible to restart the processor.

When the processor enters debug state, the ARM9E-S core is interrogated to determine its state. In the case of a watchpoint, the PC contains a value that is five instructions on from the address of the next instruction to be executed. Therefore, if on entry to debug state, in ARM state, the instruction `SUB PC, PC, #20` is scanned in and the processor restarted, execution flow returns to the next instruction in the code sequence.

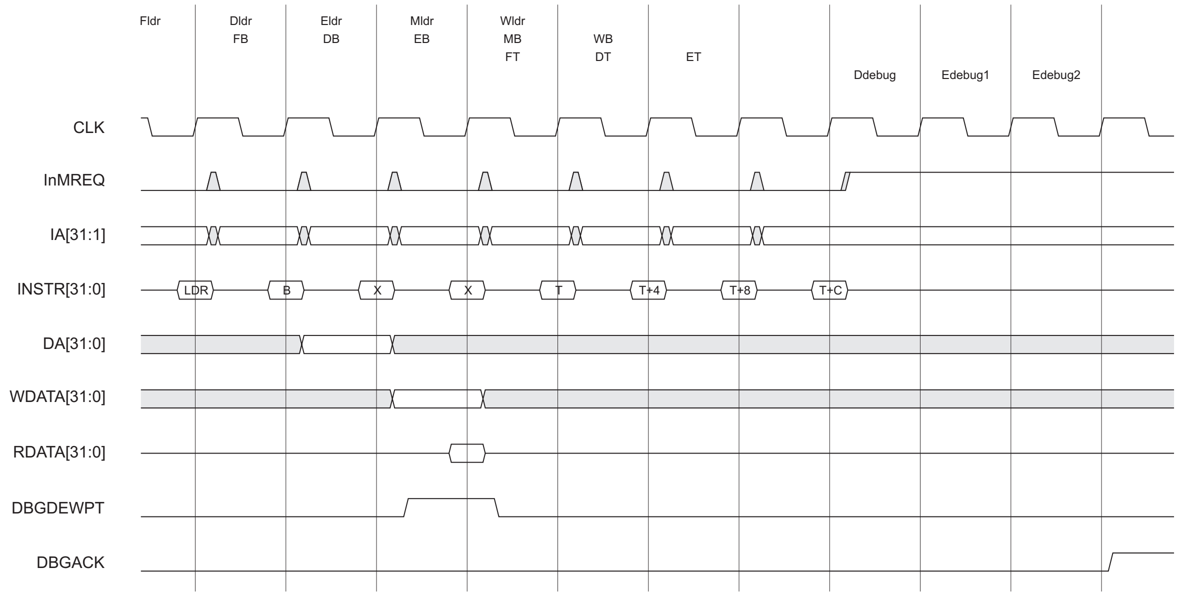


Figure 8-6 Watchpoint entry with branch

8.4.4 Watchpoints and exceptions

If there is an abort with the data access as well as a watchpoint, the watchpoint condition is latched, the exception entry sequence performed, and then the processor enters debug state. If there is an interrupt pending, again the ARM9E-S allows the exception entry sequence to occur and then enters debug state.

8.4.5 Debug request

A debug request can take place through the EmbeddedICE-RT logic or by asserting the **EDBGRQ** signal. The request is synchronized and passed to the processor. Debug request takes priority over any pending interrupt. Following synchronization, the core enters debug state when the instruction at the Execute stage of the pipeline is completed (when Memory and Write stages of the pipeline have completed). While waiting for the instruction to finish executing, no more instructions are issued to the Execute stage of the pipeline.

——— **Caution** ————

Asserting **EDBGRQ** in monitor mode results in unpredictable behavior.

———————————

8.4.6 Actions of the ARM9E-S in debug state

When the ARM9E-S is in debug state, both memory interfaces indicate internal cycles. This ensures that both the tightly-coupled SRAM within the ARM966E-S and the AHB interface are quiescent, allowing the rest of the AHB system to ignore the ARM9E-S and function as normal. Since the rest of the system continues operation, the ARM9E-S ignores aborts and interrupts.

The **nRESET** signal must be held stable during debug. If the system applies reset to the ARM966E-S (**nRESET** is driven LOW), the ARM9E-S changes state without the knowledge of the debugger.

8.5 ARM9E-S core clock domains

The ARM966E-S single clock, **CLK**, is qualified by two clock enables:

- **SYSCLKEN** controls access to the memory system
- **DBGTCKEN** controls debug operations.

During normal operation, **SYSCLKEN** conditions **CLK** to clock the core. When the ARM966E-S is in debug state, **DBGTCKEN** conditions **CLK** to clock the core.

8.6 Determining the core and system state

When the ARM966E-S is in debug state, you can examine the core and system state by forcing the load and store multiples into the instruction pipeline.

Before you can examine the core and system state, the debugger must determine whether the processor entered debug from Thumb state or ARM state, by examining bit 4 of the EmbeddedICE-RT debug status register. When bit 4 is HIGH, the core enters debug from Thumb state.

8.7 About the EmbeddedICE-RT

The ARM9E-S EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM9E-S core within the ARM966E-S.

EmbeddedICE-RT is programmed serially using the ARM9E-S TAP controller. Figure 8-7 illustrates the relationship between the core, EmbeddedICE-RT, and the TAP controller, showing only the signals that are pertinent to EmbeddedICE-RT.

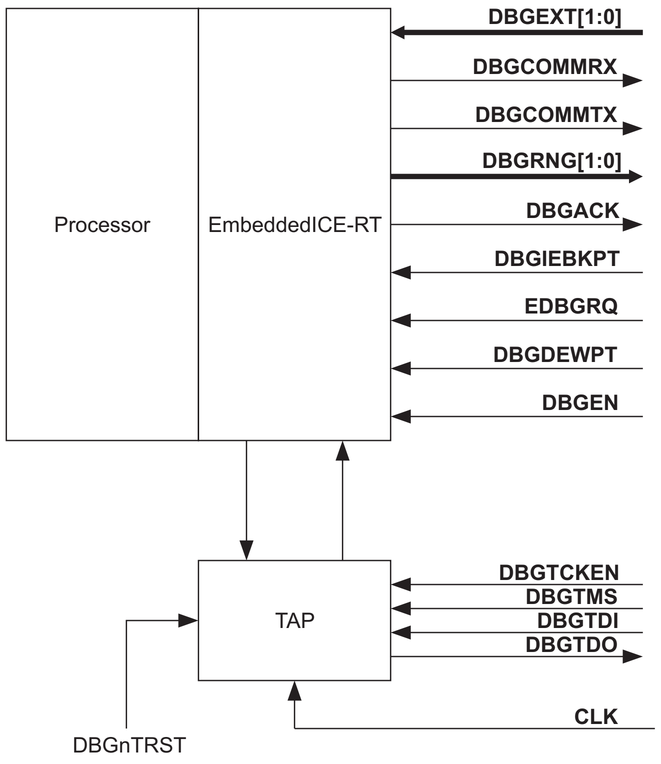


Figure 8-7 The ARM9E-S, TAP controller and EmbeddedICE-RT

The EmbeddedICE-RT logic comprises:

- two real-time watchpoint units
- two independent registers, the debug control register and the debug status register
- debug communications channel.

The debug control register and the debug status register provide overall control of EmbeddedICE-RT operation.

You can program one or both watchpoint units to halt the execution of instructions by the core. Execution halts when the values programmed into EmbeddedICE-RT match the values currently appearing on the address bus, data bus, and various control signals.

Note

Any bit can be masked so that its value does not affect the comparison.

Each watchpoint unit can be configured to be either a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). Watchpoints and breakpoints can be data-dependent.

8.8 Disabling EmbeddedICE-RT

You can disable EmbeddedICE-RT by setting the **DBGEN** input LOW.

——— **Caution** ———

Hard-wiring the **DBGEN** input LOW *permanently* disables debug access.

———————————

When **DBGEN** is LOW, it inhibits **DBGDEWPT**, **DBGIEBKPT**, and **EDBGRQ** to the core, and **DBGACK** from the ARM966E-S is always LOW.

8.9 The debug communications channel

The ARM9E-S EmbeddedICE-RT logic contains a communications channel for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The communications channel comprises:

- a 32-bit communications data read register
- a 32-bit wide communications data write register
- a 6-bit wide communications control register for synchronized handshaking between the processor and the asynchronous debugger.

These registers are located in fixed locations in the EmbeddedICE-RT logic register map and are accessed from the processor using `MCR` and `MRC` instructions to coprocessor 14.

In addition to the communications channel registers, the processor can access a 1-bit debug status register for use in the real-time debug configuration.

8.9.1 Debug communication channel registers

CP14 contains four registers, that have the following register allocations in coprocessor 14 as shown in Table 8-3.

Table 8-3 Coprocessor 14 register map

Register name	Register number	Notes
Communications channel status	C0	Read-only
Communications channel data read	C1	For reads
Communications channel data write	C1	For writes
Communications channel monitor mode debug status	C2	Read or write

8.9.2 Debug communications channel status register

The debug communications channel status register is read-only. It controls synchronized handshaking between the processor and the debugger. The debug communications channel status register is shown in Figure 8-8 on page 8-20.

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
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W	R

Figure 8-8 Debug communications channel status register

The function of each register bit is described here:

Bits 31:28 Contain a fixed pattern that denotes the EmbeddedICE-RT version number (in this case 0011).

Bits 27:2 Are reserved.

Bit 1 Denotes whether the communications data write register is available (from the viewpoint of the processor).

If, from the viewpoint of the processor, the communications data write register is free (W=0), new data can be written.

If the register is not free (W=1), the processor must poll until W=0.

From the viewpoint of the debugger, when W=1, new data is written that can be scanned out.

Bit 0 Denotes whether there is new data in the communications data read register.

If, from the viewpoint of the processor, R=1, there is some new data that can be read using an `MRC` instruction.

From the viewpoint of the debugger, if R=0, the communications data read register is free, and new data can be placed there through the scan chain. If R=1, this denotes that data previously placed there through the scan chain is not collected by the processor, and so the debugger must wait.

From the viewpoint of the debugger, the registers are accessed using the scan chain in the usual way. From the viewpoint of the processor, these registers are accessed using coprocessor register transfer instructions.

You must use the following instructions:

`MRC p14, 0, Rd, c0, c0`

This returns the debug communications control register into Rd.

`MCR p14, 0, Rn, c1, c0` This writes the value in Rn to the communications data write register.

`MRC p14, 0, Rd, c1, c0` This returns the debug data read register into Rd.

Because the Thumb instruction set does not contain coprocessor instructions, you are advised to access this data using `SWI` instructions when in Thumb state.

8.9.3 Communications channel monitor mode debug status register

The coprocessor 14 debug status register is provided for use by a debug monitor when the ARM9E-S is configured into monitor mode.

The coprocessor 14 debug status register is a 1-bit wide read or write register having the format shown in Figure 8-9.

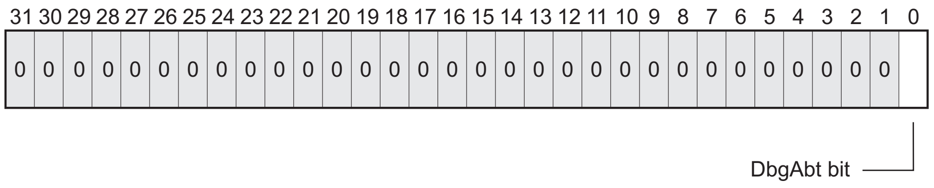


Figure 8-9 Coprocessor 14 debug status register format

Bit 0 of the register, the **DbgAbt** bit, indicates whether the processor took a Prefetch or Data Abort in the past because of a breakpoint or watchpoint. If the ARM9E-S core takes a Prefetch Abort as a result of a breakpoint or watchpoint, then the bit is set. If on a particular instruction or data fetch, both the debug abort and external abort signals are asserted, the external abort takes priority and the **DbgAbt** bit is not set. You can read or write the **DbgAbt** bit by means of `MRC` or `MCR` instructions.

This bit can be used by a real-time debug aware abort handler. This examines the **DbgAbt** bit to determine whether the abort is externally or internally generated. If the **DbgAbt** bit is set, the abort handler initiates communication with the debugger over the communications channel.

8.9.4 Communications via the communications channel

Messages can be sent and received using the communications channel as described in:

- *Sending a message to the debugger*
- *Receiving a message from the debugger* on page 8-22

Sending a message to the debugger

When the processor wishes to send a message to the debugger, it must check the communications data write register is free for use by finding out whether the W bit of the debug communications control register is clear.

The processor reads the debug communications control register to check status of the W bit.

- If W bit is clear, the communications data write register is clear.
- If the W bit is set, previously written data is not read by the debugger. The processor must continue to poll the control register until the W bit is clear.

When the W bit is clear, a message is written by a register transfer to coprocessor 14. Because the data transfer occurs from the processor to the communications data write register, the W bit is set in the debug communications control register.

The debugger sees both the R and W bits when it polls the debug communications control register through the JTAG interface. When the debugger sees that the W bit is set, it can read the communications data write register, and scan the data out. The action of reading this data register clears the debug communications control register W bit. At this point, the communications process can begin again.

Receiving a message from the debugger

Transferring a message from the debugger to the processor is similar to sending a message to the debugger. In this case, the debugger polls the R bit of the debug communications control register.

- if the R bit is LOW, the communications data read register is free, and data can be placed there for the processor to read
- if the R bit is set, previously deposited data is not yet collected, so the debugger must wait.

When the communications data read register is free, data is written there using the JTAG interface. The action of this write sets the R bit in the debug communications control register.

The processor polls the debug communications control register. If the R bit is set, there is data that can be read using an `MRC` instruction to coprocessor 14. The action of this load clears the R bit in the debug communications control register. When the debugger polls this register and sees that the R bit is clear, the data is taken, and the process can be repeated.

8.10 Monitor mode debug

The ARM9E-S within ARM966E-S contains logic that allows the debugging of a system without stopping the core entirely. This allows the continued servicing of critical interrupt routines while the core is being interrogated by the debugger. Setting bit 4 of the debug control register enables the real-time debug features of ARM9E-S. When this bit is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the ARM to enter abort mode, taking the Prefetch Abort or Data Abort vectors respectively. When the ARM is configured for real-time debugging you must be aware of the following restrictions:

- Breakpoints or watchpoints might not be data dependent. No support is provided for use of the range and chain functionality. Breakpoints or watchpoints can only be based on:
  - instruction or data addresses
  - external watchpoint conditioner (**DBGEXTERN**)
  - user or privileged mode access (**DnTRANS** and **InTRANS**)
  - read or write access (watchpoints)
  - access size (breakpoints, **ITBIT**, and watchpoints, **DMAS[1:0]**).
- The single-step hardware is not enabled.
- External breakpoints and watchpoints are not supported.
- The vector catching hardware can be used but must not be configured to catch the Prefetch or Data Abort exceptions.

Caution

No support is provided to mix halt mode and monitor mode debug functionality. When the core is configured into the monitor mode, asserting the external **EDBGRQ** signal results in unpredictable behavior. Setting the internal **EDBGRQ** bit results in unpredictable behavior.

When an abort is generated by the monitor mode it is recorded in the debug status register in coprocessor 14 (see *Communications channel monitor mode debug status register* on page 8-21).

Because the monitor mode debug does not put the ARM9E-S into debug state, it is necessary to change the contents of the watchpoint registers while external memory accesses are taking place, rather than being changed when in debug state. If the watchpoint registers are written to during an access, all matches from the affected watchpoint unit using the register being updated are disabled for the cycle of the update.

If there is a possibility of false matches occurring during changes to the watchpoint registers, caused by old data in some registers and new data in others, then you must:

1. Disable that watchpoint unit using the control register for that watchpoint unit
2. Change the other registers
3. Re-enable the watchpoint unit by rewriting the control register.

8.11 Debug additional reading

A more detailed description of the ARM9E-S debug features and JTAG interface is provided in the ARM9E-S Technical Reference Manual, Appendix D Debug in Depth.

Chapter 9
Embedded Trace Macrocell Interface

This chapter describes the ARM966E-S *Embedded Trace Macrocell* (ETM) interface. It contains the following sections:

9.1 About the ETM interface

The ARM966E-S supports the connection of an external *Embedded Trace Module* (ETM) to provide real time code tracing of the ARM966E-S in an embedded system.

The ETM interface is primarily *one way*. In order to provide code tracing, the ETM block must be able to monitor various ARM9E-S inputs and outputs. The required ARM9E-S inputs and outputs are collected and driven out from the ARM966E-S from the ETM interface registers, as shown in Figure 9-1.

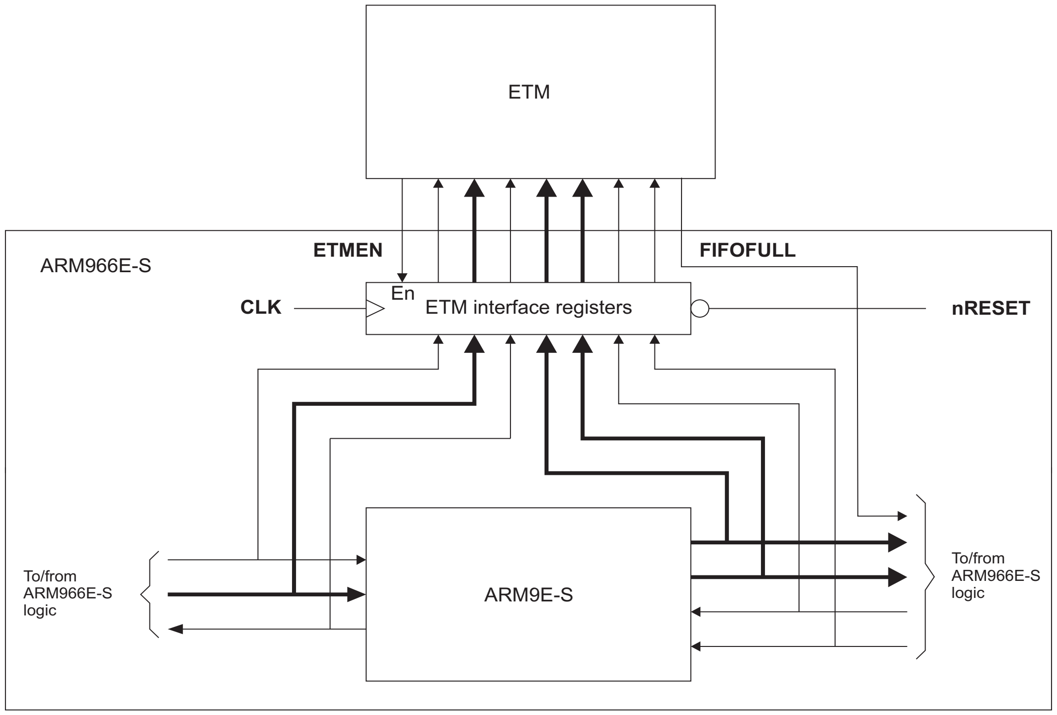


Figure 9-1 ARM966E-S ETM interface

The ETM interface outputs are pipelined by a single clock cycle to provide early output timing and to isolate any ETM input load from the critical ARM966E-S signals. The latency of the pipelined outputs does not effect ETM trace behavior, as all outputs are delayed by the same amount.

9.2 Enabling the ETM interface

The ETM interface on the ARM966E-S is enabled by the top-level pin **ETMEN**. When this input is HIGH, the ETM interface is enabled and the outputs are driven so that an external ETM can begin code tracing.

When the **ETMEN** input is driven LOW, the ETM interface outputs are held at their last value before the interface was disabled. At reset, all ETM interface outputs are reset LOW.

The **ETMEN** input is usually driven by the ETM, and driven HIGH once the ETM is programmed using its TAP controller.

——— **Note** ———

If an ETM is not used in an embedded ARM966E-S design, the **ETMEN** input must be tied LOW to save power.

9.3 ARM966E-S trace support features

The trace support uses the following features:

- *FIFOFULL*
- *Register 15, trace control register*
- *Register 1, Trace process identifier*.

9.3.1 FIFOFULL

The signal, **FIFOFULL**, is an input to the ARM966E-S driven by the ETM9. Whenever the programmed upper watermark of the ETM FIFO is filled, **FIFOFULL** is asserted. The ARM966E-S uses **FIFOFULL** to stall the ARM9E-S core, preventing trace loss. The ARM9E-S core remains stalled until **FIFOFULL** is deasserted.

The ARM966E-S can only stall on instruction boundaries enabling any current AHB transfers to complete. You must take this into consideration when programming the ETM FIFO watermark. If the current instruction is either a `LDM` or a `STM`, the FIFO might have to accept up to 16 words after the assertion of **FIFOFULL**.

——— **Note** ———

Using **FIFOFULL** to stall the ARM966E-S affects real-time operating performance.

———————————

9.3.2 Register 15, trace control register

The trace control register allows the masking of interrupts during trace. This register allows **nIRQ** and **nFIQ** interrupt priority over **FIFOFULL** to be programmed. The operation of this register is described in *Register 15, Test* on page 2-9.

9.3.3 Register 1, Trace process identifier

The ARM966E-S contains a trace process identifier register that allows Real-time Trace tools to identify the currently executing process in multi-tasking environments. The operation of this register is described in *Register 13, Trace process identifier* on page 2-9.

Chapter 10
Test Support

This chapter describes the test methodology employed for the ARM966E-S synthesized logic and tightly-coupled SRAM. It contains the following sections:

10.1 About the ARM966E-S test methodology

To achieve a high level of fault coverage, scan insertion and ATPG techniques are used on the ARM9E-S core and ARM966E-S control logic as part of the synthesis flow. BIST is used to provide high fault coverage of the compiled SRAM.

10.2 Scan insertion and ATPG

This technique is covered in detail in the *ARM966E-S Implementation Guide*. Scan insertion requires that all register elements are replaced by scannable versions that are then connected up into a number of large scan chains. These scan chains are used to set up data patterns on the combinatorial logic between the registers, and capture the logic outputs. The logic outputs are then scanned out while the next data pattern is scanned in.

Automatic Test Pattern Generation (ATPG) tools are used to create the necessary scan patterns to test the logic, when the scan insertion has been performed. This technique enables very high fault coverage to be achieved for the standard cell combinatorial logic, typically in the 95-99% range.

Scan insertion does have an impact on the area and performance of the synthesized design, due to the larger scan register elements and the serial routing between them. However, to minimize these effects, the scan insertion is performed early in the synthesis cycle and the design re-optimized with the scan elements in place.

10.2.1 ARM966E-S INTEST wrapper

To facilitate testing of the shadow logic between the ARM966E-S scan chains and the scan chains in an OEM ASIC, a synthesis option allows an INTEST wrapper to be inserted into the ARM966E-S. The INTEST wrapper is a scan chain around the boundary of the ARM966E-S, connecting to all input and output pins.

Note

- Shadow logic is logic that is not ordinarily tested.
- The INTEST wrapper is only required for embedded ARM966E-S.
- The order of this scan chain is predetermined and must be maintained through synthesis and place and route of the macrocell.

10.3 BIST of tightly-coupled SRAM

Adding a simple memory test controller allows an exhaustive test of the memory arrays to be performed. BIST test is activated by an `MCR` to the CP15 BIST control register and can be run on one or both of the I-SRAM and D-SRAM simultaneously.

When a BIST test is performed on an SRAM, the functional enable for that SRAM is automatically disabled, forcing all memory accesses to that SRAM address space to go to the AHB. This enables BIST tests to be run in the background. For instance, the instruction SRAM can be BIST tested, while code is executed over the AHB.

Full programmer control over the BIST mechanism is achieved through five registers that are mapped to CP15 register 15 address space. For details of the `MCR` or `MRC` instructions used to access these registers, see *Register 15, Test* on page 2-9. Access to these registers is also available in debug mode, see *ARM966E-S scan chain 15* on page 8-7.

10.3.1 BIST control register

This controls the operation of the SRAM memory BIST. Before initiating a BIST test, a `MCR` is first performed to the BIST control register to set up the size of the test and enable the SRAM to be tested. A further `MCR` is required to initiate the test.

The current status of a BIST test and result of a completed test can be accessed by performing an `MRC` to the BIST control register. This returns flags to indicate that a test is:

- running
- paused
- failed
- completed.

In addition to returning the state for the size of the test and SRAM enable status, having completed a BIST test, the BIST enable must first be cleared by writing to the BIST control register if the SRAM is to be used by you for functional operation. The SRAM must then be re-enabled by writing to CP15 register 1. This is necessary as the BIST test enable automatically clears the functional enable.

Note

Clearing the functional SRAM enable when BIST is enabled prevents the programmer from trying to run from tightly coupled SRAM following a BIST test, without having first reprogrammed the SRAM. This is necessary as the BIST algorithm corrupts all tested SRAM locations.

10.3.2 BIST address and general registers

The BIST control register enables standard BIST operations to be performed on each SRAM and the size of the test to be specified. Additional registers are required however, to provide the following functionality:

- testing of the BIST hardware
- changing the seed data for a BIST test
- providing a nonzero starting address for a BIST test
- peek and poke of the SRAM
- returning an address location for a failed BIST test
- returning failed data from the failing address location.

This additional functionality is most useful for debugging faulty silicon during production test. The exception to this is the start address for a BIST test. It is possible that BIST of the SRAM is performed periodically during program execution, the memory being tested in smaller pieces rather than in one go. This requires a start address that is incremented by the size of the test each time a test is activated.

Table 10-1 and Table 10-2 on page 10-6 show how the registers are used. The pause bits from the BIST control register provide extra decode of these registers.

Table 10-1 Instruction BIST address and general registers

BIST register	IBIST pause	Read	Write
IBIST address register	0	IBIST fail address	IBIST start address
IBIST address register	1	IBIST fail address	IBIST peek/poke address
IBIST general register	0	IBIST fail data	IBIST seed data
IBIST general register	1	IBIST peek data	IBIST poke data

Table 10-2 Data BIST address and general registers

BIST register	IBIST pause	Read	Write
DBIST address register	0	DBIST fail address	DBIST start address
DBIST address register	1	DBIST fail address	DBIST peek/poke address
DBIST general register	0	DBIST fail data	DBIST seed data
DBIST general register	1	DBIST peek data	DBIST poke data

10.3.3 Pause modes

The suggested production test sequence for the SRAM is:

1. Test each SRAM using a full test.
2. Test the BIST hardware for each SRAM. To allow testing of the BIST hardware, a pause mechanism enables the BIST test to be halted and data within the SRAM to be corrupted. The sequence for this is:
   a. Writing the address for the location to be corrupted with a `MCR` to the relevant BIST address register.
   b. Writing the corrupted data using a `MCR` to the BIST general register.
   c. Restarting the test by an `MCR` to the BIST control register.
   d. Checking that the corrupted data causes the test to fail by reading the failed address and data from the BIST address and general registers.

In addition to controlling the addressing within the address and general registers, the pause bit also controls the progression of the BIST algorithm as follows:

- *Auto pause*
- *User pause* on page 10-7

Auto pause

If the pause bit is set in the BIST control register before the test is activated, the test runs in auto pause mode. The BIST test pauses at predetermined points of the BIST algorithm, for instance when the algorithm has reached the top or the bottom of the memory array being tested.

The programmer can poll the BIST control register to detect when a test has paused (the running flag is LOW). Data can then be corrupted as detailed above, before restarting the BIST test.

User pause

If the pause bit is clear when the test is activated, the test is run in user pause mode. The BIST algorithm is paused by an MCR to the BIST control register, setting the pause bit for the SRAM being tested. The SRAM contents are then corrupted as previously. This stops the BIST algorithm at a potentially unknown point, resulting in the possibility that the corrupted data is overwritten by the BIST algorithm and therefore not cause a test to fail.

Note

User pause mode is provided for production test debugging to shorten a test by pausing the algorithm early. The auto pause mechanism is recommended to provide or BIST hardware testing for all other occasions.

Chapter 11
Instruction cycle timings

This chapter describes the instruction cycle timings for the ARM966E-S. It contains the following sections:

11.1 Introduction to instruction cycle timings

The ARM9E-S core within the ARM966E-S implements a pipelined architecture where several instructions in different pipeline stages overlap. The instruction cycle timing tables in the *ARM9E-S Technical Reference Manual* show the number of cycles required by an instruction, once it has reached the execute stage of the ARM9E-S core pipeline.

The instruction cycle timing numbers quoted in the *ARM9E-S Technical Reference Manual* assume that the ARM9E-S is permanently enabled with the **CLKEN** input tied HIGH. This implies that both instruction and data memory connected to the ARM9E-S are able to perform zero wait state responses to all accesses.

In a system such as the ARM966E-S, the **CLKEN** input to the ARM9E-S core might be pulled LOW to stall the processor until the memory system is able to respond to the access. These stall cycles must be taken into account when calculating the ARM966E-S instruction cycle timings.

Stall cycles are introduced by the ARM966E-S system controller in the following circumstances:

- the internal SRAM cannot always be accessed in a single cycle
- the access requires an AHB transfer
- the write buffer is full or being drained.

This chapter describes the cycle counts for both normal operation and the above circumstances.

11.2 When stall cycles do not occur

Before describing the various stall cycle scenarios, it is useful to consider the circumstances where the ARM9E-S core can run within the ARM966E-S with no stall cycles introduced by the system controller. When this is the case, the ARM966E-S is running at peak efficiency and the instruction cycles exactly match those quoted in the *ARM9E-S Technical Reference Manual.*

The fundamental requirement for no stall cycles is that the I-SRAM is enabled and the necessary instructions have been previously programmed into it. Additionally, if the D-SRAM is enabled, it can be accessed for reads without incurring a stall penalty, even if the I-SRAM is being simultaneously accessed for an instruction fetch.

When a write is performed, the access can be zero stall if the write buffer is used and there is space available. If the write is to the D-SRAM, the write is a single cycle in most circumstances, and any store multiple to the D-SRAM can be executed as one write per cycle. As long as these writes are not to the I-SRAM address space, instruction fetches from the I-SRAM can be performed simultaneously without incurring a stall penalty.

To maximize performance, it is therefore desirable to ensure that frequently accessed code is preloaded into the I-SRAM and that data accesses map to the D-SRAM address space. It is also advisable to enable the write buffer and use bufferable areas of memory where possible, when AHB writes are performed.

Note

If the data interface of the ARM9E-S core accesses the I-SRAM memory, in most cases stall cycles are incurred. An example of where this type of access is unavoidable, is the fetching of inline code literals from the I-SRAM.

11.3 Tightly-coupled SRAM cycles

This section describes the stall cycle counts for accesses to one or both of the SRAMs. The circumstances where the internal tightly-coupled SRAM can stall are detailed in *SRAM stall cycles* on page 4-3.

Table 11-1 lists the stall cycles incurred when accessing the I-SRAM. In most cases the data accesses are to the D-SRAM so the stall penalties listed are not incurred.

Table 11-1 I-SRAM access

Instruction sequence	Stalls	Comment
Single instruction fetch	0	Assuming no data interface access to I-SRAM
Sequential instruction fetch	0	Assuming no data interface access to I-SRAM
`LDR`, no instruction fetch	0	Assuming no previous I-SRAM store
`LDR`, simultaneous instruction fetch	1	Simultaneous instruction fetch request causes stall of LDR for 1 cycle
`LDM`, instruction fetch in parallel with final load	1	Simultaneous instruction fetch request at end of `LDM` causes stall
`STR`, no instruction fetch	0	Assuming no previous ISRAM store
`STR` simultaneous instruction fetch	2	Two cycle write performed prior to instruction fetch
`STR` followed by instruction fetch	1	Stall occurs due to second cycle of store
`STR` followed by simultaneous, instruction fetch `LDR`	1	Stall occurs due to second cycle of store
`STR` followed by simultaneous instruction fetch, `STR`	2	Stall due to second cycle of second store plus instruction fetch request
`STR` followed by `LDR`/`STR`, no instruction fetch	1	Stall due to second cycle of store
`STM`, instruction fetch in parallel with final store	2	Simultaneous instruction fetch request must wait for second cycle of final write to complete

The D-SRAM can only be accessed by the ARM9E-S data interface so there are no simultaneous access contentions as found in the I-SRAM. Table 11-2 shows the stall cycles that can occur when accessing the D-SRAM.

Table 11-2 D-SRAM access

Data access	Stalls	Comment
`LDR`	0	D-SRAM provides single cycle response
`LDM`	0	D-SRAM provides single cycle response to each word
`LDR`/`LDM` followed by any load or store	0	D-SRAM provides single cycle response
`STR`	0	Assuming no following load
`STM`	0	Assuming no following load
`STR`/`STM` followed by `STR`/`STM`	0	Pipelined addresses allow back-to-back stores or store multiples
`STR`/`STM` followed by `LDR`/`LDM`	1	Second cycle of write causes stall before load can be performed

Note

All internal SRAM stall cycles are in terms of the **CLK** and are therefore not affected by the speed of the external AHB interface.

11.4 AHB memory access cycles

When a read or non-bufferable write access to the AHB is performed, stall cycles are introduced. The number of **CLK** stall cycles incurred depends on:

- the clocking ratio of the AHB interface
- the type of access being performed
- if there are further accesses to be performed.

Before an AHB transfer can be initiated, the ARM966E-S must be the granted bus master. The cycle calculations in this section assume that the ARM966E-S is granted and that it is the default bus master.

11.4.1 Synchronization penalty

At the start of an AHB access, the BIU within the ARM966E-S must wait for the first rising edge of **HCLK** (the **HCLKEN** input is true) before it can broadcast the necessary AHB control and address information for the access. This delay is the synchronization penalty. The best case is that in the cycle when the AHB access is requested, the **HCLKEN** input is HIGH, incurring a zero cycle synchronization penalty. The worst case is where the **HCLKEN** is HIGH in the cycle before the AHB access is required. The ARM966E-S must then wait until the next assertion of **HCLKEN** which is *R*-1 cycles later, where *R* is the **CLK** to **HCLK** ratio:

- Best case synchronization penalty is 0 **CLK** cycles
- Worst case synchronization penalty is *R*-1 **CLK** cycles, where *R* = 1, 2, 3, 4, 5, 6, 7, 8 for example.

If the AHB must be accessed for two transfers that were requested simultaneously by the ARM9E-S core (that is, a simultaneous instruction fetch and data load), the BIU stays synchronized after the first transfer so that the penalty is only incurred for the first access. If the transfer is part of a burst (`STM/LDM`) or a sequential instruction fetch sequence, again the BIU stays synchronized between each transfer to minimize synchronization penalty.

Note

If the clock ratio *R*=1 and the **HCLKEN** input to the ARM966E-S is tied HIGH then no synchronization penalty is incurred when accessing the AHB.

11.4.2 AHB transfer types

The ARM966E-S can perform IDLE, NONSEQ, and SEQ transfers. Depending on the implementation of the AHB system to which the ARM966E-S is connected, a varying number of **HCLK** cycles are required for the NONSEQ and SEQ transfers. Typically, a NONSEQ cycle requires a two-cycle response from the selected slave, whereas a SEQ cycle can be handled in a single cycle. The IDLE cycle takes one **HCLK** cycle by definition.

For each **HCLK** cycle required by the AHB transfer, *R* internal **CLK** cycles are taken. The AHB transfer cycles are converted to **CLK** by multiplying by *R,* the **CLK** to **HCLK** ratio, as shown in Table 11-3

Table 11-3 Key to tables

Symbol	Meaning in terms of CLK cycles
Sync	Worst-case synchronization penalty (= *R*-1)
S	**HCLK** cycles required for a SEQ transfer x R
N	**HCLK** cycles required for a NONSEQ transfer x R
I	**HCLK** cycle required for an IDLE cycle (=*R*)
n	Number of words accessed by the transfer

Table 11-4 lists the types of AHB transfers performed by the ARM966E-S and the number of **CLK** cycles required to perform them. This table indicates cycles where the ARM9E-S core must be stalled until one or more AHB accesses have completed, that is, for reads and unbuffered writes.

Table 11-4 AHB read and unbuffered write transfer cycles

AHB access	Cycles	Comment
Start of sequential instruction fetch of n words	Sync+N(n+I)	Assumes no AHB load or store activity.
Nonsequential instruction fetch	Sync+N+I	Assumes no AHB load or store activity.
Nonsequential instruction fetch follows sequential instruction fetch	N+I	Assumes no AHB load or store activity.
Single LDR or STR	Sync+N+I	Assumes no AHB instruction fetch.

Table 11-4 AHB read and unbuffered write transfer cycles (continued)

AHB access	Cycles	Comment
Back-to-back `LDR/LDR`, `LDR/STR`, `STR/STR`, `STR/LDR`	Sync+2(N+I)	Assumes no AHB instruction fetch. Synchronization penalty for first transfer only.
Simultaneous `LDR/STR` and instruction fetch	Sync+2N+I	Optimization replaces IDLE cycle after load/store with NONSEQ of instruction fetch.
`STM` of n words	Sync+N+(n-1)S+I	Assumes no AHB instruction fetch.
`STM` of n words, simultaneous instruction fetch at end	Sync+2N+(n-1)S+I	Optimization replaces IDLE cycle after final stored word with NONSEQ of instruction fetch.
`STM` of n words crosses 1KB region	Sync+2N+(n-2)S+2I	Assumes no AHB instruction fetch, sequentiality broken on boundary.
`LDM` of n words	Sync+N+(n-1)S+2I	Assumes no AHB instruction fetch. LDM requires extra IDLE at end of transfer to re-sample core interface.
`LDM` of n words, simultaneous instruction fetch at end	Sync+2N+(n-1)S+2I	Optimization replaces second IDLE cycle after final loaded word with NONSEQ of instruction fetch.
`LDM` of n words crosses 1KB region	Sync+2N+(n-2)S+4I	Assumes no AHB instruction fetch, sequentiality broken on boundary.

See *AHB bus master interface* on page 6-7 for diagrams of the cycles listed in Table 11-4 on page 11-7.

Table 11-5 on page 11-9 shows the cycles required to perform buffered writes. These writes usually take place in parallel with program execution and the ARM9E-S core is not stalled while the buffered writes take place. However, whenever a load or instruction fetch to the AHB is required, the core is stalled and the write buffer drained before program execution can continue.

Table 11-5 AHB buffered writes cycles

AHB access	Cycles	Comment
Single `STR`	Sync+N+I	Assumes no following AHB instruction fetch
Back-to-back `STR`/`STR`	Sync+2(N+I)	Assumes no following AHB instruction fetch
`STM`	Sync+N+(n-1)S+I	Assumes no following AHB instruction fetch
Last `STR` in write buffer drain followed by unbuffered data access	2(N+I)	Core stalled until write buffer empty and data access has been performed
Last `STR` in write buffer drain followed by instruction fetch	2N+I	Optimization replaces IDLE cycle after store with NONSEQ of instruction fetch

11.5 Interrupt latency calculation

The ARM9E-S has a worst-case interrupt latency figure that is listed in the *ARM9E-S Technical Reference Manual*. The number quoted assumes that the **CLKEN** input to the core is HIGH, ensuring no stall cycles.

In the ARM966E-S, the best-case figure could match the latency quoted for the ARM9E-S core, if the necessary data and instructions were already in the D-SRAM and I-SRAM respectively. However, when calculating the worst-case figure, it must be assumed that the necessary data and instructions are not in the tightly-coupled SRAM and must therefore be accessed over the AHB.

In addition, the worst-case is where the write buffer is full when the interrupt occurs, requiring that the buffer drain is added to the interrupt latency calculation. The worst-case sequence for the write buffer is that five nonsequential words are to be written.

For the ARM9E-S core, the worst-case interrupt latency occurs when the longest `LDM` incurs a Data Abort. However, for the ARM966E-S, this is the longest `LDM` without a Data Abort. The `LDM` that incurs a Data Abort takes extra **CLK** cycles in the core, but the abort vector is usually in the tightly-coupled SRAM and can be returned without introducing the extra stall cycles of an AHB access.

The longest `LDM` without the Data Abort is one that loads all the registers, including the PC, that causes a branch to a destination anywhere in memory. The branch destination must therefore be assumed to be outside of the tightly-coupled SRAM. The loads to the PC address and (PC+1) cause additional AHB accesses to produce the worst-case interrupt latency.

Using the symbols defined in Table 11-3 on page 11-7, the worst-case interrupt latency can be summarized in Table 11-6.

Table 11-6 Interrupt latency cycle summary

AHB access	Cycles	Comment
Write buffer drain	Sync+ 5(N+I)	**FIQ** asserted, first data transfer requested, write buffer drain stalls core.
`LDM` (r0-pc) crosses 1KB boundary	2N+14 S+4 I	No instruction fetch at end due to core pipeline bubble to calculate pc
Instruction fetch of (pc)	Sync+N+I	Synchronization lost due to core internal cycle, no AHB request
Sequential instruction fetch of (pc+1)	N+I	Synchronization retained

The `LDM` (r0-pc) must complete before the interrupt vector is fetched. The write buffer drain must be added to this, in addition to assuming that the LDM (r0-pc) crosses a 1KB boundary.

The calculation assumes that once the interrupt has entered the Decode stage of the ARM9E-S pipeline following the instruction fetch to (pc+1), the subsequent fetches to the interrupt vector are serviced by the tightly-coupled SRAM, requiring a further three **CLK** cycles for the FIQ handler to enter execute. (This is not the case if the interrupt vector resides at the HIVECS location of `0xFFFF 0000`. This requires AHB access.)

The cycles from Table 11-6 on page 11-10 are added to the three **CLK** cycles from the tightly-coupled SRAM to produce the interrupt latency equation:

Interrupt latency **CLK** = 2Sync+9N+14S+2B+11I+3

Rewriting in terms of *R*, NONSEQ, SEQ and IDLE the equation simplifies to:

Interrupt latency **CLK** =*R* (9 NONSEQ+14SEQ+13)+1

where IDLE=BUSY=*R* as this is a single **HCLK** cycle by definition.

The number of **CLK** cycles latency can now be derived for different AHB clocking ratios and for the differing AHB slave responses that might exist in the AHB system to which the ARM966E-S interfaces. Table 11-7 gives examples of interrupt latency for systems with different **CLK** to **HCLK** ratios. For each system, slaves can have different response times to NONSEQ and SEQ transfers. Table 11-7 gives some examples of different slave responses and the resultant interrupt latency in **CLK** cycles.

Table 11-7 Interrupt latency calculated examples

CLK to HCLK Ratio - R	Latency when NONSEQ = 1, SEQ = 1	Latency when NONSEQ= 2, SEQ = 1	Latency when NONSEQ = 2, SEQ = 2
1	37	46	60
2	73	91	119
3	109	136	178
4	145	181	237

Appendix A
Signal Descriptions

This appendix describes the ARM966E-S signals. It contains the following sections:

A.1 Signal properties and requirements

In order to ensure ease of integration of the ARM966E-S into embedded applications and to simplify synthesis flow, the following design techniques have been used:

- a single rising edge clock times all activity
- all signals and buses are unidirectional
- all inputs are required to be synchronous to the single clock.

These techniques simplify the definition of the top-level ARM966E-S signals because all outputs change from the rising edge and all inputs are sampled with the rising edge of the clock. In addition, all signals are either input or output only, as bidirectional signals are not used.

Note

Asynchronous signals (for example interrupt sources) must first be synchronized by external logic before being applied to the ARM966E-S macrocell.

A.2 Clock interface signals

Table A-1 describes the ARM966E-S clock interface signals.

Table A-1 Clock interface signals

Name	Direction	Description
CLK System clock	Input	This clock times all operations in the ARM966E-S design. All outputs change from the rising edge and all inputs are sampled on the rising edge. The clock might be stretched in either phase. Through the use of the **HCLKEN** signal, this clock also times AHB operations. Through the use of the **DBGTCKEN** signal, this clock also times debug operations.
HCLKEN	Input	Synchronous enable for AHB transfers. When HIGH indicates that the next rising edge of **CLK** is also a rising edge of **HCLK** in the AHB system in which the ARM966E-S is embedded. **HCLK** must be tied HIGH in systems where **CLK** and **HCLK** are intended to be the same frequency.
DBGTCKEN	Input	Synchronous enable for debug logic accessed by the JTAG interface. When HIGH on the rising edge of **CLK** the debug logic is able to advance.
HRESETn Not reset	Input	Asynchronously asserted LOW input used to initialize the ARM966E-S system state. Synchronously de-asserted.

A.3 AHB signals

Table A-2 describes the ARM966E-S AHB signals.

Table A-2 AHB signals

Name	Direction	Description
HADDR[31:0] Address bus	Output	The 32-bit AHB system address bus.
HTRANS[1:0] Transfer type	Output	Indicates the type of ARM966E-S transfer, which can be IDLE (00), NONSEQ (10), or SEQ (11).
HWRITE Transfer direction	Output	When HIGH indicates a write transfer. When LOW indicates a read transfer.
HSIZE[2:0] Transfer size	Output	Indicates the size of an ARM966E-S transfer, which can be Byte (000), Half-word (001) or Word (010).
HBURST[2:0] Burst type	Output	Indicates if the transfer forms part of a burst. The ARM966E-S supports SINGLE transfer (000) and INCRemental burst of unspecified length (001).
HPROT[3:0] Protection control	Output	Indicates that the ARM966E-S transfer is an opcode fetch (0--0) or a data access (0--1) or a User mode access (0-0-) or a Supervisor mode access (0-1-). Also indicates that an access is not bufferable (00--) or bufferable (01--). Bit [3] is driven to 0 indicating not cacheable.
HWDATA[31:0] Write data bus	Output	The 32-bit write data bus is used to transfer data from the ARM966E-S to a selected bus slave during write operations.
HRDATA[31:0] Read data bus	Input	The 32-bit read data bus is used to transfer data from a selected bus slave to the ARM966E-S during read operations.
HREADY Transfer done	Input	When HIGH indicates that a transfer has finished on the bus. This signal can be driven LOW by the selected bus slave to extend a transfer.
HRESP[1:0] Transfer response	Input	The transfer response from the selected slave provides additional information on the status of the transfer. The response can be OKAY (00), ERROR (01), RETRY (10), or SPLIT (11).

Table A-2 AHB signals (continued)

Name	Direction	Description
HBUSREQ Bus request	Output	Indicates that the ARM966E-S requires the bus.
HLOCK Request locked transfers	Output	When HIGH, indicates that the ARM966E-S requires locked access to the bus and no other master is granted until this signal has gone LOW. Asserted by the ARM966E-S when executing SWP instructions to AHB address space.
HGRANT Bus grant	Input	Indicates that the ARM966E-S is currently the highest priority master. Ownership of the address and control signals changes at the end of a transfer when **HREADY** is HIGH, so the ARM966E-S gets access to the bus when both **HREADY** and **HGRANT** are HIGH.

A.4 Coprocessor interface signals

Table A-3 describes the ARM966E-S coprocessor interface signals.

Table A-3 Coprocessor interface signals

Name	Direction	Description
CPCLKEN Coprocessor clock enable	Output	Synchronous enable for coprocessor pipeline follower. When HIGH on the rising edge of **CLK** the pipeline follower logic is able to advance.
CPINSTR[31:0] Coprocessor instruction data	Output	The 32-bit coprocessor instruction bus over which instructions are transferred to the coprocessor pipeline follower.
CPDOUT[31:0] Coprocessor read data	Output	The 32-bit coprocessor read data bus for transferring data to the coprocessor.
CPDIN[31:0] Coprocessor write data	Input	The 32-bit coprocessor write data bus for transferring data from the coprocessor.
CPPASS	Output	Indicates that there is a coprocessor instruction in the Execute stage of the pipeline, and it must be executed.
CPLATECANCEL	Output	If HIGH during the first memory cycle of a coprocessor instruction, then the coprocessor must cancel the instruction without changing any internal state. This signal is only asserted in cycles where the previous instruction caused a Data Abort to occur.
CHSDE[1:0] Coprocessor handshake decode	Input	The handshake signals from the Decode stage of the coprocessor's pipeline follower. Indicates ABSENT (10), WAIT (00), GO (01), or LAST (11).
CHSEX[1:0] Coprocessor handshake execute	Input	The handshake signals from the Execute stage of the coprocessor's pipeline follower. Indicates ABSENT (10), WAIT (00), GO (01), or LAST (11).

Table A-3 Coprocessor interface signals (continued)

Name	Direction	Description
CPTBIT Coprocessor instruction Thumb bit	Output	When HIGH indicates that the ARM966E-S in is Thumb state. When LOW indicates that the ARM966E-S is in ARM state. Sampled by the coprocessor pipeline follower.
nCPMREQ Not coprocessor instruction request	Output	When LOW on the rising edge of **CLK** and **CPCLKEN** is HIGH, the instruction on **CPINSTR** must enter the coprocessor pipeline.
nCPTRANS Not coprocessor memory translate	Output	When LOW indicates that the ARM966E-S is in User mode. When HIGH indicates that the ARM966E-S is in privileged mode. Sampled by the coprocessor pipeline follower.

A.5 Debug signals

Table A-4 describes the ARM966E-S debug signals.

Table A-4 Debug signals

Name	Direction	Description
DBGIR[3:0] TAP controller instruction register	Output	These four bits reflect the current instruction loaded into the TAP controller control register. These bits change when the TAP controller is in the UPDATE-IR state.
DBGnTRST Not test reset	Input	This is the active low reset signal for the EmbeddedICE internal state. This signal is a level sensitive asychronous reset signal.
DBGnTDOEN Not **DBGTDO** enable	Output	When LOW, this signal denotes that the serial data is being driven out of the **DBGTDO** output. Normally used as an output enable for a **DBGTDO** pin in a packaged part.
DBGSCREG[4:0]	Output	These five bits reflect the ID number of the scan chain currently selected by the TAP controller. These bits change when the TAP controller is in the UPDATE-DR state.
DBGSDIN External scan chain serial input data	Output	Contains the serial data to be applied to an external scan chain.
DBGSDOUT External scan chain serial data output	Input	Contains the serial data out of an external scan chain. When an external scan chain is not connected, this signal must be tied LOW.
DBGTAPSM[3:0] TAP controller state machine	Output	This bus reflects the current state of the TAP controller state machine.
DBGTDI	Input	Test data input for debug logic.
DBGTDO	Output	Test data output from debug logic.
DBGTMS	Input	Test mode select for TAP controller.
COMMRX Communications channel receive	Output	When HIGH denotes that the communications channel receive buffer contains valid data waiting to be read.

Table A-4 Debug signals (continued)

Name	Direction	Description
COMMTX Communications channel transmit	Output	When HIGH, denotes that the comms channel transmit buffer is empty.
DBGACK Debug acknowledge	Output	When HIGH indicates that the processor is in debug state.
DBGEN Debug enable	Input	Enables the debug features of the processor. This signal must be tied LOW if debug is not required.
DBGRQI Internal debug request	Output	Represents the debug request signal that is presented to the core debug logic. This is a combination of **EDBGRQ** and bit 1 of the debug control register.
EDBGRQ External debug request	Input	An external debugger forces the processor into debug state by asserting this signal.
DBGEXT[1:0] EmbeddedICE external input	Input	Input to the EmbeddedICE-RT logic allows breakpoints/watchpoints to be dependent on external conditions.
DBGINSTREXEC Instruction executed	Output	Indicates that the instruction in the Execute stage of the processor pipeline has been executed.
DBGRNG[1:0] EmbeddedICE Rangeout	Output	Indicates that the corresponding EmbeddedICE-RT watchpoint register has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint enable control bit.

Table A-4 Debug signals (continued)

Name	Direction	Description
TAPID[31:0] Boundary scan ID code	Input	Specifies the ID code value shifted out on **DBGTDO** when the IDCODE instruction is entered into the TAP controller.
DBGIEBKPT Instruction breakpoint	Input	Asserted by external hardware to halt execution of the processor for debug purposes. If HIGH at the end of an instruction fetch, it causes the ARM966E-S to enter debug state if that instruction reaches the Execute stage of the processor pipeline.
DBGDEWPT Data watchpoint	Input	Asserted by external hardware to halt execution of the processor for debug purposes. If HIGH at the end of a data memory request cycle, it causes the ARM966E-S to enter debug state.

A.6 Miscellaneous signals

Table A-5 describes the ARM966E-S miscellaneous signals.

Table A-5 Miscellaneous signals

Name	Direction	Description
nFIQ Not fast interrupt request	Input	This is the Fast Interrupt Request signal. This signal must be synchronous to **CLK**.
nIRQ Not interrupt request	Input	This is the Interrupt Request signal. This signal must be synchronous to **CLK**.
VINITHI Exception vector location at reset	Input	Determines the reset location of the exception vectors. When LOW, the vectors are located at `0x00000000`. When HIGH, the vectors are located at `0xFFFF0000`.
INITRAM Tightly-coupled SRAM enable at reset	Input	Determines the tightly-coupled SRAM reset enable.When HIGH, the instruction and data SRAM are both enabled during reset, when LOW, the SRAM are disabled during reset.
BIGENDOUT	Output	When HIGH, the ARM966E-S treats bytes in memory as being in big-endian format. When LOW, memory is treated as little-endian.

A.7 ETM interface signals

Table A-6 describes the ARM966E-S ETM interface signals.

Table A-6 ETM interface signals

Name	Direction	Description
ETMEN	Input	Synchronous ETM interface enable. This signal must be tied LOW if an ETM is not used.
FIFOFULL	Input	Asserted when ETM FIFO fills. This signal must be tied LOW if an ETM is not used.
ETMBIGEND	Output	Big-endian configuration indication for the ETM.
ETMHIVECS	Output	Exception vectors configuration for the ETM.
ETMIA[31:1]	Output	Instruction address for the ETM.
ETMInMREQ	Output	Instruction memory request for the ETM.
ETMISEQ	Output	Sequential instruction access for the ETM.
ETMITBIT	Output	Thumb state indication for the ETM.
ETMDA[31:0]	Output	Data address for the ETM.
ETMDMAS[1:0]	Output	Data size indication for the ETM.
ETMDMORE	Output	More sequential data indication for the ETM.
ETMDnMREQ	Output	Data memory request for the ETM.
ETMDnRW	Output	Data not read or write for the ETM.
ETMDSEQ	Output	Sequential data indication for the ETM.
ETMRDATA[31:0]	Output	Read data for the ETM.
ETMWDATA[31:0]	Output	Write data for the ETM.
ETMDABORT	Output	Data Abort for the ETM.
ETMnWAIT	Output	ARM9E-S stalled indication for the ETM.
ETMDBGACK	Output	Debug state indication for the ETM.
ETMINSTREXEC	Output	Instruction execute indication for the ETM.
ETMINSTRVALID	Output	Instruction valid indication for the ETM.
ETMRNGOUT[1:0]	Output	Watchpoint register match indication for the ETM.

Table A-6 ETM interface signals (continued)

Name	Direction	Description
ETMID31TO25[31:25]	Output	Instruction data field for the ETM.
ETMID15TO11[15:11]	Output	Instruction data field for the ETM.
ETMCHSD[1:0]	Output	Coprocessor handshake decode signals for the ETM.
ETMCHSE[1:0]	Output	Coprocessor handshake execute signals for the ETM.
ETMPASS	Output	Coprocessor instruction execute indication for the ETM.
ETMLATECANCEL	Output	Coprocessor late cancel indication for the ETM.
ETMPROCID	Output	Process ID for the ETM.
ETMPROCIDWR	Output	Asserted when **ETMPROCID** is written.

A.8 INTEST wrapper signals

Table A-7 describes the ARM966E-S INTEST wrapper signals.

Table A-7 INTEST wrapper signals

Name	Direction	Description
SI	Input	Serial input data for the INTEST wrapper scan chain.
SO	Output	Serial output data from the INTEST wrapper scan chain.
SCANEN	Input	Enables scanning of data through the INTEST wrapper scan chain.
TESTEN	Input	Selects the INTEST wrapper scan chain as the source for ARM966E-S inputs.
SERIALEN	Input	Enables the INTEST wrapper BIST activation mode where the scan chain is used to apply serialized ARM instructions to the ARM966E-S to activate BIST test of the tightly-coupled SRAM.
ICAPTUREEN	Input	1 = INTEST wrapper in INTEST mode 0 = INTEST wrapper in EXTEST mode.

A.9 DMA Signals

DMA signals are listed in Table A-8.

Table A-8 DMA signals

Name	Direction	Description
DMAENABLE	Input	Enable ARM966E-S DMA port. Must be tied LOW if DMA not required.
DMAnREQ	Input	DMA not memory request. Must be tied HIGH if DMA not required.
DMAA[25:0]	Input	DMA address. Accesses up to 64Mbyte of memory. Unused address bits must be tied LOW.
DMAnRW	Input	DMA write not read: 0 = read 1 = write.
DMAMAS[1:0]	Input	DMA Memory Access Size. Encodes the size of writes. Reads are always word wide: 00 = byte 01 = halfword 10 = word 11 = reserved.
DMAD[31:0]	Input	DMA write data.
DMAWait	Input	DMA Wait. Used to stall the ARM966E-S to allow a DMA access to take place. This functionality is only required if the data RAM is single-port. This signal must be tied LOW if the data RAM is dual-port. This signal has the same functionality internal to the ARM966E-S as **FIFOFULL**.
DMAReady	Output	DMA Ready. Asserted HIGH when the ARM966E-S is stalled. Only needs to be sampled when the data RAM is single port, for example when the ARM966E-S stall was requested by **DMAWait**.
DMARData[31:0]	Output	DMA read data.

Appendix B
AC Parameters

This appendix describes the AC timing parameters for the ARM966E-S. It contains the following sections:

B.1 Timing diagrams

The timing diagrams in this section are:

- *Clock, reset and AHB enable timing*
- *AHB bus request and grant related timing* on page B-3
- *AHB bus master timing* on page B-4
- *Coprocessor interface timing* on page B-5
- *Debug interface timing* on page B-6
- *JTAG interface timing* on page B-7
- *DBGSDOUT to DBGTDO timing* on page B-8
- *Exception and configuration timing* on page B-8
- *INTEST wrapper timing* on page B-9
- *ETM interface timing* on page B-10.

Clock, reset and AHB enable timing parameters are shown in Figure B-1.

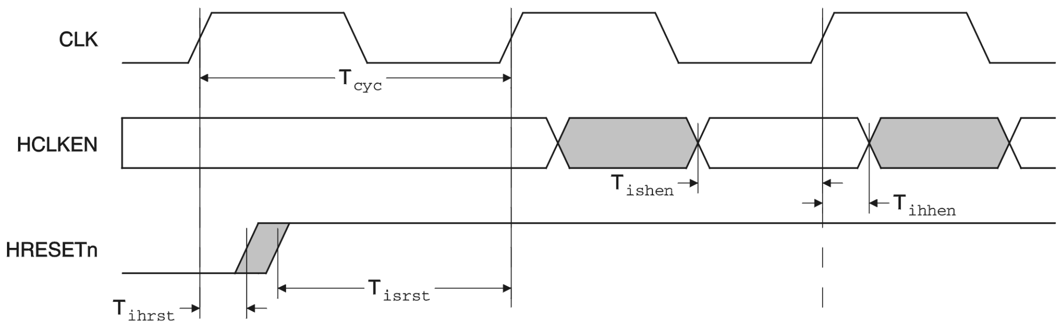


Figure B-1 Clock, reset and AHB enable timing

AHB bus request and grant related timing parameters are shown in Figure B-2 on page B-3.

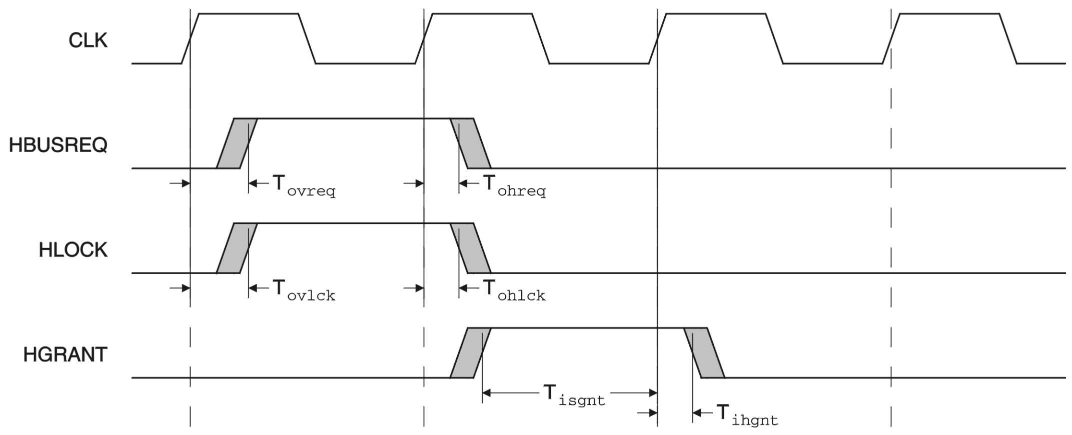


Figure B-2 AHB bus request and grant related timing

AHB bus master timing parameters are shown in Figure B-3 on page B-4.

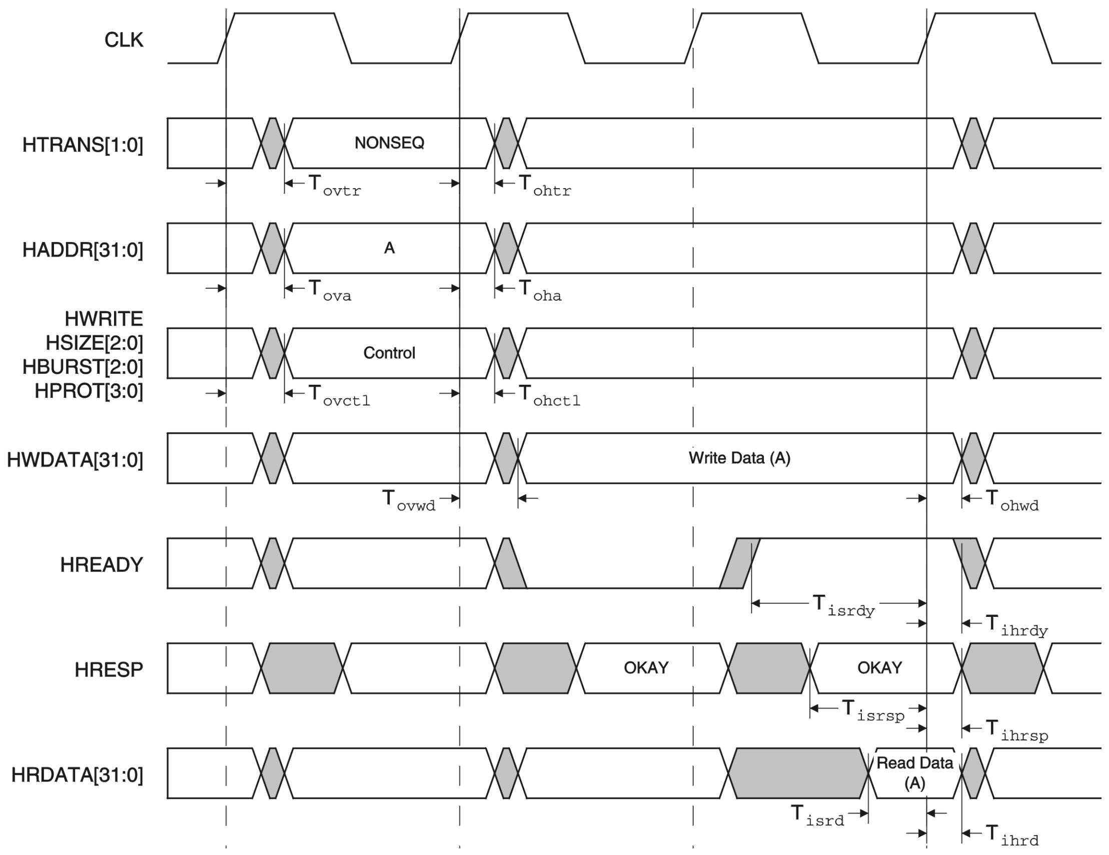


Figure B-3 AHB bus master timing

Coprocessor interface timing parameters are shown in Figure B-4 on page B-5.

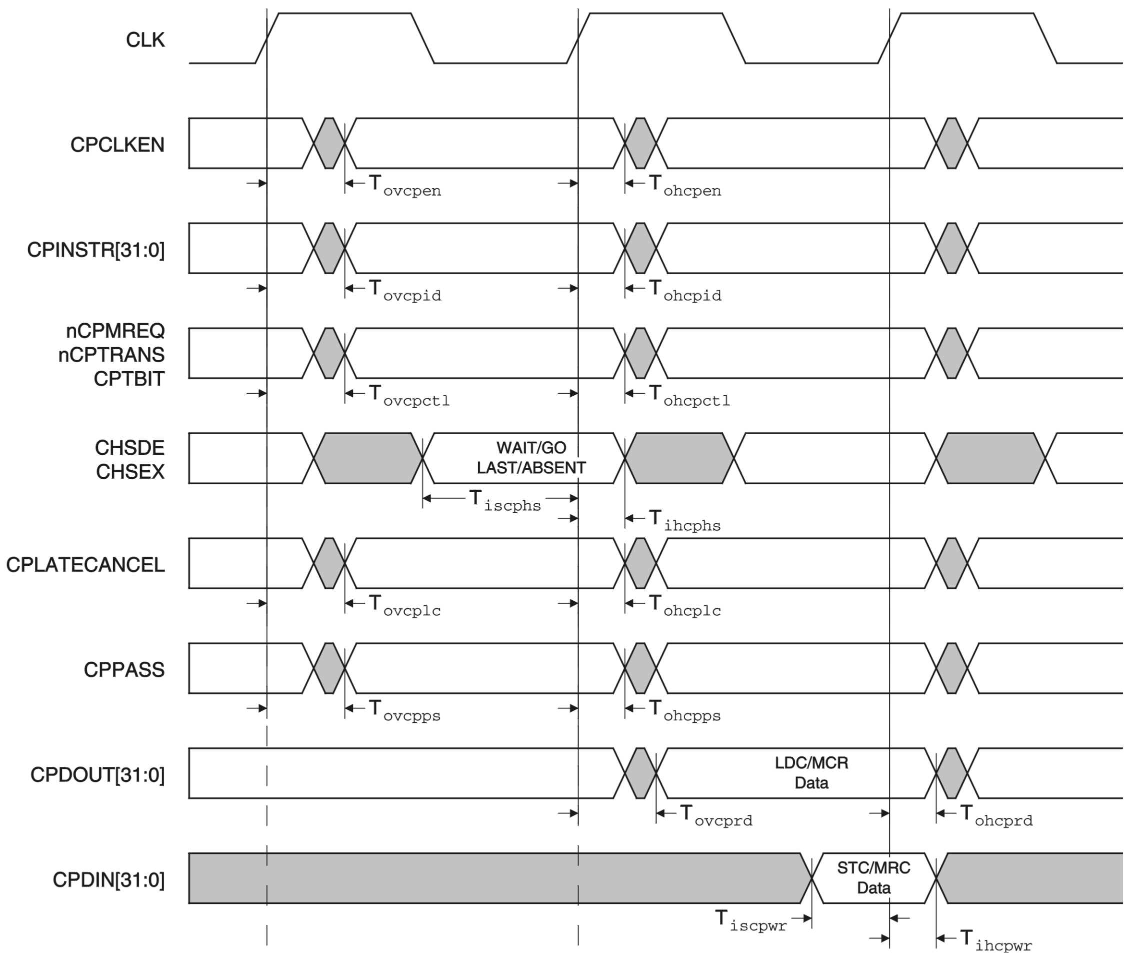


Figure B-4 Coprocessor interface timing

Debug interface timing parameters are shown in Figure B-5 on page B-6.

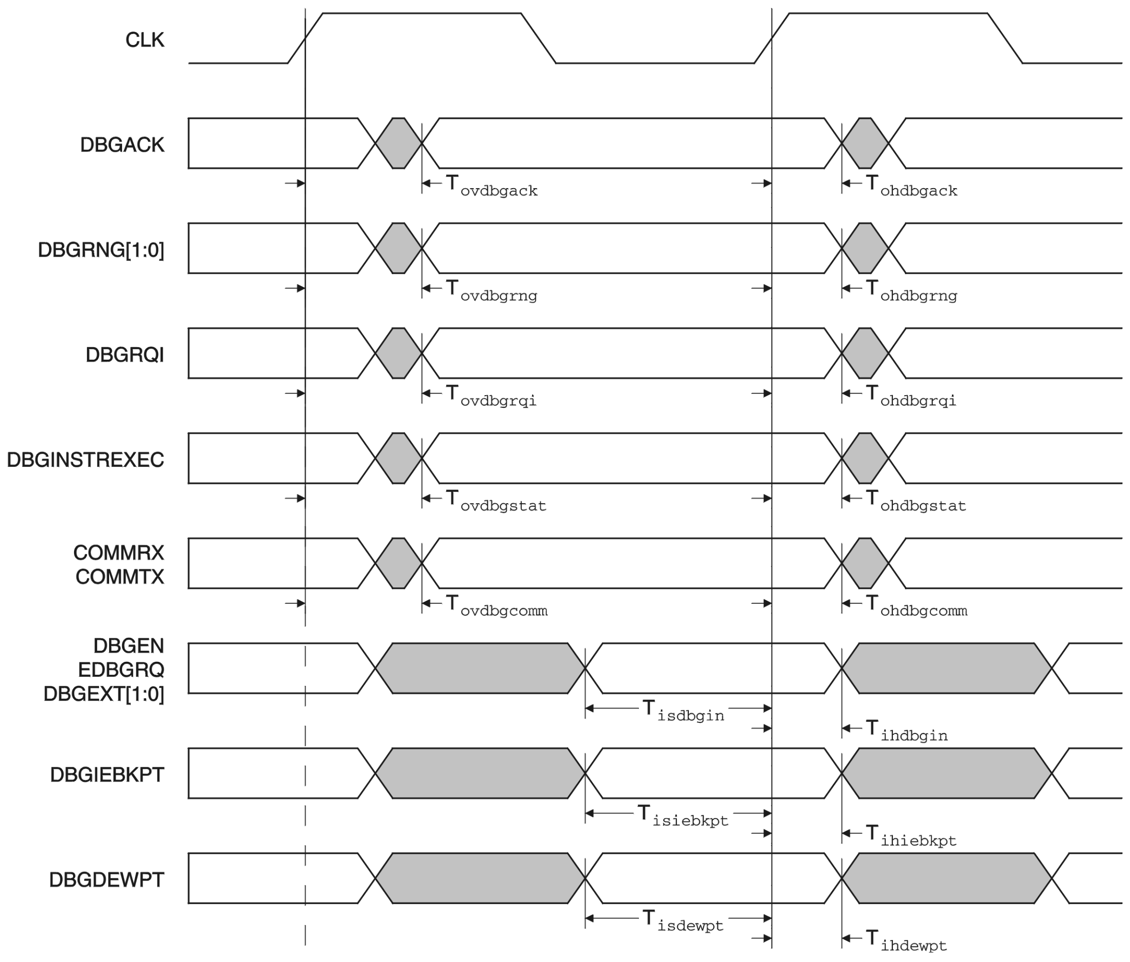


Figure B-5 Debug interface timing

JTAG interface timing parameters are shown in Figure B-6 on page B-7.

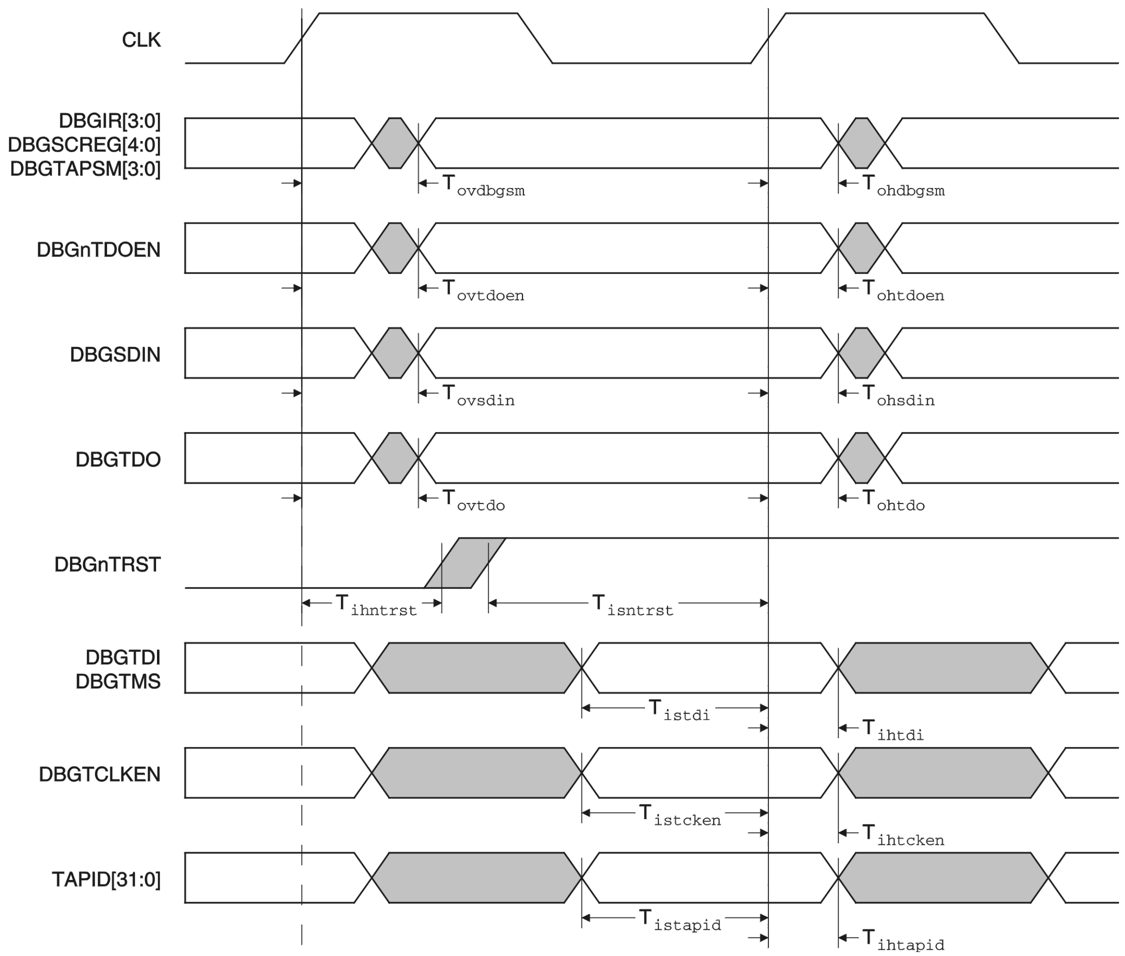


Figure B-6 JTAG interface timing

A combinatorial path timing parameter exists from the **DBGSDOUT** input to the **DBGTDO** output. This is shown in Figure B-7 on page B-8.

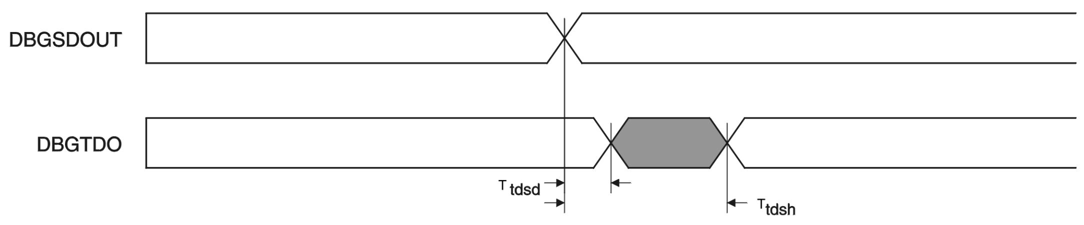


Figure B-7 DBGSDOUT to DBGTDO timing

Exception and configuration timing parameters are shown in Figure B-8.

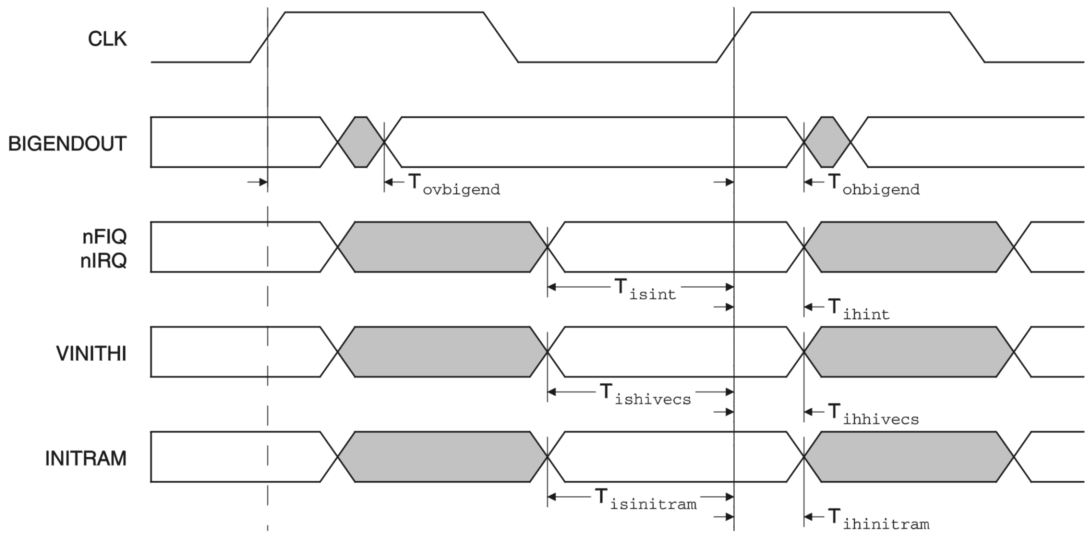


Figure B-8 Exception and configuration timing

The INTEST wrapper timing parameters are shown in Figure B-9 on page B-9.

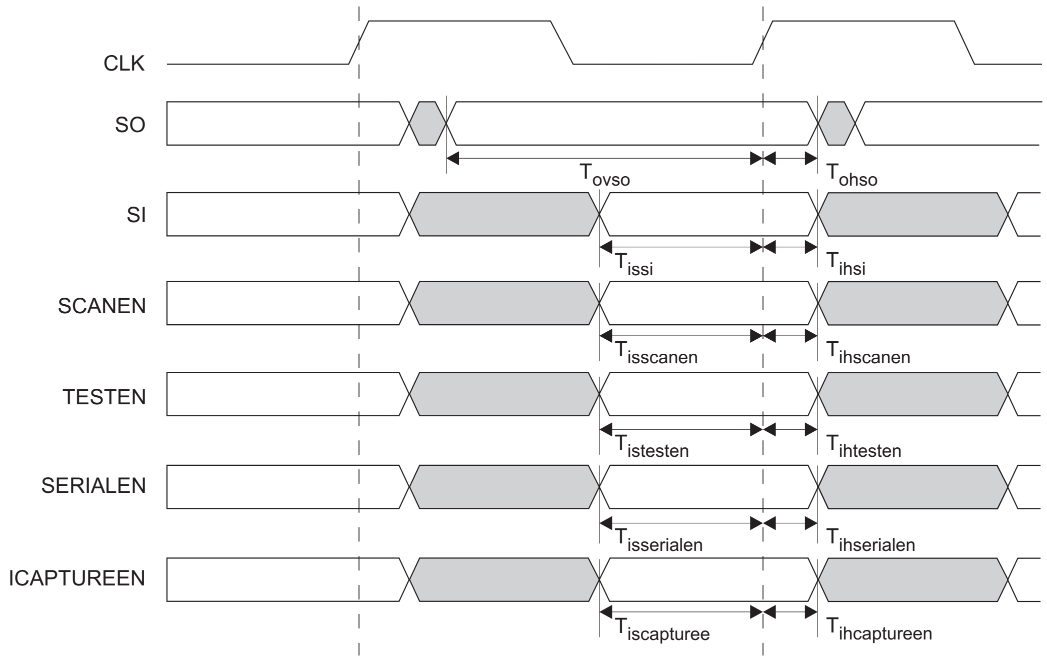


Figure B-9 INTEST wrapper timing

The ETM interface timing parameters are shown in Figure B-10 on page B-10.

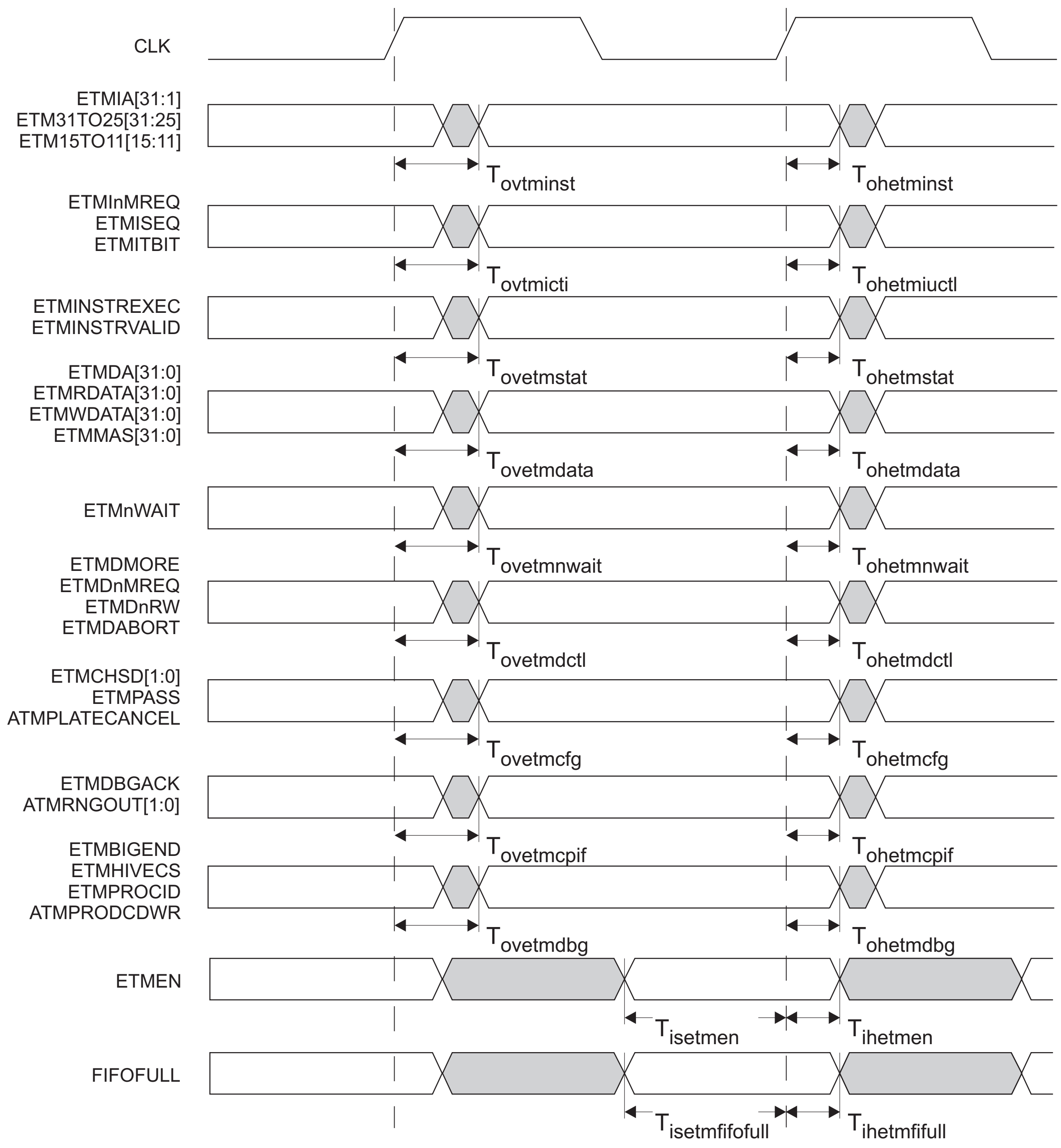


Figure B-10 ETM interface timing

The DMA interface timing parameters are shown in Figure B-11

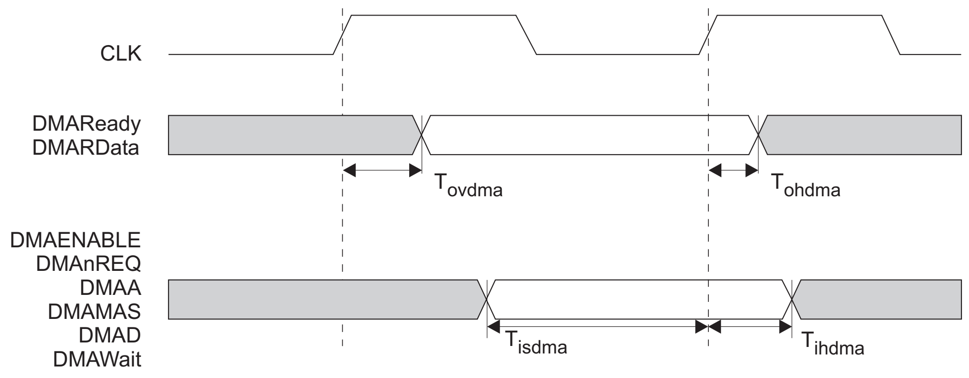


Figure B-11 DMA interface timing

B.2 AC timing parameter definitions

Table B-1 shows target AC parameters. All figures are expressed as percentages of the **CLK** period at maximum operating frequency.

——— **Note** ———

The figures quoted are relative to the rising clock edge after the clock skew for internal buffering has been added. Inputs given a 0% hold figure therefore require a positive hold relative to the top- level clock input. The amount of hold required is equivalent to the internal clock skew.

Table B-1 AC parameters

Symbol	Parameter	Min	Max
T_{cyc}	**CLK** cycle time	100%	-
T_{ishen}	**HCLKEN** input setup time to rising CLK	85%	-
T_{ihhen}	**HCLKEN** input hold time from rising CLK	-	0%
T_{isrst}	**HRESETn** deassertion input setup time to rising CLK	90%	-
T_{ihrst}	**HRESETn** deassertion input hold time from rising **CLK**	-	0%
T_{ovreq}	Rising **CLK** to **HBUSREQ** valid	-	30%
T_{ohreq}	**HBUSREQ** hold time from rising **CLK**	>0%	-
T_{ovlck}	Rising **CLK** to **HLOCK** valid	-	30%
T_{ohlck}	**HLOCK** hold time from rising **CLK**	>0%	-
T_{isgnt}	**HGRANT** input setup time to rising **CLK**	40%	-
T_{ihgnt}	**HGRANT** input hold time from rising **CLK**	-	0%
T_{ovtr}	Rising **CLK** to **HTRANS[1:0]** valid	-	30%
T_{ohtr}	**HTRANS[1:0]** hold time from rising **CLK**	>0%	-
T_{ova}	Rising **CLK** to **HADDR[31:0]** valid	-	30%
T_{oha}	**HADDR[31:0]** hold time from rising **CLK**	>0%	-
T_{ovct}l	Rising **CLK** to AHB control signals valid	-	30%
T_{ohctl}	AHB control signals hold time from rising CLK	>0%	-

Table B-1 AC parameters (continued)

Symbol	Parameter	Min	Max
T_{ovwd}	Rising **CLK** to **HWDATA[31:0]** valid	-	30%
T_{ohwd}	**HWDATA[31:0]** hold time from rising **CLK**	>0%	-
T_{isrdy}	**HREADY** input setup time to rising **CLK**	75%	-
T_{ihrdy}	**HREADY** input hold time from rising **CLK**	-	0%
T_{isrsp}	**HRESP[1:0]** input setup time to rising **CLK**	50%	-
T_{ihrsp}	**HRESP[1:0]** input hold time from rising **CLK**	-	0%
T_{isrd}	**HRDATA[31:0]** input setup time to rising **CLK**	40%	-
T_{ihrd}	**HRDATA[31:0]** input hold time from rising **CLK**	-	0%
T_{ovcpen}	Rising **CLK** to **CPCLKEN** valid	-	30%
T_{ohcpen}	**CPCLKEN** hold time from rising **CLK**	>0%	-
T_{ovcpid}	Rising **CLK** to **CPINSTR[31:0]** valid	-	30%
T_{ohcpid}	**CPINSTR[31:0]** hold time from rising **CLK**	>0%	-
$T_{ovcpctl}$	Rising **CLK** to transaction control valid	-	30%
$T_{ohcpctl}$	Transaction control hold time from rising **CLK**	>0%	-
T_{iscphs}	Coprocessor handshake input setup time to rising **CLK**	50%	-
T_{ihcphs}	Coprocessor handshake input hold time from rising **CLK**	-	0%
T_{ovcplc}	Rising **CLK** to **CPLATECANCEL** valid	-	30%
T_{ohcplc}	**CPLATECANCEL** hold time from rising **CLK**	>0%	-
T_{ovcpps}	Rising **CLK** to **CPPASS** valid	-	30%
T_{ohcpps}	**CPPASS** hold time from rising **CLK**	>0%	-
T_{ovcprd}	Rising **CLK** to **CPDOUT[31:0]** valid	-	30%
T_{ohcprd}	**CPDOUT[31:0]** hold time from rising **CLK**	>0%	-
T_{iscpwr}	**CPDIN[31:0]** input setup time to rising **CLK**	40%	-
T_{ihcpwr}	**CPDIN[31:0]** input hold time from rising **CLK**	-	0%

Table B-1 AC parameters (continued)

Symbol	Parameter	Min	Max
$T_{ovdbgack}$	Rising **CLK** to **DBGACK** valid	-	60%
Tohdbgack	**DBGACK** hold time from rising **CLK**	>0%	-
$T_{ovdbgrng}$	Rising **CLK** to **DBGRNG[1:0]** valid	-	60%
$T_{ohdbgrng}$	**DBGRNG[1:0]** hold time from rising **CLK**	>0%	-
$T_{ovdbgrqi}$	Rising **CLK** to **DBGRQI** valid	-	45%
$T_{ohdbgrqi}$	**DBGRQI** hold time from rising **CLK**	>0%	-
$T_{ovdbgstat}$	Rising **CLK** to **DBGINSTREXEC** valid	-	30%
$T_{ohdbgstat}$	**DBGINSTREXEC** hold time from rising **CLK**	>0%	-
$T_{ovdbgcomm}$	Rising **CLK** to communications channel outputs valid	-	30%
$T_{ohdbgcomm}$	Communications channel outputs hold time from rising **CLK**	>0%	-
$T_{isdbgin}$	Debug inputs setup time to rising **CLK**	30%	-
$T_{ihdbgin}$	Debug inputs hold time from rising **CLK**	-	0%
$T_{isiebkpt}$	**DBGIEBKPT** input setup time to rising **CLK**	20%	-
$T_{ihiebkpt}$	**DBGIEBKPT** input hold time from rising **CLK**	-	0%
$T_{isdewpt}$	**DBGDEWPT** input setup time to rising **CLK**	20%	-
$T_{ihdewpt}$	**DBGDEWPT** input hold time from rising **CLK**	-	0%
$T_{ovdbgsm}$	Rising **CLK** to debug state valid	-	30%
$T_{ohdbgsm}$	Debug state hold time from rising **CLK**	>0%	-
$T_{ovtdoen}$	Rising **CLK** to **DBGnTDOEN** valid	-	40%
$T_{ohtdoen}$	**DBGnTDOEN** hold time from rising **CLK**	>0%	-
T_{ovsdin}	Rising **CLK** to **DBGSDIN** valid	-	20%
T_{ohsdin}	**DBGSDIN** hold time from rising **CLK**	>0%	-
T_{ovtdo}	Rising **CLK** to **DBGTDO** valid	-	65%
T_{ohtdo}	**DBGTDO** hold time from rising **CLK**	>0%	-

Table B-1 AC parameters (continued)

Symbol	Parameter	Min	Max
$T_{isntrst}$	**DBGnTRST** de-asserted input setup time to rising **CLK**	35%	-
$T_{ihntrst}$	**DBGnTRST** input hold time from rising **CLK**	-	0%
T_{istdi}	Tap state control input setup time to rising **CLK**	25%	-
T_{ihtdi}	Tap state control input hold time from rising **CLK**	-	0%
$T_{istcken}$	**DBGTCKEN** input setup time to rising **CLK**	50%	-
$T_{ihtcken}$	**DBGTCKEN** input hold time from rising **CLK**	-	0%
$T_{istapid}$	**TAPID[31:0]** input setup time to rising **CLK**	20%	-
$T_{ihtapid}$	**TAPID[31:0]** input hold time from rising **CLK**	-	0%
T_{dsd}	**DBGTDO** delay from **DBGSDOUTBS** changing	-	30%
T_{dsh}	**DBGTDO** hold time from **DBGSDOUTBS** changing	>0%	-
$T_{ovbigend}$	Rising **CLK** to **BIGENDOUT** valid	-	30%
$T_{ohbigend}$	**BIGENDOUT** hold time from rising **CLK**	>0%	-
T_{isint}	Interrupt input setup time to rising **CLK**	15%	-
T_{ihint}	Interrupt input hold time from rising **CLK**	-	0%
$T_{ishivecs}$	**VINITHI** input setup time to rising **CLK**	95%	-
$T_{ihhivecs}$	**VINITHI** input hold time from rising **CLK**	-	0%
$T_{isinitram}$	**INITRAM** input setup time to rising **CLK**	95%	-
$T_{ihinitram}$	**INITRAM** input hold time from rising **CLK**	-	0%
T_{ovso}	Rising **CLK** to **SO** valid	-	30%
T_{ohso}	**SO** hold time from rising **CLK**	>0%	-
T_{issi}	**SI** input setup time to rising **CLK**	95%	-
T_{ihsi}	**SI** input hold time from rising **CLK**	-	0%
$T_{isscanen}$	**SCANEN** input setup time to rising **CLK**	95%	-
$T_{ihscanen}$	**SCANEN** input hold time from rising **CLK**	-	0%

Table B-1 AC parameters (continued)

Symbol	Parameter	Min	Max
$T_{istesten}$	**TESTEN** input setup time to rising **CLK**	95%	-
$T_{ihtesten}$	**TESTEN** input hold time from rising **CLK**	-	0%
$T_{isserialen}$	**SERIALEN** input setup time to rising **CLK**	95%	-
$T_{ihserialen}$	**SERIALEN** input hold time from rising **CLK**	-	0%
$T_{iscaptureen}$	**CAPTUREEN** input setup time to rising **CLK**	95%	-
$T_{ihcaptureen}$	**CAPTUREEN** input hold time from rising **CLK**	-	0%
$T_{ovetminst}$	Rising **CLK** to ETM instruction interface valid	-	30%
$T_{ohetminst}$	ETM instruction interface hold time from rising **CLK**	>0%	-
$T_{ovetmictl}$	Rising **CLK** to ETM instruction control valid	-	30%
$T_{ohetmictl}$	ETM instruction control hold time from rising **CLK**	>0%	-
$T_{ovetmstat}$	Rising **CLK** to **ETMINSTREXEC** valid	-	30%
$T_{ohetmstat}$	**ETMINSTREXEC** hold time from rising **CLK**	>0%	-
$T_{ovetmdata}$	Rising **CLK** to ETM data interface valid	-	30%
$T_{ohetmdata}$	ETM data interface hold time from rising **CLK**	>0%	-
$T_{ovetmnwait}$	Rising **CLK** to **ETMnWAIT** valid	-	30%
$T_{ohetmnwait}$	**ETMnWAIT** hold time from rising **CLK**	>0%	-
$T_{ovetmdctl}$	Rising **CLK** to ETM data control valid	-	30%
$T_{ohetmdctl}$	ETM data control hold time from rising **CLK**	>0%	-
$T_{ovetmcfg}$	Rising **CLK** to ETM configuration valid	-	30%
$T_{ohetmcfg}$	ETM configuration hold time from rising **CLK**	>0%	-
$T_{ovetmcpif}$	Rising **CLK** to ETM coprocessor signals valid	-	30%
$T_{ohetmcpif}$	ETM coprocessor signals hold time from rising **CLK**	>0%	-
$T_{ovetmdbg}$	Rising **CLK** to ETM debug signals valid	-	30%
$T_{ohetmdbg}$	ETM debug signals hold time from rising **CLK**	>0%	-

Table B-1 AC parameters (continued)

Symbol	Parameter	Min	Max
$T_{isetmen}$	**ETMEN** input setup time to rising **CLK**	50%	-
$T_{ihetmen}$	**ETMEN** input hold time from rising **CLK**	-	0%
$T_{isfifofull}$	**FIFOFULL** input setup time to rising **CLK**	50%	-
$T_{ihetmen}$	**FIFOFULL** input hold time from rising **CLK**	-	0%
T_{ovdma}	Rising **CLK** to DMA signals valid	50%	-
T_{ohdm}a	DMA signals hold time from rising **CLK**	0%	-
T_{isdma}	DMA input setup time to rising **CLK**	50%	-
T_{ihdma}	DMA input hold time from rising **CLK**	-	0%

Note

- The **VINITHI** and **INITRAM** pins are specified as 95% of the cycle as they are for input configuration during reset and can be considered static.
- The INTEST wrapper inputs and outputs are specified as 95% of the cycle as they are production test related and expected to operate at typically 50% of the functional clock rate.

Appendix C
SRAM Stall Cycles

This appendix describes the tightly-coupled SRAM in the ARM966E-S. It contains the following section:

- *About SRAM stall cycles* on page C-2.

For details of the ARM9E-S interface signals referenced in this section, refer to the *ARM9E-S Technical Reference Manual*.

C.1 About SRAM stall cycles

Stall cycles can occur in both the instruction and data SRAMs, with one stall mechanism being shared between the SRAMs and additional stall mechanism attributed to the I-SRAM only. Any stall requirement is detected by the SRAM control and factored into its response to the ARM966E-S system controller. The ARM9E-S **SYSCLKEN** input is then deasserted until the SRAM has performed the access.

C.1.1 Read-follows-write

This stall mechanism is shared by both instruction and data SRAM because of the pipelined nature of write data from the ARM9E-S core. The write data appears on the core interface in the cycle after the address, so that it is not possible to perform the write until the next rising clock edge. The address from the core must therefore be pipelined to line up with the write data. A write with pipelined address is shown in Figure C-1.

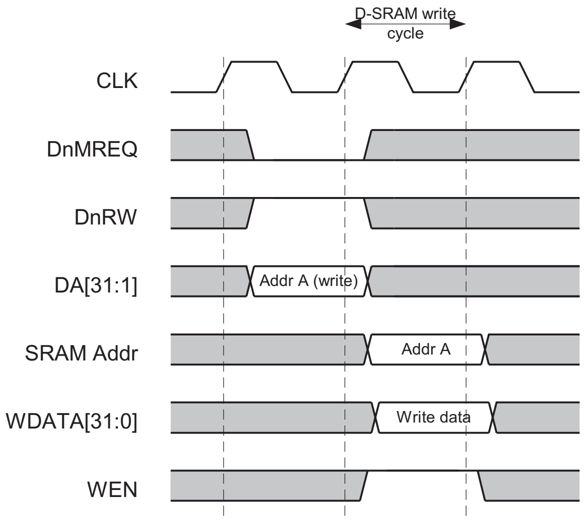


Figure C-1 SRAM write cycle

Note

The write is performed on the second rising edge of the period marked D-SRAM write cycle.

In the case of back-to-back writes, stalls do not occur because the pipelined address is being used and this keeps in step with the data. However, if a read follows the write, the write must first be allowed to complete before the lookup for the read can be performed.

Figure C-2 shows this example and how the SRAM control must pipeline and select between the write and read address. The ARM9E-S core is stalled for a cycle by the system controller by deasserting **SYSCLKEN**.

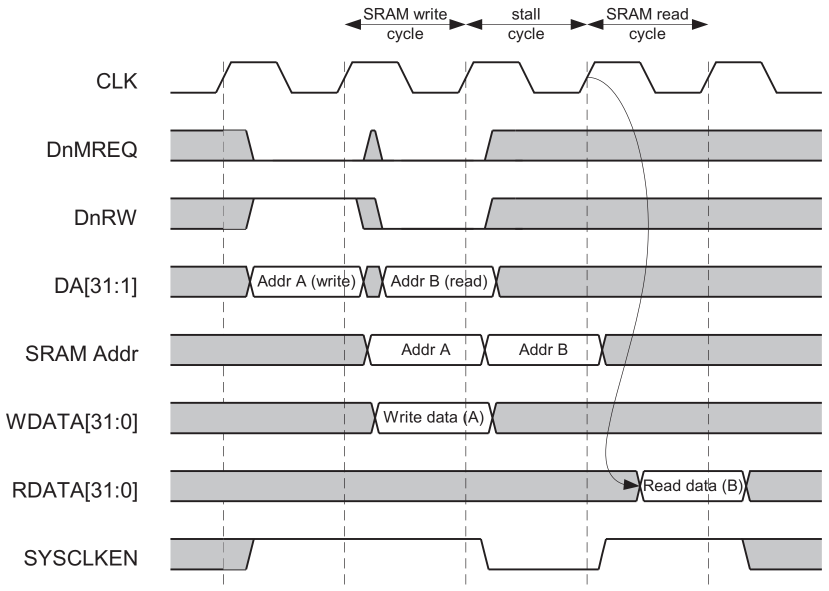


Figure C-2 Read follows write

——— **Note** ———

The second rising edge of the SRAM write cycle is the same edge that is required for the SRAM read (of Addr B). It is not possible to read and write concurrently so a stall must occur before the read of Addr B.

C.1.2 Additional Instruction SRAM stalls

The I-SRAM has additional stall cycles that arise because of the following operations:

- data reads to the I-SRAM are pipeline
- simultaneous instruction fetches and data accesses can occur
- any access can occur during two cycle data reads and writes.

Simultaneous instruction fetch, data read

The ARM9E-S data interface is able to access the I-SRAM for programming purposes and for access to literal tables during program execution.

It is possible for the ARM9E-S to issue a simultaneous instruction and data request, and if the data request addresses the I-SRAM, a stall cycle is required (see Figure C-3).

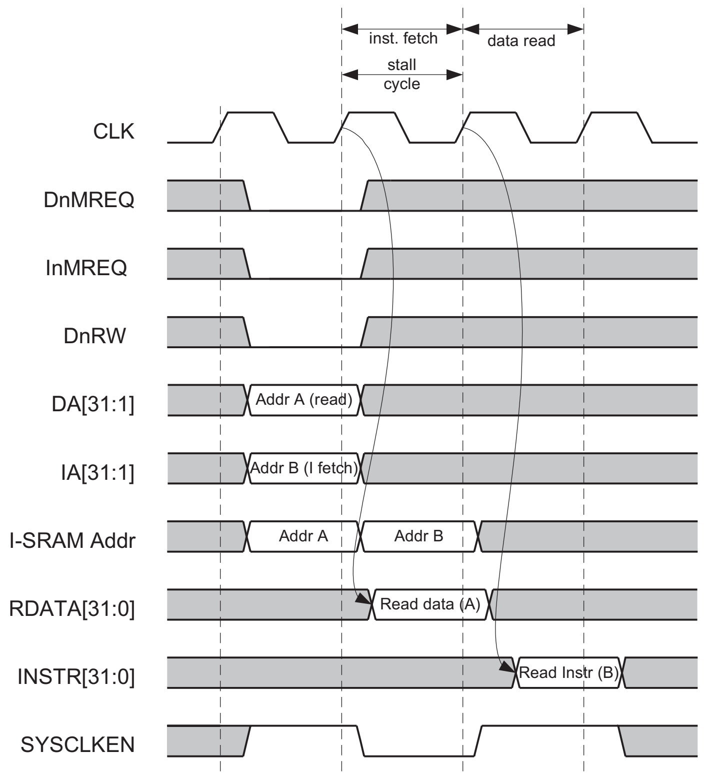


Figure C-3 Simultaneous instruction fetch, data read

Note

In the case of simultaneous I-SRAM and D-SRAM read access requests from the ARM9E-S core, the instruction fetch is always performed first, followed by the data read. The core is disabled until both accesses have completed.

Data read

To maximize the I-SRAM interface frequency performance, data read requests to this RAM are pipelined. This adds a stall cycle for every data read instruction. An example of a data read from the I-SRAM is shown in on page C-5.

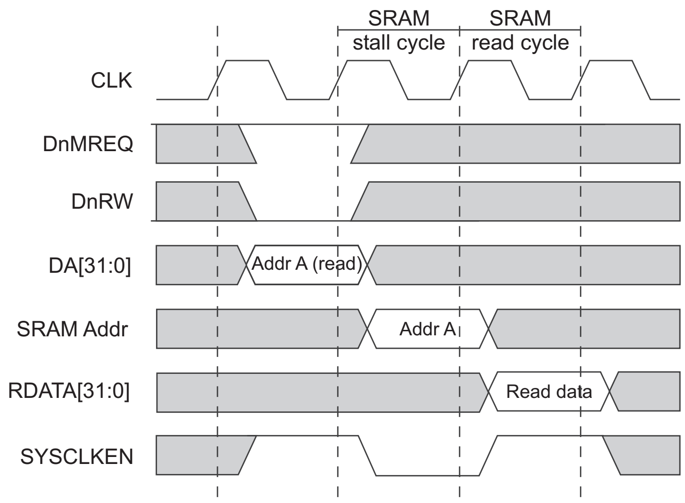


Figure C-4 Data read from I-SRAM

The stall cycle is only incurred for the first read of a read instruction. If an `LDM` is performed, there is a stall cycle inserted only for the first read of the `LDM`. Back-to- back `LDR`s will incur a stall cycle at the start of each `LDR`.

Data read followed by instruction fetch

Data reads to the I-SRAM are pipelined. An instruction fetch in the cycle after a data read request coincides with the stalled data read and so the instruction fetch is stalled for 1 cycle. This is shown in on page C-6.

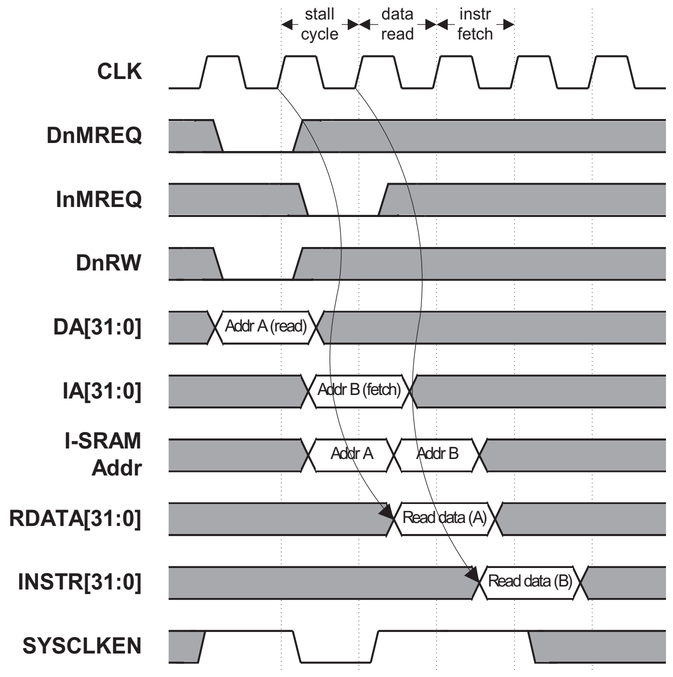


Figure C-5 Data read followed by instruction fetch

Simultaneous instruction fetch, data write

If the ARM9E-S performs a simultaneous data write and instruction fetch that both map to I-SRAM address space, two stall cycles occur. The first cycle allows for the pipelined write, the second cycle allows for the instruction fetch. The core cannot be enabled until both accesses have completed (see Figure C-6 on page C-7).

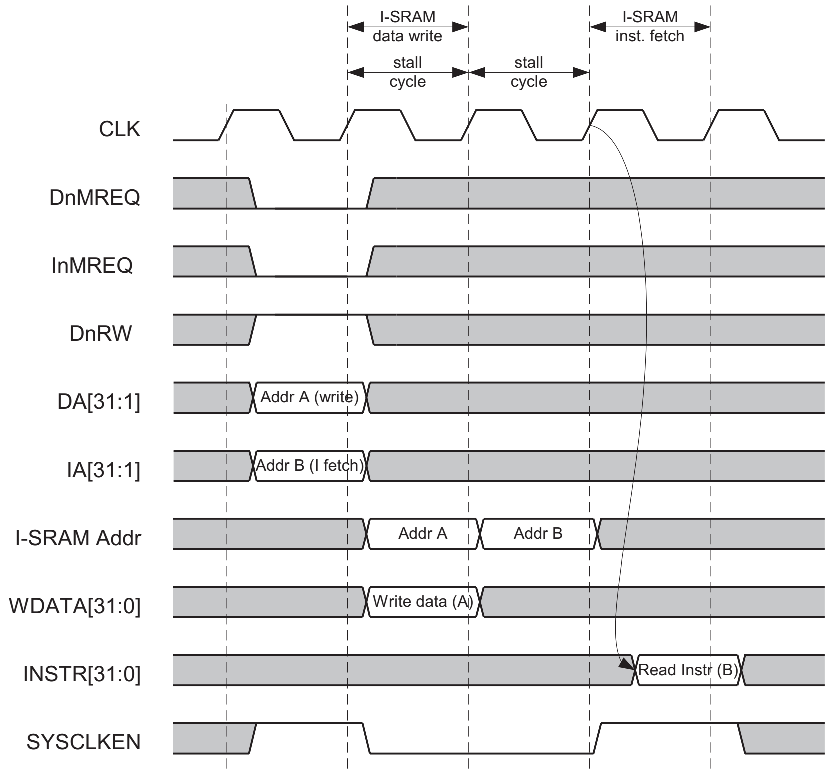


Figure C-6 Simultaneous instruction fetch, data write

I-SRAM data write followed by instruction fetch

This class of stall occurs when a data write to the I-SRAM address space is performed, followed by an instruction fetch request in the next cycle. It is similar to the generic read follows write scenario of each SRAM except that the read is an instruction fetch rather than a data load. The instruction fetch must be held off until the write has completed, requiring that the ARM9E-S core is stalled for a cycle (see Figure C-7 on page C-8).

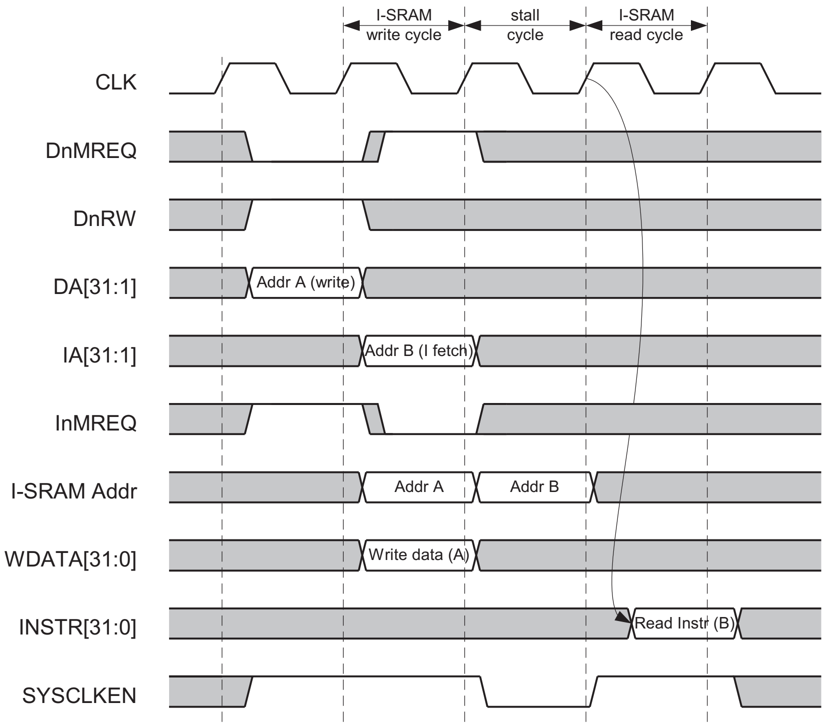


Figure C-7 I-SRAM data write followed by instruction fetch

I-SRAM write followed by instruction fetch, data write

This case is where a write is taking place to the I-SRAM that is immediately followed by both an instruction fetch and a data write. The second write is performed immediately after the current write without penalty. However, the core must be stalled until both the second write and instruction fetch have completed, so there are two stall cycles (see Figure C-8 on page C-9).

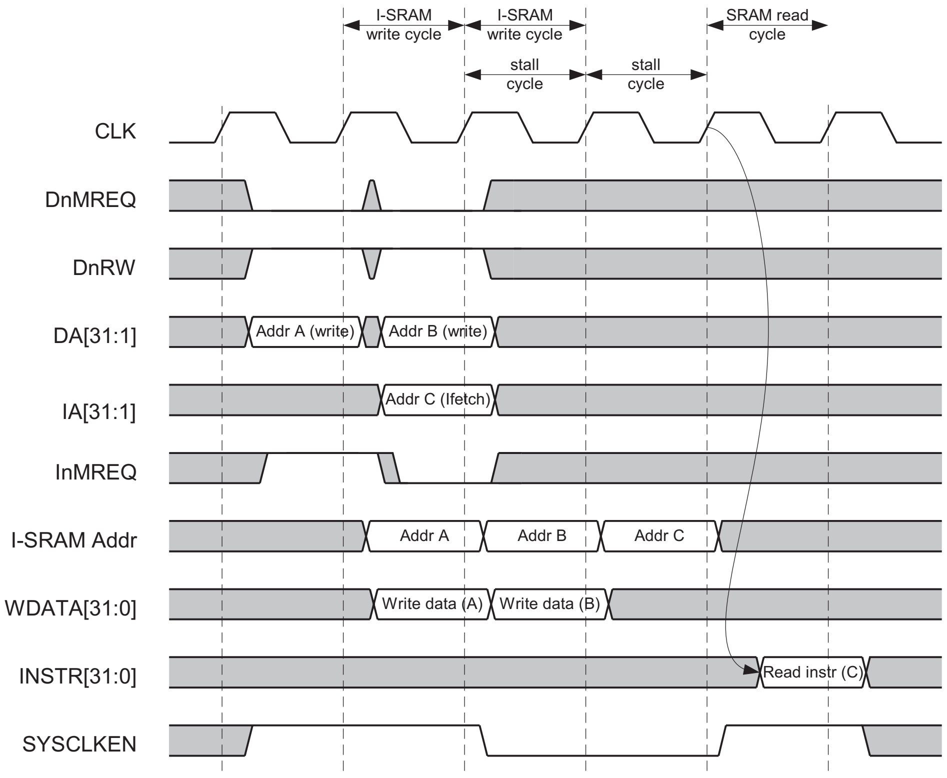


Figure C-8 I-SRAM write followed by instruction fetch, data write

I-SRAM write followed by instruction fetch, data read

This is where a write is taking place to the I-SRAM that is immediately followed by both an instruction fetch and a data read. This has the same two-stall cycle response as the previous scenario, although the I-SRAM control behaves differently. The first write must complete before the data read can be performed. The instruction fetch can then be performed in the next cycle (see Figure C-9 on page C-10).

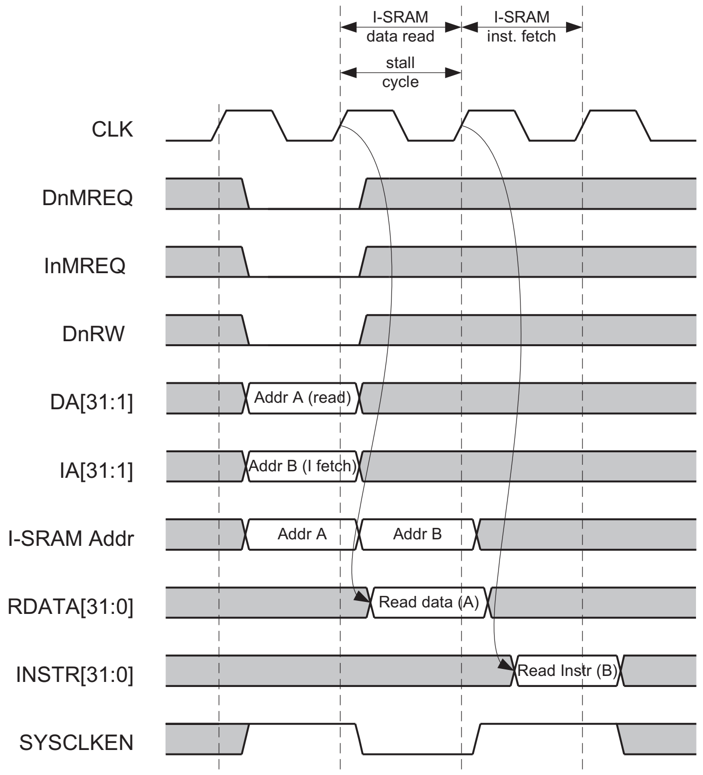


Figure C-9 I-SRAM write followed by instruction fetch, data read

Index