ARM940T

(Rev 2)

Technical Reference Manual

ARM

ARM DDI 0144B

ARM940T (Rev 2)
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
12th February 1999	A	First release.
22nd November 2000	B	Second release.

Proprietary Notice

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

The information in this document is final (information on a developed product).

Web Address

`http://www.arm.com`

Contents
ARM940T (Rev 2) Technical Reference Manual

Preface

This preface introduces the ARM940T and its reference documentation. It contains the following sections:

About this document

This document is the technical reference manual for the ARM940T.

Intended audience

This document has been written for hardware and software engineers who want to design or develop products based upon the ARM940T processor. It assumes no prior knowledge of ARM products.

Using this manual

This document is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for a functional description of the ARM940T.

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

Read this chapter to learn how to program the ARM940T system control registers to configure operation of the macrocell.

Chapter 3 ***Protection Unit***

Read this chapter to learn how to use the protection unit and the memory interface to partition memory and set individual partition attributes.

Chapter 4 ***Caches and Write Buffer***

Read this chapter to learn how to use the instruction and data caches and the write buffer.

Chapter 5 ***Clock Modes***

Read this chapter to learn how to use the Fastbus, Synchronous, and Asynchronous processor clock modes to connect to memory using the AMBA ASB bus.

Chapter 6 ***Bus Interface Unit***

Read this chapter to learn how to use the bus interface unit and the AMBA ASB and AHB interface to handle single and burst transfers.

Chapter 7 ***Coprocessor Interface***

Read this chapter to learn how to interface the ARM940T to an off-chip coprocessor, and how to execute coprocessor instructions.

Chapter 8 ***Debug Support***

Read this chapter to learn how to use the debug interface to implement a debugging system using scan chains and the EmbeddedICE unit.

Chapter 9 ***TrackingICE***

Read this chapter to learn how to connect a second external ARM9TDMI to precisely track the inputs to the ARM940T using TrackingICE mode.

Chapter 10 ***Test Support***

Read this chapter to learn how to use the test support provided by the ARM940T for the ARM9TDMI core and the ARM940T macrocell.

Chapter 11 ***Instruction Cycle Summary***

Read this chapter for details of instruction cycle times. This chapter contains timing diagrams for interlock timing.

Chapter 12 ***AC Characteristics***

Read this chapter for a description of the timing parameters used in the ARM940T.

Appendix A ***Signal Descriptions***

Read this chapter for a detailed description of the signals used in the ARM940T.

Typographical conventions

The following typographical conventions are used in this book:

bold Highlights ARM processor signal names, and interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.

italic Highlights special terminology, cross-references, and citations.

`typewriter` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

`type`writer Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`typewriter italic`

Denotes arguments to commands or functions, where the argument is to be replaced by a specific value.

`typewriter bold`

Denotes language keywords when used outside example code.

Timing diagram conventions

This manual contains a number of timing diagrams. Figure P-1 on page xi explains the components used in these diagrams. Any variations are clearly labeled when they occur. Therefore, you must not attach any additional meaning unless specifically stated.

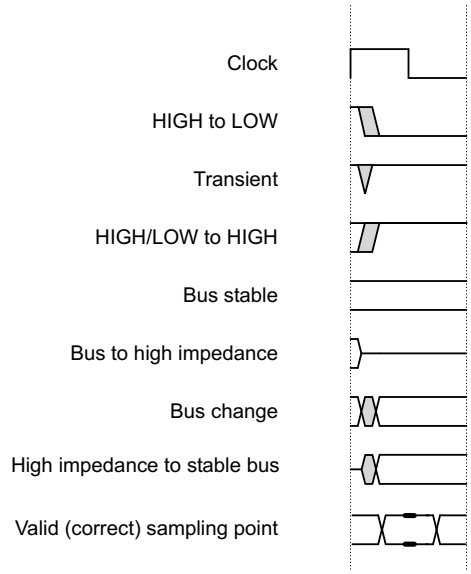

Figure P-1 Key to timing diagram conventions

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

This document contains information that is specific to the ARM940T. Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM9TDMI Data Sheet* (ARM DDI 0029)
- *AMBA Specification* (ARM IHI 0011)
- *Application Note 41 TrackingICE* (ARM DAI 0041).

Other publications

This section lists relevant documents published by third parties:

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

Feedback

ARM Limited welcomes feedback both on the ARM940T, and on the documentation.

Feedback on the ARM940T

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 ARM940T Technical Reference Manual

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

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

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM940T processor. It contains the following sections:

1.1 About the ARM940T

The ARM940T is a member of the ARM9TDMI family of general-purpose microprocessors. This family includes:

- ARM9TDMI, the core
- ARM940T, the core plus cache and protection unit
- ARM920T, the core plus cache and MMU.

The ARM9TDMI processor core is a Harvard architecture device implemented using a five-stage pipeline consisting of Fetch, Decode, Execute, Memory, and Write stages. It can be provided as a standalone core that can be embedded into more complex devices. The standalone core has a simple bus interface that allows you to design your own caches and memory systems around it.

The ARM9TDMI family of microprocessors supports both the 32-bit ARM and 16-bit Thumb instruction sets, allowing you to trade off between high performance and high code density.

The ARM940T processor has a Harvard cache architecture with separate 4KB instruction and 4KB data caches, each with a 4-word line length. A protection unit allows you to define eight regions of memory for data and eight regions for instructions, each with individual cache and write buffer configurations and access permissions. The cache system is software configurable to provide highest average performance or to meet the requirements of real-time systems. Software configurable options include:

- random or round-robin replacement algorithm
- write-through or write-back cache operation (independently selectable for each memory region)
- cache locking with granularity 1/64th of cache size.

The cache and write buffers improve CPU performance and minimize accesses to the AMBA bus and to any off-chip memory, reducing overall system power consumption.

The ARM940T supports the ARM debug architecture and includes logic to assist in both hardware and software debug. The ARM940T also includes support for coprocessors, providing access to the instruction and data buses and handshaking signals.

The ARM940T interface to the rest of the system is over unified address and data buses. This interface enables implementation of either an *Advanced Microcontroller Bus Architecture* (AMBA) *Advanced System Bus* (ASB) or *Advanced High-performance Bus* (AHB) bus scheme with the ARM940T ASB to AHB bridge block available from ARM Ltd. You can implement an ASB scheme either as a fully-compliant AMBA bus master, or as a slave for production test. The ARM920T also has a *Tracking ICE* mode that allows an approach similar to a conventional ICE mode of operation.

1.2 Processor functional block diagram

Figure 1-1 on page 1-3 shows the functional block diagram of the ARM940T.

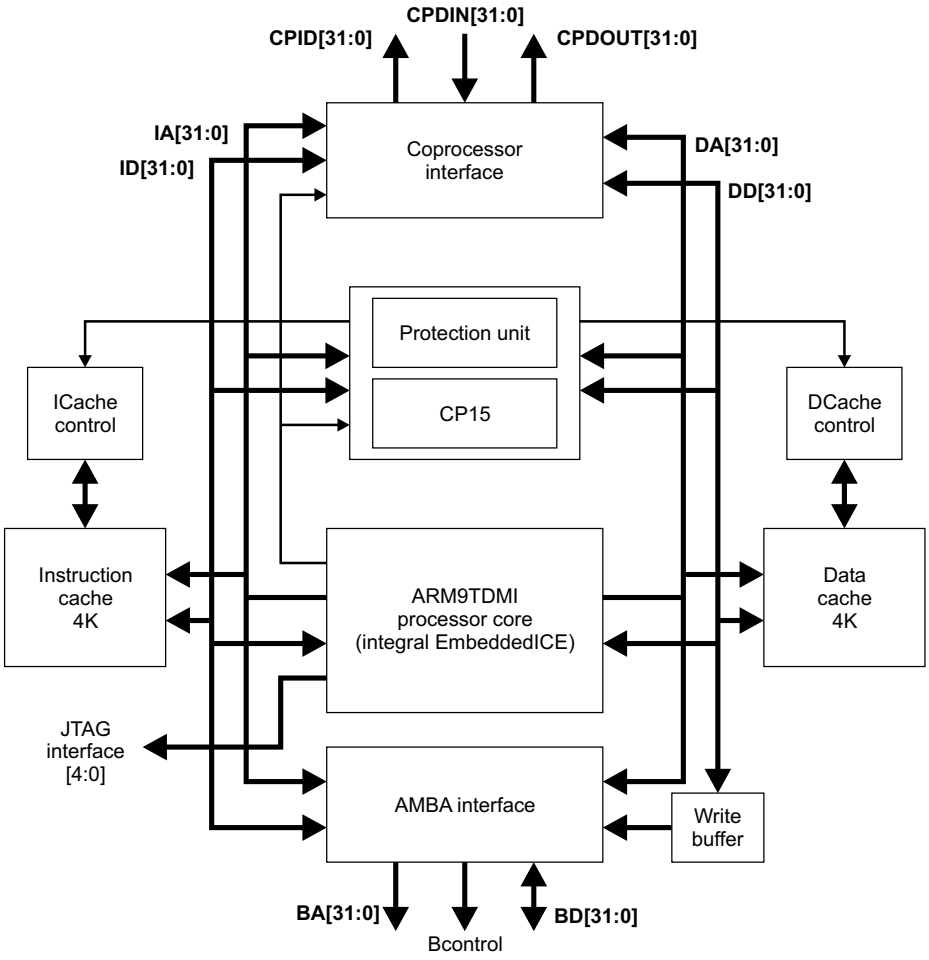

Figure 1-1 ARM940T functional block diagram

The blocks shown in Figure 1-1 on page 1-3 are described as follows:

- The ARM9TDMI is described in the *ARM9TDMI Technical Reference Manual*.
- Coprocessor 15 is described in Chapter 2 *Programmer's Model*.

- The coprocessor interface, and instruction and data cache control are described in Chapter 7 *Coprocessor Interface*.
- The protection unit is described in Chapter 3 *Protection Unit*.
- The instruction cache, data cache, and write buffer are described in Chapter 4 *Caches and Write Buffer*.
- The AMBA interface is described in Chapter 6 *Bus Interface Unit*

Chapter 2
Programmer's Model

This chapter describes the programmer's model for the ARM940T. It contains the following sections:

2.1 About the programmer's model

The ARM940T cached processor macrocell includes the ARM9TDMI microprocessor core, instruction and data caches, a write buffer, and a protection unit for defining the attributes of regions of memory.

The programmer's model of the ARM940T consists of the programmer's model of the ARM9TDMI (see *About the ARM9TDMI programmer's model* on page 2-3) with the following additions and modifications:

- The ARM940T incorporates two coprocessors:
  - CP14, which allows software access to the debug communications channel. You can access the registers defined in CP14 using `MCR` and `MRC` instructions. These are described in *Accessing CP15 registers* on page 2-6.
  - CP15, the system control coprocessor, which provides additional registers that are used to configure and control the caches, protection unit, and other system options of the ARM940T, such as big or little-endian operation. You can access the registers defined in CP15 using `MCR` and `MRC` instructions. These are described in *Accessing CP15 registers* on page 2-6.
- The ARM940T also features an external coprocessor interface that allows the attachment of a closely-coupled coprocessor on the same chip, for example, a floating point unit. You can access registers and operations provided by any coprocessors attached to the external coprocessor interface using appropriate coprocessor instructions.
- Memory accesses for instruction fetches, and data loads and stores can be cached or buffered. Cache and write buffer configuration and operation is described in detail in Chapter 4 *Caches and Write Buffer*.

2.2 About the ARM9TDMI programmer's model

The ARM9TDMI processor core implements the ARM architecture v4T and executes the ARM 32-bit instruction set and the compressed Thumb 16-bit instruction set. The ARM9TDMI programmer's model is fully described in the *ARM Architecture Reference Manual*. The *ARM9TDMI Technical Reference Manual* gives implementation details, including instruction execution cycle times.

ARMv4T specifies a small number of implementation options. The options selected in the ARM9TDMI implementation are listed in Table 2-1 on page 2-3. For comparison, the options selected for the ARM7TDMI implementation are also shown.

Table 2-1 ARM9TDMI implementation option

Processor core	ARM architecture	Data abort model	Value stored by direct STR, STRT, and STM of PC
ARM7TDMI	v4T	Base updated	Address of Inst + 12
ARM9TDMI	v4T	Base restored	Address of Inst + 12

The ARM9TDMI is code-compatible with the ARM7TDMI, with two exceptions:

- the ARM9TDMI implements the *base restored data abort model*, which significantly simplifies the software Data Abort handler
- the ARM9TDMI fully implements the instruction set extension spaces added to the ARM (32-bit) instruction set in ARMv4 and ARMv4T.

These differences are explained in more detail in the following sections:

- *Data Abort model* on page 2-3
- *Instruction set extension spaces* on page 2-4.

2.2.1 Data Abort model

The *base restored data abort model* 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 is 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.

2.2.2 Instruction set extension spaces

All ARM processors implement the undefined instruction space as one of the entry mechanisms for the undefined instruction exception. That is, ARM instructions with opcode[27:25] = 0b011 and opcode[4] = 1 are *undefined* on all ARM processors including the ARM9TDMI and ARM7TDMI.

ARMv4 and ARMv4T also introduce a number of instruction set extension spaces to the ARM instruction set. These are:

- arithmetic instruction extension space
- control instruction extension space
- coprocessor instruction extension space
- load/store instruction extension space.

Instructions in these spaces are undefined, and cause an undefined instruction exception. The ARM9TDMI fully implements all the instruction set extension spaces defined in ARMv4T as undefined instructions, allowing emulation of future instruction set additions.

2.3 CP15 register map summary

CP15 defines 16 registers. The register map for CP15 is shown in Table 2-2 on page 2-5.

Table 2-2 CP15 register map

Register	Read	Write
0	ID code [a]	Unpredictable
0	Cache type [a]	Unpredictable
1	Control	Control
2	Cachable [a]	Cachable [a]
3	Write buffer control	Write buffer control
4	Reserved	Reserved
5	Protection region access permissions [a]	Protection region access permissions [a]
6	Protection region base and size control [a]	Protection region base and size control [a]
7	Unpredictable	Cache operations
8	Reserved	Reserved
9	Cache lockdown	Cache lockdown
10-14	Reserved	Reserved
15	Test [b]	Test [b]

a. Register locations 0, 2, 5, and 6 each provide access to more than one register. The register accessed depends on the value of the `opcode_2` field.
b. Not accessed in normal operations

2.3.1 Accessing CP15 registers

The terms and abbreviations shown in Table 2-3 on page 2-6 are used throughout this section.

Table 2-3 CP15 abbreviations

Term	Abbreviation	Description
Unpredictable	UNP	For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration.
Should be zero	SBZ	When writing to this location, all bits of this field should be 0.

In all cases, reading from, or writing any data values to any CP15 registers, including those fields specified as *unpredictable* or *should be zero*, does not cause any permanent damage.

All CP15 register bits that are defined and contain state, are set to zero by **BnRES** except the V bit in register 1, which takes the value of macrocell input **VINITHI** when **BnRES** is asserted.

You can only access CP15 registers with `MRC` and `MCR` instructions in a privileged mode. The instruction bit pattern of the `MCR` and `MRC` instructions is shown in Figure 2-1 on page 2-6. The assembler for these instructions is:

```
MCR/MRC{cond} P15,opcode_1,Rd,CRn,CRm,opcode_2
```

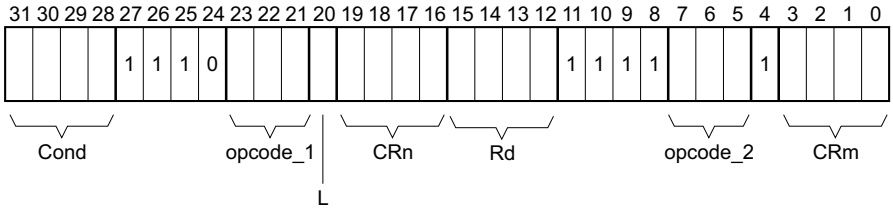

Figure 2-1 CP15 MRC and MCR bit pattern

Instructions CDP, LDC, and STC, together with unprivileged MRC and MCR instructions to CP15, cause the undefined instruction trap to be taken. The CRn field of MRC and MCR instructions specifies the coprocessor register to access. The CRm field and opcode_2 fields specify a particular action when addressing registers. The L bit distinguishes between an MRC (L=1) and an MCR (L=0).

Note

Attempting to read from a nonreadable register, or to write to a nonwritable register causes unpredictable results.

The opcode_1, opcode_2, and CRm fields should be zero, except when the values specified are used to select the desired operations, in all instructions that access CP15. Using other values results in unpredictable behavior.

2.3.2 Register 0, ID code

This is a read-only register that returns a 32-bit device ID code. You can access the ID code register by reading CP15 register 0 with the opcode_2 field set to any value other than 1. For example:

```
MRC cp15, 0, rd, c0, c0,{0,2-7}; returns ID register
```

The contents of the ID code are shown in Table 2-4 on page 2-7.

Table 2-4 ID code register

Register bits	Function	Value
31:12	Implementer	0x41 (identifies ARM)
23:16	Architecture version	0x2
15:4	Part number	0x940
3:0	Layout revision	Revision

2.3.3 Register 0, cache type

This is a read-only register that contains information about the size and architecture of the *Instruction Cache* (ICache) and *Data Cache* (DCache), allowing operating systems to establish how to perform such operations as cache cleaning and lockdown. All ARMv4T and later cached processors contain this register, allowing RTOS vendors to produce future-proof versions of their operating systems.

You can access the cache type register by reading CP15 register 0 with the opcode_2 field set to 1. For example:

```
MRC cp15, 0, rd, c0, c0, 1; returns cache details
```

The register contains information about the size and architecture of the caches. The format of the register is shown in Figure 2-2 on page 2-8.

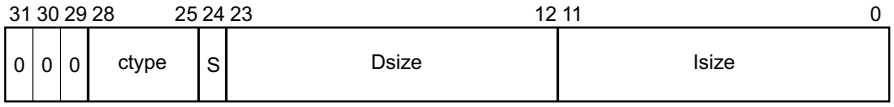

Figure 2-2 Cache type register format

ctype The ctype field determines the cache type.

S bit Specifies if the cache is a unified cache or separate ICache and DCache.

Dsize Specifies the size, line length, and associativity of the DCache.

Isize Specifies the size, line length, and associativity of the ICache.

The Dsize and Isize fields in the cache type register have the same format. This is shown in Figure 2-3 on page 2-8

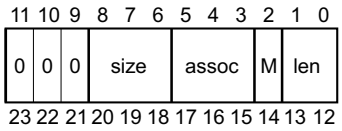

Figure 2-3 Dsize and Isize field format

size The size field determines the cache size in conjunction with the M bit.

assoc The assoc field determines the cache associativity in conjunction with the M bit.

M bit The multiplier bit. Determines the cache size and cache associativity values in conjunction with the size and assoc fields.

len The len field determines the line length of the cache.

The register values for the ARM920T cache type register are listed in Table 2-5 on page 2-9.

Table 2-5 Cache type register format

Function		Register bits	Value
Reserved		31:29	0b000
ctype		28:25	0b0111
S		24	0b1 = Harvard cache
Dsize	Reserved	23:21	0b000
	size	20:18	0b011 = 4KB
	assoc	17:15	0b110 = 64 way
	M	14	0b0 = 1x base parameters
	len	13:12	0b01 = 4 words per line
Isize	Reserved	11:9	0b000
	size	8:6	0b011 = 4KB
	assoc	5:3	0b110 = 64 way
	M	2	0b0 = 1x base parameters
	len	1:0	0b01 = 4 words per line

The size of the cache is determined by the size field and the M bit. The M bit is 0 for the data and instruction caches. Bits [20:18] for the *Data Cache* (DCache) and bits [8:6] for the *Instruction Cache* (ICache) are the size field. Table 2-6 on page 2-9 shows the cache size encoding.

Table 2-6 Cache size encoding (M=0)

size field	Cache size
0b000	512B
0b001	1KB
0b010	2KB
0b011	4KB

Table 2-6 Cache size encoding (M=0) (continued)

size field	Cache size
0b100	8KB
0b101	16KB
0b110	32KB
0b111	64KB

The associativity of the cache is determined by the assoc field and the M bit. The M bit is 0 for the data and instruction caches. Bits [17:15] for the DCache and bits [5:3] for the ICache are the assoc field. Table 2-7 on page 2-10 shows the cache associativity encoding.

Table 2-7 Cache associativity encoding (M=0)

assoc field	Associativity
0b000	Direct mapped
0b001	2-way
0b010	4-way
0b011	8-way
0b100	16-way
0b101	32-way
0b110	64-way
0b111	128-way

The line length of the cache is determined by the len field. Bits [13:12] for the DCache and bits [1:0] for the ICache are the len field. Table 2-8 on page 2-11 shows the line length encoding.

Table 2-8 Line length encoding

len field	Cache line length
00	2 words (8 bytes)
01	4 words (16 bytes)
10	8 words (32 bytes)
11	16 words (64 bytes)

2.3.4 Register 1, control register

This contains the global control bits of the ARM940T. 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. The format of CP15 register 1 is shown in Table 2-9.

Table 2-9 CP15 register 1

Register bits	Name	Function	Value
31	iA bit	Asynchronous clocking select	See Table 2-10 on page 2-12.
30	nF bit	nFastBus select	See Table 2-10 on page 2-12.
29:14	-	Reserved	Read = Unpredictable. Write = Should be zero.
13	V bit	Base location of exception registers	0 = Low addresses = `0x00000000`. 1 = High addresses = `0xFFFF0000`. This bit is cleared to 0 at reset.
12	I bit	ICache enable bit	0 = ICache disabled. 1 = ICache enabled. See *Enabling and disabling the ICache* on page 4-5.
11:8	-	Reserved	Should be zero.

Table 2-9 CP15 register 1 (continued)

Register bits	Name	Function	Value
7	B bit	Big-end bit	0 = Little-endian operation. 1 = Big-endian operation. This bit is cleared to 0 at reset.
6:3	-	Reserved	Should be one.
2	C bit	DCache enable bit	0 = DCache disabled. 1 = DCache enabled. See *Enabling and disabling the DCache* on page 4-8.
1	-	Reserved	Should be zero.
0	P bit	Protection unit enable	0 = Protection unit disabled. 1 = Protection unit enabled. See *Enabling the protection unit* on page 3-3.

Register 1 bits [31:30] select the clocking mode of the ARM940T, as shown in Table 2-10 on page 2-12. Clocking modes are described in Chapter 5 *Clock Modes*.

Table 2-10 Clocking modes

Clocking mode	iA	nF
FastBus mode	0	0
Synchronous	0	1
Reserved	1	0
Asynchronous	1	1

2.3.5 Register 2, instruction and data cachable registers

This location provides access to two registers that contain the cachable attributes for each of eight memory areas. The two registers provide individual control for the I and D address spaces. The opcode_2 field determines if the instruction or data cachable attributes are programmed:

- If the opcode_2 field = 0, the data cachable bits are programmed. For example:

  ```
  MCR p15,0,Rd,c2,c0,0; Write data cachable bits
  MRC p15,0,Rd,c2,c0,0; Read data cachable bits
  ```

- If the opcode_2 field = 1 the instruction cachable bits are programmed. For example:

  ```
  MCR p15,0,Rd,c2,c0,1; Write instruction cachable bits
  MRC p15,0,Rd,c2,c0,1; Read instruction cachable bits
  ```

The format of the data and instruction cachable bits are the same, as shown in Table 2-11 on page 2-13. Setting a bit makes an area cachable, clearing it makes it noncachable.

Table 2-11 Cachable register format

Register bit	Function
7	Cachable bit (C_7) for area 7
6	Cachable bit (C_6) for area 6
5	Cachable bit (C_5) for area 5
4	Cachable bit (C_4) for area 4
3	Cachable bit (C_3) for area 3
2	Cachable bit (C_2) for area 2
1	Cachable bit (C_1) for area 1
0	Cachable bit (C_0) for area 0

The use of register 2 is described in Chapter 3 *Protection Unit*.

2.3.6 Register 3, write buffer control register

This register contains a write buffer control (bufferable) attribute bit for each of the eight areas of memory. Each bit is used in conjunction with the cachable bit to control write buffer operation. For a description of buffer behavior, see *The write buffer* on page 4-12.

Setting a bit makes an area bufferable, clearing a bit makes an area unbuffered. For example:

```
MCR p15,0,Rd,c3,c0,0    ; Write data bufferable bits
MRC p15,0,Rd,c3,c0,0    ; Read data bufferable bits
```

Note

The opcode_2 field should be 0 because the write buffer only operates on data regions. The following table, therefore, only applies to the DCache.

The format of the write buffer control register is shown in Table 2-12 on page 2-14.

Table 2-12 Write buffer control register

Register bit	Function
7	Write buffer control bit (B_d7) for data area 7
6	Write buffer control bit (B_d6) for data area 6
5	Write buffer control bit (B_d5) for data area 5
4	Write buffer control bit (B_d4) for data area 4
3	Write buffer control bit (B_d3) for data area 3
2	Write buffer control bit (B_d2) for data area 2
1	Write buffer control bit (B_d1) for data area 1
0	Write buffer control bit (B_d0) for data area 0

The use of register 3 is described in Chapter 3 *Protection Unit*.

2.3.7 Register 4, reserved

You must not access (read or write) this register because it causes unpredictable behavior.

2.3.8 Register 5, instruction and data space protection registers

These registers contain the access permission bits for the instruction and data protection regions. The opcode_2 field determines if the instruction or data access permissions are programmed:

- If the opcode_2 field = 0, the data space bits are programmed. For example:

```
MCR p15,0,Rd,c5,co,0     ; Write data space access permissions
MRC p15,0,Rd,c5,co,0     ; Read data space access permissions
```

- If the opcode_2 field =1, the instruction space bits are programmed. For example:

```
MCR p15,0,Rd,c5,co,1     ; Write instruction space access permissions
MRC p15,0,Rd,c5,co,1     ; Read instruction space access permissions
```

Each register contains the access permission bits, apn[1:0], for the eight areas of instruction or data memory, as shown in Table 2-13 on page 2-15.

Table 2-13 Protection space register format

Register bit	Function
15:14	ap7[1:0] bits of area 7
13:12	ap6[1:0] bits of area 6
11:10	ap5[1:0] bits of area 5
9:8	ap4[1:0] bits of area 4
7:6	ap3[1:0] bits of area 3
5:4	ap2[1:0] bits of area 2
3:2	ap1[1:0] bits of area 1
1:0	ap0[1:0] bits of area 0

The values of the Iapn[1:0] and Dapn[1:0] bits define the access permission for each area of memory. The access encoding is shown in Table 2-14 on page 2-16.

Note

On reset, the values of the Iapn[1:0] and Dapn[1:0] bits for all areas are undefined. However, on reset, the protection unit is disabled and all areas are effectively set to *no access*. Therefore, you must program the protection space registers before you enable the protection unit.

Table 2-14 Permission encoding

I/Dapn[1:0]	Permission
00	No access
01	Privileged mode access only
10	Privileged mode full access, User mode read only
11	Full access

The use of register 5 is described in Chapter 3 *Protection Unit*.

2.3.9 Register 6, protection region base and size registers

This register allows you to define 16 programmable regions in memory, made up of eight instruction and eight data regions. Individual control is provided for the instruction and data memory regions. These registers define the base and size of each of the eight areas of memory. The values are ignored when the protection unit is disabled.

On reset, only the region enable bit for each region is reset to 0, all other bits are undefined. You must program at least one instruction and data memory region before you enable the protection unit.

The `opcode_2` field defines if the data or instruction protection regions are to be programmed. The CRm field selects the region number. Table 2-15 on page 2-16 shows the data protection region registers.

Table 2-15 CP15 data protection region registers

ARM instruction	Protection region register
`MCR/MRC p15, 0, Rd, c6, c7, 0`	Data memory region 7
`MCR/MRC p15, 0, Rd, c6, c6, 0`	Data memory region 6
`MCR/MRC p15, 0, Rd, c6, c5, 0`	Data memory region 5
`MCR/MRC p15, 0, Rd, c6, c4, 0`	Data memory region 4
`MCR/MRC p15, 0, Rd, c6, c3, 0`	Data memory region 3

Table 2-15 CP15 data protection region registers (continued)

ARM instruction	Protection region register
`MCR/MRC p15, 0, Rd, c6, c2, 0`	Data memory region 2
`MCR/MRC p15, 0, Rd, c6, c1, 0`	Data memory region 1
`MCR/MRC p15, 0, Rd, c6, c0, 0`	Data memory region 0

Table 2-16 on page 2-17 shows the instruction protection region registers.

Table 2-16 CP15 instruction protection region registers

ARM instruction	Protection region register
`MCR/MRC p15, 0, Rd, c6, c7, 1`	Instruction memory region 7
`MCR/MRC p15, 0, Rd, c6, c6, 1`	Instruction memory region 6
`MCR/MRC p15, 0, Rd, c6, c5, 1`	Instruction memory region 5
`MCR/MRC p15, 0, Rd, c6, c4, 1`	Instruction memory region 4
`MCR/MRC p15, 0, Rd, c6, c3, 1`	Instruction memory region 3
`MCR/MRC p15, 0, Rd, c6, c2, 1`	Instruction memory region 2
`MCR/MRC p15, 0, Rd, c6, c1, 1`	Instruction memory region 1
`MCR/MRC p15, 0, Rd, c6, c0, 1`	Instruction memory region 0

Each protection region register has the format shown in Table 2-17 on page 2-17.

Table 2-17 CP15 protection region register format

Register bit	Function
31:12	Base address
11:6	Unused
5:1	Area size (See Table 2-18 on page 2-18)
0	Region enable. Reset to disable (0)

The region base must be aligned to an area size boundary, where the area size is defined in its respective protection region register. The behavior is undefined if this is not the case. Area sizes are given in Table 2-18 on page 2-18.

Table 2-18 Area size encoding

Bit encoding	Area size
0b00000 to 0b01010	Reserved
0b01011	4KB
0b01100	8KB
0b01101	16KB
0b01110	32KB
0b01111	64KB
0b10000	128KB
0b10001	256KB
0b10010	512KB
0b10011	1MB
0b10100	2MB
0b10101	4MB
0b10110	8MB
0b10111	16MB
0b11000	32MB
0b11001	64MB
0b11010	128MB
0b11011	256MB
0b11100	512MB
0b11101	1GB
0b11110	2GB
0b11111	4GB

Register 6 is described in Chapter 3 *Protection Unit*.

Example base setting

An 8KB size region aligned to the 8KB boundary at `0x00002000` (covering the address range `0x00002000-0X00003FFF`) would be programmed to `0x00002019`.

2.3.10 Register 7, cache operations register

Register 7 is a write-only register used to manage the ICache and DCache. A write to this register can be used to perform the following operations:

- flush ICache and DCache
- prefetch an ICache line
- wait for interrupt
- drain the write buffer
- clean and flush the DCache.

The ARM940T uses a subset of the ARMv4 functions, as defined in the *ARM Architecture Reference Manual*. The available operations are summarized in Table 2-19 on page 2-19.

Table 2-19 Cache operations writing to register 7

ARM instruction	Data	Function
`MCR p15, 0, Rd, c7, c5, 0`	Should be zero	Flush ICache
`MCR p15, 0, Rd, c7, c5, 2`	Index/segment	Flush ICache single entry
`MCR p15, 0, Rd, c7, c6, 0`	Should be zero	Flush DCache
`MCR p15, 0, Rd, c7, c6, 2`	Index/segment	Flush DCache single entry
`MCR p15, 0, Rd, c7, c10, 2`	Index/segment	Clean DCache single entry
`MCR p15, 0, Rd, c7, c13, 1`	Address	Prefetch ICache line
`MCR p15, 0, Rd, c7, c14, 2`	Index/segment	Clean and flush DCache single entry
`MCR p15, 0, Rd, c7, c8, 2`	Should be zero	Wait for interrupt
`MCR p15, 0, Rd, c7, c10, 4`	Should be zero	Drain write buffer

Should be zero means the value transferred in the Rd.

A read from this register returns an unpredictable value.

Index/segment format

Where the required value is an index/segment, the format is shown in Figure 2-4 on page 2-20.

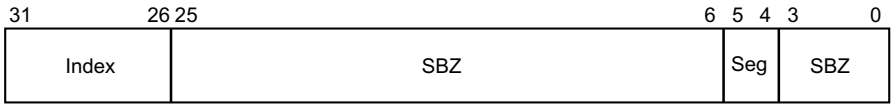

Figure 2-4 Index/segment format for cache operations

ICache prefetch data format

For the ICache prefetch operation, the data format is shown in Figure 2-5 on page 2-20.

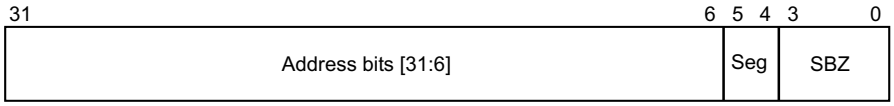

Figure 2-5 Address format for ICache prefetch operations

The use of register 7 is described in Chapter 4 *Caches and Write Buffer*.

Wait for interrupt

Wait for interrupt allows you to place the ARM940T in a low-power standby mode. When the operation is invoked, all clocks in the processor are frozen 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
```

The following instruction causes the same affect and has been added for backward compatibility with StrongARM SA-1:

```
MCR p15, 0, Rd, c15, c8, 2
```

This stalls the processor, with internal clocks held HIGH from the time that this instruction is executed until one of the signals **nFIQ**, **nIRQ**, or **EDBGRQ** is asserted. Also, if the debugger sets the debug request bit in the EmbeddedICE unit control register, the wait-for-interrupt condition is terminated.

In the case of **nFIQ** and **nIRQ**, the processor 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 the interrupts are enabled, the ARM core is guaranteed to take the interrupt before executing the instruction after the wait-for-interrupt. If you use debug request to wake up the system, the processor enters debug-state before executing any more instructions.

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 function is invoked by a write to CP15 register 7 using the following ARM instruction:

```
MCR p15, 0, Rd, c7, c10, 4
```

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

2.3.11 Register 8, reserved

You must not access (read or write) this register because it causes unpredictable behavior.

2.3.12 Register 9, instruction and data lockdown registers

These registers allow regions of the cache to be locked down. The `opcode_2` field determines if the instruction or data caches are programmed:

- If the `opcode_2` field = 0, the data lockdown bits are programmed. For example:

  ```
  MCR/MRC p15, 0, Rd, c9, c0, 0    ; data lockdown control
  ```

- If the `opcode_2` field = 1, the instruction lockdown bits are programmed. For example:

  ```
  MCR/MRC p15, 0, Rd, c9, c0, 1    ; instruction lockdown control
  ```

The format of the registers, Rd, transferred during this operation, is shown in Table 2-20 on page 2-22.

Table 2-20 Lockdown register format

Register bit	Function
31	Load bit
30:6	Reserved
5:0	Cache index

Note

The segment number is not specified because cache lines are locked down across all four segments (16-word granularity).

The use of register 9 is described in Chapter 4 *Caches and Write Buffer*.

2.3.13 Registers 10 to 14, reserved

You must not access (read or write) these registers because it causes unpredictable behavior.

2.3.14 Register 15, test/debug register

The DTRRobin and ITRRobin bits set the respective caches into a pseudo round-robin replacement mode. The format of register 15 is shown in Table 2-21 on page 2-22.

Table 2-21 CP15 register 15

Register bit	Function
31:4	Reserved
3	ITRRobin
2	DTRRobin
1:0	Reserved

Chapter 3
Protection Unit

This chapter describes the ARM940T protection unit. It contains the following sections:

3.1 About the protection unit

The protection unit is used to partition memory and set individual protection attributes for each partition. The instruction address space and the data address space can each be divided into up to eight regions of variable size.

Figure 3-1 on page 3-2 shows how the protection unit functions.

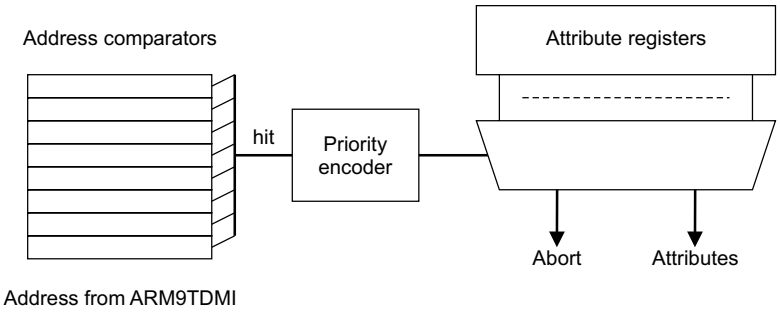

Figure 3-1 ARM940T protection unit

The protection unit is programmed using CP15 registers 1, 2, 3, 5, and 6.

3.2 Enabling the protection unit

Before the protection unit is enabled, you must program at least one valid data and instruction protection region. If they are not programmed, the ARM940T can enter a state that is recoverable only by reset. Setting bit 0 of CP15 register 1, the control register, enables the protection unit.

When the protection unit is disabled, all instruction fetches are noncachable and all data accesses are noncachable and nonbufferable.

3.3 Memory regions

The instruction and data address spaces can both be partitioned into a maximum of eight regions. Each region is specified by the following:

- *Region base address* on page 3-4
- *Region size* on page 3-5
- cache and write buffer configuration
- read and write access permissions.

The ARM architecture uses constants known as *inline literals* to perform address calculations. These constants are automatically generated by the assembler and compiler and are stored with the instruction. To ensure correct operation, you must define an area of memory from where code is to be executed for both the instruction and data address spaces.

The base address and size properties are programmed using CP15 register 6. Table 3-1 on page 3-4 shows the format of the protection register.

Table 3-1 Protection register format

Register bit	Function
31:12	Region base address
11:6	Unused
5:1	Region size
0	Region enable Reset to disable (0)

3.3.1 Region base address

The base address defines the start of the memory region. This must be aligned to a region-sized boundary. For example, if a region size of 8KB is programmed for a given region, the base address must be a multiple of 8KB.

If the region is not aligned correctly, this results in unpredictable behavior.

3.3.2 Region size

The region size is specified as a 5-bit value, encoding a range of values from 4KB to 4GB. The encoding is shown in Table 3-2 on page 3-5.

Table 3-2 Region size encoding

Bit encoding	Area size
0b00000 to 0b01010	Reserved
0b01011	4KB
0b01100	8KB
0b01101	16KB
0b01110	32KB
0b01111	64KB
0b10000	128KB
0b10001	256KB
0b10010	512KB
0b10011	1MB
0b10100	2MB
0b10101	4MB
0b10110	8MB
0b10111	16MB
0b11000	32MB
0b11001	64MB
0b11010	128MB
0b11011	256MB
0b11100	512MB
0b11101	1GB
0b11110	2GB
0b11111	4GB

Note

Any value less than 0b01011 programmed in Rd[5:1] results in unpredictable behavior.

3.3.3 Partition attributes

Each region has a number of attributes associated with it. These control how a memory access is performed when the processor core issues an address that falls within a given region. The attributes are:

- cachable
- bufferable (for data regions only)
- read/write permissions.

This information is specified by programming CP15 registers 2, 3, and 5 (see Chapter 2 *Programmer's Model*). If an access fails its protection check (for example, if a User mode application attempts to access a privileged-mode access only region), a memory abort occurs. The processor enters the abort exception mode, branching to the Data Abort or Prefetch Abort vector accordingly.

The cachable and bufferable bits in CP15 registers 2 and 3 are together used to select one of four cache and write buffer configurations. These are described in *Write buffer operation* on page 4-12.

3.4 Overlapping regions

The protection unit can be programmed with two or more overlapping regions. When overlapping regions are programmed, a fixed priority scheme is applied to determine the region attributes that are applied to the memory access.

Attributes for region 7 take highest priority. Attributes for region 0 take lowest priority. For example:

- Data region 2 is programmed to be 4KB in size, starting from address `0x3000` with Dap[1:0]=10 (Privileged mode full access, User mode read only).
- Data region 1 is programmed to be 16KB in size, starting from address `0x0` with Dap[1:0]=01 (Privileged mode access only).

When the processor performs a data load from address `0x3010` while in User mode, the address falls into both region 1 and region 2, as shown in Figure 3-2 on page 3-7. Because there is a clash, the attributes associated with region 2 are applied. In this case, the load does not abort.

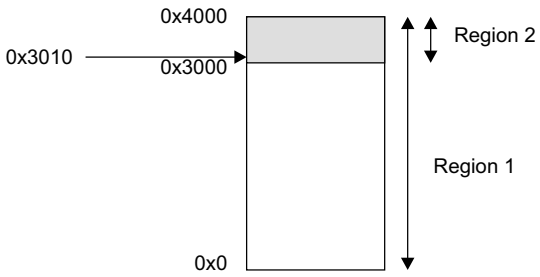

Figure 3-2 Overlapping memory regions

3.4.1 Background regions

Overlapping regions increase the flexibility of mapping the eight regions onto physical memory devices in the system. You can also use the overlapping properties to specify a background region. For example, there might be a number of physical memory areas sparsely distributed across the 4GB address space. If a programming error occurs therefore, it might be possible for the processor to issue an address that does not fall into any defined region.

If the address issued by the processor falls outside any of the defined regions, the ARM940T protection unit is hard-wired to abort the access. This behavior can be overridden by programming region 0 to be a 4GB background region. In this way, if the address does not fall into any of the other seven regions, the access is controlled by the attributes specified for region 0.

Chapter 4
Caches and Write Buffer

This chapter describes the *Instruction Cache* (ICache), *Data Cache* (DCache), and the write buffer. It contains the following sections:

4.1 Cache architecture

The ARM940T has a 4KB *Instruction Cache* (ICache), a 4KB *Data Cache* (DCache), and an 8-word write buffer. The ICache and DCache have similar architectures, as shown in Figure 4-1 on page 4-2.

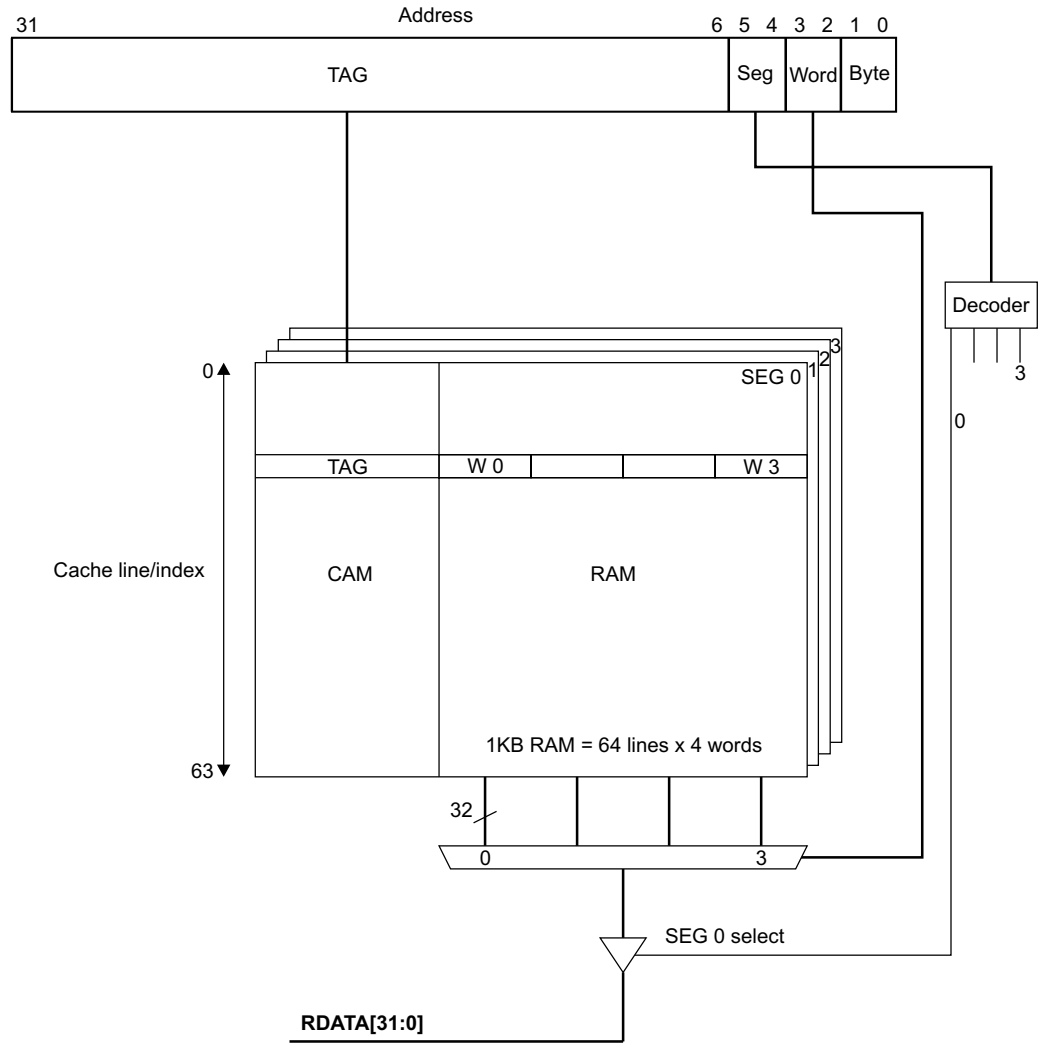

Figure 4-1 4KB cache used for the instruction and data caches

Each cache comprises four fully-associative 1KB segments that support single-cycle reads, and either one or two-cycle writes depending on the sequentiality of the access.

Each cache segment consists of 64 *Content Addressable Memory* (CAM) rows that each select one of 64 RAM four-word long lines. During a cache access, a segment is selected and the access address is compared with the 64 TAGs in the CAM. If a match occurs (or a *hit*), the matched line is enabled and the data can be accessed. If none of the TAGs match (a *miss*), then external memory must be accessed, unless the access is a buffered write, when the write buffer is used.

If a read access from a cachable memory region misses, new data is loaded into one of the 64 row lines of the selected segment. This is an *allocate on read-miss* replacement policy. Selection is performed by a randomly clocked target row counter.

Critical or frequently-accessed instructions or data can be locked into the cache by restricting the range of the target counter. Locked lines cannot be replaced and remain in the cache until they are unlocked or flushed.

The CAM allows 64 address TAGs to be stored for an address that selects a given segment (64-way associativity). This reduces the chance of an address sequence in, for example, a program loop that constantly selects the same segment, from replacing data that is required again in a later iteration of the loop. The overhead for high associativity is the requirement to store a larger TAG. In the case of the ARM940T, this is 26 bits per line.

Figure 4-2 on page 4-3 shows how the 4KB ICache and 4KB DCache are addressed.

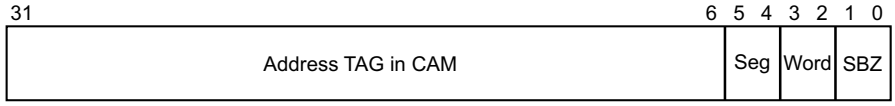

Figure 4-2 ARM940T Instruction/data cache addressing

The address bits are assigned as follows:

Bits 31:6 Select an address tag in CAM.

Bits 5:4 Selects one of the four cache segments.

Bits 3:2 Selects a word in the cache line.

Two additional bits are used on each segment row line:

Valid bit This is set when the cache line has been written with valid data. Only a valid line can return a hit during a CAM lookup. On reset, all the valid bits are cleared.

Dirty bit This is associated with write operations in the DCache and is used to indicate that a cache line contains data that differs from data stored at the address in external memory. Data can only be marked dirty if it resides in a writeback protection region.

4.2 ICache

The ARM940T has a 4KB ICache comprising four 64-way associative segments of 16 bytes per line per segment. The ICache uses the physical address generated by the processor core. It uses a policy of *allocate on read-miss* and is always reloaded one cache line (four words) at a time, through the external interface.

The ICache is always disabled on reset.

4.2.1 Enabling and disabling the ICache

You can enable the ICache by setting bit 12 of the CP15 control register. You must only enable the cache if you have already enabled the protection unit, or if you enable them simultaneously. When the ICache is enabled, a cachable read-miss causes lines to be placed in the ICache.

You can enable the ICache and protection unit simultaneously with a single write to the CP15 control register, although you must have programmed at least one protection region before you enable the protection unit. You can lock critical or frequently-accessed instructions into the ICache with a granularity of 64 bytes.

Note

Instructions in this lockdown region are not replaced and remain in the ICache although they are not immune to being flushed.

4.2.2 ICache operation

When enabled, the ICache operation is also controlled by the *Gated Cachable instruction* (GCi) bit stored in the protection unit. This selectively enables or disables caching for different memory regions. The GCi bit affects ICache operation as follows:

- Successful cache read:

  Data is returned to the core regardless of the state of the GCi bit.

- Unsuccessful cache read:

  If the GCi bit is 1, a cachable code area and protection unit are enabled, and a linefetch of four words is performed. The data is written into a randomly chosen line in the ICache. If the GCi bit is 0, a single-word external access is performed to fetch the requested instruction. The cache is not updated.

Locked down code is always found on ICache searches. Lines containing locked down code cannot be selected for replacement during a linefetch.

You can disable the ICache by clearing bit 12 of the CP15 control register. This has the effect of preventing all ICache look-ups and linefills, and forces all instruction fetches to be performed by single external accesses.

4.2.3 ICache validity

The ARM940T does not support external memory snooping. Therefore, if self-modifying code is written, the instructions in the ICache might become invalid. Similarly, if the instruction protection regions are reprogrammed, code might exist in the cache that should be in a noncachable region. In either of these cases, the ICache must be flushed by the programmer.

You can flush the entire ICache by software in one operation, or you can flush it one line at a time by writing to the CP15 cache operations register (register 7). The ICache is automatically flushed during reset. The ICache never has to be cleaned because its only source of data is from external memory. The processor only ever performs reads from the ICache.

Flushing the entire cache

As shown in Table 2-19 on page 2-19, you can flush the entire ICache using an `MCR` instruction. In this case, the contents of the ARM register transferred to CP15 should be zero. The code segment shown below can be used. The use of r0 is arbitrary:

```
M0V r0,#0           ; Clear r0
MCR p15,r0,c7,c5,0  ; Flush entire ICache
```

Flushing the entire cache also flushes any locked down code. If the ICache contains locked down code, the programmer must flush lines individually, avoiding the lines used for the locked down code.

Flushing a single cache line

Single cache lines can be flushed. To do this, the cache line must be specified in Rd. The ARM940T ICache comprises four segments, each with 64 lines, which means that both the segment and line number index must be specified.

The format of Rd for this operation is shown in Table 4-1 on page 4-7.

Table 4-1 CP15 register 7

Rd bit position	Function
31:26	Index
25:6	Should be zero
5:4	Segment
3:0	Should be zero

For example, the following code sequence can be used to flush line 25 of segment 2 in the ICache:

```
MOV r0,#0x64000000      ; Specify line 25
ORR r0,r0,#0x20         ; Specify cache segment 2,
                        ; r0=0x64000020
MCR p15,0,r0,c7,c5,1    ; Flush the ICache line
```

4.3 DCache

The ARM940T has a 4KB DCache comprising 256 lines of 16 bytes (four words), arranged as four 64-way associative segments. The DCache uses the physical address generated by the processor core. It uses an *allocate on read-miss* policy, and is always reloaded a cache line (four words) at a time through the external interface.

The DCache supports both *Write-Back* (WB) and *Write-Through* (WT) modes. For data stores that hit in the DCache, in WB mode the cache line is updated, and an additional dirty bit associated with the cache line is set. This indicates that the internal version of the data differs from that in the external memory. In WT mode, a store that hits in the DCache causes the cache line to be updated but not marked as dirty, because the data store is also written to the write buffer to keep the external memory consistent. In both WB and WT modes, a store that misses in the cache is sent to the write buffer. When a line fetch causes a cache line to be evicted from the DCache, the dirty bit for the victim line is read and if the line contains valid and dirty data, it is written back to the write buffer before the linefill replaces it.

The GCd bit and the GBd bit control the DCache behavior. For this reason the protection unit must be enabled when the DCache is enabled.

4.3.1 Enabling and disabling the DCache

You can enable the DCache by writing to bit 2 of the CP15 control register. You must only enable the cache if you have already enabled the protection unit, or if you enable them simultaneously. You can enable the DCache and protection unit simultaneously with a single write to the CP15 control register.

You can disable the DCache by clearing bit 2 of the CP15 control register.

The DCache is automatically disabled and flushed on reset.

When the DCache is disabled, DCache searches are prevented. This has the effect of making all data accesses noncachable and forcing the ARM940T to perform external accesses. The write buffer control is still decoded from the GBd and the GCd bit. The GCd bit is forced to 0 (noncachable).

Note

When the caches are disabled their contents are preserved. This means that if a write to an address that was held in the data cache occurs while the data cache is disabled, the update does not affect the data cache. If the data cache is then switched back on, it still holds the out of date version of the data, which appears valid. This results in unrecoverable data corruption. To prevent this, you are recommended to always clean and flush the data cache before you disable it

4.3.2 Operation of the GCd bit and GBd bit

The GCd bit determines if data being read must be placed in the DCache and used for subsequent reads. Typically, main memory is marked as cachable to reduce memory access time and therefore increase system performance. It is usual to mark I/O space as noncachable. For example, if a processor is polling a memory-mapped register in I/O space, it is important that the processor is forced to read data direct from the external peripheral, and not from a copy of initial data held in the DCache.

The GBd and GCd bits affect writes that both hit and miss in the DCache. For details of the ways these bits are decoded to perform different types of writes, see *The write buffer* on page 4-12.

4.3.3 DCache operation

When the DCache is enabled, it is searched when the processor performs a data load or store. If the cache hits on a load, data is returned to the core regardless of the state of the GCd bit. If the cache read misses, the GCd bit is examined:

- If the GCd bit is 1, the cachable data area and protection unit are enabled. A linefill of four words is performed, and the data is written into a randomly chosen line in the DCache.
- If the GCd bit is 0, a single or multiple external access is performed and the cache is not updated.

Stores that hit in the cache always update the cache line, regardless of the GCd bit. Stores that miss the cache use the GCd and GBd bits to determine if the write is buffered (see *The write buffer* on page 4-12). A write miss is not loaded into the cache as a result of that miss.

Noncachable load multiples and *NonCachable NonBufferable* (NCNB) store multiples are broken up on 4KB boundaries (the minimum protection region size), allowing a protection check to be performed in case the *Load Multiple* (`LDM`) or *Store Multiple* (`STM`) crosses into a region with different protection properties.

DCache lockdown is supported with 16-word granularity. Data that is locked down always hits on DCache searches, and lines containing locked down data cannot be selected for replacement during a linefill.

Back-to-back stores from adjacent store instructions to the same segment within the DCache cause a cache stall, requiring two cycles for the cache write. A burst of stores from a single store multiple instruction does not cause stalls and allows one write per cycle to be performed. Single back-to-back stores to different segments are also performed without a stall, allowing one write per cycle.

4.3.4 DCache validity

The ARM940T does not support memory translation so the data in the DCache can always be considered valid within the context of the ARM940T. However, if external memory translation is used, and the mappings are changed, the DCache data is no longer consistent with external memory, and the DCache must be flushed by the programmer.

The ARM940T does not support external memory snooping. Any shared data memory space therefore, must not be cachable. Additionally, if the data protection regions are reprogrammed, data already in the cache might now be in a noncachable region, and the cache must be flushed.

4.3.5 DCache clean and flush

The DCache has flexible cleaning and flushing utilities that allow the following operations:

- The whole DCache can be invalidated (*flush DCache*) in one operation without writing back dirty data.
- Individual lines can be invalidated without writing back any dirty data (*flush DCache single entry*).
- Cleaning can be performed on a line-by-line basis. The data is only written back through the write buffer when a dirty line is encountered, and the cleaned line remains in the cache (*clean DCache single entry*).
- Individual lines can be cleaned and flushed in one operation (*clean and flush DCache single entry*).

Note

Flushing the entire DCache also flushes any locked down code, without resetting the victim counter range.

The cleaning and flushing utilities are performed using CP15 register 7, in a similar manner to that described in *ICache* on page 4-5 for ICache. The format of Rd transferred to CP15 is as shown in Figure 4-2 on page 4-3 for all register 7 operations.

It is usual for the cache to be cleaned before being flushed, so that external memory is updated with any dirty data. Example 4-1 on page 4-11 shows how the entire cache can be cleaned and flushed:

Example 4-1 Clean and flush the entire DCache

```
    MOV r1,#0               ; Initialize line counter, r1
outer_loop
    MOV r0,#0               ; Initialize segment counter, r0
inner_loop
    ORR r2,r1,r0            ; Make segment and line address
    MCR p15,0,r2,c7,c14,2   ; Clean and flush that line
    ADD r0,r0,#0x10         ; Increment segment counter
    CMP r0,#0x40            ; Complete all 4 segments?
    BNE inner_loop          ; If not, branch back to inner_loop
    ADD r1,r1,#0x04000000   ; Increment line counter
    CMP r1,#0x0             ; Complete all lines?
    BNE outer_loop          ; If not, branch back to outer_loop
```

4.4 The write buffer

The ARM940T provides a write buffer to improve system performance. The write buffer can buffer up to eight words of data at up to four nonsequential addresses. The write buffer is used for memory that is marked as one of the following:

- NCB
- WB
- WT.

Write buffer behavior is controlled by the protection region attributes of the store being performed and the DCache and control bits (GCd and GBd) from the protection unit. These control bits are generated as follows:

GCd bit The GCd bit is generated from the cachable attribute of the protection region AND the DCache enable AND the protection unit enable.

GBd bit The GBd bit is generated from the bufferable attribute for the protection region AND the protection unit enable.

All accesses are initially noncachable and nonbufferable until the protection unit has been programmed and enabled. It follows that the write buffer cannot be used while the protection unit is disabled.

On reset, the buffer is flushed.

4.4.1 Write buffer operation

The write buffer is used when the DCache hits and/or misses, depending on the mode of operation. Table 4-2 on page 4-12 shows how the GCd and GBd bits control the behavior of the write buffer.

Table 4-2 Data write modes

GCd	GBd	Access mode
0	0	NCNB (Noncachable, nonbufferable). Reads and writes are not cached. They always perform accesses on the AMBA ASB interface. Writes are not buffered. The CPU halts until the write is completed on the AMBA ASB interface. Reads and writes can be externally aborted. Cache hits never occur under normal operation.

Table 4-2 Data write modes (continued)

GCd	GBd	Access mode
0	1	NCB (Noncachable, bufferable). Reads and writes are not cached. They always perform accesses on the AMBA ASB interface. Writes are placed in the write buffer and appear on the AMBA ASB interface. The CPU continues execution as soon as the write is placed in the write buffer. Reads can be externally aborted. Writes cannot be externally aborted. Cache hits never occur under normal operation. If the DCache hits for this type of access, there has been a programming error. This error is treated like a write-through in that the DCache line is updated and the data is buffered. Swap instruction operations on data in an NCB region are made to perform NCNB type accesses and are not buffered.
1	0	WT (Write-through). Reads that hit in the cache read the data from the cache and do not perform an access on the AMBA ASB interface. Reads that miss in the cache cause a linefill. Writes that hit in the cache update the cache but do not mark the cache line as dirty. All writes are placed in the write buffer and appear on the AMBA ASB interface. The CPU continues execution as soon as the write is placed in the write buffer. Reads and writes cannot be externally aborted.
1	1	WB (Write-back). Reads that hit in the cache read the data from the cache and do not perform an access on the AMBA ASB interface. Reads that miss in the cache cause a linefill. Writes that hit in the cache update the cache and mark the appropriate half of the cache line as dirty. They do not cause an AMBA ASB interface access. Writes that miss in the cache are placed in the write buffer and appear on the AMBA ASB interface. The CPU continues execution as soon as the write is placed in the write buffer. Cache write-backs are buffered. Reads, writes, and write-backs cannot be externally aborted.

4.4.2 Enabling and disabling the write buffer

You cannot directly enable or disable the write buffer. However, setting the properties of a memory region to be NCNB or disabling the protection unit prevents the write buffer from being used.

4.4.3 Buffered writes

The write buffer is non-merging, so even if two separate buffered external memory writes are performed that are sequentially related, they still take two address locations within the buffer, and are treated as nonsequential accesses. This is also true for non-word writes to the same word address. In this instance two address and two data locations are used in the write buffer.

The write buffer splits any accesses caused by an `STM` instruction on 4-word boundaries. Each set of words uses one address location in the write buffer. This mechanism allows privileges to be rechecked in the case where the access crosses a memory region and the memory region privileges might change, therefore protecting any regions of reserved memory.

Figure 4-3 on page 4-14 shows the write buffer behavior for the following code sequence:

```
MOV     r11, #0x10c         ; set pointer
MOV     r12, #0x20c         ; set pointer
STMIA   r11, {r0-r5}        ; store 6 registers
STMIA   r12, {r6-r10}       ; store 5 registers
```

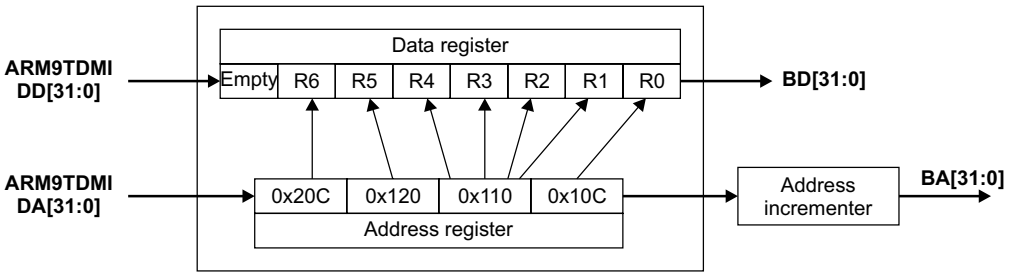

Figure 4-3 Write buffer allocation

In this code, a pointer has been set to address `0x10C`. A store multiple of six registers is then executed. This instruction uses six data registers, and three address registers within the write buffer. Another store to address `0x20c` is then executed using the remaining address location. The internal ARM9TDMI is then stalled until an address register becomes free.

Note

When a cache line is evicted from the DCache to the write buffer, it only uses one address register, because cache lines are aligned to 4-word boundaries.

4.4.4 Drain write buffer

You can drain the write buffer under software control, so that additional instructions are not executed until the write buffer is drained, using the following methods:

- store to nonbufferable memory
- load from noncachable memory
- `MCR` drain write buffer:

  ```
  MCR p15, 0, Rd, c7, c10, 4
  ```

4.5 Cache lockdown

To provide predictable code behavior in embedded systems, a mechanism for locking code and data into the ICache and DCache respectively is provided. For example, this feature can be used to hold high-priority interrupt routines where there is a hard real-time constraint, or to hold the coefficients of a DSP filter routine to reduce external bus traffic.

Locking down a region of the ICache or DCache is achieved by executing a short software routine, taking note of these requirements:

- the program must be held in a noncachable area of memory
- the cache must be enabled and interrupts must be disabled
- software must ensure that the code or data to be locked down is not already in the cache
- if the caches have been used after the last reset, the software must ensure that the cache in question is cleaned, if appropriate, and then flushed.

Lockdown in the DCache is achieved through use of CP15 register 9. ICache lockdown uses both CP15 registers 7 and 9.

As described in *Cache architecture* on page 4-2, the ARM940T ICache and DCache each comprise four segments. Each cache segment comprises 64 lines of four words. Each segment is 1KB in size. Lockdown can be performed with a granularity of one line across each of the four segments. The smallest space that can be locked down is 16 words. Lockdown starts at line zero, and can continue until up to 63 of the 64 lines are locked.

4.5.1 Locking down the caches

The procedures for locking down a line in the ICache and in the DCache are slightly different. In both cases:

1. The cache must be put into lockdown mode by programming register 9.
2. A linefill must be forced.
3. The corresponding data must be locked in the cache.

If more than one line is to be locked, a software loop must repeat this procedure.

Data cache lockdown

For the DCache, the procedure is as follows:

1. Write to CP15 register 9, setting DL=1 and Dindex=0.
2. Initialize the pointer to the first of the 16 words to be locked.
3. Execute an `LDR` from that location. This forces a linefill from that location, and the resulting four words are captured by the cache.
4. Increment the pointer by 16 to select cache segment 1.
5. Execute an `LDR` from that location. The resulting linefill is captured in cache segment 2.
6. Repeat steps 1 to 5 for cache segments 3 and 4.
7. Write to CP15 register 9, setting DL=0 and Dindex=1.

If there is more data to lockdown, at the final step, step 7, the DL bit must be left HIGH, Dindex incremented by 1 line, and the process repeated. The DL bit must only be set LOW when all the lockdown data has been loaded.

Instruction cache lockdown

For the ICache, this procedure is as follows:

1. Write to CP15 register 9, setting IL=1 and Iindex=0.
2. Initialize the pointer to the first of the sixteen words to lockdown.
3. Force a linefill from that location by writing to CP15 register 7.
4. Increment the pointer by 16 to select cache segment 1.
5. Force a linefill from that location by writing to CP15 register 7. The resulting linefill is captured in segment 1.
6. Repeat for cache segments 3 and 4.
7. Write to CP15 register 9, setting IL=0 and Iindex=1.

If there is more data to lockdown, at the final step 7, the IL bit must be left HIGH, Iindex incremented by 1 line, and the process repeated. The IL bit must be set LOW when all the lockdown data has been loaded.

The only significant difference between the sequence of operations for the Dcache and ICache is that an MCR instruction must be used to force the linefill in the ICache, instead of an LDR. This is because of the Harvard nature of the processor. During the MCR, the value set up in the pointer register is output on the instruction address bus, and a memory access is forced. Because this misses in the cache, as a result of earlier flushing, a linefill occurs.

The rest of the sequence of operations is exactly the same as for DCache lockdown.

The MCR to perform the ICache lookup is a CP15 register 7 operation:

```
MCR p15,0,Rd,c7,c13,1
```

A subroutine used to lockdown code in the instruction cache is given in Example 4-2 on page 4-18.

Example 4-2 ICache lockdown subroutine

```
;   Subroutine lock_i_cache
;   r1 contains start address of code to be locked down
;
;   The subroutine performs a lock-down of instructions in the
;   I Cache. It first reads the current lock_down index and then
;   locks down the number of lines requested.
;

;   Note that this subroutine must be located in a noncachable
;   region of memory to work, or these instructions
;   themselves will be locked into the cache. Interrupts must
;   also be disabled.
;   The subroutine must be called using the 'BL' instruction.
;
;   This subroutine returns the next free cache line number in
;   r0, or 0 in r0 if an error occurs.

lock_i_cache
    STMFD r13!, {r1-r3}         ; save corrupted registers
    BIC r1, r1, #0x3f           ; align address to cache line
    MRC p15, 0, r3, c9, c0, 1   ; get current instruction cache index
    AND r2, r2, #0x3f           ; mask off unwanted bits
    ADD r3, r2, r0              ; Check to see if current index
    CMP r3, #0x3f               ; plus line count is greater than 63
                                ; If so, branch to error as
                                ; more lines are being locked down
                                ; than permitted
    BGT error
    ORR r2, r2, #0x80000000     ; set lock bit, r2 contains the cache
                                ; line number to lockdown
```



```
lock_loop
    MCR p15, 0, r2, c9, c0, 1   ; write lockdown register
    MCR p15, 0, r1, c7, c13, 1  ; force line fetch from external
                                ; memory
    ADD r1, r1, #16             ; add 4 words to address
    MCR p15, 0, r1, c7, c13, 1  ; force line fetch from external memory
    ADD r1, r1, #16             ; add 4 words to address
    MCR p15, 0, r1, c7, c13, 1  ; force line fetch from external
                                ; memory
    ADD r1, r1, #16             ; add 4 words to address
    MCR p15, 0, r1, c7, c13, 1  ; force line fetch from external
                                ; memory
    ADD r1, R1, #16             ; add 4 words to address
    ADD r2, r2, #0x1            ; increment cache line in lockdown
                                ; register
    SUBS r0, r0, #0x1           ; decrement line count & set flags
    BNE lock_loop               ; if r0! = 0 then branch round
    BIC r0, r2, #0x80000000     ; clear lock bit in lockdown
                                ; register
    MCR p15, 0, r0, c9, c0, 1   ; restrict victim counter to lines
                                ; r0 to 63
    LDMFD r13!, {r1-r3}         ; restore corrupted registers and
                                ; return
    MOV PC, LR                  ; r0 contains the first free cache
                                ; line number
error
    LDR r0, =0                  ; make r0 = 0 to indicate error
    LDMFD r13!, {r1-r3}         ; restore corrupted registers and
                                ; return
    MOV PC, LR
```


Chapter 5
Clock Modes

This chapter describes the different clock modes available on the ARM940T. It contains the following sections:

5.1 About ARM940T clocking

The ARM940T has two functional clock inputs, **BCLK** and **FCLK**. Internally, the ARM940T is clocked by **GCLK**. You can see this on the **CPCLK** output as shown in Figure 5-1 on page 5-2. **GCLK** can be sourced from either **BCLK** or **FCLK** depending on the clocking mode, selected using nF bit and iA bit in CP15 register 1 (see *Register 1, control register* on page 2-11), and external memory access. The three clocking modes are:

- *FastBus mode* on page 5-3
- *Synchronous mode* on page 5-4
- *Asynchronous mode* on page 5-6.

The ARM940T is a static design and you can stop both clocks indefinitely without loss of state. Figure 5-1 on page 5-2 shows that some of the ARM940T macrocell signals have timing specified with relation to **GCLK**. This can be either **FCLK** or **BCLK** depending on the clocking mode.

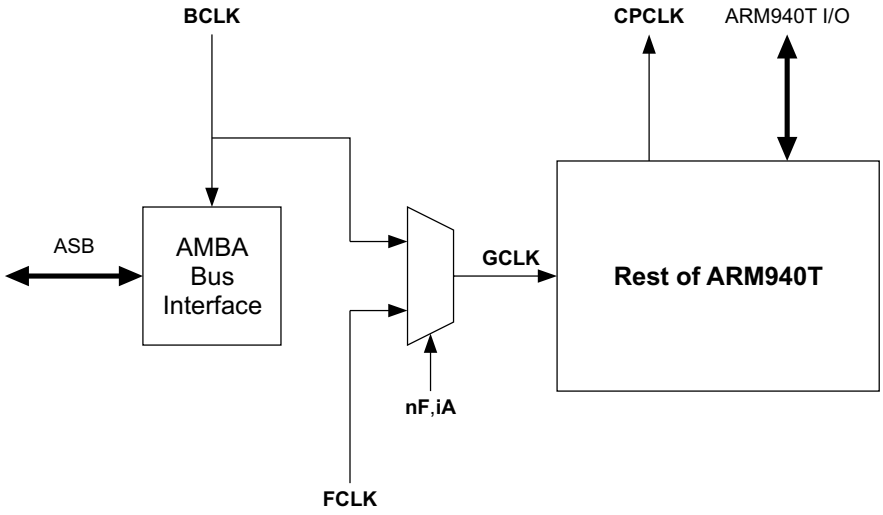

Figure 5-1 ARM940T clocking

5.2 FastBus mode

In FastBus mode **GCLK** is sourced from **BCLK**.

In this mode of operation the **BCLK** input is used to control:

- the internal ARM9TDMI
- cache operations
- the AMBA bus interface.

This mode is typically used in systems with high-speed memory.

The **FCLK** input is ignored. This means that **BCLK** is used to control the AMBA ASB interface and the internal ARM940T processor core.

On reset, the ARM940T is put into FastBus mode and operates using **BCLK**. A typical use for FastBus mode is to execute startup code while configuring a PLL under software control to produce **FCLK** at a higher frequency. When the PLL has stabilized and locked, you can switch the ARM940T to synchronous or asynchronous clocking using **FCLK** for normal operation.

5.3 Synchronous mode

This mode is typically used in systems with low-speed memory. In this mode both the **BCLK** and **FCLK** inputs are used. In this mode of operation **GCLK** is sourced from **BCLK** or **FCLK**. There are three restrictions that apply to **BCLK** and **FCLK**:

- **FCLK** must have a higher frequency than **BCLK**
- **FCLK** must be an integer multiple of the **BCLK** frequency
- **FCLK** must be HIGH whenever there is a **BCLK** transition.

BCLK is used to control the AMBA ASB interface, and **FCLK** is used to control the internal ARM940T processor core and any cache operations. When an external memory access is required the core either continues to clock using **FCLK** or is switched to **BCLK**, as shown in Table 5-1 on page 5-4. This is the same as for asynchronous mode.

Table 5-1 Clock selection for external memory accesses

External memory access operation	GCLK =
Buffered write	**FCLK**
Nonbuffered write	**BCLK**
Cachable read (linefill), noncachable read	**BCLK**

The penalty in switching from **FCLK** to **BCLK** and from **BCLK** to **FCLK** is symmetric, from zero to one phase of the clock to which the core is re-synchronizing. That is, switching from **FCLK** to **BCLK** has a penalty of between zero and one **BCLK** phase, and switching back from **BCLK** to **FCLK** has a penalty of between zero and one **FCLK** phase.

Figure 5-2 on page 5-5 shows an example zero **BCLK** phase delay when switching from **FCLK** to **BCLK** in synchronous mode.

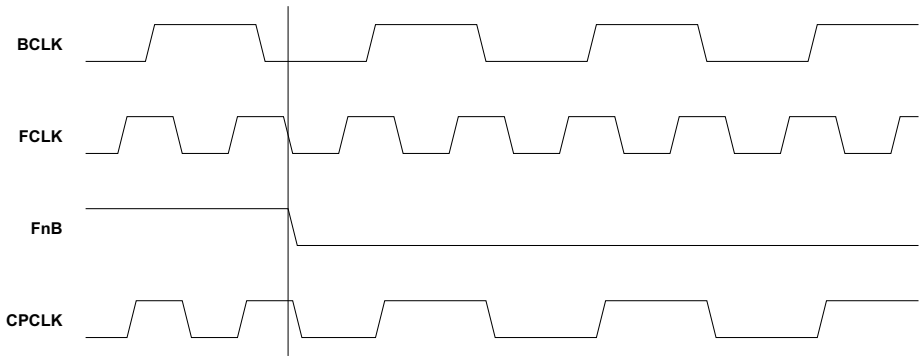

Figure 5-2 Synchronous mode FCLK to BCLK zero phase delay

Figure 5-3 on page 5-5 shows an example one **BCLK** phase delay when switching from **FCLK** to **BCLK** in synchronous mode.

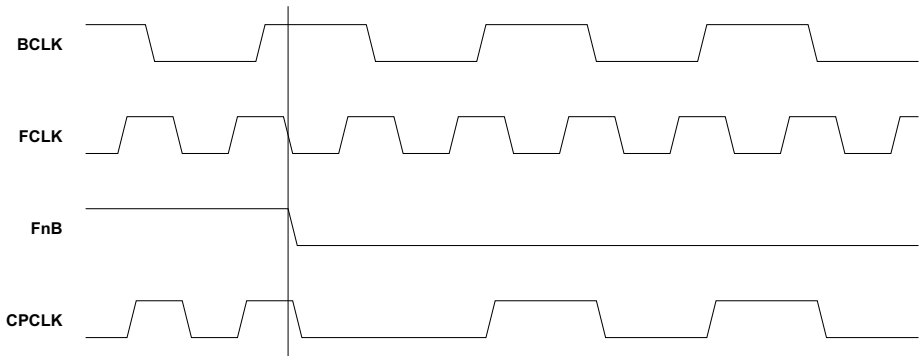

Figure 5-3 Synchronous mode FCLK to BCLK one phase delay

5.4 Asynchronous mode

This mode is typically used in systems with low-speed memory. In this mode of operation **GCLK** is sourced from **BCLK** or **FCLK**. **FCLK** and **BCLK** can be completely asynchronous to one another, with the one restriction that **FCLK** must have a higher frequency than **BCLK**.

BCLK is used to control the AMBA ASB interface, and **FCLK** is used to control the internal ARM940T processor core and any cache operations. When an external memory access is required the core either continues to clock using **FCLK** or is switched to **BCLK**. This is the same as for synchronous mode.

The penalty in switching from **FCLK** to **BCLK** and from **BCLK** to **FCLK** is symmetric, from zero to one cycle of the clock to which the core is resynchronizing. That is, switching from **FCLK** to **BCLK** has a penalty of between zero and one **BCLK** cycle, and switching back from **BCLK** to **FCLK** has a penalty of between zero and one **FCLK** cycle.

Figure 5-4 on page 5-6 shows an example zero **BCLK** cycle delay when switching from **FCLK** to **BCLK** in asynchronous mode.

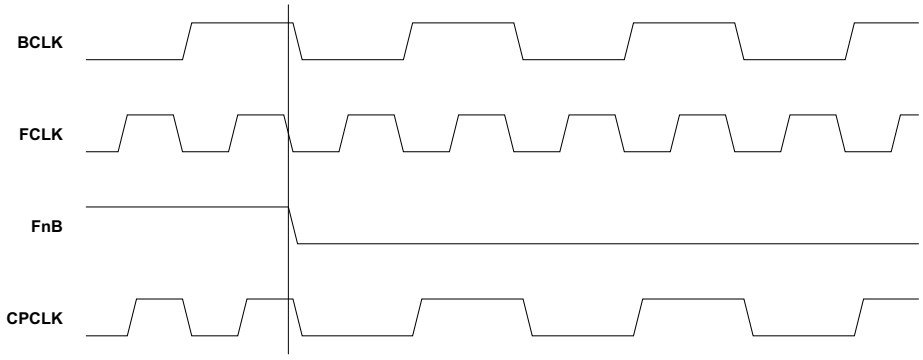

Figure 5-4 Asynchronous mode FCLK to BCLK zero cycle delay

Figure 5-5 on page 5-7 shows an example one **BCLK** cycle delay when switching from **FCLK** to **BCLK** in asynchronous mode.

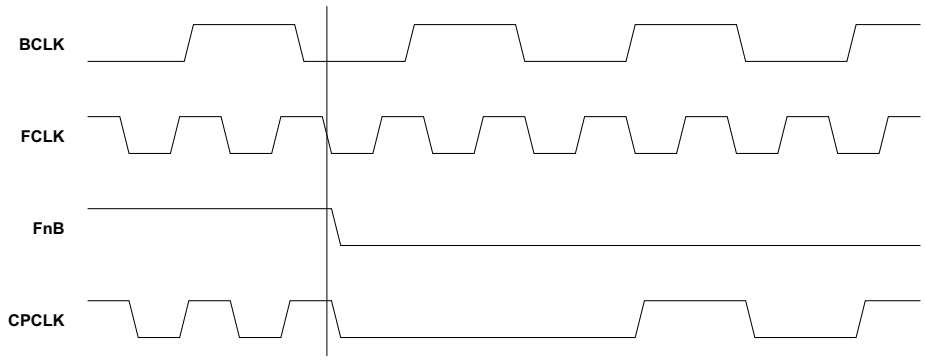

Figure 5-5 Asynchronous mode FCLK to BCLK one cycle delay

Chapter 6
Bus Interface Unit

The ARM940T has an *Advanced Microprocessor Bus Architecture* (AMBA) interface. This chapter describes the operation of this interface. It contains the following sections:

For more details of AMBA, see the *AMBA Specification*.

6.1 About the ARM940T bus interface

The *AMBA Specification (REV 2.0)* defines two high-performance system buses:

- the *Advanced High-performance Bus* (AHB)
- the *Advanced System Bus* (ASB).

The ARM940T has been designed with a bidirectional ASB interface, plus the necessary extra control signals to enable efficient implementation of both the AHB and ASB interface. The ARM940T implements a fully-compliant ASB interface, either as an ASB bus master, or as a slave for production test. With the addition of a synthesizable wrapper, the ARM940T implements a full AHB interface, either as an AHB bus master, or as a slave for production test.

In this section the following abbreviations are used:

NCNB Noncachable and nonbufferable

NCB Noncachable and bufferable

NC Noncachable

WT Cachable and write-through

WB Cachable and write-back.

6.2 ASB transfers

The AMBA ASB specification describes three transfer types that are encoded in **BTRAN[1:0]**. Table 6-1 on page 6-3 shows these transfer types.

Table 6-1 AMBA ASB transfer types

BTRAN[1:0]	Transfer type	Description
00	Address-only (A-TRAN)	Used when no data movement is required. The three main uses for address-only transfers are: • for IDLE cycles • for bus handover cycles • for speculative address decoding without committing to a data transfer.
01	-	Reserved.
10	Nonsequential (N-TRAN)	Used for single transfers or the first transfer of a burst. The address of the transfer is unrelated to the previous bus access.
11	Sequential (S-TRAN)	Used for successive transfers in burst. The address of a SEQUENTIAL transfer is always related to the previous transfer.

The output signals **ASTB**, **BURST[1:0]**, **NCMAHB**, and **BUFFSTRAHB** have been added to the ARM940T bus interface. These are necessary to support the AMBA AHB wrapper, but can also be used to provide optimized accesses in an AMBA ASB system:

ASTB This signal distinguishes between an IDLE cycle and the A-TRAN cycle of a nonsequential transfer. It is asserted with the same timing as **AOUT[31:0]**, changing in phase 2. Usually a memory controller only commits to a transfer when it sees the S-TRAN cycle, perhaps only decoding the address during the A-TRAN cycle. **ASTB** is asserted in the preceding A-TRAN cycle, indicating that the current A-TRAN is followed by an S-TRAN, providing **AGNT** is HIGH on the next rising edge of **BCLK**.

BURST[1:0] Burst transfers are used for cache linefills, and for buffered writes caused by cache lines that have been evicted or cleaned. In each case, a transfer of four words takes place.

These signals indicate a sequential burst, as shown in Table 6-2 on page 6-4.

Table 6-2 Burst transfers

BURST[1:0]	Transfer
00	No sequential information available (default)
01	Reserved
10	Current access is part of a four-word transfer
11	Reserved

The **BURST[1:0]** signals change in phase 2 and are asserted in the phase when **ASTB** is asserted. **BURST[1:0]** then remain unchanged until the next transfer.

BURST[1:0] only indicates a four-word transfer when either a cache linefill takes place, or when a dirty line within a write back protection region has been evicted. In all other circumstances, **BURST[1:0]** indicates single word transfers. This is true for `LDM` and `STM` instructions, regardless of the number of registers being transferred.

BURST[1:0] can be factored into both the arbiter and decoder of the AMBA system, and can be used to prevent a new bus master taking control of the ASB, giving a more efficient transfer.

NCMAHB This signal indicates for noncached load multiples and store multiples whether more words are requested as part of the current burst transfer. When HIGH this indicates more words are requested. When LOW, on the last S-TRAN of the burst, this indicates that the current transfer is the last word of the burst. It is asserted in phase 2 and is only valid if **AGNT** remains asserted throughout the transfer.

BUFFSTRAHB

This signal indicates when the NCMAHB signal is set but not valid. This can happen when some buffered stores are occuring.

The following timing diagrams show the types of transfer that can be initiated by the ARM940T rev1:

- *Example LDR from address 0x108* on page 6-6
- *Example LDM of 5 words from 0x108* on page 6-7
- *Example nonbuffered STR* on page 6-9
- *Example STM* on page 6-10
- *Example linefill from 0x100* on page 6-12
- *Cache linefill and write back* on page 6-13
- *Example 4-word data eviction* on page 6-14
- *Example swap operation* on page 6-15.

The **AREQ** and **AGNT** signals and the responses from the ASB slave are not shown in these diagrams. It is assumed that **AGNT** is asserted and the ASB slave response is DONE.

Different slave responses and bus master handover are covered in the *AMBA Specification (Rev 2.0)*. It is assumed that you are using the ARM940T macrocell within a multi-master ASB system, so unidirectional ASB timing diagrams are not provided.

6.2.1 Noncached LDRs and noncached fetches

The only difference between these noncached `LDR`s and noncached fetches is the **BPROT[1:0]** information, as shown in Table 6-3 on page 6-5.

Table 6-3 Noncached LDR and fetch

BPROT[0]	Transfer
0	Opcode fetch
1	Data access

The address is word-aligned for an `LDR` and fetch. An example `LDR` is shown in Figure 6-1 on page 6-6.

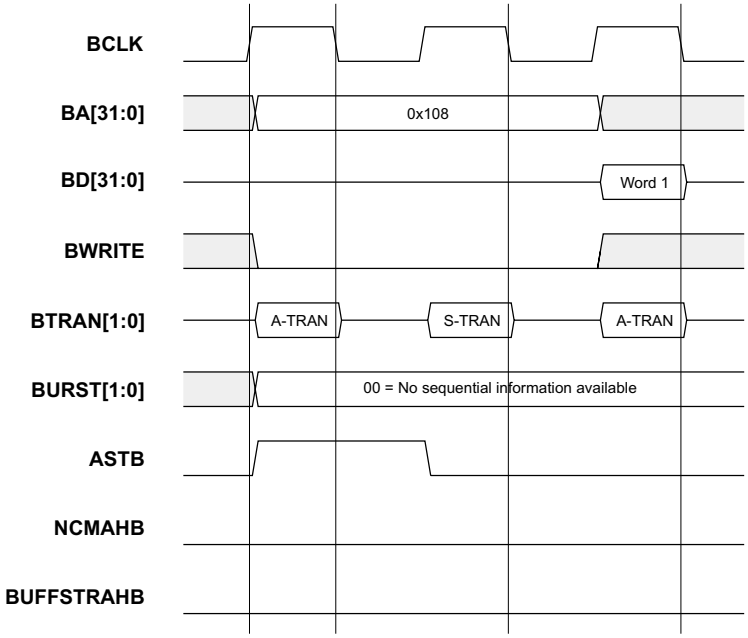

Figure 6-1 Example LDR from address 0x108

6.2.2 Noncached LDM

For a noncached `LDM` the **BURST[1:0]** information is always:

00 No sequential information available.

The **NCMAHB** signal gives one cycle advance warning of the end of the burst transfer if **AGNT** remains asserted throughout the burst transfer. The address is word-aligned. An example `LDM` is shown in Figure 6-2 on page 6-7.

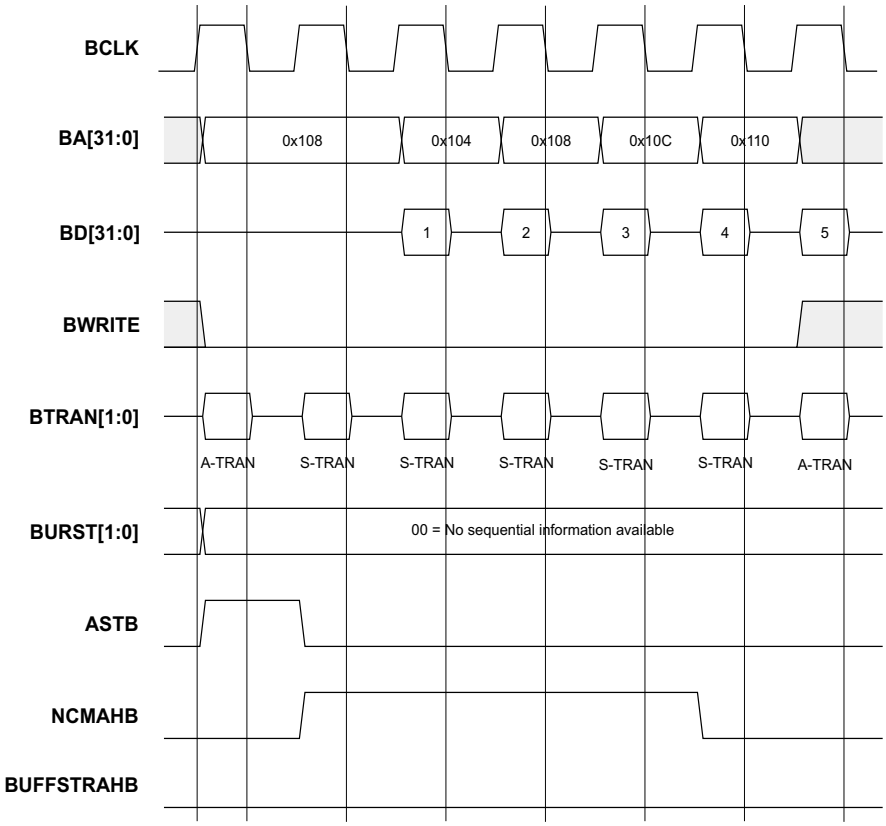

Figure 6-2 Example LDM of 5 words from 0x108

6.2.3 Noncached LDM crossing a 4KB boundary

An `LDM` instruction can transfer all 16 general-purpose registers in one instruction. If this instruction is executed, and the address being accessed lies in a noncachable region of memory, a 16-word sequential load takes place on the AMBA interface. If the access crosses a 4KB boundary, it is split. This allows the region properties to be checked in the case where there is a transition between memory protection regions. Figure 6-3 on page 6-8 shows an `LDM` operation crossing a 4KB boundary.

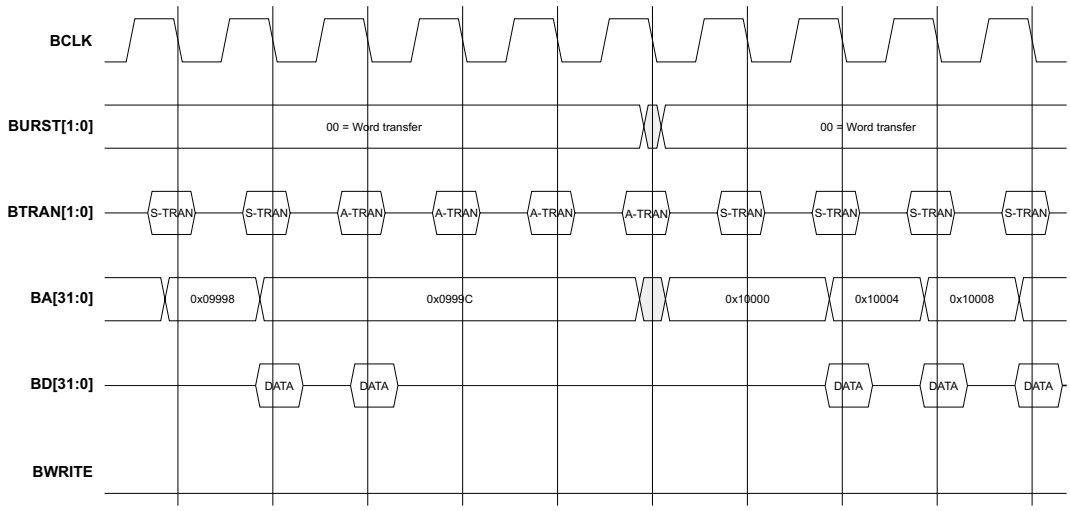

Figure 6-3 LDM operation crossing a 4KB boundary

Note

In Figure 6-3 the **BURST[1:0]** bus is only indicating word transfers during the LDM operation.

As the LDM transfer takes place on the ASB, the time taken to complete the operation is dependent on the **BCLK** frequency, any bus arbitration, and the speed of the slave being accessed. An LDM instruction must therefore be completed before an interrupt can be serviced.

6.2.4 Buffered and nonbuffered STR

For a buffered or nonbuffered STR the **BURST[1:0]** information is:

10 Current access is part of a four-word transfer

00 No sequential information available.

The address is word-aligned. An example STR is shown in Figure 6-4 on page 6-9.

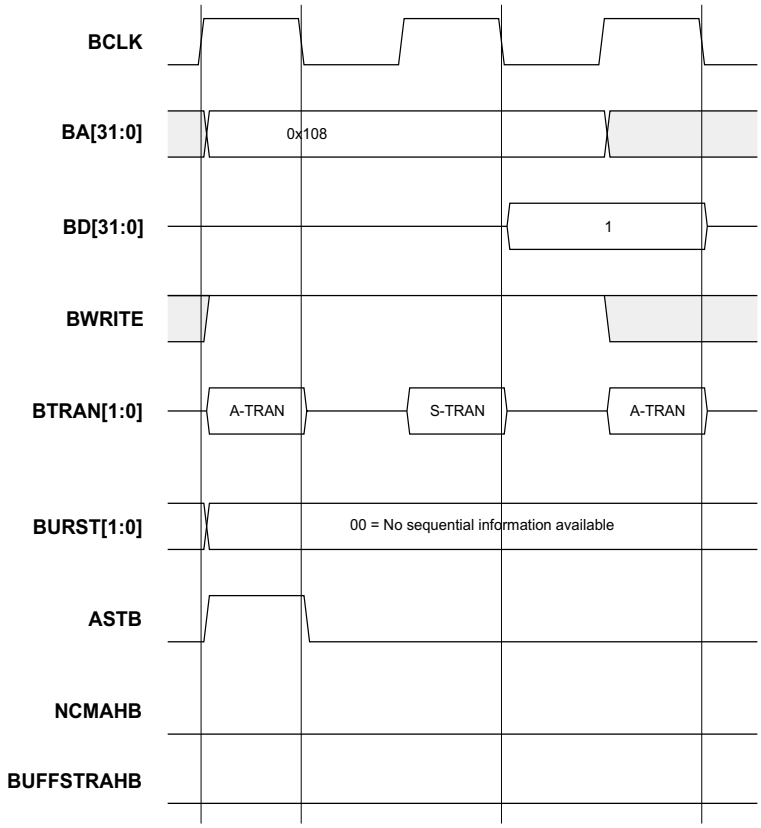

Figure 6-4 Example nonbuffered STR

6.2.5 Buffered and nonbuffered STM

For a buffered or nonbuffered STM the **BURST[1:0]** information is:

10 Current access is part of a four-word transfer

00 No sequential information available.

The address is word aligned. An example STM is shown in Figure 6-5 on page 6-10.

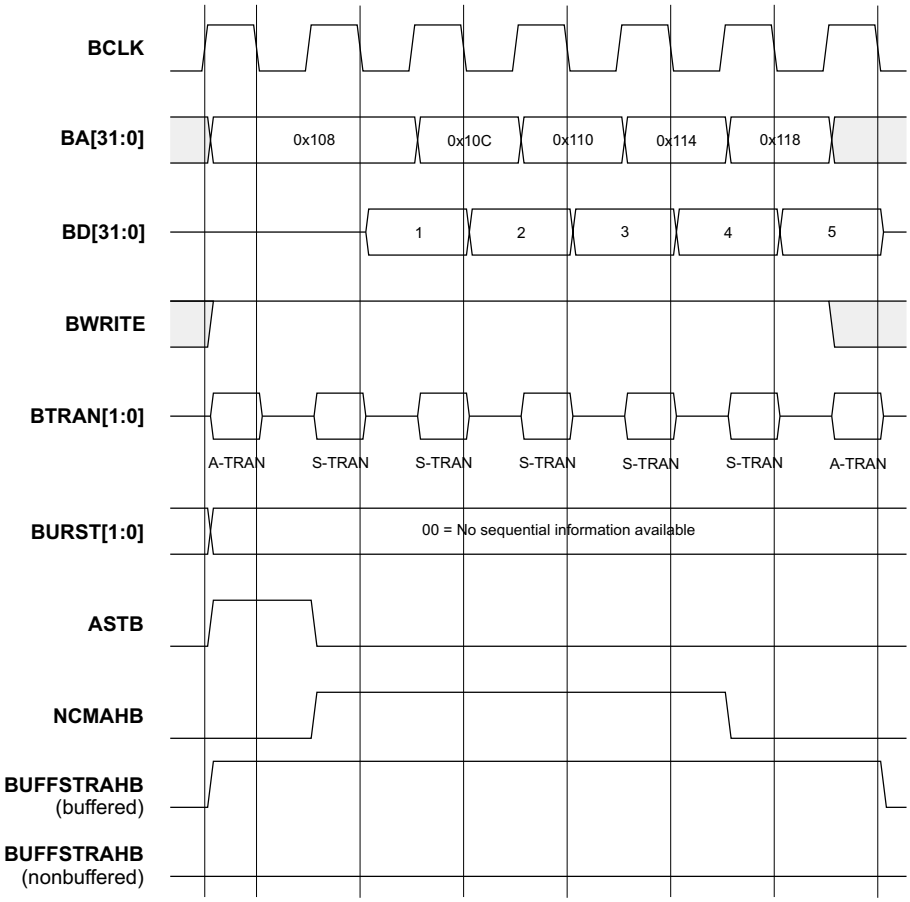

Figure 6-5 Example STM

6.2.6 NCNB STM crossing a 4KB boundary

An `STM` instruction can transfer all 16 general-purpose registers in one instruction. If this instruction is executed, and the address being accessed lies in an NCNB region of memory, a 16-word sequential write takes place on the AMBA interface. If the access crosses a 4KB boundary, the access is split. This allows the region properties to be checked in the case where there is a transition between memory regions. Figure 6-6 on page 6-11 show an `STM` operation crossing a 4KB boundary.

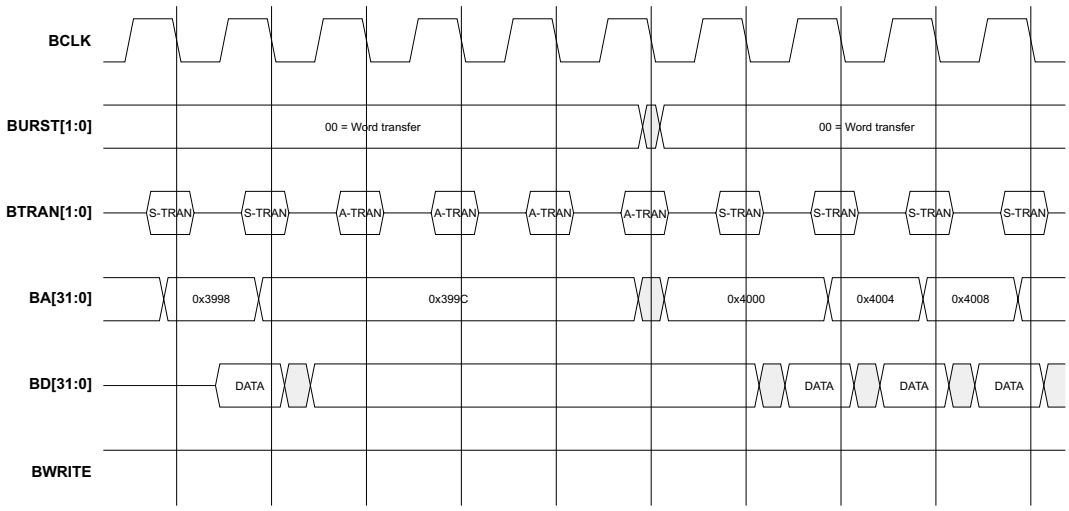

Figure 6-6 STM operation crossing a 4KB boundary

Note

In Figure 6-6 on page 6-11 **BURST[1:0]** bus is only indicating word transfers during the `STM` operation.

6.2.7 Cached LDR, cached LDM, and cached fetch

A cached `LDR` or `LDM`, and a cached fetch, are equivalent to a linefill operation. The **BURST[1:0]** information is always:

10 4 words.

The address is word-aligned and increases from the lowest address. The lowest five bits always increase from `0x00` to `0x1C`. An example linefill is shown in Figure 6-7 on page 6-12.

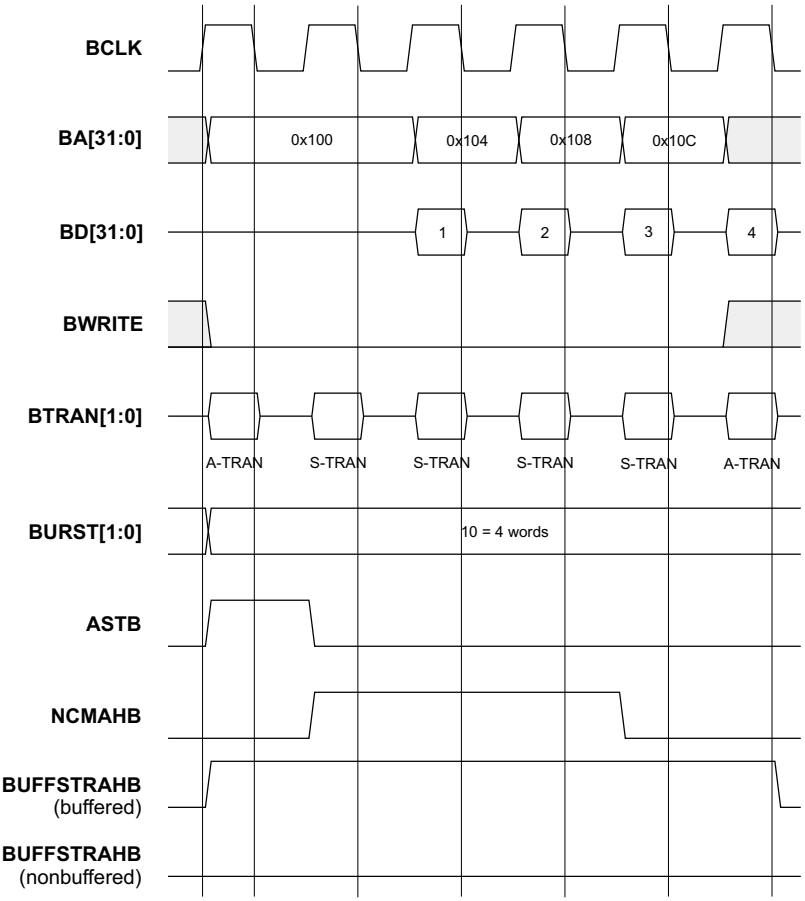

Figure 6-7 Example linefill from 0x100

Figure 6-8 on page 6-13 shows a cache linefill followed by a buffered write where a cache line has been evicted and is being written back.

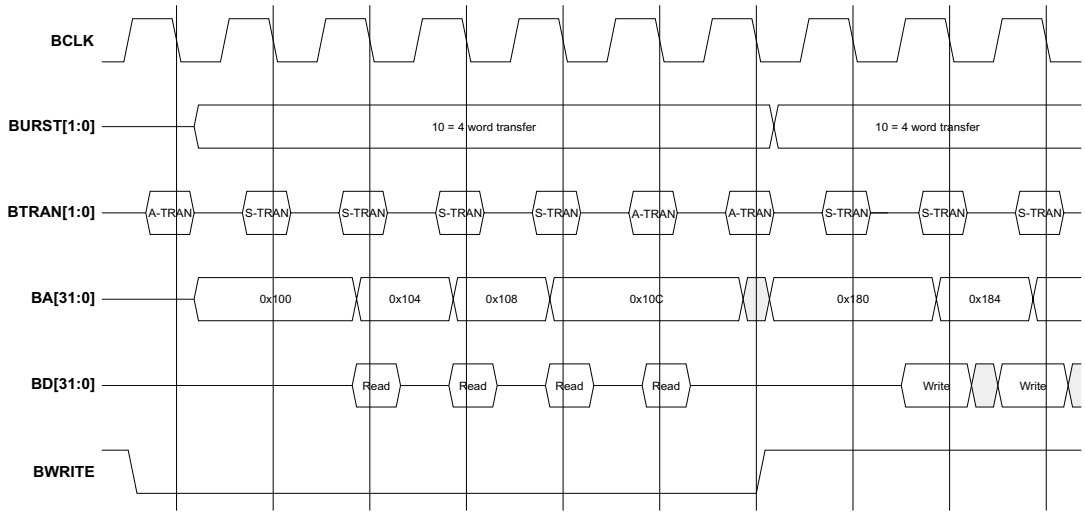

Figure 6-8 Cache linefill and write back

Note

An evicted cache line is only written back to main memory from the DCache when a protection region is marked as a write back area, and the dirty bit of the line has been set.

Cache linefills are performed by reading four words of data aligned to a four-word boundary. The word of data aligned onto the four-word boundary is always fetched first. The ARM940T supports streaming, so when the addressed word is fetched, it is transferred to the cache and to the ARM9TDMI simultaneously. If the next access is sequential, subsequent words can also be streamed to the ARM9TDMI.

6.2.8 Dirty data eviction, write-back of 4 words

Dirty data is evicted from a cache line as all four words of the cache line. The address is word-aligned and increases from the lowest address. **BURST[1:0]** = 10. The lowest five bits of the address are `0x00` to `0x1C`. Figure 6-9 on page 6-14 shows an example four-word dirty data eviction of a cache line.

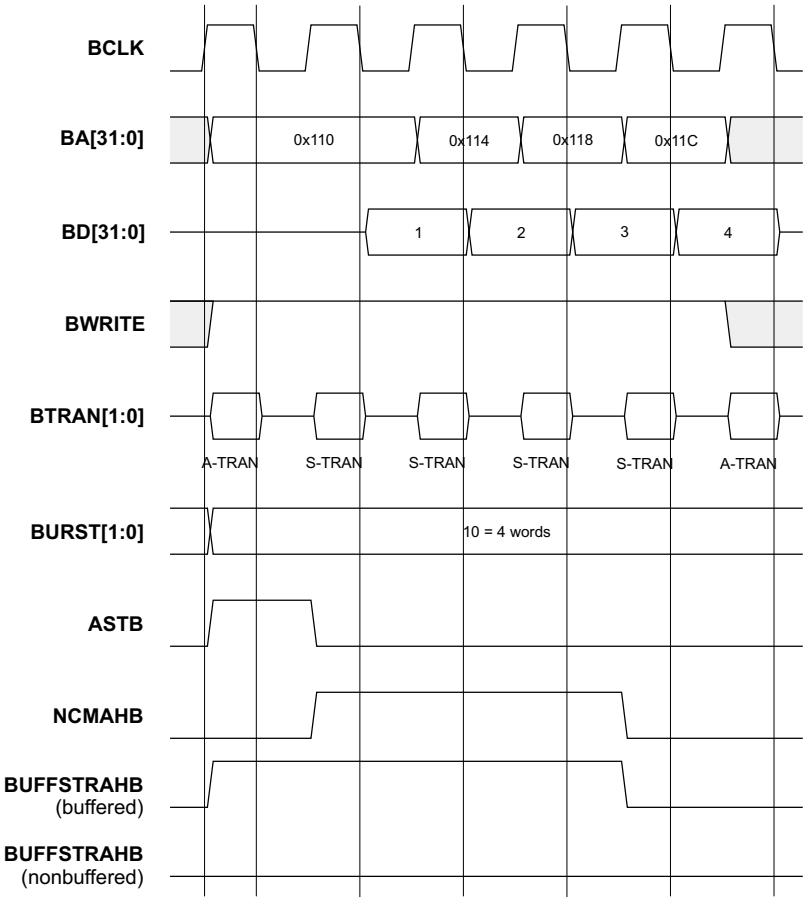

Figure 6-9 Example 4-word data eviction

6.2.9 Swap

The *Swap* (SWP) operation is implemented as a single read transfer followed by a single write transfer. The **BLOK** signal is asserted so that the write transfer is locked to the preceding read transfer. This must be used by the arbiter to ensure that no other bus master is given access to the bus between the read and write transfers. An example swap operation is shown in Figure 6-10 on page 6-15.

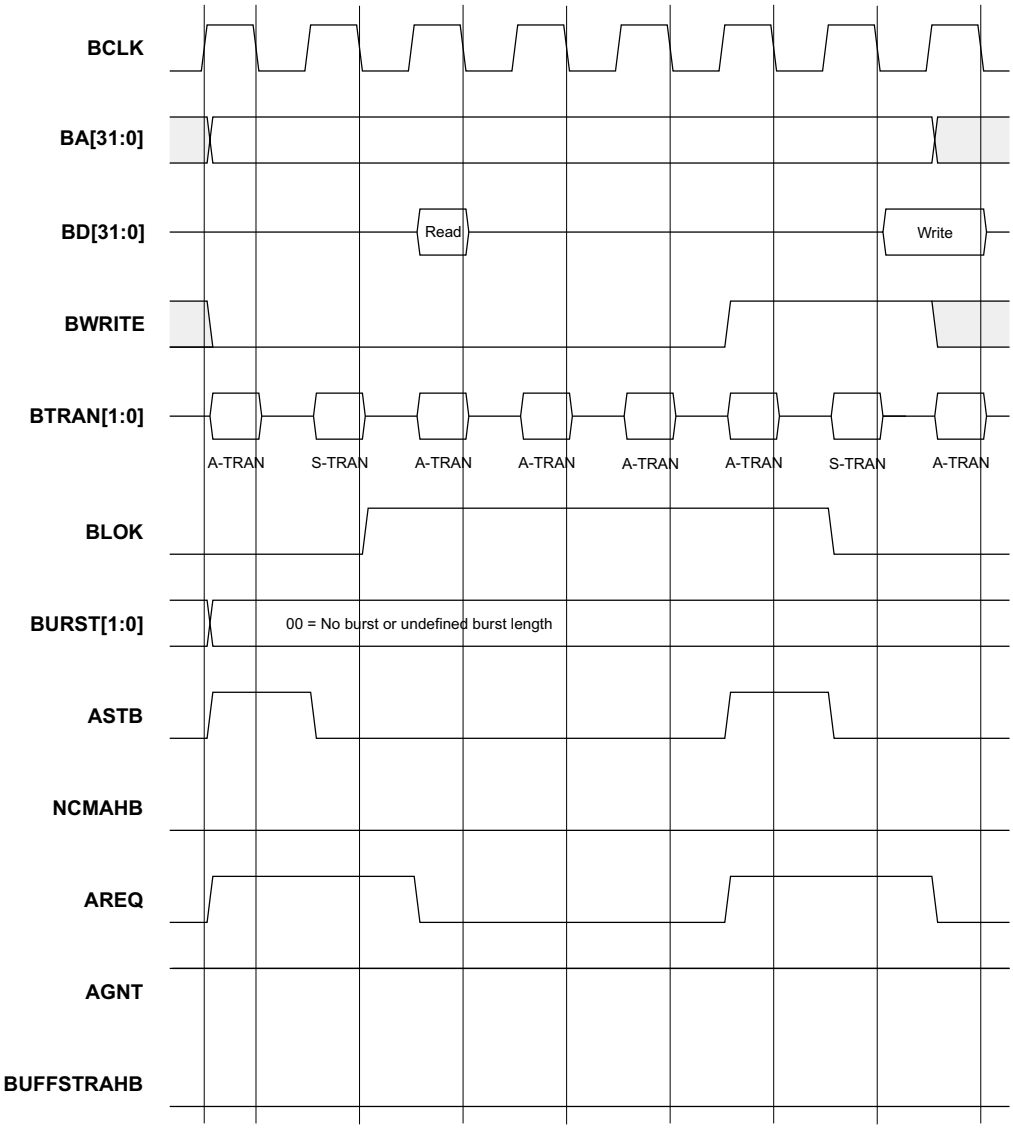

Figure 6-10 Example swap operation

When a SWP instruction is executed on the ARM940T, the behavior is dependent on the memory region being accessed, and it is up to the programmer to ensure correct operation.

Typically for multi-master operations, the SWP instruction is used for passing semaphores between the masters. For this type of operation, the semaphore must be stored in an NCNB or NCB region of memory. When a SWP instruction is executed, any cache linefills complete and the write buffer drains before the SWP instruction memory accesses take place. During the SWP access, the **BLOK** signal goes HIGH to indicate that the two memory accesses are indivisible.

For SWP instructions that access an NCB region of memory, any cache linefills complete, and the write buffer drains before the read takes place. During the read, **BLOK** is driven HIGH. The write operation then takes place as an unbuffered write. This is to allow external aborts to be taken.

When a SWP instruction accesses a cachable region of memory, the access is protected as a normal data access. The **BLOK** signal remains LOW throughout this operation.

If a region of memory is changed from being cachable to noncachable and the cache is not flushed, it is possible for a cache hit to occur for the read access of the SWP instruction. This is a programming error that must be avoided.

6.2.10 AMBA ASB slave transfers

You can test the ARM940T as an individual module within an AMBA system, responding only to transfers from the AMBA ASB. In this mode of operation the ARM940T is never granted the ASB as a bus master, and responds as an ASB slave, detecting the assertion of **DSEL**. This is described in detail in the *AMBA Specification (REV 2.0)*.

6.3 External aborts

External aborts are ignored for buffered write operations or for cache linefills. In all other cases, the external abort causes the abort exception to be taken.

6.4 Memory access order

If a simultaneous data access and instruction fetch both cause cache misses, the data access takes precedence and is completed first. Typically, instructions tend to require frequent sequential accesses and data requires infrequent nonsequential accesses. This type of behavior results in more efficient ASB usage, and improves the chances of streaming linefill words to the ARM9TDMI core.

Figure 6-11 on page 6-18 shows how misses in both the ICache and DCache result in external accesses, with the data access taking place first, followed by the instruction fetch.

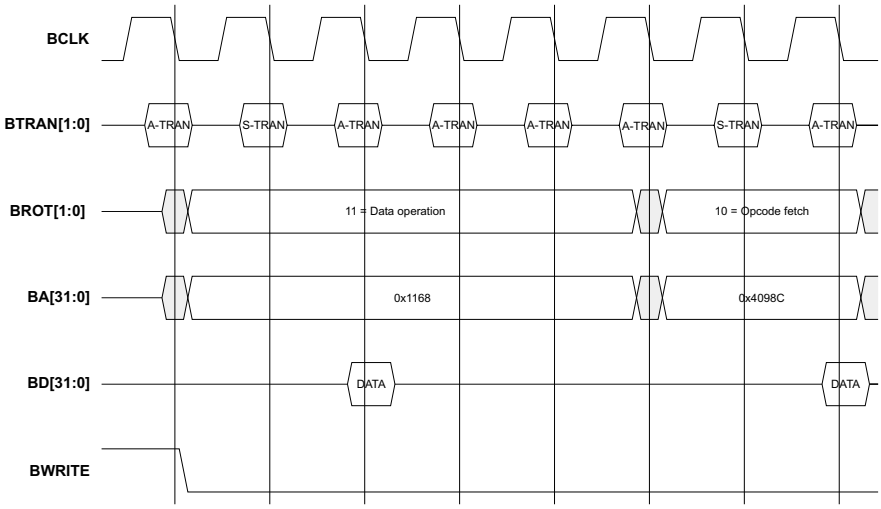

Figure 6-11 Simultaneous cache misses

Chapter 7
Coprocessor Interface

This chapter describes the ARM940T coprocessor interface. It contains the following sections:

7.1 About the coprocessor interface

The ARM940T coprocessor interface allows you to attach specially designed coprocessor hardware to the ARM940T. Uses include:

- attachment of accelerators for floating point math
- DSP
- 3-D graphics
- encryption
- decryption.

The ARM instruction set supports the connection of up to 16 coprocessors, numbered 0 to 15, to an ARM processor.

7.1.1 User-assignable coprocessor numbers

The ARM940T contains two internal coprocessors:

- CP14 for debug control
- CP15 for cache, protection unit, and clocking mode control.

This means that external coprocessors cannot be assigned to coprocessor numbers 15 or 14. Other coprocessor numbers, allocated by ARM for internal usage, are listed in Table 7-1 on page 7-2.

Table 7-1 Coprocessor availability

Coprocessor number	Allocation
15	System control
14	Debug controller
13:8	Reserved
7:4	Available to users
3:0	Reserved

The register map of CP15 is described in *CP15 register map summary* on page 2-5. The functionality of CP14 is described in *Debug communications channel* on page 8-48.

7.1.2 External coprocessors

Coprocessors determine the instructions that they must execute by using a *pipeline follower* in the coprocessor. As each instruction arrives from memory, it enters both the ARM pipeline and the coprocessor pipeline. To avoid a critical path for the instruction being latched by the coprocessor, the coprocessor pipeline must operate one clock phase behind the ARM940T pipeline. The ARM940T then informs the coprocessor when instructions move from Decode into Execute, and if the instruction has to be executed.

To enable coprocessors to continue execution of coprocessor data operations while the ARM940T pipeline is stalled (for example, waiting for a cache linefill to occur), the coprocessor must monitor a clock **CPCLK**, and a clock stall signal **nCPWAIT**. If **nCPWAIT** is LOW on the rising edge of **CPCLK**, the ARM940T pipeline is stalled and the coprocessor pipeline must not advance.

Figure 7-1 on page 7-3 indicates the timing for these signals and when the coprocessor pipeline can advance its state. In this diagram, Coproc clock shows the result of ORing **CPCLK** with the inverse of **nCPWAIT**. This is one technique for generating a clock that reflects the ARM940T pipeline advancing.

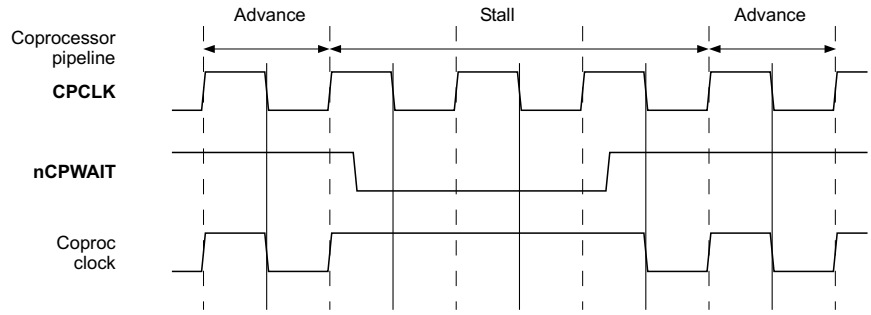

Figure 7-1 ARM940T coprocessor clocking

Coprocessor instructions

There are three classes of coprocessor instructions:

LDC or STC Load from coprocessor register to memory and store to coprocessor register from memory.

MCR or MRC Register transfer between coprocessor and ARM processor core.

CDP Coprocessor data operation.

Examples of how a coprocessor must execute these instruction classes are given in:

- *LDC or STC* on page 7-5
- *MCR/MRC* on page 7-9
- *Interlocked MCR* on page 7-11
- *CDP* on page 7-13.

7.2 LDC or STC

The cycle timings for LDC or STC operations are shown in Figure 7-2 on page 7-5. The diagram assumes that the requested data is in the DCache and that the writes hit the DCache.

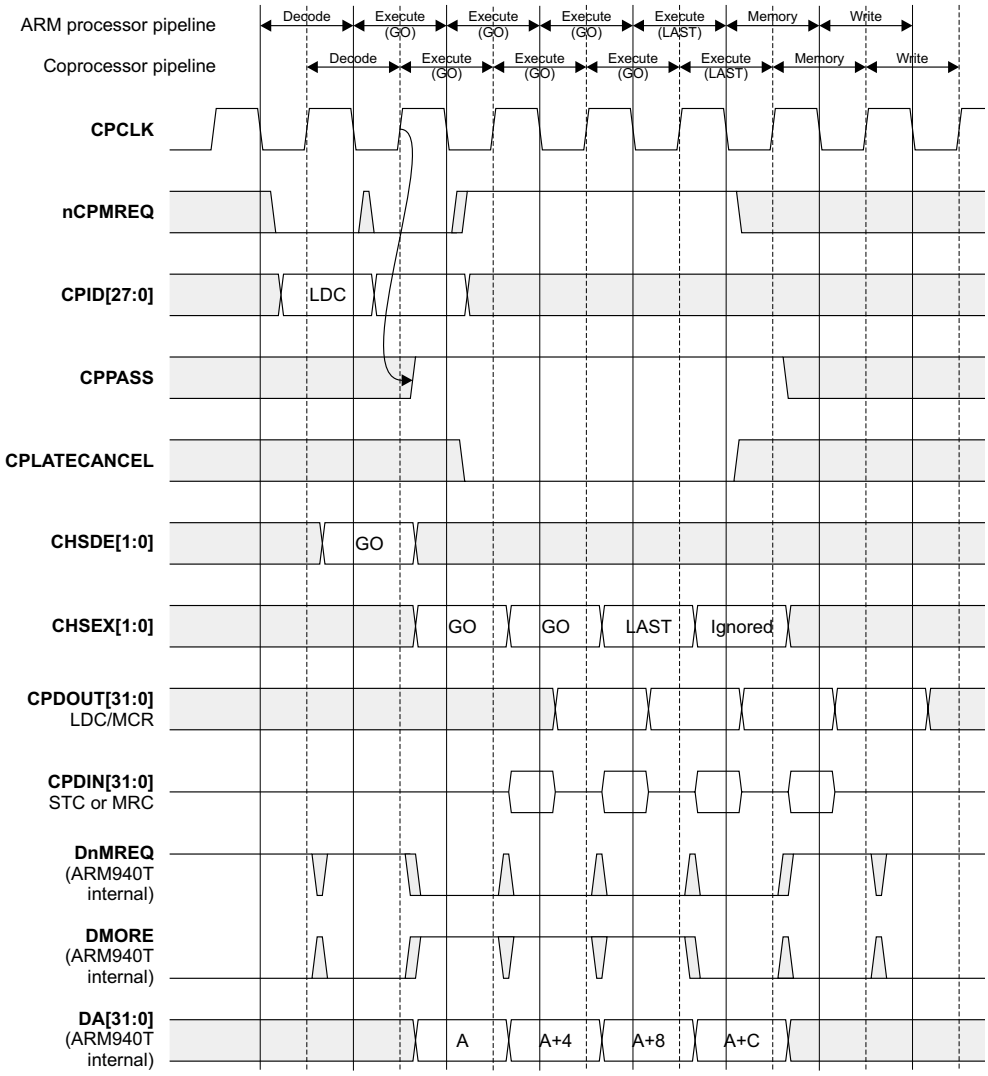

Figure 7-2 ARM940T LDC/STC cycle timing

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 ARM940T processor core performs the main instruction 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 must keep in step with the ARM940T by monitoring **nCPMREQ**, a latched copy of the ARM940T instruction memory request signal **InMREQ**. Whenever **nCPMREQ** is LOW, an instruction fetch is occurring and **CPID** is updated with the fetched instruction in the next cycle. This means that the instruction currently on **CPID** must enter the Decode stage of the coprocessor pipeline, and that the instruction in the Decode stage of the coprocessor pipeline must enter its Execute stage.

During the Execute stage, the condition codes are combined with the flags to determine if the instruction must be executed 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 must stop execution of the coprocessor instruction.

An additional output, **CPLATECANCEL**, is used to cancel a coprocessor instruction when the instruction preceding it caused a Data Abort. This is valid on the rising edge of **CPCLK** on the cycle after the first Execute cycle of the coprocessor instructions. **CPLATECANCEL** is only asserted during the first Memory cycle of a coprocessor instruction execution.

On the falling edge of the clock, the ARM940T processor core 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 coprocessor instruction currently in Execute requires another Execute cycle, it examines **CHSEX[1:0]**.

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 ARM940T processor core takes the undefined instruction exception.

WAIT If there is a coprocessor attached that can execute the instruction but not immediately, the coprocessor handshake signals must be driven to indicate that the ARM940T processor core must stall until the coprocessor can catch up. This is known as the *busy-wait* condition.

In this case, the ARM940T 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 exception is taken. If **CHSEX[1:0]** changes to GO or LAST, the instruction proceeds as described below.

If an interrupt occurs, the ARM940T processor core 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 (change any of the coprocessor states) until it has seen **CPPASS** go HIGH, and 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 another cycle of execution. Both the ARM940T 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 must drive the handshake signals with GO when two or more words still have to be transferred. When only one more word is required, the coprocessor must drive the handshake signals with the LAST condition.

In phase 2 of the Execute stage, the ARM940T processor core outputs the address for the `LDC`/`STC`. Also in this phase, **DnMREQ** is driven LOW, indicating to the memory system that a memory access is required at the data end of the device. The timing for the data on **CPDOUT[31:0]** for an `LDC` and **CPDIN[31:0]** for an `STC` is as shown in Figure 7-2 on page 7-5.

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 must drive the coprocessor handshake signals with a number of GO states, and in the penultimate cycle LAST. The LAST indicating that the next transfer is the final one. If there is only one transfer, the sequence is [WAIT,[WAIT,...]],LAST.

7.2.1 Coprocessor handshake encoding

Table 7-2 on page 7-8 shows how the handshake signals **CHSDE[1:0]** and **CHSEX[1:0]** are encoded.

Table 7-2 Handshake encoding

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

If a coprocessor is not attached to the ARM940T, then the handshake signals must be driven with ABSENT.

If you attach multiple coprocessors to the interface, the handshaking signals can be combined by ANDing bit 1, and ORing bit 0. In the case of two coprocessors that 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]**

Consequently, if the coprocessor does not recognize a coprocessor instruction, it must drive **CHSDE[1:0]** and **CHSEX[1:0]** with ABSENT.

7.3 MCR/MRC

These cycles look very similar to STC or LDC. An example, with a busy-wait state, is shown in Figure 7-3 on page 7-9.

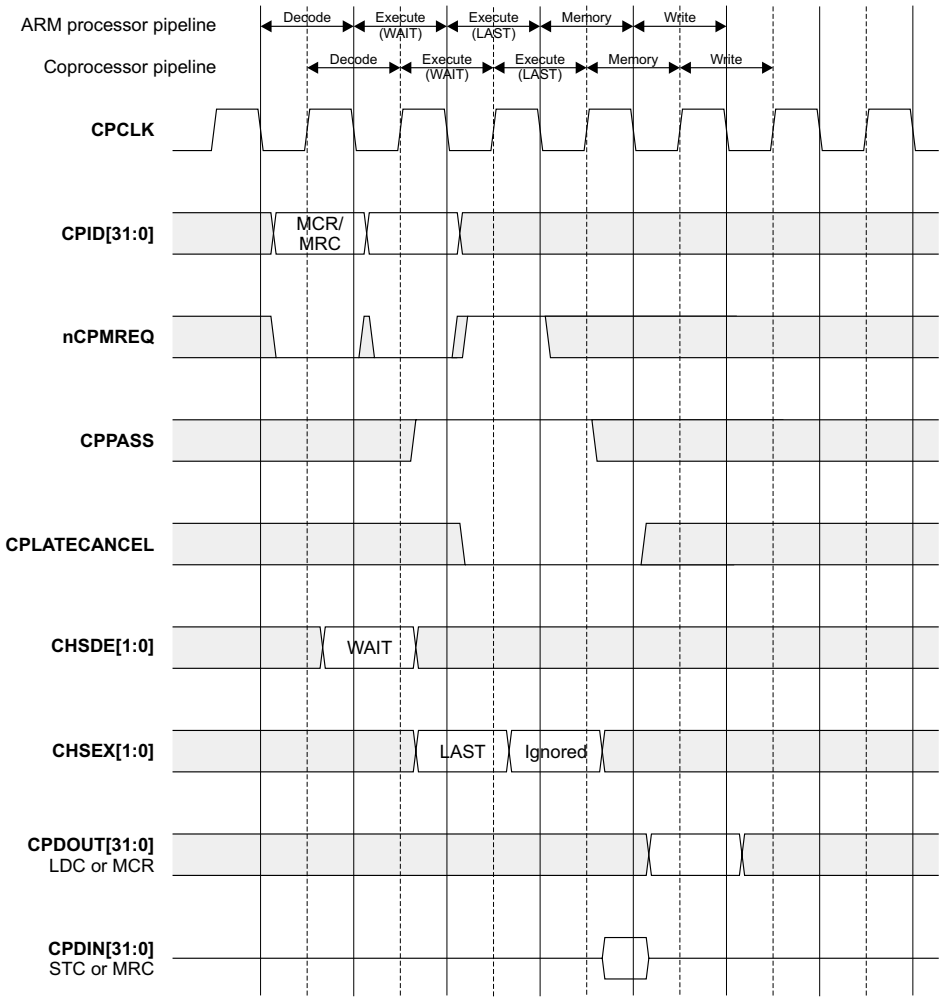

Figure 7-3 ARM940T MCR/MRC transfer timing

First **nCPMREQ** is driven LOW to denote that the instruction on **CPID** is entering the Decode stage of the pipeline. This causes the coprocessor to decode the new instruction and drive **CHSDE[1:0]** as required.

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 pass, 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).

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 an `MCR`, the **CPDOUT[31:0]** bus is driven with the register data. In the case of an `MRC`, **CPDIN[31:0]** is sampled at the end of the ARM940T Memory stage and written to the destination register during the next cycle.

For an `MCR` or `MRC` with no busy-wait states, the coprocessor drives **CHSDE[1:0]** with LAST. This commits the instruction for execution in the next cycle. The value on **CHSEX[1:0]** is ignored.

7.4 Interlocked MCR

If the data for an MCR operation is not available inside the ARM940T pipeline during its first Decode cycle, the ARM940T 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 remains there for a number of cycles before entering the Execute stage. Figure 7-4 on page 7-11 gives an example of an interlocked MCR.

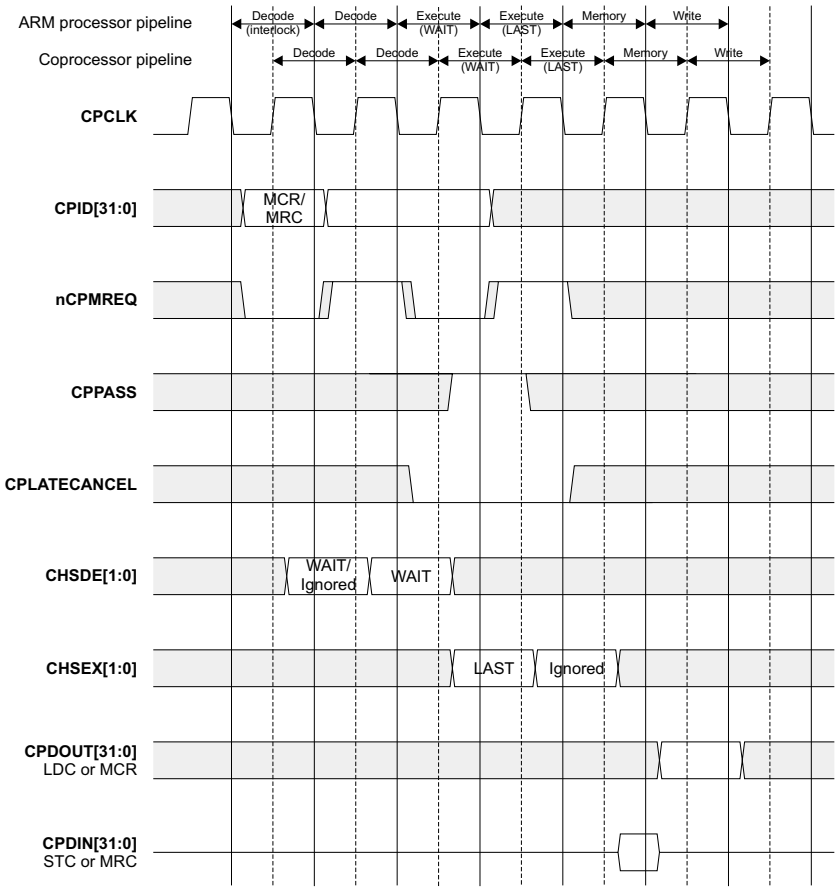

Figure 7-4 ARM940T interlocked MCR

In this example the MCR busy-waits the ARM9TDMI. When the instruction enters the Decode stage of the coprocessor pipeline, the coprocessor drives **CHSDE[1:0]** with WAIT. Due to an interlock in the ARM9TDMI, the instruction remains in Decode for an extra cycle. This is signaled to the coprocessor by **nCPMREQ** going HIGH, holding the instruction in the Decode stage of the coprocessor pipeline follower. The coprocessor signals WAIT to the ARM9TDMI during its second Decode cycle. The interlock in the ARM9TDMI resolves, **nCPMREQ** goes LOW, and the instruction moves from Decode into Execute.

7.5 CDP

CDPs 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 is to be executed, the **CPPASS** signal is driven HIGH during phase 2 of the Execute stage
- 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 7-5 on page 7-14 shows a CDP that is cancelled because the previous instruction causes a Data Abort. The CDP instruction enters the Execute stage of the pipeline, and is signalled to execute by **CPPASS**. In the following phase **CPLATECANCEL** is asserted. This causes the coprocessor to terminate execution of the CDP instruction, and ensures that the instruction does not cause any state changes in the coprocessor

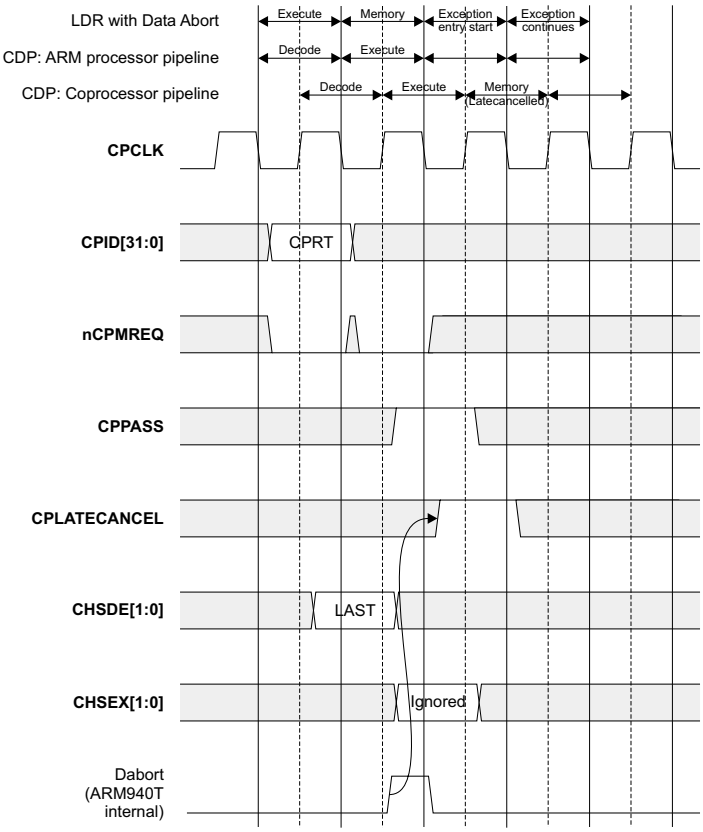

Figure 7-5 ARM940T late-cancelled CDP

7.6 Privileged instructions

The coprocessor restricts certain instructions for use in privileged modes only. To do this, the coprocessor must track the **nCPTRANS** output. Figure 7-6 on page 7-15 shows how **nCPTRANS** changes after a mode change.

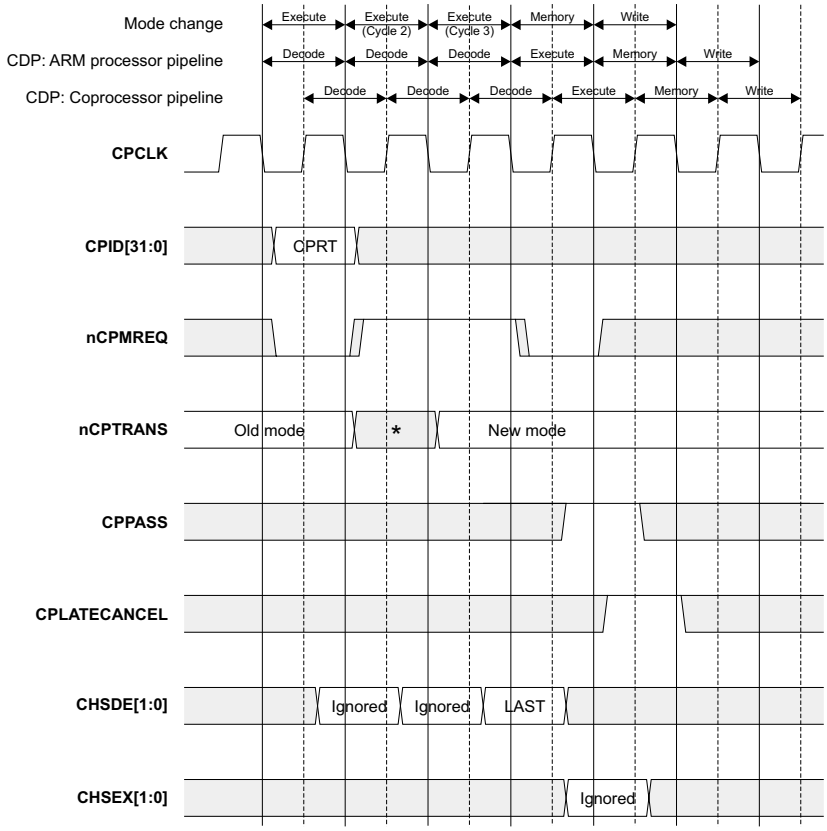

Figure 7-6 ARM940T privileged instructions

In Figure 7-6 on page 7-15 the mode change (marked with an asterisk) occurs as follows:

- for mode changes that do not use an MSR, the mode changes after the first Execute cycle
- for mode changes that use an MSR, the mode changes after the second Execute cycle.

Note

The first two **CHSDE** responses are ignored by the ARM940T because it is only the final **CHSDE** response, as the instruction moves from Decode into Execute, that is relevant. This allows the coprocessor to change its response as **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 must drive WAIT onto **CHSDE[1:0]**. When the instruction concerned enters the Execute stage of the pipeline, the coprocessor can drive WAIT onto **CHSEX[1:0]** for as many cycles as required to keep the instruction in the busy-wait loop.

For interrupt latency reasons, the coprocessor can be interrupted while busy-waiting causing the instruction to be abandoned. Abandoning execution is done using **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 7-7 on page 7-18 shows a busy-waited coprocessor instruction being abandoned because of an interrupt.

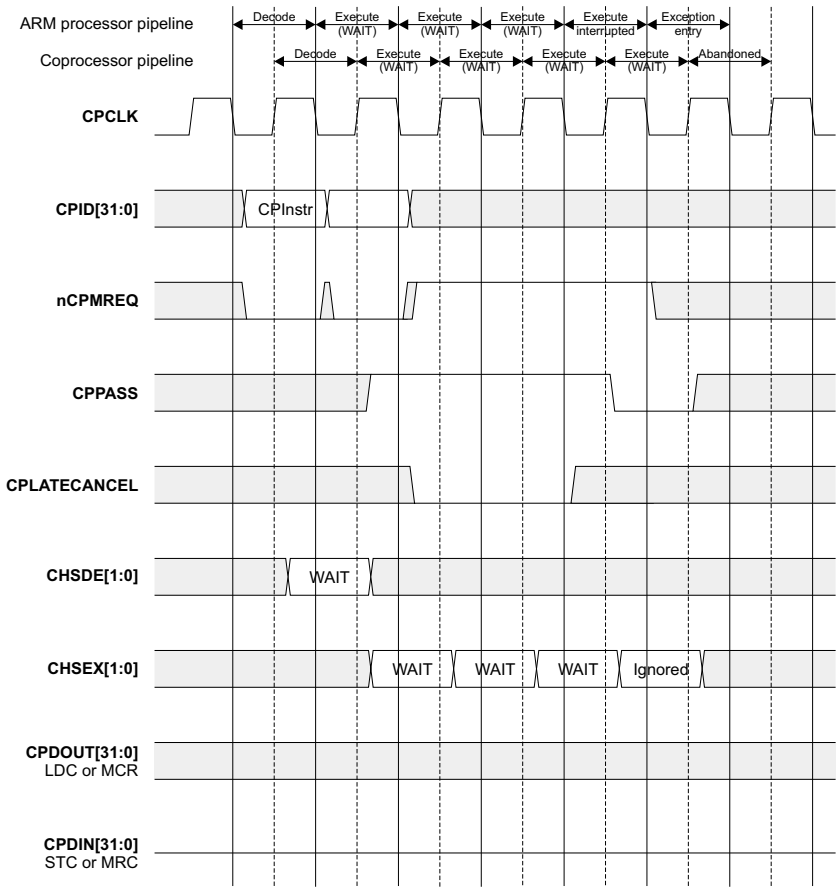

Figure 7-7 ARM940T busy-waiting and interrupts

Chapter 8
Debug Support

This chapter describes the debug support for the ARM940T. It contains the following sections:

8.1 About debug support

The ARM940T debug interface is based on *IEEE Std. 1149.1- 1990, Standard Test Access Port and Boundary-Scan Architecture*. See this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

The ARM940T contains hardware extensions for advanced debugging features to simplify the development of application software, operating systems, and hardware.

The debug extensions allow the core to be stopped by one of the following:

- a given instruction fetch (breakpoint)
- a data access (watchpoint)
- asynchronously by a debug request.

When this happens, the ARM940T is said to be in *debug state*. At this point, the internal state of the core and the external state of the system can be examined. When examination is complete, the core and system state can be restored and program execution resumed.

The ARM940T is forced into debug state either by a request on one of the external debug interface signals, or by an internal functional unit known as the EmbeddedICE unit. In debug state, the core isolates itself from the memory system. The core can then be examined while all other system activity continues as normal.

You can examine the internal state of the ARM940T using a JTAG-style serial interface. This allows instructions to be serially inserted into the core pipeline without using the external data bus. So, when in debug state, you can insert a *Store Multiple* (`STM`) into the instruction pipeline to export the contents of the ARM9TDMI registers. This data can be serially shifted out without affecting the rest of the system.

8.2 Debug systems

The ARM940T forms one component of a debug system that interfaces from the high-level debugging performed by the user to the low-level interface supported by the ARM940T. A typical system is shown in Figure 8-1 on page 8-3.

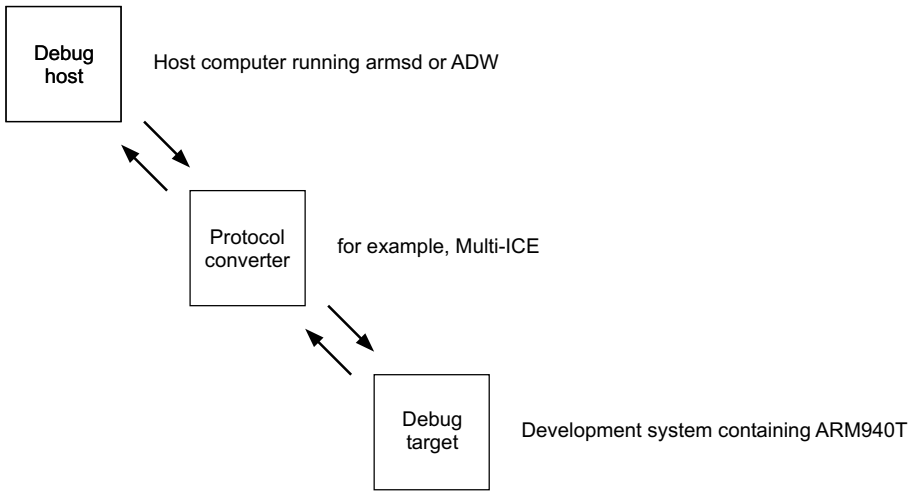

Figure 8-1 Typical debug system

Such a system typically has three parts:

- *Debug host* on page 8-3
- *Protocol converter* on page 8-3
- *Debug target (ARM940T)* on page 8-4.

8.2.1 Debug host

The debug host is a computer such as a PC, running a software debugger such as ADW. The debug host allows you to issue high-level commands such as *set breakpoint at location XX*, or *examine the contents of memory from 0x0 to 0x100*.

8.2.2 Protocol converter

The debug host is connected to the ARM940T development system using an interface (an RS232 interface, for example). The messages broadcast over this connection must be converted to the interface signals of the ARM940T, and this function is performed by the protocol converter (for example, Multi-ICE).

8.2.3 Debug target (ARM940T)

The ARM940T, with hardware extensions to support debugging, is the lowest level of the system. The debug extensions allow you to

- stall the core from program execution
- examine its internal state and the state of the memory system
- resume program execution.

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

8.3 Debug interface signals

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

- **IEBKPT**, **DEWPT**, and **EDBGRQ**, that are used by system requests to cause the ARM9TDMI to enter debug state
- **DBGACK**, that is used by the ARM940T to flag back to the system when it is in debug state
- **DBGEN**, that must be HIGH to allow the use of the EmbeddedICE unit debug facilities.

8.3.1 Entry into debug state on breakpoint

Any instruction being fetched for memory is latched at the end of phase2. To apply a breakpoint to that instruction, the breakpoint signal must be asserted by the end of the following phase1. This minimizes the set-up time, giving the EmbeddedICE unit an entire phase to perform the comparison in. This is shown in Figure 8-2 on page 8-5.

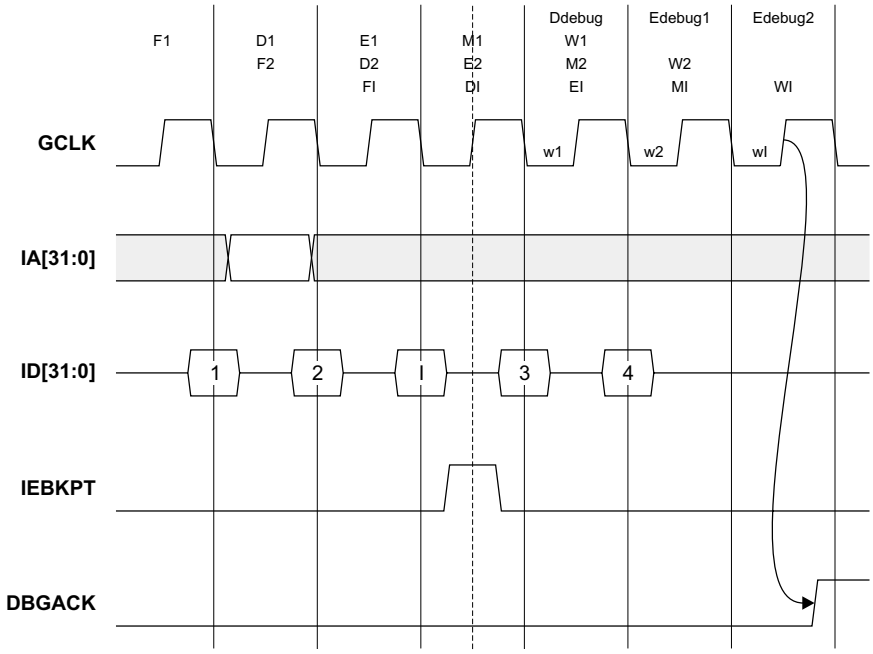

Figure 8-2 Breakpoint timing

You can build external logic, such as additional breakpoint comparators, to extend the functionality of the EmbeddedICE unit. You must apply the external logic output to the **IEBKPT** input. This signal is ORed with the internally generated breakpoint signal before being applied to the ARM9TDMI core control logic.

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.

8.3.2 Breakpoints and exceptions

If a breakpointed instruction has a Prefetch Abort associated with it, the Prefetch Abort takes priority and the breakpoint is ignored. (If there is a Prefetch Abort, instruction data could be invalid, the breakpoint might have been data-dependent, and because the data could 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 (**IRQ** or **FIQ**), the interrupt is taken and the breakpointed instruction is discarded. When the interrupt has been serviced, the execution flow is returned to the original program. This means that the instruction that previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters the debug state when it reaches the Execute stage of the pipeline.

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

8.3.3 Watchpoints

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

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 until the instruction completes. The timing of debug entry following a watchpointed load in this case is shown in Figure 8-3 on page 8-7.

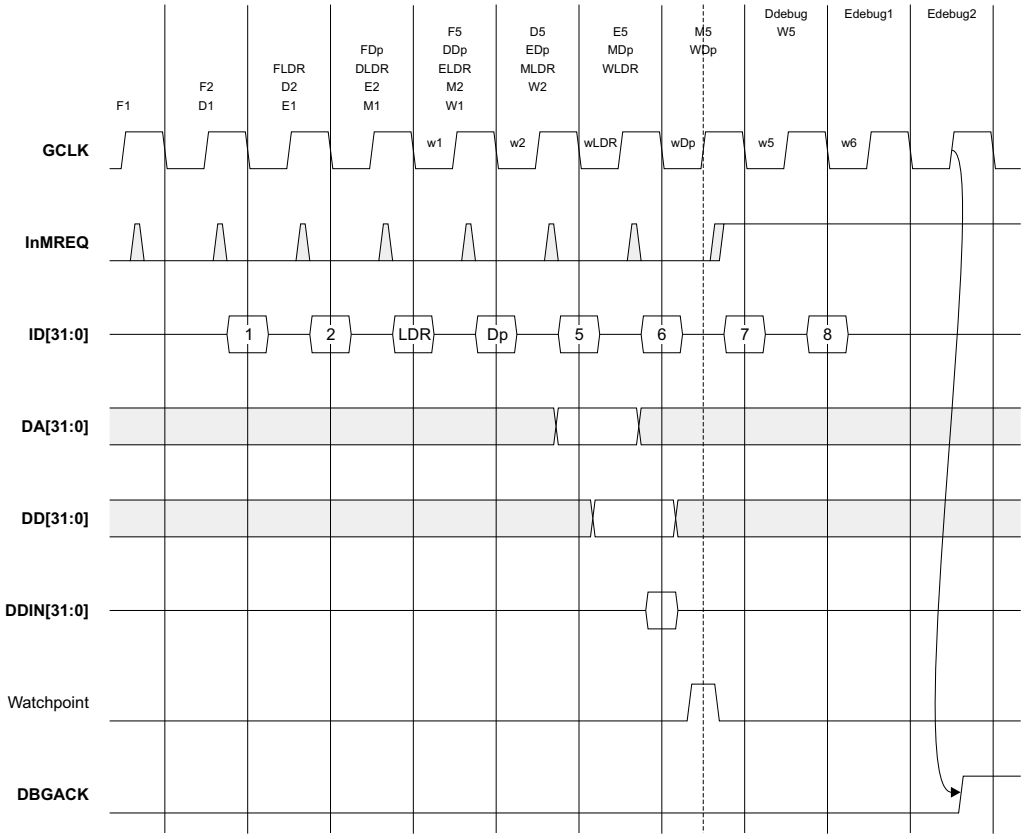

Figure 8-3 Watchpoint entry with data processing instruction

Note

Although instruction 5 enters the Execute state, it is not executed. Also, there is no state update as a result of this instruction. When the debugging session is complete, normal execution continues with a return to instruction 5, the next instruction in the code sequence that has not yet been executed.

The instruction following the instruction that generated the watchpoint might have modified the *Program Counter* (PC). If this has happened, 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 8-4 on page 8-9. However, you can always restart the processor.

When the processor has entered debug state, you can interrogate the ARM940T core 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 is restarted, execution flow returns to the next instruction in the code sequence

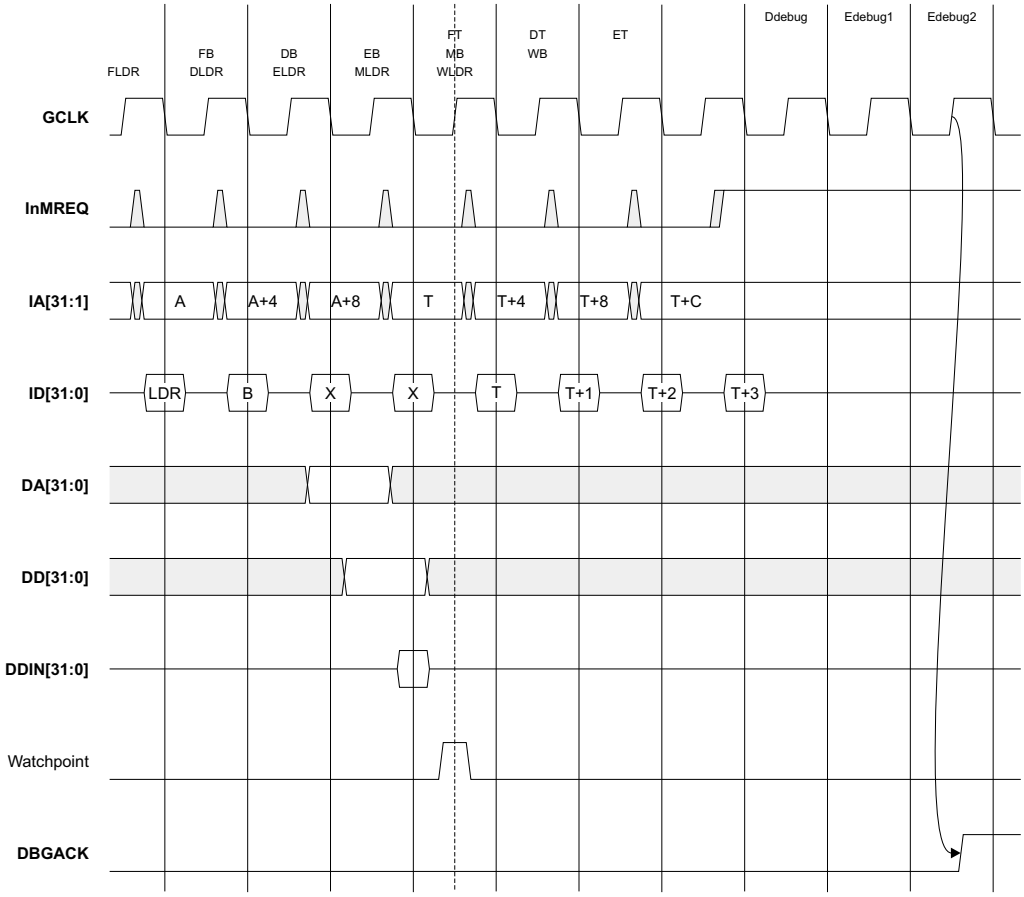

Figure 8-4 Watchpoint entry with branch

8.3.4 Watchpoints and exceptions

If an abort occurs during a watchpointed data access, 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 ARM940T allows the exception entry sequence to occur and then enters debug state.

8.3.5 Debug request

A debug request can take place through the EmbeddedICE unit 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 Execution stage of the pipeline has completely finished executing (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.

8.3.6 Actions of the ARM940T in debug state

When the ARM940T is in debug state, both memory interfaces indicate internal cycles. Because the rest of the system continues operation, the ARM940T ignores aborts and interrupts.

8.4 Scan chains and JTAG interface

The ARM940T provides a JTAG-style *Test Access Port* (TAP) controller that supports 32 scan chains. Of these, scan chains 0 to 15 are reserved for use by ARM.

The ARM940T implements six internal scan chains, scan chains 0, 1, 2, 4, 5, and 15, that allow testing, debugging, and programming of the EmbeddedICE units. An additional seventh scan chain (scan chain 3) can be implemented to provide an external boundary scan chain around the pads of a packaged device.

External scan chains can be implemented in the remaining space (16-31). The signals **SCREG[4:0]** indicate the external scan chain that is being accessed.

The active scan chain is selected by using the scan chain select register. For a complete listing of the ARM940T scan chains, see *Scan chain select register* on page 8-19.

8.5 The JTAG state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure 8-5 on page 8-12 shows the state transitions that occur in the TAP controller. The state numbers shown on the diagram are output from the ARM940T on the **TAPSM[3:0]** bits.

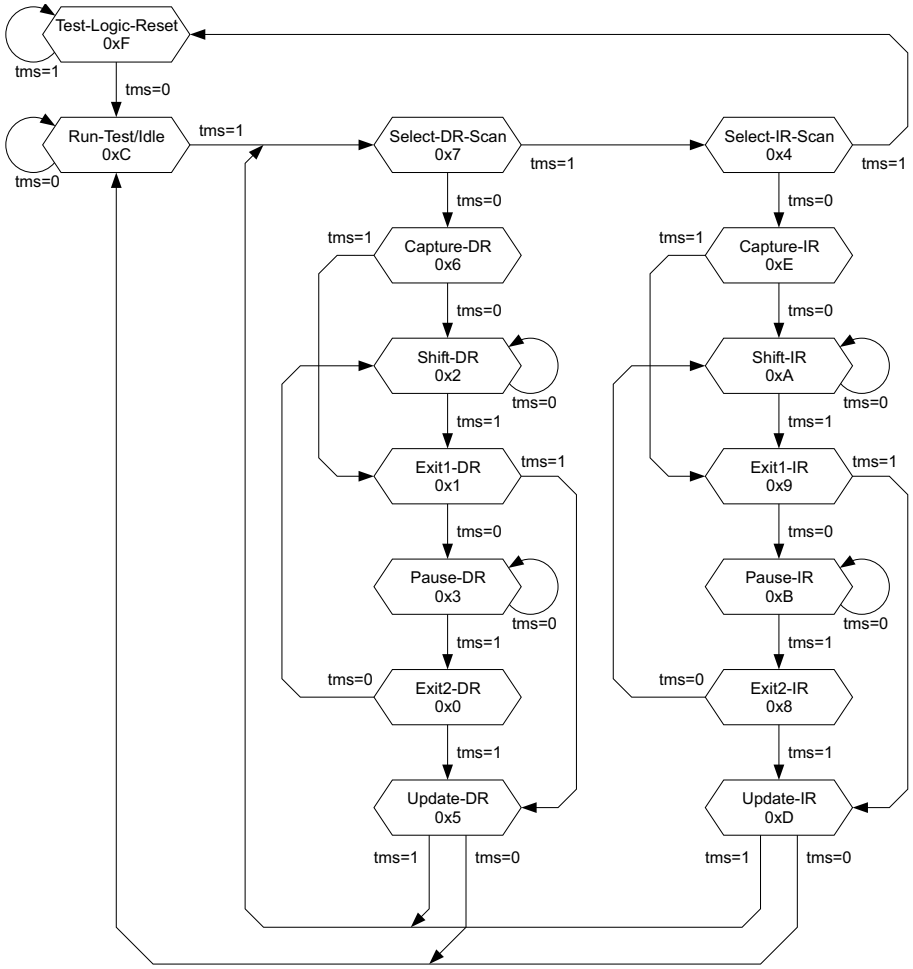

Figure 8-5 Test access port (TAP) controller state transitions[1]

1. From IEEE Std 1149.1-1990. Copyright 2000 IEEE. All rights reserved.

8.5.1 Reset

The JTAG interface includes a state-machine controller (the TAP controller). To force the TAP controller into the correct state after power-up of the device, a reset pulse must be applied to the **nTRST** signal. If the JTAG interface is to be used, **nTRST** must be driven LOW, and then HIGH again. If the boundary scan interface is not to be used, the **nTRST** input can be tied permanently LOW.

Note

A clock on **TCK** is not necessary to reset the device.

The action of reset is as follows:

1. System mode is selected. The boundary scan chain cells do not intercept any of the signals passing between the external system and the core.
2. The IDCODE instruction is selected. If the TAP controller is put into the Shift-DR state and **TCK** is pulsed, the contents of the ID register are clocked out of **TDO**.

8.5.2 Pull-up resistors

The IEEE 1149.1 standard effectively requires **TDI** and **TMS** to have internal pull-up resistors. To minimize static current draw, these resistors are not fitted to the ARM940T. Accordingly, the four inputs to the test interface (the **TDO**, **TDI**, and **TMS** signals plus **TCK**) must all be driven to valid logic levels to achieve normal circuit operation.

8.5.3 Instruction register

The instruction register is four bits in length. There is no parity bit. The fixed value loaded into the instruction register during the CAPTURE-IR controller state is 0001.

8.5.4 Public instructions

Table 8-1 on page 8-13 shows the public instructions that are supported.

Table 8-1 Public instructions

Instruction	Binary code
EXTEST	0000
SCAN_N	0010
INTEST	1100

Table 8-1 Public instructions (continued)

Instruction	Binary code
IDCODE	1110
BYPASS	1111
CLAMP	0101
HIGHZ	0111
CLAMPZ	1001
SAMPLE/PRELOAD	0011
RESTART	0100

Note

The EXTEST, HIGHZ, and CLAMPZ instructions for scan chains 0-15 are reserved for production test purpose only and must not be used.

In the descriptions that follow, **TDI** and **TMS** are sampled on the rising edge of **TCK** and all output transitions on **TDO** occur as a result of the falling edge of **TCK**.

EXTEST (0000)

The selected scan chain is placed in test mode by the EXTEST instruction.
The EXTEST instruction connects the selected scan chain between **TDI** and **TDO**.

When the instruction register is loaded with the EXTEST instruction, all the scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, inputs from the system logic and outputs from the output scan cells to the system are captured by the scan cells.

In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain using **TDO**, while new test data is shifted in using the **TDI** input. This data is applied immediately to the system logic and system pins.

SCAN_N (0010)

This instruction connects the Scan Path Select register between **TDI** and **TDO**.

During the CAPTURE-DR state, the fixed value 10000 is loaded into the register.

During the SHIFT-DR state, the ID number of the desired scan path is shifted into the scan path select register.

In the UPDATE-DR state, the scan register of the selected scan chain is connected between **TDI** and **TDO**, and remains connected until a subsequent SCAN_N instruction is issued. On reset, scan chain 3 is selected by default. The scan path select register is five bits long in this implementation, although no finite length is specified.

INTEST (1100)

The selected scan chain is placed in test mode by the INTEST instruction. The INTEST instruction connects the selected scan chain between **TDI** and **TDO**.

When the instruction register is loaded with the INTEST instruction, all the scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, the value of the data applied from the core logic to the output scan cells, and the value of the data applied from the system logic to the input scan cells is captured.

In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain through the **TDO** pin, while new test data is shifted in using the **TDI** pin.

IDCODE (1110)

The IDCODE instruction connects the device identification register (or ID register) between **TDI** and **TDO**. The ID register is a 32-bit register that allows the manufacturer, part number and version of a component to be determined through the TAP.

When the instruction register is loaded with the IDCODE instruction, all the scan cells are placed in their normal (system) mode of operation.

In the CAPTURE-DR state, the device identification code is captured by the ID register.

In the SHIFT-DR state, the previously captured device identification code is shifted out of the ID register using the **TDO** pin, while data is shifted in using the **TDI** pin into the ID register.

In the UPDATE-DR state, the ID register is unaffected.

BYPASS (1111)

The BYPASS instruction connects a 1-bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the BYPASS instruction is loaded into the instruction register, all the scan cells are placed in their normal (System) mode of operation. This instruction has no effect on the system pins.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register.

In the SHIFT-DR state, test data is shifted into the bypass register using **TDI** and out using **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a zero.

The bypass register is not affected in the UPDATE-DR state.

Note

All unused instruction codes default to the BYPASS instruction.

CLAMP (0101)

This instruction connects a 1-bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the CLAMP instruction is loaded into the instruction register, the state of all the output signals is defined by the values previously loaded into the currently loaded scan chain.

Note

This instruction must only be used when scan chain 0 is the currently selected scan chain.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register.

In the SHIFT-DR state, test data is shifted into the bypass register using **TDI** and out using **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a zero.

The bypass register is not affected in the UPDATE-DR state.

HIGHZ (0111)

This instruction connects a 1-bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the HIGHZ instruction is loaded into the instruction register and scan chain 0 is selected, all ARM9TDMI outputs are driven to the high-impedance state, and the external **HIGHZ** signal is driven HIGH. This functions as if the ARM9TDMI signal **TBE** had been driven LOW.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register. In the SHIFT-DR state, test data is shifted into the bypass register using **TDI** and out using **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a zero.

The bypass register is not affected in the UPDATE-DR state.

CLAMPZ (1001)

This instruction connects a 1-bit shift register (the BYPASS register) between **TDI** and **TDO**.

When the CLAMPZ instruction is loaded into the instruction register and scan chain 0 is selected, all the tristate outputs (as described above) are placed in their inactive state, but the data supplied to the outputs is derived from the scan cells. The purpose of this instruction is to ensure that, during production test, each output can be disabled when its data value is either a logic 0 or logic 1.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register.

In the SHIFT-DR state, test data is shifted into the bypass register using **TDI** and out using **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a zero.

The bypass register is not affected in the UPDATE-DR state.

SAMPLE/PRELOAD (0011)

When the instruction register is loaded with the SAMPLE/PRELOAD instruction, all the scan cells of the selected scan chain are placed in the normal mode of operation.

In the CAPTURE-DR state, a snapshot of the signals of the boundary scan is taken on the rising edge of **TCK**. Normal system operation is unaffected.

In the SHIFT-DR state, the sampled test data is shifted out of the boundary scan using the **TDO** pin, while new data is shifted in using the **TDI** pin to preload the boundary scan parallel input latch. This data is not applied to the system logic or system pins while the SAMPLE/PRELOAD instruction is active.

This instruction must be used to preload the boundary scan register with known data prior to selecting INTEST or EXTEST instructions.

RESTART (0100)

This instruction is used to restart the processor on exit from debug state. The RESTART instruction connects the bypass register between **TDI** and **TDO** and the TAP controller behaves as if the BYPASS instruction had been loaded. The processor resynchronizes back to the memory system when the RUN-TEST/IDLE state is entered.

8.6 Test data registers

The following test data registers can be connected between **TDI** and **TDO**:

- *Bypass register* on page 8-18
- *ARM940T device identification (ID) code register* on page 8-18
- *Instruction register* on page 8-19
- *Scan chain select register* on page 8-19
- *Scan chains 0, 1, 2, 3, 4, 5, and 15* on page 8-21.

8.6.1 Bypass register

Purpose Bypasses the device during scan testing by providing a path between **TDI** and **TDO**.

Length 1 bit

Operating mode When the BYPASS instruction is the current instruction in the instruction register, serial data is transferred from **TDI** to **TDO** in the SHIFT-DR state with a delay of one **TCK** cycle. There is no parallel output from the bypass register. A logic 0 is loaded from the parallel input of the bypass register in CAPTURE-DR state.

8.6.2 ARM940T device identification (ID) code register

Purpose Reads the 32-bit device identification code. No programmable supplementary identification code is provided.

Length 32 bits

Operating mode When the IDCODE instruction is current, the ID register is selected as the serial path between **TDI** and **TDO**. There is no parallel output from the ID register. The 32-bit identification code is loaded into the register from its parallel inputs of the input bus **TAPID[31:0]** during the CAPTURE-DR state.

The IEEE format of the ID register is as follows:

Table 8-2 ID code register

Bits	Contents	Value
31–28	Version number	0x2
27–12	Part number	0x0940
11–1	Manufacturer identity	Default = 0b11110000111
0	IEEE standard specified	0b1

The **TAPID[31:0]** pins allow this value to be set when the macrocell is instantiated in a design.

8.6.3 Instruction register

Purpose Changes the current TAP instruction.

Length 4 bits

Operating mode When in SHIFT-IR state, the instruction register is selected as the serial path between **TDI** and **TDO**.

During the CAPTURE-IR state, the value 0b0001 is loaded into this register. This is shifted out during SHIFT-IR (least significant bit first), while a new instruction is shifted in (least significant bit first). During the UPDATE-IR state, the value in the instruction register becomes the current instruction. On reset, IDCODE becomes the current instruction.

8.6.4 Scan chain select register

Purpose Changes the current active scan chain.

Length 5 bits

Operating mode After SCAN_N has been selected as the current instruction, when in SHIFT-DR state, the scan chain select register is selected as the serial path between **TDI** and **TDO**.

During the CAPTURE-DR state, the value 0b10000 is loaded into this register. This is shifted out during SHIFT-DR (least significant bit first), while a new value is shifted in (least significant bit first).

During the UPDATE-DR state, the value in the register selects a scan chain to become the currently active scan chain. All following instructions such as INTEST then apply to that scan chain.

The currently selected scan chain only changes when a SCAN_N instruction is executed, or a reset occurs. On reset, scan chain 3 is selected as the active scan chain.

The number of the currently selected scan chain is reflected on the **SCREG[4:0]** output bus. The TAP controller can be used to drive external scan chains in addition to those within the ARM940T macrocell. The external scan chain must be assigned a number and control signals for it, and can be derived from **SCREG[4:0]**, **IR[3:0]**, **TAPSM[3:0]**, **TCK1**, and **TCK2**.

The list of scan chain numbers allocated by ARM is shown in Table 8-3 on page 8-20. An external scan chain can take any other number. The serial data stream applied to the external scan chain is made present on **SDINBS**. The serial data back from the scan chain must be presented to the TAP controller on the **SDOUTBS** input.

The scan chain present between **SDINBS** and **SDOUTBS** is connected between **TDI** and **TDO** whenever scan chain 3 is selected, or when any of the unassigned scan chain numbers is selected. If there is more than one external scan chain, a multiplexor must be built externally to apply the desired scan chain output to **SDOUTBS**. The multiplexor can be controlled by decoding **SCREG[4:0]**.

Table 8-3 Scan chain number allocation

Scan chain number	Function
0	Macrocell scan test
1	Debug
2	EmbeddedICE unit programming
3	External boundary scan
4	ICache CAM
5	DCache CAM
6-14	Reserved
15	Control coprocessor
16–31	Unassigned

8.6.5 Scan chains 0, 1, 2, 3, 4, 5, and 15

These allow serial access to the core logic, and to the EmbeddedICE unit for programming purposes. Each scan cell can perform two basic functions, capture and shift.

Scan chain 0

Purpose Primarily for inter-device testing (EXTEST), and testing the ARM9TDMI core (INTEST). Scan chain 0 is selected using the SCAN_N instruction.

Length 184 bits

INTEST allows serial testing of the ARM9TDMI core. The TAP controller must be placed in the INTEST mode after scan chain 0 has been selected.

During CAPTURE-DR, the current outputs from the core logic are captured in the output cells.

During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in. This ensures that known stimuli are applied to the inputs.

During RUN-TEST/IDLE, the core is clocked. Normally, the TAP controller only spends one cycle in RUN-TEST/IDLE. The whole operation can then be repeated.

EXTEST can be used to drive the outputs of the ARM9TDMI core and to capture its inputs. It exists on the ARM940T to aid diagnostic testing but in its embedded form is of limited use.

Scan chain 1

Purpose Primarily for debugging. Scan chain 1 is selected using the SCAN_N TAP controller instruction.

Length 67 bits

This scan chain is 67 bits long, 32 bits for data values, 32 bits for instruction data, and 3 control bits, SYSSPEED, WPTANDBKPT, and **DDEN**. The three control bits serve four different purposes:

- Under normal INTEST test conditions, the **DDEN** signal can be captured and examined.
- During EXTEST test conditions, a known value can be scanned into **DDEN** to be driven into the rest of the system. If a logic 1 is scanned into **DDEN**, the data data bus **DD[31:0]** drives out the values stored in its scan cells. If a logic 0 is scanned into **DDEN**, **DD[31:0]** captures the current input values.
- While debugging, the value placed in the SYSSPEED control bit determines if the ARM9TDMI synchronizes back to system speed before executing the instruction.
- After the ARM9TDMI has entered debug state, the first time SYSSPEED is captured and scanned out, its value tells the debugger if the core has entered debug state in response to a breakpoint (SYSSPEED LOW), or a watchpoint (SYSSPEED HIGH). If the instruction directly following one that causes a watchpoint has a breakpoint set on it, then the WPTANDBKPT bit is set. This situation does not effect how to restart the code.

Scan chain 2

Purpose Allows access to the EmbeddedICE unit registers. The order of the scan chain from **TDI** to **TDO** is shown in Table 8-4 on page 8-22.

Table 8-4 Scan chain 4 addressing mode bit order

Bits	Contents
37	Read = 0 Write = 1
36:32	EmbeddedICE register address
31:0	Data

Length 38 bits

To access this serial register, scan chain 2 must first be selected using the SCAN_N TAP controller instruction. The TAP controller must then be placed in INTEST mode.

No action is taken during CAPTURE-DR.

During SHIFT-DR, a data value is shifted into the serial register. Bits 32 to 36 specify the address of the EmbeddedICE unit register to be accessed.

During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read).

The EmbeddedICE register map is shown in *Control registers* on page 8-41.

Scan chain 3

Purpose Allows the ARM940T to control an external boundary scan chain.

Length User-defined

Scan chain 3 is provided so that an optional external boundary scan chain can be controlled using the ARM940T. Typically this is used for a scan chain around the pad ring of a packaged device. The following control signals are provided and are generated only when scan chain 3 has been selected. These outputs are inactive at all other times:

DRIVEOUTBS This is used to switch the scan cells from system mode to test mode. This signal is asserted whenever an INTEST, EXTEST, CLAMP, or CLAMPZ instruction is selected.

PCLKBS This is the update clock, generated in the UPDATE-DR state. Typically the value scanned into the chain is transferred to the cell output on the rising edge of this signal.

ICAPCLKBS, ECAPCLKBS

These are the capture clocks used to sample data into the scan cells during INTEST and EXTEST respectively. These clocks are generated in the CAPTURE-DR state.

SHCLK1BS, SHCLK2BS

These are non-overlapping clocks generated in the SHIFT-DR state that are used to clock the master and slave element of the scan cells respectively. When the state machine is not in the SHIFT-DR state, both these clocks are LOW.

RSTCLKBS This signal is used to reset the cells of the Boundary Scan Chain when **nTRST** is asserted or the TAP controller state machine is in the reset state.

In addition to these control outputs, **SDINBS** outputs and **SDOUTBS** inputs are provided to support external scan chains. When an external scan chain is in use, **SDOUTBS** must be connected to the serial data output of the external chain and **SDINBS** must be connected to the serial data input of the external chain.

Scan chain 4

Purpose Allows access to the ICache CAM array. The scan chain has two modes of operation, addressing mode and reading mode.

In addressing mode, the order of the scan chain **TDI** to **TDO** is:

Table 8-5 Scan chain 4 addressing mode bit order

Bits	Contents
28:22	CAM index
21:6	SBZ
5:0	Segment

In reading mode, the order of the scan chain **TDI** to **TDO** is:

Table 8-6 Scan chain 4 reading mode bit order

Bits	Contents
28	Valid
27	Dirty
26:0	TAG

To access this serial register, scan chain 4 must first be selected using the SCAN_N TAP controller instruction. The TAP controller must then be placed in INTEST mode.

During SHIFT-DR, a CAM index can be addressed by shifting data into the serial register in the addressing mode format. Bits 5 to 0 define the cache segment and bits 28 to 22 the CAM index to be accessed. However, because the ARM940T has four segments in the ICache, only bits 1 and 0 are required. Bits 5:2 should be zero. Similarly, the 64 CAM indexes are selected with bits 27 to 22, and bit 28 should be zero. The extra segment and index bits are reserved for future implementations.

During UPDATE-DR, the addressed CAM index data is transferred to the serial register in the reading mode format.

Scan chain 5

Purpose Allows access to the DCache CAM array. The scan chain has two modes of operation. In addressing mode, the order of the scan chain **TDI** to **TDO** is shown in Table 8-7 on page 8-25.

Table 8-7 Scan chain 5 addressing mode bit order

Bits	Contents
28:22	CAM index
21:6	SBZ
5:0	Segment

In reading mode, the order of the scan chain **TDI** to **TDO** is shown in Table 8-8 on page 8-25.

Table 8-8 Scan chain 5 reading mode bit order

Bits	Contents
28	Valid
27	Dirty
26:0	TAG

To access this serial register, scan chain 5 must first be selected using the SCAN_N TAP controller instruction. The TAP controller must then be placed in INTEST mode.

During SHIFT-DR, a CAM index can be addressed by shifting data into the serial register in the addressing mode format. Bits 5 to 0 define the cache segment and bits 28 to 22 the CAM index to be accessed. However, because the ARM940T has four segments in the ICache, only bits 1 and 0 are required. Bits 5:2 should be zero. Similarly, the 64 CAM indexes are selected with bits 27 to 22, and bit 28 should be zero. The extra segment and index bits are reserved for future implementations.

During UPDATE-DR, the addressed CAM index data is transferred to the serial register in the reading mode format.

Scan chain 15

Purpose Allows access to the control coprocessor (CP15) registers. The order of the scan chain **TDI** to **TDO** is shown in Table 8-9 on page 8-26.

Table 8-9 Scan chain 4 addressing mode bit order

Bits	Contents
38	Read = 0 Write = 1
37:32	CP15 register address
31:0	Register value

Length 39 bits

To access this serial register, scan chain 15 must first be selected using the SCAN_N TAP controller instruction. The TAP controller must then be placed in INTEST mode.

No action is taken during CAPTURE-DR.

During SHIFT-DR, a data value is shifted into the serial register. Bits 32 to 37 specify the address of the CP15 register to be accessed.

During UPDATE-DR, this register is either read or written depending on the value of bit 38 (0 = read).

8.7 ARM940T core clocks

The source **GCLK** applied to the internal ARM9TDMI core is dependent on the current selected clock mode and the operation being performed. See Clock Modes for more details.

The ARM9TDMI core has two clocks, the memory clock **GCLK**, and an internally **TCK** generated clock, **DCLK**. During normal operation, the core is clocked by **GCLK**, and internal logic holds **DCLK** LOW. When the ARM940T is in the debug state, the ARM9TDMI core is clocked by **DCLK** under control of the TAP state machine, and **GCLK** can free run. The selected clock is output on the **ECLK** signal for use by the external system.

Note

When the core is being debugged and is running from **DCLK**, **nWAIT** has no effect.

There are two cases where the clocks switch, during debug and during testing.

8.7.1 Clock switching during debug

When the ARM9TDMI core enters debug state, it must switch from **GCLK** to **DCLK**. This is handled automatically by logic in the ARM9TDMI. On entry to debug state, the ARM9TDMI core asserts **DBGACK** in the HIGH phase of **GCLK**. The switch between the two clocks occurs on the next falling edge of **GCLK**, see Figure 8-6 on page 8-27.

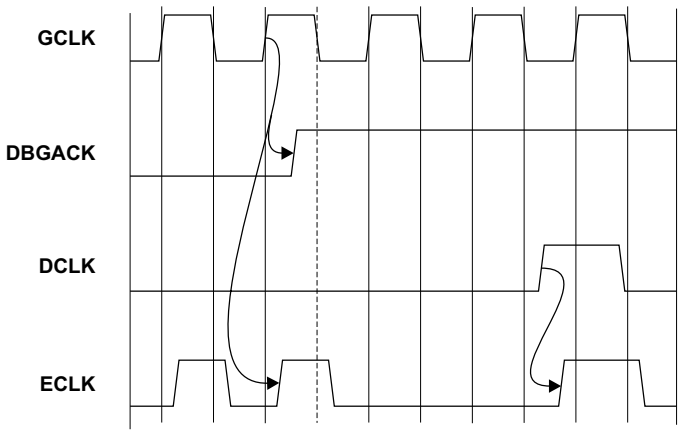

Figure 8-6 Clock switching on entry to debug state

The ARM9TDMI core is forced to use **DCLK** as the primary clock until debugging is complete. On exit from debug, the core must be allowed to synchronize back to **GCLK**. This must be done in the following sequence:

1. The final instruction of the debug sequence must be shifted into the instruction data bus scan chain, and clocked in by asserting **DCLK**. At this point, RESTART must be clocked into the TAP controller register.
2. The ARM9TDMI automatically resynchronizes back to **GCLK** when the TAP controller enters to the RUN-TEST/IDLE mode and starts fetching instructions from memory at **GCLK** speed. For more information, see *Exit from debug state* on page 8-33.

8.7.2 Clock switching during test

Under serial test conditions, when test patterns are being applied to the ARM9TDMI core through the JTAG interface, the ARM9TDMI must be clocked using **DCLK**. Entry into test is less automatic than debug and some care must be taken.

On entry into test, **GCLK** must be held LOW. The TAP controller can now be used to perform serial testing on the ARM9TDMI. If scan chain 0 and INTEST are selected, **DCLK** is generated while the state machine is in RUN-TEST/IDLE state.

During EXTEST, **DCLK** is not generated.

On exit from test, RESTART must be selected as the TAP controller instruction. When this is done, **GCLK** can be allowed to resume. After INTEST testing, care must be taken to ensure that the core is in a sensible state before switching back. The safest way to do this is to either select RESTART and then cause a system reset, or to insert `MOV PC,#0` into the instruction pipeline before switching back.

8.8 Determining the core and system state

You can examine the core and system state when the ARM940T is in debug state. You do this by forcing load and store multiples into the pipeline.

Before the core and system state can be examined, the debugger must first determine if the processor has entered debug from Thumb or ARM state. This is achieved by examining bit 4 of the EmbeddedICE unit debug status register. If this is HIGH, debug has been entered from Thumb state.

8.8.1 Determining the core state

If the processor has entered debug state from Thumb state, the simplest course of action is for the debugger to force the core back into ARM state. When this is done, the debugger can always execute the same sequence of instructions to determine the processor state.

To force the processor into ARM state, the following sequence of Thumb instructions must be executed on the core:

```
STR r0, [r1]    ; Save r0 before use
MOV r0,  PC     ; Copy PC into R0
STR r0, [r1]    ; Save the PC in R0
BX PC           ; Jump into ARM state
MOV r8, r8      ; NOP
MOV r8, r8      ; NOP
```

The above use of R1 as the base register for the stores is for illustration only. Any register can be used.

Because all Thumb instructions are only 16 bits long, the simplest course of action when shifting them into scan chain 1 is to repeat the instruction twice on the instruction data bus bits. For example, the encoding for `BX r0` is `0x4700`. If `0x47004700` is shifted into the 32 bits of the instruction data bus of scan chain 1, then the debugger does not have to keep track of the half of the bus that the processor expects to use to read instructions.

From this point on, the processor state can be determined by the sequences of ARM instructions described below.

When the processor is in ARM state, typically the first instruction executed is:

```
STMIA r0, {r0-r15}
```

This causes the contents of the registers to be made visible on the data data bus. These values can then be sampled and shifted out.

After determining the values in the current bank of registers, you might want to access banked registers. This can only be done by changing mode. Normally, a mode change can only occur if the core is already in a privileged mode. However, while in debug state, a mode change from any mode into any other mode might occur.

Note

The debugger must restore the original mode before exiting debug state.

For example, assume that the debugger has been asked to return the state of the User mode and FIQ mode registers, and debug state has been entered from supervisor mode. The instruction sequence might be:

```
STMIA r0, {r0-r15}      ; Save current registers
MRS   r0, CPSR
STR   r0, {r0}          ; Save CPSR to determine current mode
BIC   r0,r0, #0x1F      ; Clear mode bits
ORR   r0,r0, #0x10      ; Select USER mode
MSR   CPSR_c, r0        ; Enter USER mode
STMIA r0, {r13-r14}     ; Save registers not previously visible
ORR   r0,r0, #0x01      ; Select FIQ mode
MSR   CPSR_c, r0        ; Enter FIQ mode
STMIA r0, {r8-r14}      ; Save banked FIQ registers
```

All these instructions are said to execute at debug speed. Debug speed is much slower than system speed. This is because 67 scan clocks occur between each core clock to shift an instruction in, or shift data out. Executing instructions at debug speed presents no problems for accessing the core state because the ARM9TDMI core is fully static. However, this method cannot be used for determining the state of the rest of the system.

While in debug state, you can only insert the following instructions into the instruction pipeline for execution:

- all data processing operations
- all load, store, load multiple and store multiple instructions
- `MSR` and `MRS`.

8.8.2 Determining system state

To meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously. Therefore, the ARM9TDMI core must be forced to synchronize back to system speed. The 33rd bit of scan chain 1, SYSSPEED, controls this.

A legal debug instruction can be placed in the instruction data bus of scan chain 1 with bit 33 (the SYSSPEED bit) LOW. This instruction is then executed at debug speed. To execute an instruction at system speed, a `NOP` (such as `MOV R0, R0`) must be scanned in as the next instruction with bit 33 set HIGH.

After the system speed instructions have been scanned into the instruction data bus and clocked into the pipeline, the RESTART instruction must be loaded into the TAP controller. This causes the ARM9TDMI core to automatically resynchronize back to **GCLK** when the TAP controller enters RUN-TEST/IDLE state, and execute the instruction at system speed. Debug state is re-entered when the instruction completes execution, when the processor switches itself back to the internally generated **DCLK**. When the instruction has completed, **DBGACK** is HIGH. At this point INTEST can be selected in the TAP controller, and debugging can resume.

Note

When performing system speed accesses, the caches operate as usual, for example, performing cache lookups, linefills, and evicting lines. To prevent the contents of the caches being altered, it is necessary to disable them first. However, when the caches are disabled their contents are preserved. This means that if a write to an address that was held in the data cache occurs while the data cache is disabled, the updatedoes not affect the data cache. If the data cache is then switched back on, it still holds the out of date version of the data, which appears valid. This results in unrecoverable data corruption. To prevent this, you are recommended to always clean and flush the data cache before you disable it.

To determine if a system speed instruction has completed, the debugger must look at SYSCOMP (bit 3 of the debug status register). To access memory, the ARM9TDMI core must access memory through the data bus interface, as this access might be stalled indefinitely by **nWAIT**. The only way to determine if the memory access has completed, is to examine the SYSCOMP bit. When this bit is HIGH, the instruction has completed.

By the use of system speed load multiples and debug store multiples, the state of the system memory can be passed to the debug host.

8.8.3 Instructions that can have the SYSSPEED bit set

The only valid instructions that can have this bit set are:

- loads
- stores
- load multiple
- store multiple.

When the ARM940T returns to debug state after a system speed access, the SYSSPEED bit is set HIGH.

8.9 Exit from debug state

Leaving debug state involves restoring the ARM940T internal state, causing a branch to the next instruction to be executed, and synchronizing back to **GCLK**. After restoring the internal state, a branch instruction must be loaded into the pipeline. For details on calculating the branch, see *The behavior of the program counter during debug* on page 8-36.

Bit 33 of scan chain 1 is used to force the ARM940T to resynchronize back to **GCLK**. The penultimate instruction in the debug sequence is a branch to the instruction where execution is to resume. This is scanned in with bit 33 set LOW. The core is then clocked to load the branch into the pipeline. The final instruction to be scanned in is a NOP (such as `MOV r0, r0`), with bit 33 set HIGH. The core is then clocked to load this instruction into the pipeline, and the RESTART instruction is selected in the TAP controller. When the state machine enters the RUN-TEST/IDLE state, the scan chain reverts back to system mode and clock resynchronization to **GCLK** occurs within the ARM940T. Normal operation then resumes, with instructions being fetched from memory.

The delay, until the state machine is in RUN-TEST/IDLE state, allows conditions to be set up in other devices in a multiprocessor system without taking immediate effect. When RUN-TEST/IDLE state is subsequently entered, all the processors resume operation simultaneously.

The function of **DBGACK** is to tell the rest of the system when the ARM940T is in debug state. This can be used to inhibit peripherals such as watchdog timers that have real time characteristics. **DBGACK** can also be used to mask out memory accesses that are caused by the debugging process. For example, when the ARM940T enters debug state after a breakpoint, the instruction pipeline contains the breakpointed instruction plus two other instructions that have been prefetched. On entry to debug state, the pipeline is flushed. On exit from debug state, the pipeline must then be refilled to its previous state. Because of the debugging process, more memory accesses occur than normally expected. Any system peripheral that might be sensitive to the number of memory accesses can be inhibited with **DBGACK**.

Note

DBGACK can only be used in such a way using breakpoints. It does not mask the correct number of memory accesses after a watchpoint.

For example, consider a peripheral that counts the number of instruction fetches. This device must return the same answer after a program has run both with or without debugging.

Figure 8-7 on page 8-34 shows the behavior of the ARM940T on exit from debug state.

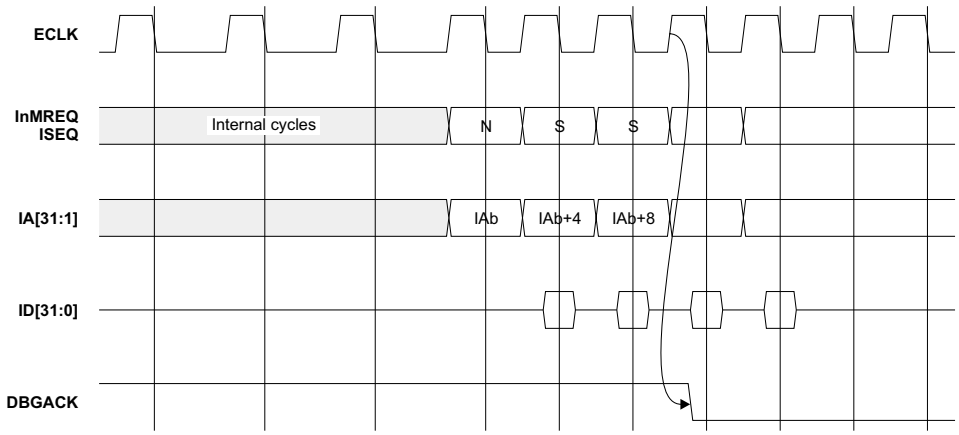

Figure 8-7 Debug exit sequence

Figure 8-8 on page 8-34 shows that two instructions are fetched after the one that breakpoints. **DBGACK** masks the first three instruction fetches out of debug state, corresponding to the breakpoint instruction and the two instructions prefetched after it.

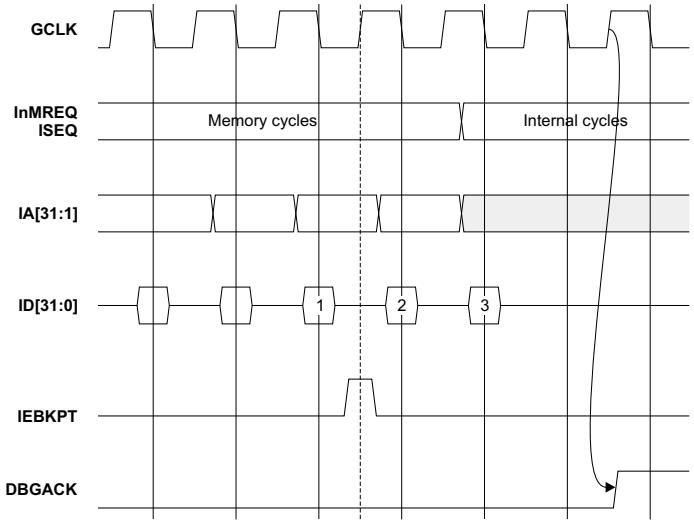

Figure 8-8 Debug state entry

Note

When a system speed access occurs **DBGACK** remains HIGH, masking any memory access.

8.10 The behavior of the program counter during debug

To force the ARM940T to branch back to the place where program flow has been interrupted by debug, the debugger must keep track of what happens to the PC. There are six cases:

- *Breakpoint* on page 8-36
- *Watchpoint* on page 8-36
- *Watchpoint with another exception* on page 8-37
- *Watchpoint and breakpoint* on page 8-37
- *Debug request* on page 8-37
- *System speed accesses* on page 8-38.

These cases are described below. In each case the same calculation is used to determine where to resume execution. This is explained in *Summary of return address calculations* on page 8-38.

8.10.1 Breakpoint

Entry to debug state from a breakpointed instruction advances the PC by 16 bytes in ARM state, or 8 bytes in Thumb state. Each instruction executed in debug state advances the PC by one address. The normal way to exit from debug state after a breakpoint is to remove the breakpoint, and branch back to the previously breakpointed address.

For example, if the ARM940T entered debug state from a breakpoint set on a given address and two debug speed instructions were executed, a branch of minus 7 addresses must occur (four for debug entry, plus two for the instructions, plus one for the final branch). The following sequence shows ARM instructions scanned into scan chain 1. This is MSB first, and so the first digit represents the value to be scanned into the SYSSPEED bit, followed by the instruction:

```
0 EAFFFFF9      ; B -7 addresses (two's complement)
1 E1A00000      ; NOP (MOV r0, r0), SYSSPEED bit is set
```

For small branches, the final branch can be replaced with a subtract having the PC as the destination (`SUB PC,PC,#28` for ARM code in the above example).

8.10.2 Watchpoint

Returning to the program execution after entering debug state from a watchpoint is done in the same way as the procedure described in *Breakpoint*. Debug entry adds four addresses to the PC, and every instruction adds one address. Because the instruction after the one that caused the watchpoint has executed, instruction execution resumes at the one after that.

8.10.3 Watchpoint with another exception

If a watchpoint access simultaneously causes a Data Abort, the ARM940T enters debug state in abort mode. Entry into debug is held off until the core has changed into abort mode, and fetched the instruction from the abort vector.

A similar sequence is followed when an interrupt, or any other exception, occurs during a watchpointed memory access. The ARM940T enters debug state in the same mode as the exception, and the debugger must check that this happens by looking at the current and previous mode (in the CPSR and SPSR), and the value of the PC. If an exception takes place, you must be able to choose to service the exception before debugging.

For example, suppose an abort occurred on a watchpoint access and ten instructions had been executed to determine this, the following sequence can be used to return program execution:

```
0 EAFFFFFC      ; B -15 addresses (two's complement)
1 E1A00000      ; NOP (MOV r0, r0), SYSSPEED bit is set
```

This forces a branch back to the abort vector, causing the instructions at that location to be refetched and executed. After the abort service routine, the instruction that caused the abort and watchpoint is re-executed. This causes the watchpoint to be generated and the ARM940T enters debug state again.

8.10.4 Watchpoint and breakpoint

It is possible to have a watchpoint and breakpoint condition occurring simultaneously. This can happen when the instruction causes a watchpoint, and the following instruction has been breakpointed. The same calculation must be performed as for breakpoint (see *Breakpoint* on page 8-36) to determine the address to resume at. In this case, it is at the breakpoint instruction because this has not been executed.

8.10.5 Debug request

Entry into debug state from a debug request is similar to a breakpoint and, as for breakpoint entry to debug state, adds four addresses to the PC. Every instruction executed in debug state adds one.

For example, the following sequence handles a situation where you have invoked a debug request, and decided to return to program execution immediately:

```
0 EAFFFFFD      ; B -5 addresses (2's complement)
1 E1A00000      ; NOP (MOV r0, r0), SYSSPEED bit is set
```

This restores the PC, and restarts the program from the next instruction.

8.10.6 System speed accesses

If a system speed access is performed during debug state, the value of the PC is increased by five addresses. Because system speed instructions access the memory system, it is possible for aborts to take place. If an abort occurs during a system speed memory access, the ARM940T enters abort mode before returning to debug state.

This is similar to an aborted watchpoint. However, this occurrence is more difficult to resolve, because the abort is not caused by an instruction in the main program, and the PC does not point to the instruction that caused the abort. An abort handler usually looks at the PC to determine the instruction that caused the abort, and so the abort address. In this case, the value of the PC is invalid, but the debugger knows the address of the location that is being accessed. Therefore, the debugger can be written to help the abort handler fix the memory system.

8.10.7 Summary of return address calculations

The calculation of the branch return address can be summarized as:

-(4 + N +5S)

where:

N Is the number of debug speed instructions executed (including the final branch).

S Is the number of system speed instructions executed.

8.11 EmbeddedICE unit

The EmbeddedICE unit is an integral component of the ARM9TDMI processor core. It has two hardware breakpoint/watchpoint units that can be configured to monitor either the instruction memory interface or the data memory interface. Each watchpoint unit has a value and mask register, with an address, data and control field. The general architecture of the EmbeddedICE unit is shown in Figure 8-9 on page 8-39.

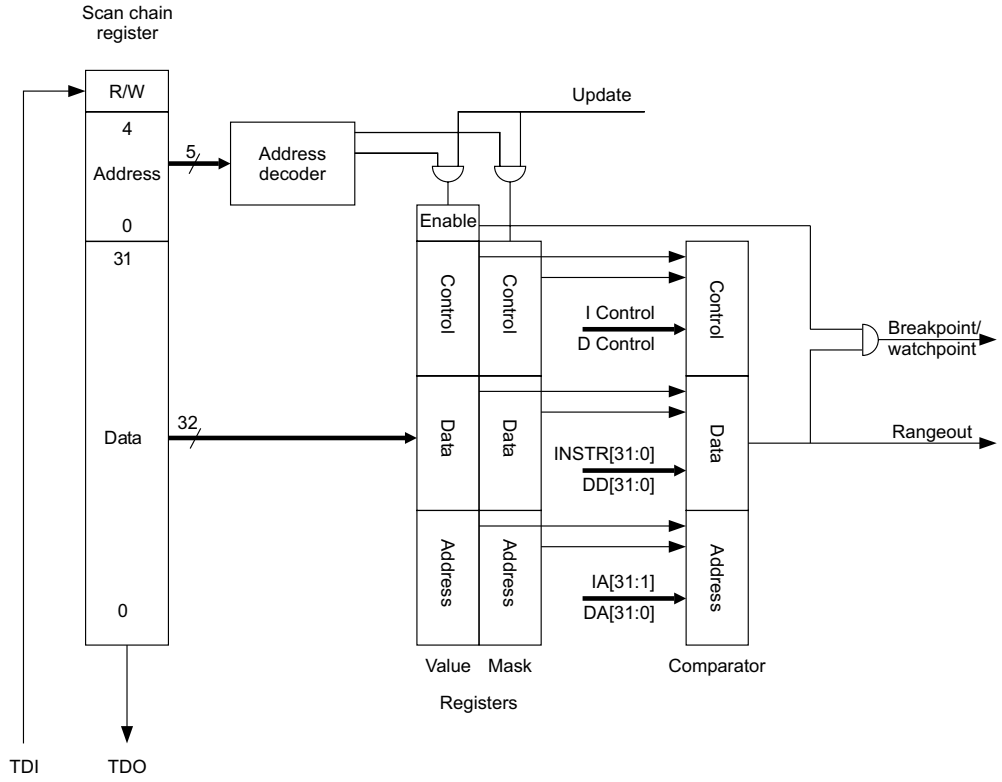

Figure 8-9 ARM940T EmbeddedICE unit

The ARM940T EmbeddedICE unit has logic that allows single-stepping through code. This reduces the work required by an external debugger, and removes the requirement to flush the instruction cache. There is also hardware to allow efficient trapping of accesses to the exception vectors. These blocks of logic free the two general-purpose hardware breakpoint/watchpoint units for use by the programmer or debugger.

Because the ARM9TDMI processor core has a Harvard architecture, you must specify if the watchpoint registers must examine the instruction memory interface or the data memory interface. This is specified by bit 3 in the control field of the watchpoint register, as follows:

- when bit 3 is set, the data must be examined
- when bit 3 is clear, the instruction must be examined.

There must not be a *don't care* case for this bit because the comparators cannot compare the values on both interfaces simultaneously. Therefore, bit 3 of the control mask registers is always clear and cannot be programmed HIGH. Bit 3 also determines if the **IBREAKPT** or **DBREAKPT** signal must be driven by the result of the comparison, as shown in Figure 8-9 on page 8-39.

8.11.1 Register map

The EmbeddedICE unit register map is shown in Table 8-10 on page 8-40.

Table 8-10 ARM940T EmbeddedICE unit register map

Address	Width	Function
0b00000	4	Debug control
0b00001	5	Debug status
0b00010	8	Vector catch control
0b00100	6	Debug comms control
0b00101	32	Debug comms data
0b01000	32	Watchpoint 0 address value
0b01001	32	Watchpoint 0 address mask
0b01010	32	Watchpoint 0 data value
0b01011	32	Watchpoint 0 data mask
0b01100	9	Watchpoint 0 control value
0b01101	8	Watchpoint 0 control mask
0b10000	32	Watchpoint 1 address value
0b10001	32	Watchpoint 1 address mask
0b10010	32	Watchpoint 1 data value

Table 8-10 ARM940T EmbeddedICE unit register map (continued)

Address	Width	Function
0b10011	32	Watchpoint 1 data mask
0b10100	9	Watchpoint 1 control value
0b10101	8	Watchpoint 1 control mask

8.11.2 Using the mask register

For each value register, there is an associated mask register in the same format. Setting a bit to 1 in the mask register causes the corresponding bit in the value register to be ignored in any comparison.

For example, if a watchpoint is requested at a particular memory location but the data value is irrelevant, the data mask register can be programmed to `0XFFFFFFFF` (all bits set to 1) so that the entire data bus is ignored.

8.11.3 Control registers

The format of the control registers depends on how bit 3 is programmed.

Bit 3 programmed to 1

The breakpoint comparators examine the data address, data, and control signals. In this case, the format of the register is as shown in Figure 8-10 on page 8-41.

Note

Bit 8 and bit 3 cannot be masked.

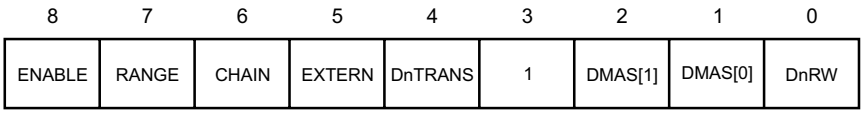

Figure 8-10 Watchpoint control register for data comparison

The control register bits have the functions for data comparison shown in Table 8-11 on page 8-42.

Table 8-11 Watchpoint control register for data comparison

Bit	Function
DnRW	Compares with the data not read/write signal from the core to detect the direction of the data bus activity. **nRW** is 0 for a read, and 1 for a write.
DMAS[1:0]	Compares with the **DMAS[1:0]** signal from the core to detect the size of the data bus activity.
DnTRANS	Compares with the data not translate signal from the core to determine between a User mode (**DnTRANS** = 0) data transfer, and a privileged mode (**DnTRANS** = 1) transfer.
EXTERN	Is an external input into the EmbeddedICE unit that allows the watchpoint to be dependent on some external condition. The **EXTERN** input for watchpoint 0 is labeled **EXTERN0**, and the **EXTERN** input for watchpoint 1 is labeled **EXTERN1**.
CHAIN	Can be connected to the **CHAIN** output of another watchpoint to implement, for example, debugger requests of the form *breakpoint on address YYY only when in process XXX*. In the ARM940T EmbeddedICE unit, the **CHAINOUT** output of watchpoint 1 is connected to the **CHAIN** input of watchpoint 0. The **CHAINOUT** output is derived from a latch. The address/control field comparator drives the write enable for the latch and the input to the latch is the value of the data field comparator. The **CHAINOUT** latch is cleared when the control value register is written or when **nTRST** is LOW.
RANGE	Can be connected to the **RANGE** output of another watchpoint register. In the ARM940T EmbeddedICE unit, the **RANGEOUT** output of watchpoint 1 is connected to the **RANGE** input of watchpoint 0. This allows two watchpoints to be coupled for detecting conditions that occur simultaneously, for example, for range-checking.
ENABLE	If a watchpoint match occurs, the **IBREAKPT** or **DBREAKPT** signal is only asserted when the ENABLE bit is set. This bit only exists in the value register. It cannot be masked.

Bit 3 programmed to 0

If bit 3 of the control register is programmed to 0, the comparators examine the instruction address, instruction data, and instruction control buses. Bits [1:0] of the mask register must be set to *don't care* (programmed to 11). The format of the register in this case is as shown in Figure 8-11 on page 8-43.

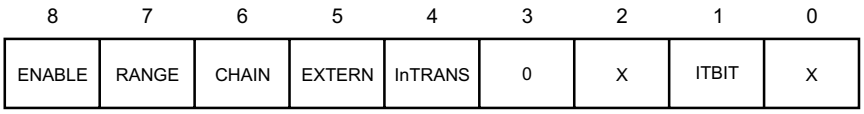

Figure 8-11 Watchpoint control register for instruction comparison

The control register bits have the functions shown in Table 8-12 on page 8-43 for instruction comparison.

Table 8-12 Watchpoint control register for instruction comparison

Bit	Function
ITBIT	Compares against the Thumb state signal from the core to determine between a Thumb (**ITBIT** = 1) instruction fetch or an ARM (**ITBIT** = 0) fetch.
InTRANS	Compares against the not translate signal from the core to determine between a User mode (**InTRANS** = 0) instruction fetch, and a privileged mode (**InTRANS** = 1) fetch.
EXTERN	Is an external input into the EmbeddedICE unit that allows the watchpoint to be dependent on some external condition. The **EXTERN** input for watchpoint 0 is labelled **EXTERN0**, and the **EXTERN** input for watchpoint 1 is labeled **EXTERN1**.
CHAIN	Can be connected to **CHAIN** output of another watchpoint to implement, for example, debugger requests of the form *breakpoint on address YYY only when in process XXX*. In the ARM940T EmbeddedICE unit, the **CHAINOUT** output of watchpoint 1 is connected to the **CHAIN** input of watchpoint 0. The **CHAINOUT** output is derived from a latch. The address/control field comparator drives the write enable for the latch, and the input to the latch is the value of the data field comparator. The **CHAINOUT** latch is cleared when the control value register is written, or when **nTRST** is LOW.
RANGE	Can be connected to the range output of another watchpoint register. In the ARM940T EmbeddedICE unit, the **RANGEOUT** output of watchpoint 1 is connected to the **RANGE** input of watchpoint 0. This allows two watchpoints to be coupled for detecting conditions that occur simultaneously, for example, for range-checking.
ENABLE	If a watchpoint match occurs, the **IBREAKPT** or **DBREAKPT** signal is only asserted when the ENABLE bit is set. This bit only exists in the value register. It cannot be masked.

8.11.4 Debug control register

The ARM940T debug control register is four bits wide and is shown in Figure 8-12 on page 8-44. Bit 3 controls the single-step hardware. This is explained in more detail in *Single-stepping* on page 8-47.

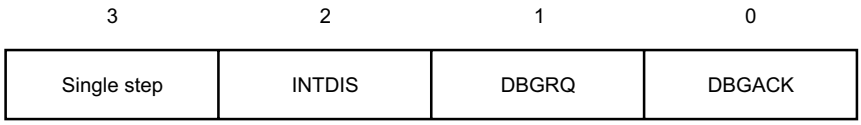

Figure 8-12 Debug control register

8.11.5 Debug status register

The debug status register is five bits wide. If it is accessed for a write (with the read/write bit set HIGH), the status bits are written. If it is accessed for a read (with the read/write bit LOW), the status bits are read. The format of the debug status register is shown in Figure 8-13 on page 8-44.

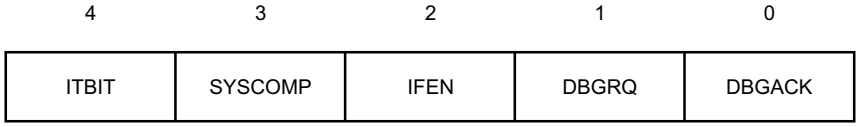

Figure 8-13 Debug status register

The function of each bit in this register is as follows:

Bits 1 and 0 Allow the values on the synchronized versions of **DBGRQ** and **DBGACK** to be read.

Bit 2 Allows the state of the core interrupt enable signal (**IFEN**) to be read. Because the capture clock for the scan chain can be asynchronous to the processor clock, the **DBGACK** output from the core is synchronized before being used to generate the **IFEN** status bit.

Bit 3 Allows the state of the **SYCOMP** signal from the core (synchronized to **TCK**) to be read. This allows the debugger to determine that a memory access from the debug state has completed.

Bit 4 Allows **ITBIT** to be read. This enables the debugger to determine what state the processor is in, and the instructions to execute.

8.11.6 Vector catch register

The ARM940T EmbeddedICE unit controls logic to enable accesses to the exception vectors to be trapped in an efficient manner. This is controlled by the vector catch register, as shown in Figure 8-14 on page 8-45. The functionality is described in *Vector catching* on page 8-46.

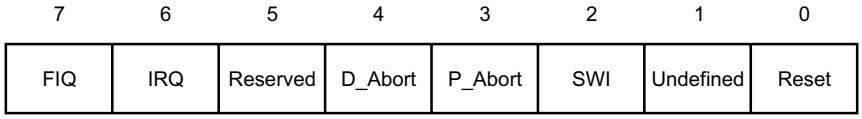

Figure 8-14 Vector catch register

8.12 Vector catching

The ARM940T EmbeddedICE unit contains logic that allows efficient trapping of fetches from the vectors during exceptions. This is controlled by the vector catch register. If one of the bits in this register is set HIGH and the corresponding exception occurs, the processor enters debug state as if a breakpoint has been set on an instruction fetch from the relevant exception vector.

For example, if the processor executes a `SWI` instruction while bit 2 of the vector catch register is set, the ARM940T fetches an instruction from location `0x8`. The vector catch hardware detects this access and forces the breakpoint signal HIGH into the ARM940T control logic. This, in turn, forces the ARM940T to enter debug state.

The behavior of this hardware is independent of the watchpoint comparators, leaving them free for general use. The vector catch register is sensitive only to fetches from the vectors during exception entry. Therefore, if code branches to an address within the vectors during normal operation, and the corresponding bit in the vector catch register is set, the processor is not forced to enter debug state.

8.13 Single-stepping

The ARM940T EmbeddedICE unit contains logic that allows efficient single-stepping through code. This leaves the hardware watchpoint comparators free for general use.

This function is enabled by setting bit 3 of the debug control register. The state of this bit must only be altered while the processor is in debug state. If the processor exits debug state and this bit is HIGH, the processor fetches an instruction, executes it, and then immediately re-enters debug state. This happens independently of the watchpoint comparators. If a system speed data access is performed while in debug state, the debugger must ensure that the control bit is clear first.

8.14 Debug communications channel

The ARM940T EmbeddedICE unit contains a communication channel for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The communications channel consists of:

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

These registers are placed in fixed locations in the EmbeddedICE unit register map (as shown in Figure 8-1 on page 8-3) and are accessed from the processor using `MCR` and `MRC` instructions to coprocessor 14.

8.14.1 Debug comms channel registers

The debug comms control register is read only. It controls synchronized handshaking between the processor and the debugger. The format of the debug comms channel registers is shown in Figure 8-15 on page 8-48.

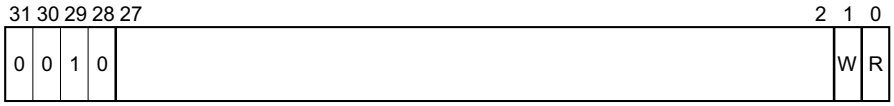

Figure 8-15 Debug comms control register

The function of each register bit is described below:

Bits 31:28 Contain a fixed pattern that denotes the EmbeddedICE unit version number, in this case 0010.

Bits 27:2 Unused.

Bit 1 Denotes if the comms data write register is free, as seen by the processor. If, from the point of view of the processor, the comms data write register is free (W=0), new data can be written. If it is not free (W=1), the processor must poll until W=0. If, from the point of view of the debugger, W=1, some new data has been written that can then be scanned out.

Bit 0 Denotes if there is some new data in the comms data read register. If, from the point of view of the processor, R=1, there is some new data that can be read using an `MRC` instruction. If, from the point of view of the debugger, R=0, the comms data read register is free and new data can be placed there through the scan chain. If R=1, data previously placed there through the scan chain has not been collected by the processor, and so the debugger must wait.

From the point of view of the debugger, the registers are accessed using the scan chain in the usual way. From the processor, these registers are accessed using coprocessor register transfer instructions. You can use the following instructions:

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

Returns the debug comms control register into Rd.

```
MCR p14, 0, Rn, c1, c0
```

Writes the value in Rn to the comms data write register.

```
MRC p14, 0, Rd, c1, c0
```

Returns the debug data read register into Rd.

Note

The Thumb instruction set does not support coprocessor instructions (to access the debug comms channel, the core must be in ARM state).

8.14.2 Communication using the comms channel

Communication can take place over the debug comms channel by either an interrupt driven mechanism or through software polling.

The interrupt driven mechanism requires the **COMMTX** and **COMMRX** signals to be factored into an interrupt controller. The comms channel is only accessed therefore, when the write channel has become free or the read channel has received data, allowing efficient communication.

Software polling requires no external hardware configuration. The program must examine the debug comms control register to determine if data has been received or if the write channel has become empty. Only when such an event has occurred can the debug comms write or read register be accessed.

8.14.3 Software polling communication

Software polling communication is achieved by the processor sending messages to, and receiving messages from, the debugger.

Sending a message to the debugger

Before sending a message to the debugger, the processor must first check that the comms data write register is available by polling the W bit. The checking process is as follows:

- If the W bit is set, previously written data has not been read by the debugger.

  The processor must continue to poll the control register until the W bit is clear.
- If W bit is clear, the comms data write register is available.

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 comms data write register, the W bit is set in the debug comms control register.

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

Receiving a message from the debugger

Message transfer 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 comms control register:

- if the R bit is LOW, the 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 has not yet been collected, so the debugger must wait.

When the comms 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 comms control register.

When the processor polls this register, it sees a **GCLK** synchronized version. If the R bit is set, there is data waiting to be collected. This data can be read using an `MRC` instruction to coprocessor 14. The action of this load clears the R bit in the debug comms control register. When the debugger polls this register and sees that the R bit is clear, the data has been taken, and the process can be repeated.

8.14.4 Interrupt driven communications

To implement interrupt driven communication, the signals **COMMRX** and **COMMTX** must be factored into any interrupt controller being used. If no interrupt controller is being used, the signals can be applied to a NOR gate with the output driving **nIRQ**.

When an interrupt occurs, the program must examine the debug comms control register to determine if an event occurred. If the W bit is clear, new data can be written into the debug comms write register. If the R bit is set, new data has been received and can be read.

If the W bit is set and the R bit clear, the debug comms channel is not the source of the interrupt.

8.15 The debugger view of the cache

The debugger can access the caches and must be able to control them. This means that the debugger must also access CP15 registers.

8.15.1 Scan access to the CP15 registers

When in debug state, the debugger is able to see the state of the memory system, including the caches. The debugger has to be able to control the cache, consequently all of CP15 registers are accessible through the scan chain. Scan chain 15 is reserved for this use. This scan chain is 39 bits long, and has a structure similar to the EmbeddedICE unit scan chain 2. The format of scan chain 15 is shown in Table 8-13 on page 8-52. An access using this scan chain allows any of the CP15 registers to be read or written.

Table 8-13 Scan chain 15 format

Scan chain bit	Function
38	R/W (Write=1)
37:32	Register address
31:0	Register value

On entry to debug state, the debugger must extract and save the state of CP15. It is advisable to switch off the cache to prevent any debug accesses to memory from altering the state of the caches. The mapping of the 6-bit address field to the CP15 register is as shown in Table 8-14 on page 8-52. For CP register 6, `CRm` corresponds to the region number.

Table 8-14 Scan access mapping to CP15 register

Register address			CP15 register
37	**36:33**	**32**	
0	0000	0	0
0	0001	0	1
0	0010	0	2 (Data)
0	0010	1	2 (Instruction)
0	0011	0	3
0	0101	0	5 (Data)

Table 8-14 Scan access mapping to CP15 register (continued)

Register address			CP15 register
37	**36:33**	**32**	
0	0101	1	5 (Instruction)
0	1001	0	9 (Data)
0	1001	1	9 (Instruction)
0	1111	0	15
1	<CRm>	0	6 (Data)
1	<CRm>	1	6 (Instruction)

In addition, the flush ICache command implemented in CP15 register 7 can be performed from the scan chain.

Flushing ICache during debug

It is possible for the debugger to flush the ICache without leaving debug state. Through scan chain 15, the debugger can perform a write to a pseudo register that causes the instruction cache to be flushed. This allows full debug without reliance on system resources. The bit pattern given in Table 8-15 on page 8-54 must be scanned into scan chain 15.

Note

In an earlier silicon version (Rev 0) it is impossible for the debugger to flush the instruction cache using the scan interface while the processor is in debug state. This means that whenever code is downloaded, the processor has to force a cache flush by downloading a small flush routine into memory. The routine is then executed in system state before continuing. The disadvantage of this method is that the debugger has to rely on system resources.

Table 8-15 Flush I-Cache

Register address				CP15 register
38	**37**	**36:33**	**32**	
1	0	0111	1	Flush ICache
0	0	0111	1	ICache flush status

Because of the asynchronous relationship between **TCK** and **GCLK**, the debugger must check that the flush has occurred by performing a read from the same register. If bit 31 of the returned data is clear, then the cache flush has happened. If bit 31 is set, then the debugger must continue to poll the register until bit 31 is clear. This is only necessary if **TCK** is significantly faster than **FCLK**.

8.15.2 Scan access to the caches

The content of the caches is determined by:

1. Extracting the contents of the CAMs.
2. Determining the contents of the RAMs using a system speed `LDR`.

The CAM arrays are read using scan chain 4 for the ICache, and scan chain 5 for the DCache. The format of these scan chains is identical and has two modes:

Addressing The CAM index and segment are specified. The format of the scan chain is as shown in Table 8-16 on page 8-55.

Reading The contents of the CAM entry are read back. The format of the data read back is shown in Table 8-17 on page 8-55. When the ICache CAM is read, the dirty bit is always read as zero.

The addressing mode format, shown in Table 8-16 on page 8-55, is used when scanning in data to address the CAM. After UPDATE-DR, the data read from the CAM array is in the reading mode format, shown in Table 8-17 on page 8-55.

Table 8-16 Scan chain 4 and 5 addressing mode

Scan chain bit	Write function
28:22	CAM index
21:6	Should be zero
5:0	Segment

Table 8-17 Scan chains 4 and 5 reading mode

Scan chain bit	Write function
28	Valid
27	Dirty
26:0	TAG

The debugger must index through all the entries in the CAM (0-63) to determine the 27-bit TAG addresses. When this information is extracted, the contents of the cache RAM array can be determined.

DCache

This is achieved by taking each TAG address, padding the bottom four bits with zeros, setting bits 5 and 6 to indicate the same segment that the TAG is scanned from, and performing a system-speed four-word `LDM` to that address, with the cache switched on. Because the TAG address is known, a cache hit occurs, and the four words in the RAM line are returned.

If a system-speed access from a TAG address is performed with the cache switched off, the external data corresponding to that address is returned. For cache lines that are marked as valid and dirty therefore, it is possible to determine the value of the cached data and the external data in main memory.

ICache

For the ICache, the system speed `LDM` must be performed with the DCache switched off. This ensures that the external memory system is accessed. Because it is impossible for the core to change the data in the instruction cache, the ICache and external memory are guaranteed coherent. However, if self-modifying code has been produced and the ICache has not been flushed, the ICache might contain an out of date copy of the external memory code.

Chapter 9
TrackingICE

This chapter describes how TrackingICE mode is used by the ARM940T. It contains the following sections:

9.1 About TrackingICE

The principle of TrackingICE is described in *ARM Technical Note AN41, TrackingICE*. The ARM940T can be switched into a mode that assists in producing a TrackingICE system.

When in TrackingICE mode, a number of the ARM940T output signals are configured to mimic the inputs to the embedded ARM9TDMI processor core. These signals can be connected to a second external ARM9TDMI to precisely monitor (or track) the inputs to the embedded processor core. The outputs from the test chip are latched copies of the outputs from the processor core that can be more easily monitored.

Figure 9-1 on page 9-2 gives an overview of how a tracking ARM9TDMI is attached to an ARM940T.

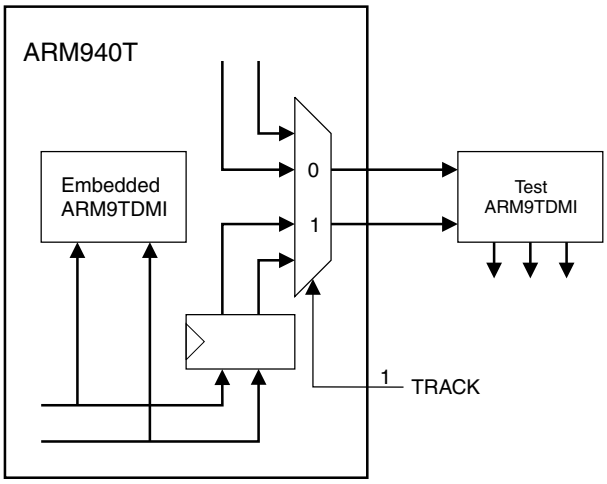

Figure 9-1 Using TrackingICE

The tracking ARM9TDMI operates one clock phase behind the actual ARM9TDMI (on the inverted clock). All required inputs to the ARM9TDMI are latched inside the ARM940T and are then brought out on various pins. The tracking ARM9TDMI can be directly attached to these outputs.

9.2 Timing requirements

To enable the ARM9TDMI processor core to be tracked correctly, all inputs must be synchronous to the ARM9TDMI processor clock. These inputs include **TCK**, which in tracking mode is latched on the falling edge of **GCLK** before it is driven onto the ARM940T tracking outputs. All other **TCK** relative signals, **TDI**, **TMS**, and **SDOUTBS**, are latched on rising **GCLK** before they are driven onto the ARM940T tracking outputs.

9.3 TrackingICE outputs

The ARM940T outputs shown in Table 9-1 on page 9-4 are re-used when the ARM940T is in TrackingICE mode.

Table 9-1 ARM940T in TrackingICE

ARM940T output	Attach to tracking ARM9TDMI input
IR[3:2]	CHSE[1:0]
IR[1:0]	CHSD[1:0]
SCREG[4]	nIRQ
SCREG[3]	nFIQ
SCREG[2]	DABORT
SCREG[1]	IABORT
TAPSM[3]	EXTERN1
TAPSM[2]	EXTERN0
TAPSM[1]	DEWPT
TAPSM[0]	IEBKPT
ICAPCLKBS	HIVECS
ECAPCLKBS	EDBGGQ
PCLKBS	nWAIT
RSTCLKBS	nRESET
SHCLK1BS	TDI
SHCLK2BS	TMS
TCK1	GCLK
TCK2	TCK
SDIN	SDOUTBS

The remaining input connections to the ARM9TDMI are:

- **ID** bus attaches to the **CPID** bus
- **DD** bus attaches to the **CPDOUT** bus
- **BIGEND** input attaches to the **BIGENDOUT**.

These can still be attached to a coprocessor when the ARM940T is in tracking mode. The only difference in behavior is that **CPDOUT** mirrors the ARM940T **DD** bus on every cycle, not only for coprocessor data transfers. The following conditions apply:

- The **ISYNC** and **nTRST** inputs must be common between the ARM940T and the tracking ARM9TDMI.
- **IABE** and **DABE** must be HIGH so that the address outputs of the tracking ARM9TDMI can be observed.
- **DDBE** must be LOW to prevent a drive clash on the bidirectional **DD** bus. It is not necessary for the tracking ARM9TDMI to drive the **DD** bus because **CPDOUT** is driven with the data from all memory access cycles.

Chapter 10
Test Support

This chapter describes the test support for the ARM940T and lists the scan chain 0 bit order. It contains the following sections:

10.1 About test support

The ARM940T test support comprises support for testing:

- *ARM9TDMI* on page 10-2
- *ARM940T macrocell* on page 10-2.

10.1.1 ARM9TDMI

The ARM9TDMI processor core has a fully JTAG-compatible scan chain that intersects all the inputs and outputs. This allows test patterns to be serialized and injected to test the ARM9TDMI core processor using the ARM940T JTAG interface.

10.1.2 ARM940T macrocell

The ARM940T supports parallel and AMBA test. The parallel test patterns are derived from assembler ARM code programs written to achieve a high fault coverage.

To test the ARM940T macrocell in an embedded system, the AMBA Test Methodology has been adopted. This allows the ARM940T to be tested in isolation from the rest of the system. The methodology applies the external drive of 32-bit parallel vectors onto the system data bus and their conversion to bus transfers. A low gate-count *Test Interface Controller* (TIC) bus master is required in the system to perform this function. The AMBA test methodology is illustrated in Figure 10-1 on page 10-2.

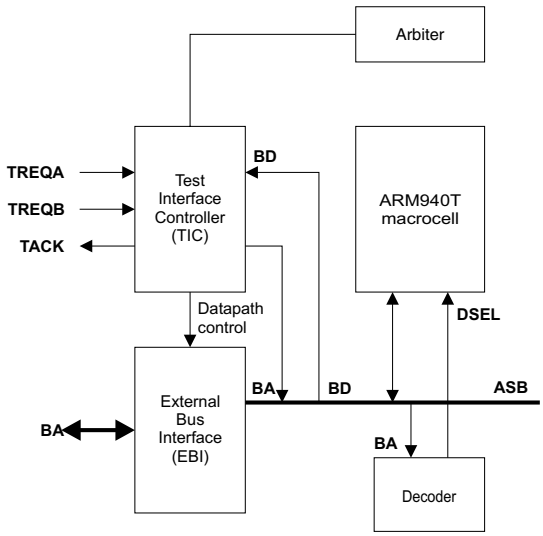

Figure 10-1 AMBA test methodology

The TIC uses a minimal three-wire handshake mechanism to control the application of test vectors. In AMBA test mode, 32 bidirectional pins are required to transfer the test data. These can be multiplexed for other uses in the normal operating mode of the chip. The datapath of an *external bus interface* (EBI) or external memory interface is commonly used to provide the 32-bit vector interface.

To support this method of test vector application, the ARM940T macrocell incorporates dedicated test logic that allows the internal inputs to be driven and the internal outputs to be monitored using transfers from the bus. The ARM940T is configured to act as an AMBA slave during test mode because the test transfers are initiated by the TIC. A simplified view of the test harness incorporated within the ARM940T is shown in Figure 10-2 on page 10-3.

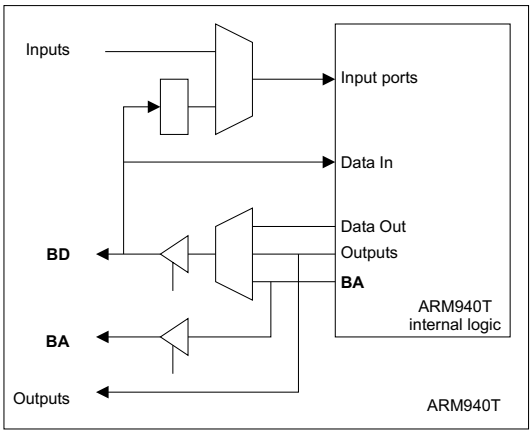

Figure 10-2 ARM940T integrated test harness

The ARM940T AMBA test logic also provides a high level of controllability over the caches. This allows you to test the caches independently of the ARM9TDMI processor core. The TAG address in the CAM array, the CAM valid and dirty bits, and the RAM data can all be read and written in the AMBA Cache Test mode.

The ARM940T macrocell is supplied with a full test vector suite that achieves a high node toggle coverage. It is, therefore, not necessary for the system designer to write test patterns for the macrocell but to provide a suitable environment in which the supplied test vectors can be applied.

You must use system-level functional vectors to test connections between ARM940T and other logic on the chip.

10.2 Scan chain 0 bit order

Table 10-1 on page 10-4 shows the scan chain 0 bit order.

Table 10-1 Scan chain 0 bit order

Number	Signal	Direction
1	**ID[0]**	Input
2	**ID[1]**	Input
3:31	**ID[2:30]**	Input
32	**ID[31]**	Input
33	**SYSSPEED**	Internal
34	**WPTANDBKPT**	Internal
35	**DDEN**	Output
36	**DD[31]**	Bidirectional
37	**DD[30]**	Bidirectional
38:66	**DD[29:1]**	Bidirectional
67	**DD[0]**	Bidirectional
68	**DA[31]**	Output
69	**DA[30]**	Output
70:98	**DA[29:1]**	Output
99	**DA[0]**	Output
100	**IA[31]**	Output
101	**IA[30]**	Output
102:129	**IA[29:2]**	Output
130	**IA[1]**	Output
131	**IEBKPT**	Input
132	**DEWPT**	Input
133	**EDBGRQ**	Input
134	**EXTERN0**	Input

Table 10-1 Scan chain 0 bit order (continued)

Number	Signal	Direction
135	**EXTERN1**	Input
136	**COMMRX**	Output
137	**COMMTX**	Output
138	**DBGACK**	Output
139	**RANGEOUT0**	Output
140	**RANGEOUT1**	Output
141	**DBGRQI**	Output
142	**DDBE**	Input
143	**InMREQ**	Output
144	**DnMREQ**	Output
145	**DnRW**	Output
146	**DMAS[1]**	Output
147	**DMAS[0]**	Output
148	**PASS**	Output
149	**LATECANCEL**	Output
150	**ITBIT**	Output
151	**InTRANS**	Output
152	**DnTRANS**	Output
153	**nRESET**	Input
154	**nWAIT**	Input
155	**IABORT**	Input
156	**IABE**	Input
157	**DABORT**	Input
158	**DABE**	Input
159	**nFIQ**	Input

Table 10-1 Scan chain 0 bit order (continued)

Number	Signal	Direction
160	**nIRQ**	Input
161	**ISYNC**	Input
162	**BIGEND**	Input
163	**HIVECS**	Input
164	**CHSD[1]**	Input
165	**CHSD[0]**	Input
166	**CHSE[1]**	Input
167	**CHSE[0]**	Input
168	**UNIEN**	Input
169	**ISEQ**	Output
170	**InM[4]**	Output
171	**InM[3]**	Output
172	**InM[2]**	Output
173	**InM[1]**	Output
174	**InM[0]**	Output
175	**DnM[4]**	Output
176	**DnM[3]**	Output
177	**DnM[2]**	Output
178	**DnM[1]**	Output
179	**DnM[0]**	Output
180	**DSEQ**	Output
181	**DMORE**	Output
182	**DLOCK**	Output
183	**ECLK**	Output

Chapter 11
Instruction Cycle Summary and Interlocks

This chapter gives the instruction cycle times and shows the timing diagrams for interlock timing. It contains the following sections:

11.1 About the instruction cycle summary

All signals quoted are ARM9TDMI signals and are internal to the ARM940T. In all cases it is assumed that all accesses are from cached regions of memory.

If an instruction causes an external access, either when prefetching instructions or when accessing data, the instruction takes more cycles to complete execution. The additional number of cycles is dependent on the system implementation.

11.2 Instruction cycle times

Table 11-1 on page 11-3 provides a key to the other tables in this chapter.

Table 11-1 Symbols used in tables

Symbol	Meaning
b	The number of busy-wait states during coprocessor accesses
m	Is in the range 1 to 4, depending on early termination (see Figure 11-1 on page 11-7)
n	The number of words transferred in an LDM/STM/LDC/STC
C	Coprocessor register transfer (C-cycle)
I	Internal cycle (I-cycle)
N	Non-sequential cycle (N-cycle)
S	Sequential cycle (S-cycle)

Table 11-2 on page 11-3 summarizes the ARM940T instruction cycle counts and bus activity when executing the ARM instruction set.

Table 11-2 Instruction cycle bus times

Instruction	Cycles	Instruction bus	Data bus	Comment
Data Op	1	1S	1I	Normal case, PC not destination
Data Op	2	1S+1I	2I	With register controlled shift, PC not destination
Data Op	3	2S + 1N	3I	PC destination register
Data Op	4	2S + 1N + 1I	4I	With register controlled shift, PC destination register
LDR	1	1S	1N	Normal case, not loading PC
LDR	2	1S+1I	1N+1I	Not loading PC and following instruction uses loaded word (1 cycle load-use interlock)
LDR	3	1S+2I	1N+2I	Loaded byte, half-word, or unaligned word used by following instruction (2 cycle load-use interlock)
LDR	5	2S+2I+1N	1N+4I	PC is destination register
STR	1	1S	1N	All cases

Table 11-2 Instruction cycle bus times (continued)

Instruction	Cycles	Instruction bus	Data bus	Comment
LDM	2	1S+1I	1S+1I	Loading 1 Register, not the PC
LDM	n	1S+(n-1)I	1N+(n-1)S	Loading n registers, n > 1, not loading the PC
LDM	n+4	2S+1N+(n+1)I	1N+(n-1)S+4I	Loading n registers including the PC, n > 0
STM	2	1S+1I	1N+1I	Storing 1 Register
STM	n	1S+(n-1)I	1N+(n-1)S	Storing n registers, n > 1
SWP	2	1S+1I	2N	Normal case
SWP	3	1S+2I	2N+1I	Loaded word used by following instruction
SWPB	3	1S+2I	2N+1I	All cases
B, BL, BX	3	2S+1N	3I	All cases
SWI, Undefined	3	2S+1N	3I	All cases
CDP	b+1	1S+bI	(1+b)I	All cases
LDC, STC	b+n	1S+(b+n-1)I	bI+1N+(n-1)S	All cases
MCR	b+1	1S+bI	bI+1C	All cases
MRC	b+1	1S+bI	bI+1C	Normal case
MRC	b+2	1S+(b+1)I	(b+I)I+1C	Following instruction uses transferred data
MRS	1	1S	1I	All cases
MSR	1	1S	1I	If only flags are updated (mask_f)
MSR	3	1S + 2I	3I	If any bits other than the flags are updated (all masks other than_f)
MUL, MLA	2+m	1S+(1+m)I	(2+m)I	All cases
SMULL, UMULL, SMLAL, UMLAL	3+m	1S+(2+m)I	(3+m)I	All cases

Table 11-3 on page 11-5 shows the instruction cycle times from the perspective of the data bus.

Table 11-3 Data bus instruction times

Instruction	Cycle time
LDR	1N
STR	1N
LDM,STM	1N+(n-1)S
SWP	1N+1S
LDC, STC	1N+(n-1)S
MCR,MRC	1C

11.2.1 Multiplier cycle counts

The number of cycles that a multiply instruction takes to complete depends on which instruction it is, and on the value of the multiplier-operand. The multiplier-operand is the contents of the register specified by bits [11:8] of the ARM multiply instructions, or bits [2:0] of the Thumb multiply instructions:

- For ARM MUL, MLA, SMULL, SMLAL, and Thumb MUL, m is:

  1 if bits [31:8] of the multiplier operand are all zero or one

  2 if bits [31:16] of the multiplier operand are all zero or one

  3 if bits [31:24] of the multiplier operand are all zero or all one

  4 otherwise.

- For ARM UMULL, UMLAL, m is:

  1 if bits [31:8] of the multiplier operand are all zero

  2 if bits [31:16] of the multiplier operand are all zero

  3 if bits [31:24] of the multiplier operand are all zero

  4 otherwise.

11.3 Interlocks

Pipeline interlocks occur when the data required for an instruction is not available because of the incomplete execution of an earlier instruction. When an interlock occurs, instruction fetches stop on the instruction memory interface of the ARM940TDMI. Four examples of this are given in:

- *Single load interlock timing* on page 11-7
- *Two cycle load interlock* on page 11-8
- *LDM interlock* on page 11-9
- *LDM dependent interlock* on page 11-10

Example 11-1 on page 11-6 shows a single load interlock code sequence.

Example 11-1 Single load interlock

In this example, the following code sequence is executed:

```
LDR r0, [r1]
ADD r2, r0, r1
```

The `ADD` instruction cannot start until the data is returned from the load. The `ADD` instruction therefore, has to delay entering the Execute stage of the pipeline by one cycle. The behavior on the instruction memory interface is shown in Figure 11-1 on page 11-7.

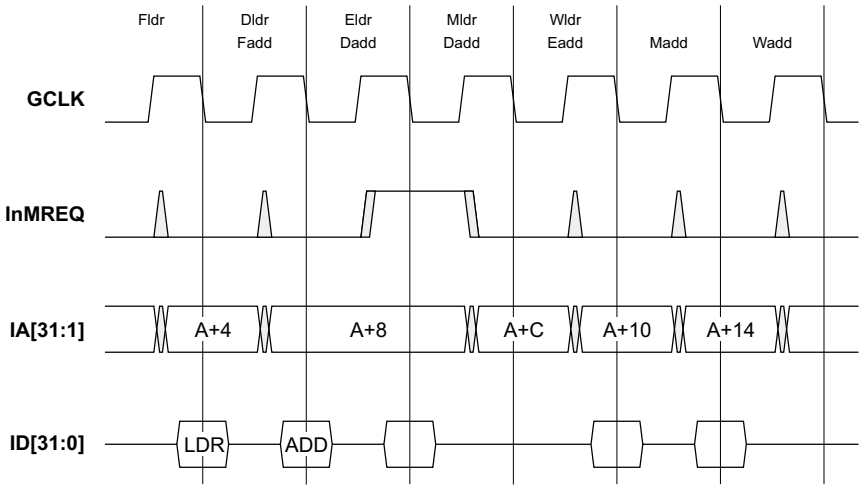

Figure 11-1 Single load interlock timing

Example 11-2 on page 11-7 shows a two cycle load interlock code sequence.

Example 11-2 Two cycle load interlock

In this example, the following code sequence is executed:

```
LDRB r0, [r1,#1]
ADD r2, r0, r1
```

Now, because a rotation must occur on the loaded data, there is a second interlock cycle. The behavior on the instruction memory interface is shown in Figure 11-2 on page 11-8.

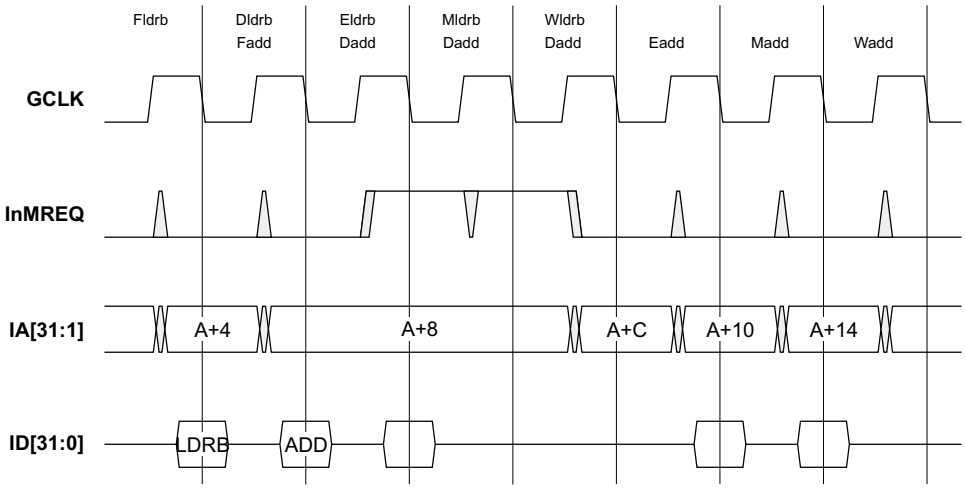

Figure 11-2 Two cycle load interlock

Example 11-3 on page 11-8 shows an LDM interlock code sequence.

Example 11-3 LDM interlock

In this example, the following code sequence is executed:

```
LDMDB r12,{r1-r3}
ADD r2, r2, r1
```

The LDM takes three cycles to execute in the memory stage of the pipeline. The ADD is therefore delayed until the LDM begins its final memory fetch. The behavior of both the instruction and data memory interface are shown in Figure 11-3 on page 11-9.

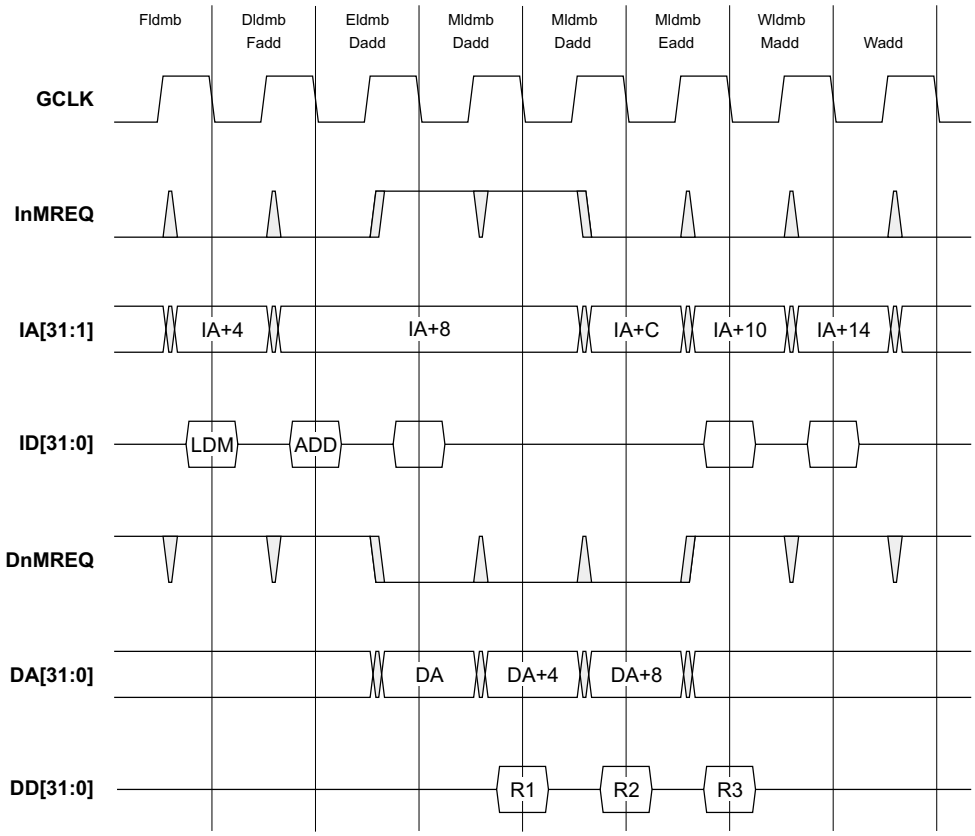

Figure 11-3 LDM interlock

Example 11-4 on page 11-9 shows an `LDM` dependent code sequence.

Example 11-4 LDM dependent interlock

In this example, the following code sequence is executed:

```
LDMDB r12,{r1-r3}
ADD r4, r3, r1
```

The code is the same code as in example 3, but in this instance the ADD instruction uses r3. Because of the nature of load multiples, the lowest register specified is transferred first, and the highest specified register last. Because the ADD is dependent on r3, there must be another cycle of interlock while r3 is loaded. The behavior on the instruction and data memory interface is shown in Figure 11-4 on page 11-10.

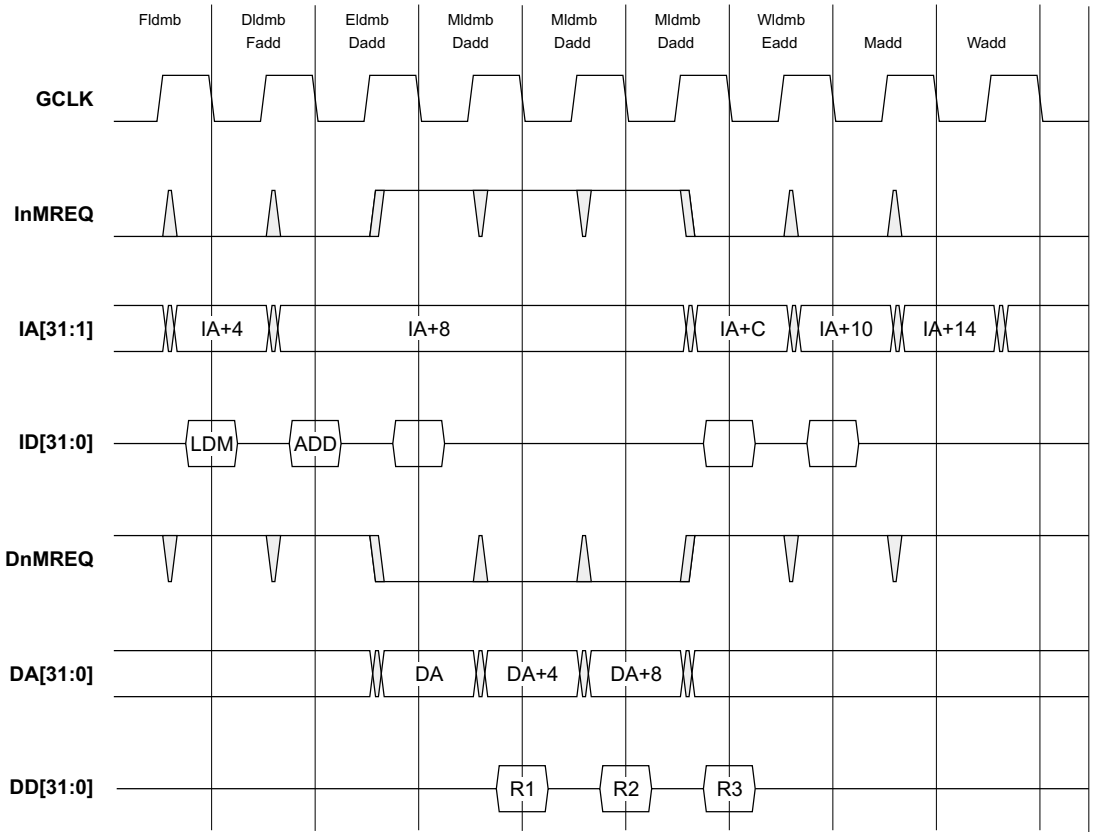

Figure 11-4 LDM dependent interlock

Chapter 12
AC Characteristics

This chapter gives the timing diagrams and timing parameters for the ARM940T. It contains the following sections:

12.1 ARM940T timing diagrams

The AMBA bus interface of the ARM940T conforms to the *AMBA Specification*. See this document for the relevant timing diagrams.

Figure 12-1 shows the **FCLK** timing parameters.

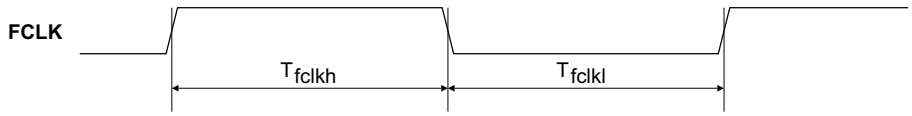

Figure 12-1 FCLK

Figure 12-2 shows the **BCLK** timing parameters.

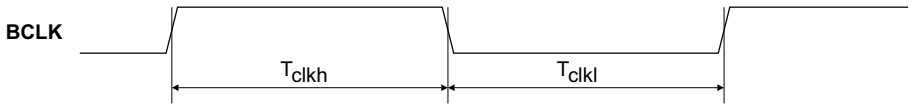

Figure 12-2 BCLK

Figure 12-3 shows the **FCLK** timed coprocessor interface timing parameters.

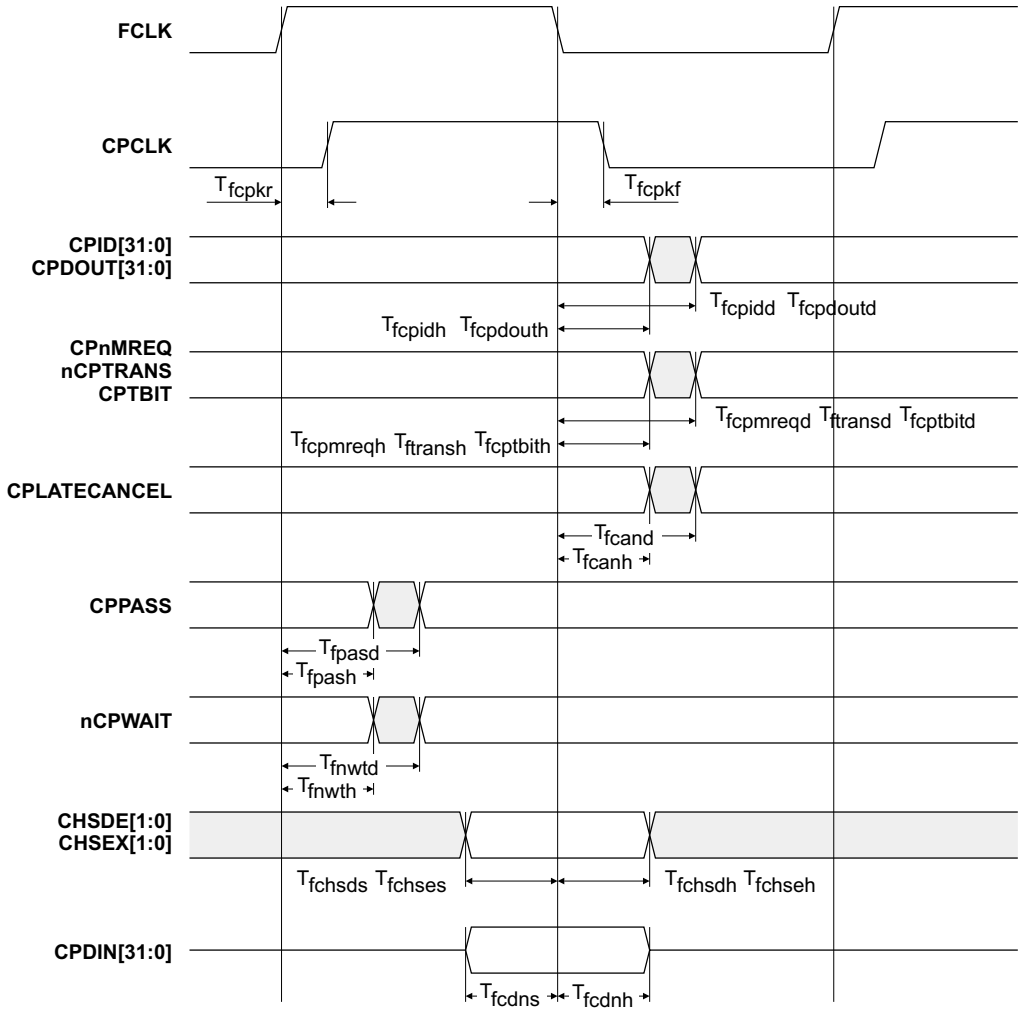

Figure 12-3 ARM940T FCLK timed coprocessor interface

Figure 12-4 shows the **BCLK** timed coprocessor interface timing parameters.

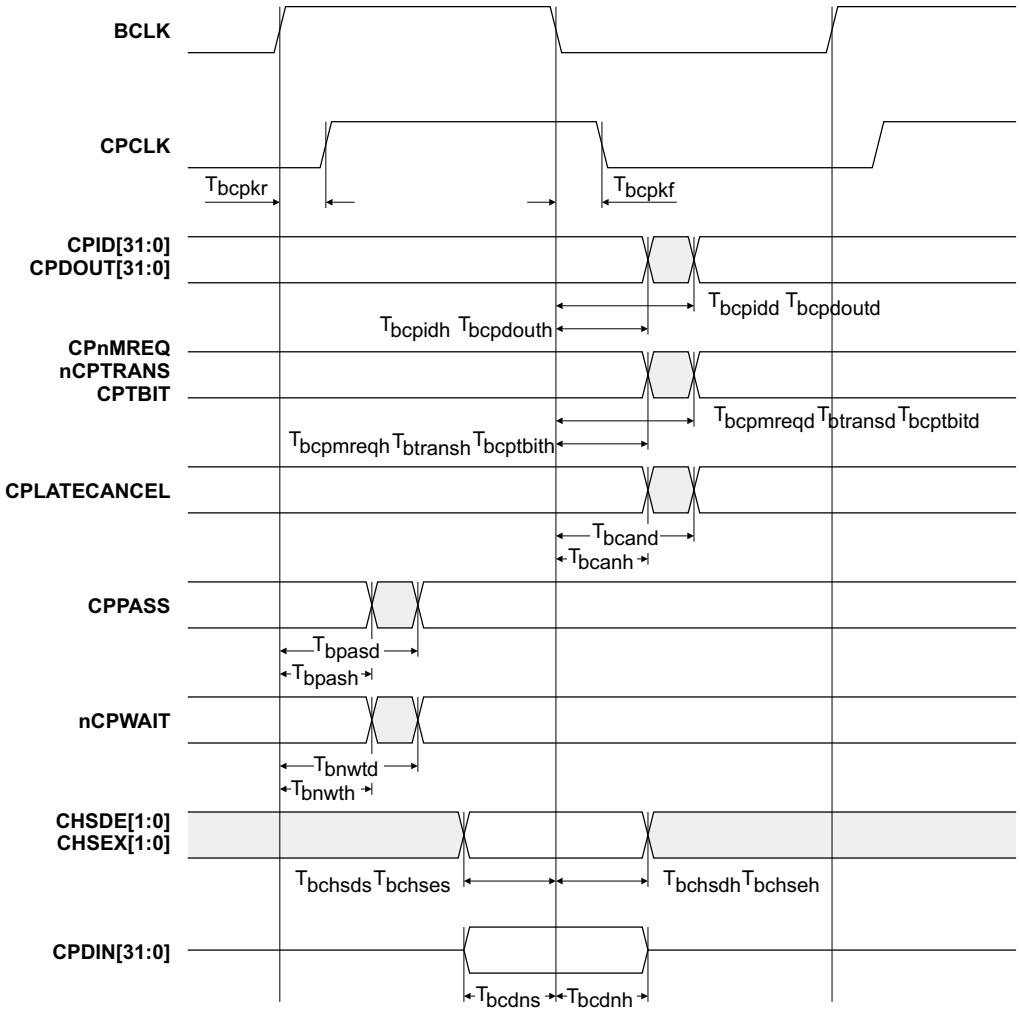

Figure 12-4 ARM940T BCLK timed coprocessor interface

Figure 12-5 shows the **FCLK** related signal timing parameters.

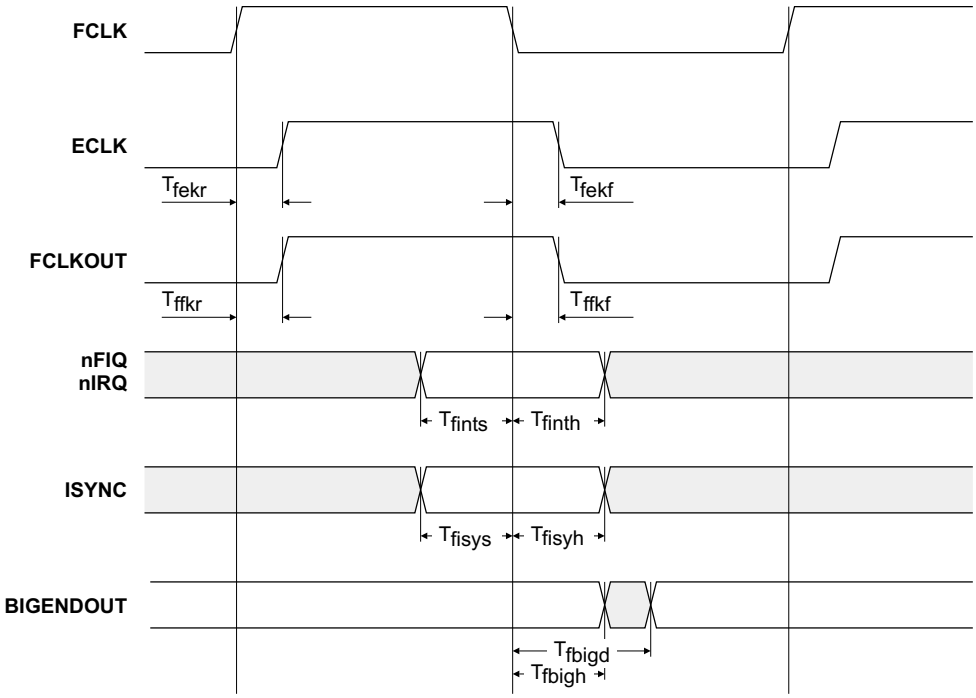

Figure 12-5 ARM940T FCLK related signal timing

Figure 12-6 shows the **BCLK** related signal timing parameters.

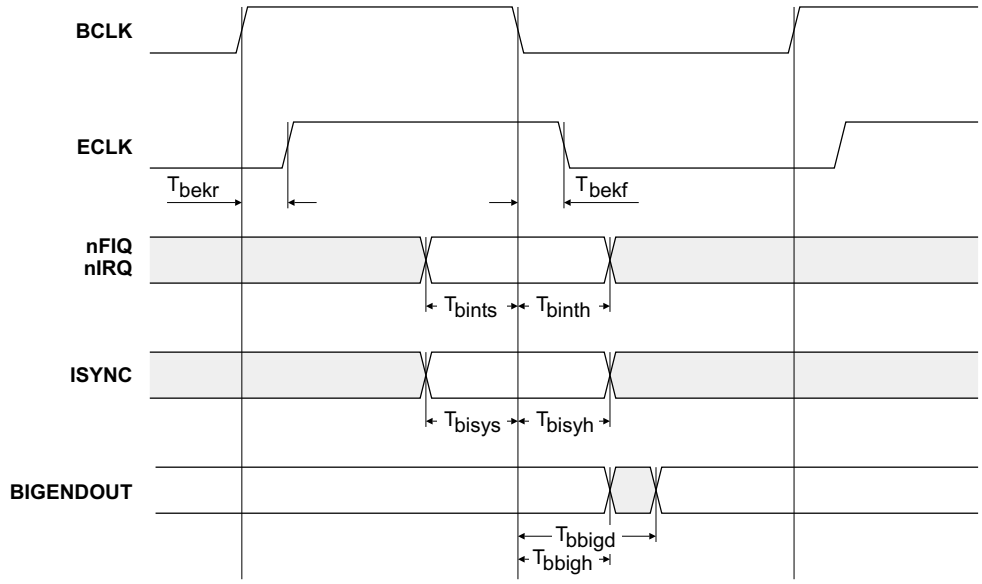

Figure 12-6 ARM940T BCLK related signal timing

Figure 12-7 shows the **SDOUTBS** to **TDO** timing relationship.

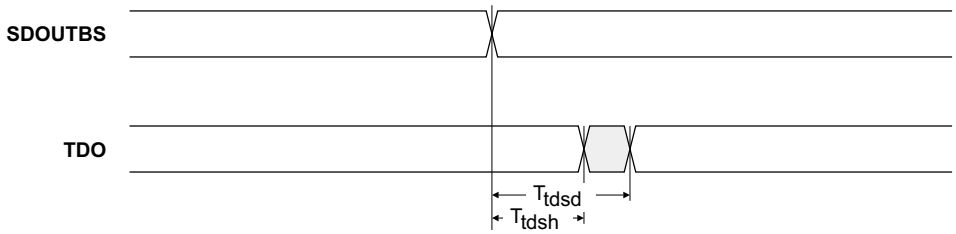

Figure 12-7 ARM940T SDOUTBS to TDO relationship

Figure 12-8 shows the **nTRST** to **RSTCLKBS** timing relationship.

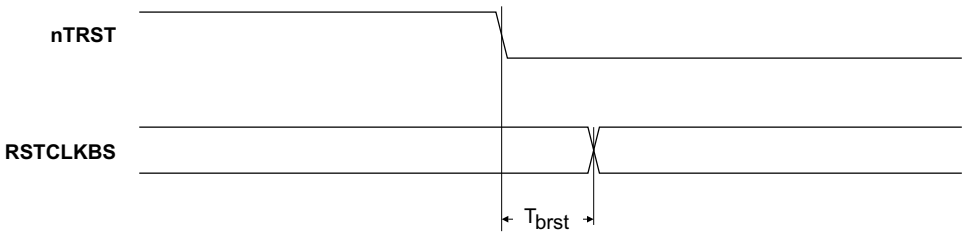

Figure 12-8 ARM940T nTRST to RSTCLKBS relationship

Figure 12-9 on page 12-8 shows the JTAG output signal timing parameters.

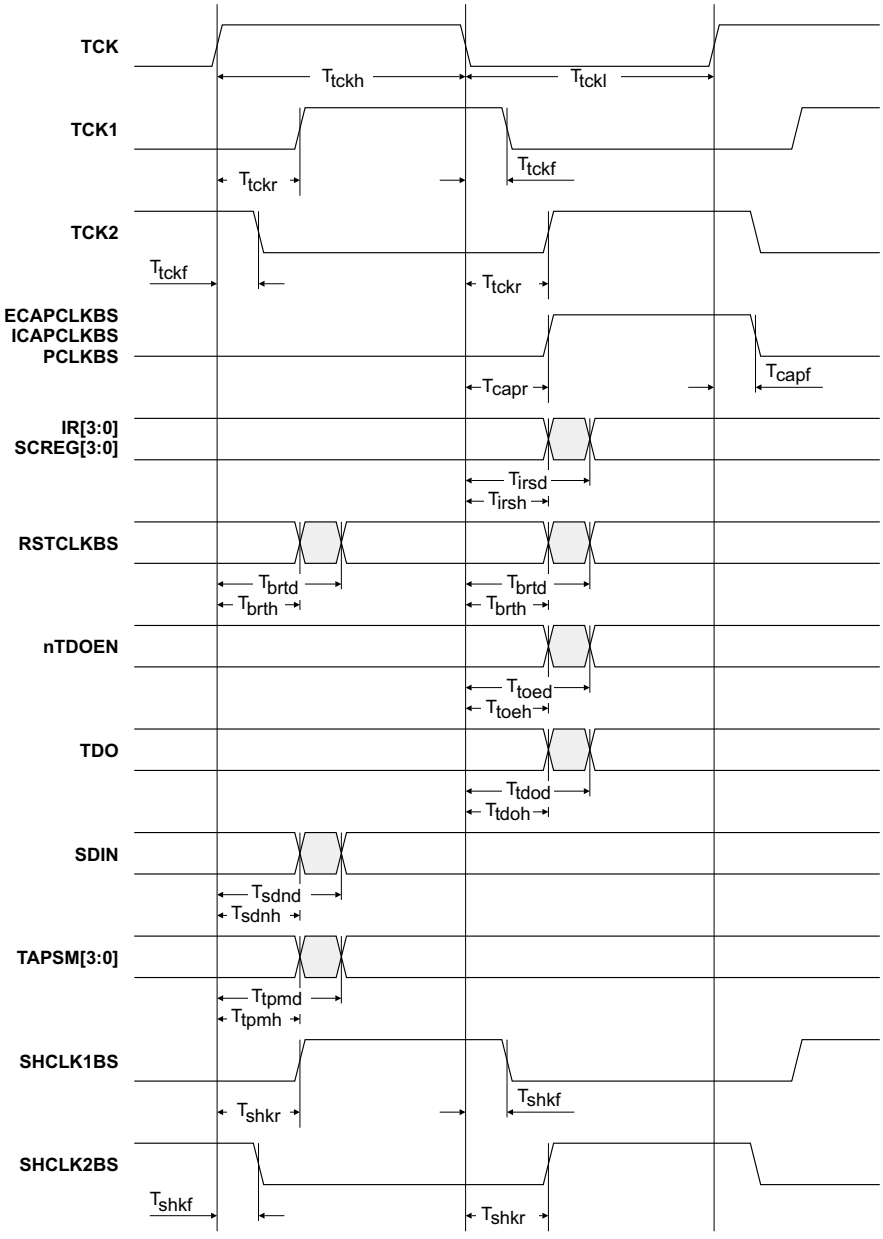

Figure 12-9 ARM940T JTAG output signal

Figure 12-10 shows the JTAG input signal timing parameters.

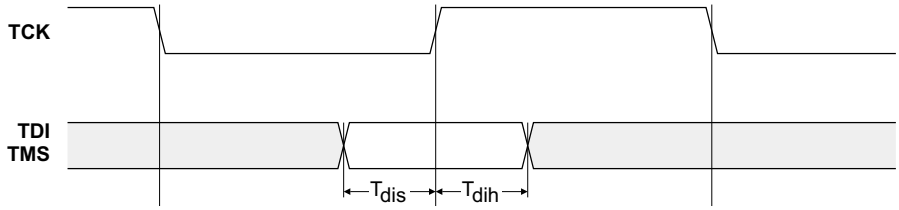

Figure 12-10 ARM940T JTAG input signal timing

Figure 12-11 shows the **FCLK** debug timing parameters.

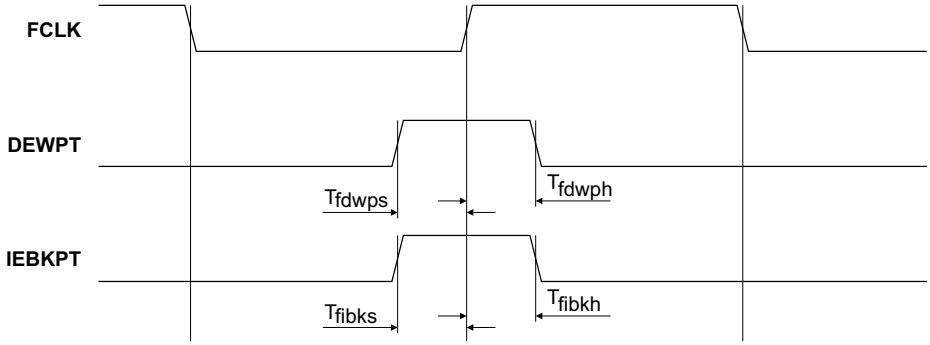

Figure 12-11 FCLK debug timing

Figure 12-12 shows the **BCLK** debug timing parameters.

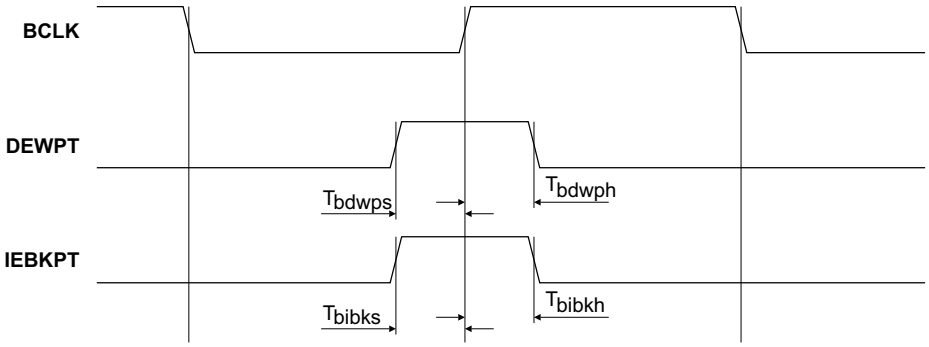

Figure 12-12 BCLK debug timing

Figure 12-13 shows the **FCLK** related debug output signal timing parameters.

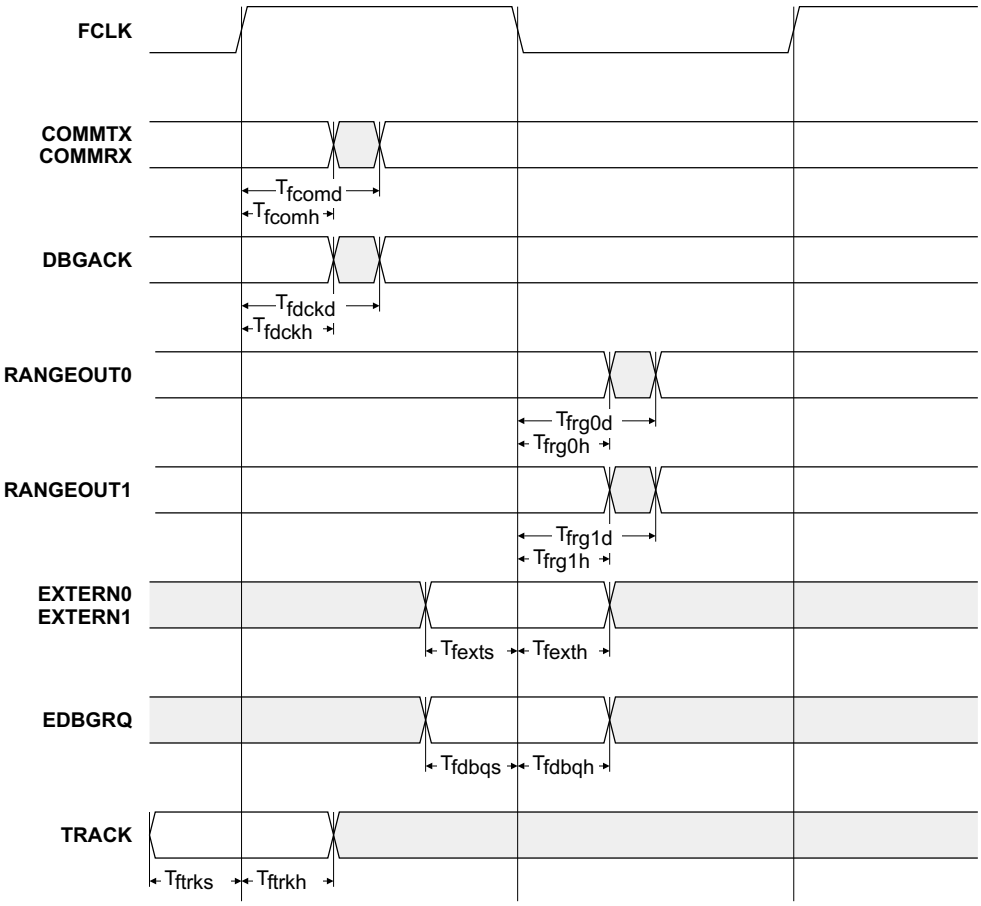

Figure 12-13 ARM940T FCLK related debug output timings

Figure 12-14 shows the **BCLK** related debug output signal timing parameters.

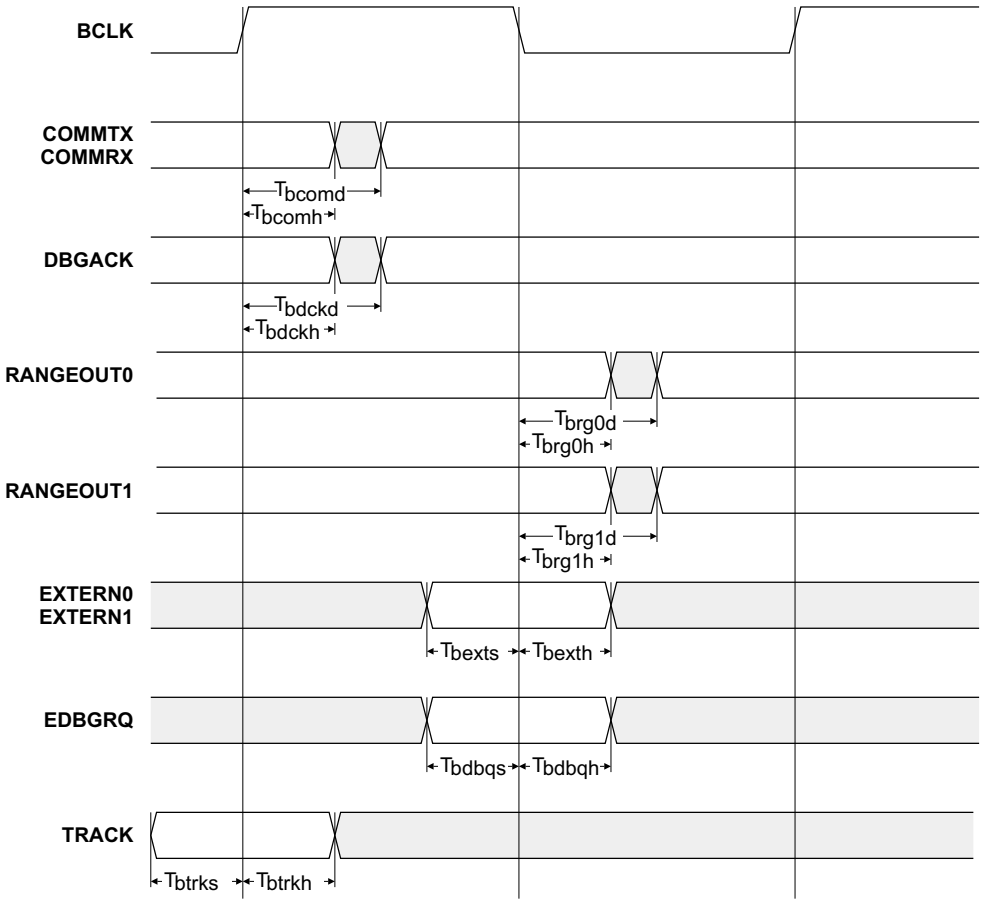

Figure 12-14 ARM940T BCLK related debug output timings

Figure 12-15 shows the **BCLK** related AHB signal timing parameters.

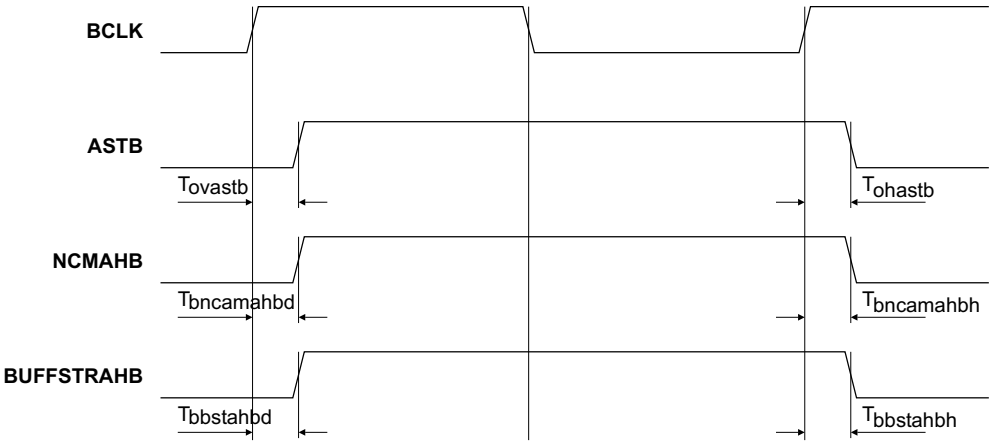

Figure 12-15 AHB signal timings

Figure 12-16 shows the **TCK**related debug output signal timing parameters.

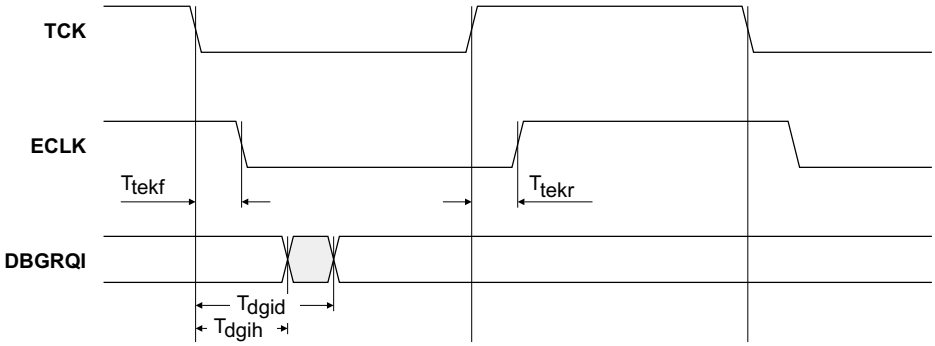

Figure 12-16 ARM940T TCK related debug output timings

Figure 12-17 shows the **nTRST** to **DBGRQI** timing relationship.

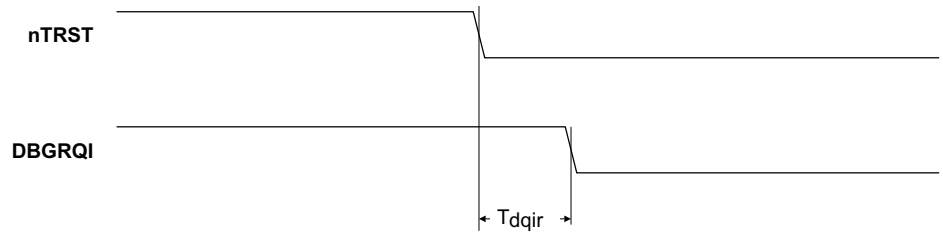

Figure 12-17 nTRST to DBGRQI relationship

Figure 12-18 shows the **EDBGRQ** to **DBGRQI** timing relationship.

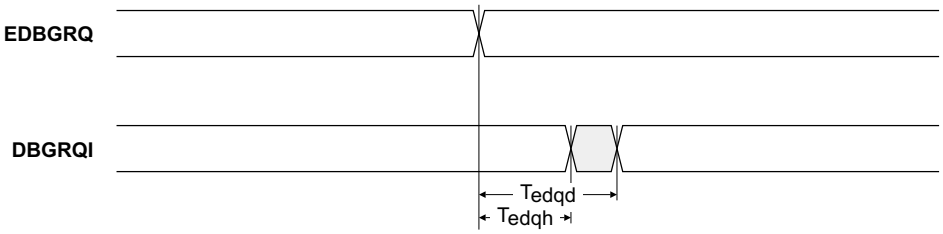

Figure 12-18 ARM940T EDBGRQ to DBGRQI relationship

Figure 12-19 shows the **DBGEN** to output signal timing relationship.

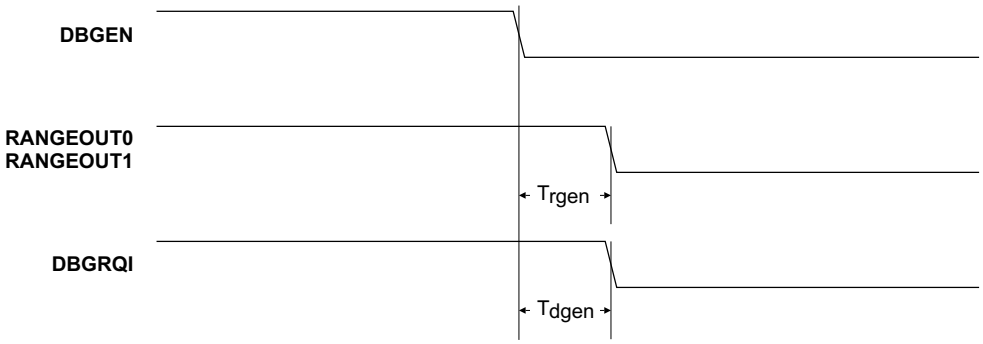

Figure 12-19 ARM940T DBGEN to output relationship

12.2 ARM940T timing parameters

Table 12-1 on page 12-15 lists the ARM940T timing parameters.

Table 12-1 ARM940T timing parameters

Parameter	Description
Tbbigd	**BIGENDOUT** output delay from **BCLK** falling
Tbbigh	**BIGENDOUT** output hold from **BCLK** falling
Tbbstahbd	**BUFFSTRAHB** output delay from **BCLK** rising
Tbbstahbh	**BUFFSTRAHB** output hold from **BCLK** rising
Tbcand	**CPLATECANCEL** output delay from **BCLK** falling
Tbcanh	**CPLATECANCEL** output hold from **BCLK** falling
Tbcdnh	**CPDIN[31:0]** input hold from **BCLK** falling
Tbcdns	**CPDIN[31:0]** input setup to **BCLK** falling
Tbchsh	**CHSDE[1:0]/CHSEX[1:0]** input hold from **BCLK** falling
Tbchss	**CHSDE[1:0]/CHSEX[1:0]** input setup to **BCLK** falling
Tbcomd	**COMMTX/COMMRX** output delay from **BCLK** rising
Tbcomh	**COMMTX/COMMRX** output hold from **BCLK** rising
Tbcpdd	**CPID[31:0]/CPDOUT[31:0]** output delay from **BCLK** falling
Tbcpdh	**CPID[31:0]/CPDOUT[31:0]** output hold from **BCLK** falling
Tbcpkf	**CPCLK** falling output delay from **BCLK** falling
Tbcpkr	**CPCLK** rising output delay from **BCLK** rising
Tbctld	**CPnMREQ/nCPTRANS/CPTBIT** output delay from **BCLK** falling
Tbctlh	**CPnMREQ/nCPTRANS/CPTBIT** output hold from **BCLK** falling
Tbdbqh	**EDBGRQ** input hold from **BCLK** falling
Tbdbqs	**EDBGRQ** input setup to **BCLK** falling
Tbdckd	**DBGACK** output delay from **BCLK** rising
Tbdckh	**DBGACK** output hold from **BCLK** rising
Tbdwph	**DEWPT** input hold from **BCLK** rising

Table 12-1 ARM940T timing parameters (continued)

Parameter	Description
Tbdwps	**DEWPT** input setup to **BCLK** rising
Tbekf	**ECLK** falling output delay from **BCLK** falling
Tbekr	**ECLK** rising output delay from **BCLK** rising
Tbexth	**EXTERN0/EXTERN1** input hold from **BCLK** falling
Tbexts	**EXTERN0/EXTERN1** input setup to **BCLK** falling
Tbibkh	**IEBKPT** input hold from **BCLK** rising
Tbibks	**IEBKPT** input setup to **BCLK** rising
Tbinth	**nFIQ/nIRQ** input hold from **BCLK** falling
Tbints	**nFIQ/nIRQ** input setup to **BCLK** falling
Tbinxd	**INSTREXEC** output delay from **BCLK** falling
Tbinxh	**INSTREXEC** output hold from **BCLK** falling
Tbisyh	**ISYNC** hold from **BCLK** falling
Tbisys	**ISYNC** setup to **BCLK** falling
Tbncamahbd	**NCMAHB** output valid from **BCLK** rising
Tbncamahbh	**NCMAHB** output hold from **BCLK** rising
Tbnwtd	**nCPWAIT** output delay from **BCLK** rising
Tbnwth	**nCPWAIT** output hold from **BCLK** rising
Tbpasd	**CPPASS** output delay from **BCLK** rising
Tbpash	**CPPASS** output hold from **BCLK** rising
Tbrg0d	**RANGEOUT0** output delay from **BCLK** falling
Tbrg0h	**RANGEOUT0** output hold from **BCLK** falling
Tbrg1h	**RANGEOUT1** output hold from **BCLK** falling
Tbrg1d	**RANGEOUT1** output delay from **BCLK** falling
Tbrst	**RSTCLKBS** rising delay from **nTRST** falling
Tbrtd	**RSTCLKBS** output delay from **TCK** falling

Table 12-1 ARM940T timing parameters (continued)

Parameter	Description
Tbrth	**RSTCLKBS** hold from **TCK** falling
Tbtrakd	Tracking mode outputs (**ICAPCLKBS/TAPSM[4:1]/IR[3:0]/SCREG[3:0]/ ECAPCLKBS/PCLKBS/RSTCLKBS/SDIN/SHCLK1BS/SHCLK2BS/ TCK2**) output delay from **BCLK** falling
Tbtrakh	Tracking mode outputs (**ICAPCLKBS/TAPSM[4:1]/IR[3:0]/SCREG[3:0]/ ECAPCLKBS/PCLKBS/RSTCLKBS/SDIN/SHCLK1BS/SHCLK2BS/ TCK2**) hold from **BCLK** falling
Tbtrgkf	Tracking mode delay from **BCLK** falling to **TCK1** falling
Tbtrgkr	Tracking mode delay from **BCLK** rising to **TCK1** rising
Tbtrks	**TRACK** input setup to **BCLK** rising
Tbtrkh	**TRACK** input hold from **BCLK** rising
Tcapf	**ECAPCLKBS/ICAPCLKBS/PCLKBS** falling output delay from **TCK** rising
Tcapr	**ECAPCLKBS/ICAPCLKBS/PCLKBS** rising output delay from **TCK** rising
Tclkh	**BCLK** minimum width high phase
Tclkl	**BCLK** minimum width low phase
Tdebugd	**DBGACK** output delay from **TCK** transition **RANGEOUT0/RANGEOUT1** output delay from **TCK** falling **COMMTX/COMMRX** output delay from **TCK** rising
Tdebugh	**DBGACK** output hold from **TCK** transition **RANGEOUT0/RANGEOUT1** output hold from **TCK** falling **COMMTX/COMMRX** output hold from **TCK** rising
Tdgid	**DBGRQI** output delay from **TCK** falling
Tdgih	**DBGRQI** output hold from **TCK** falling
Tdih	**TDI/TMS** input hold from **TCK** rising
Tdis	**TDI/TMS** input setup to **TCK** rising
Tdqen	Delay from **DBGEN** falling to **DBGRQI** falling
Tdqir	Delay from **nTRST** falling to **DBGRQI** falling

Table 12-1 ARM940T timing parameters (continued)

Parameter	Description
Tdrbsd	**DRIVEOUTBS** output delay from **TCK** falling
Tdrbsh	**DRIVEOUTBS** output hold from **TCK** falling
Tedqd	**DBGRQI** output delay from **EDBGRQ** falling
Tedqh	**DBGRQI** output hold from **EDBGRQ** falling
Tend	**ENAMBADRV/ENBTRAN** output delay from **BCLK** transition
Tenh	**ENAMBADRV/ENBTRAN** output hold from **BCLK** transition
Tfbigd	**BIGENDOUT** output delay from **FCLK** falling
Tfbigh	**BIGENDOUT** output hold from **FCLK** falling
Tfcand	**CPLATECANCEL** output delay from **FCLK** falling
Tfcanh	**CPLATECANCEL** output hold from **FCLK** falling
Tfcdnh	**CPDIN[31:0]** input hold from **FCLK** falling
Tfcdns	**CPDIN[31:0]** input set up to **FCLK** falling
Tfchsh	**CHSDE[1:0]/CHSEX[1:0]** input hold to **FCLK** falling
Tfchss	**CHSDE[1:0]/CHSEX[1:0]** input setup to **FCLK** falling
Tfclkh	**FCLK** minimum width high phase
Tfclkl	**FCLK** minimum width low phase
Tfcomd	**COMMTX/COMMRX** output delay from **FCLK** rising
Tfcomh	**COMMTX/COMMRX** output hold from **FCLK** rising
Tfcpdd	**CPID[31:0]/CPOUT[31:0]** output delay from **FCLK** falling
Tfcpdh	**CPID[31:0]/CPOUT[31:0]** output hold from **FCLK** falling
Tfcpkf	**CPCLK** falling output delay from **FCLK** falling
Tfcpkr	**CPCLK** rising output delay from **FCLK** rising
Tfctld	**CPnMREQ/nCPTRANS/CPTBIT** output delay from **FCLK** falling
Tfctlh	**CPnMREQ/nCPTRANS/CPTBIT** output hold from **FCLK** falling
Tfdbqh	**EDBGRQ** input hold from **FCLK** falling

Table 12-1 ARM940T timing parameters (continued)

Parameter	Description
Tfdbqs	**EDBGRQ** input setup to **FCLK** falling
Tfdckd	**DBGACK** output delay from **FCLK** rising
Tfdckh	**DBGACK** output hold from **FCLK** rising
Tfdwph	**DEWPT** input hold from **FCLK** rising
Tfdwps	**DEWPT** input setup to **FCLK** rising
Tfekf	**ECLK** falling output delay from **FCLK** falling
Tfekr	**ECLK** rising output delay from **FCLK** rising
Tfexth	**EXTERN0/EXTERN1** input hold from **FCLK** falling
Tfexts	**EXTERN0/EXTERN1** input setup to **FCLK** falling
Tffkf	**FCLKOUT** falling output delay from **FCLK** falling
Tffkr	**FCLKOUT** rising output delay from **FCLK** rising
Tfibkh	**IEBKPT** input hold from **FCLK** rising
Tfibks	**IEBKPT** input setup to **FCLK** rising
Tfinth	**nFIQ/nIRQ** hold from **FCLK** falling
Tfints	**nFIQ/nIRQ** setup to **FCLK** falling
Tfinxd	**INSTREXEC** output delay from **FCLK** falling
Tfinxh	**INSTREXEC** output hold from **FCLK** falling
Tfisyh	**ISYNC** input hold from **FCLK** falling
Tfisys	**ISYNC** input setup to **FCLK** falling
Tfnwtd	**nCPWAIT** output delay from **FCLK** rising
Tfnwth	**nCPWAIT** output hold from **FCLK** rising
Tfpasd	**CPPASS** output delay from **FCLK** rising
Tfpash	**CPPASS** output hold from **FCLK** rising
Tfrg0d	**RANGEOUT0** output delay from **FCLK** falling
Tfrg0h	**RANGEOUT0** output hold from **FCLK** falling

Table 12-1 ARM940T timing parameters (continued)

Parameter	Description
Tfrg1d	**RANGEOUT1** output delay from **FCLK** falling
Tfrg1h	**RANGEOUT1** output hold from **FCLK** falling
Tftrakd	Tracking mode outputs (**ICAPCLKBS/TAPSM[4:1]/IR[3:0]/SCREG[3:0]/ECAPCLKBS/PCLKBS/RSTCLKBS/SDIN/SHCLK1BS/SHCLK2BS/TCK2**) output delay from **FCLK** falling
Tftrakh	Tracking mode outputs (**ICAPCLKBS/TAPSM[4:1]/IR[3:0]/SCREG[3:0]/ECAPCLKBS/PCLKBS/RSTCLKBS/SDIN/SHCLK1BS/SHCLK2BS/TCK2**) output hold from **FCLK** falling
Tftrghf	**TCK1** falling tracking mode delay from **FCLK** falling
Tftghr	**TCK1** rising tracking mode delay from **FCLK** rising
Tftrkh	**TRACK** input hold to **FCLK** rising
Tftrks	**TRACK** input setup to **FCLK** rising
Thdt	**BD** hold to **BCLK** falling (AMBA test mode)
Tihagnt	**AGNT** hold to **BCLK** falling
Tihctl	**BWRITE** hold to **BCLK** rising (AMBA test Mode)
Tihdr	**BD** hold to **BCLK** falling (normal mode)
Tihdsel	**DSEL** input hold to **BCLK** rising
Tihnres	**BnRES** input hold to **BCLK** falling
Tihresp	**BWAIT/BERROR/BLAST** hold to **BCLK** rising (normal mode)
Tirsd	**IREG[3:0]/SCREG[3:0]** output delay from **TCK** falling
Tirsh	**IREG[3:0]/SCREG[3:0]** hold from **TCK** falling
Tisagnt	**AGNT** input setup to **BCLK** rising
Tisctl	**BWRITE** setup to **BCLK** falling (AMBA test mode)
Tisdr	**BD** setup to **BCLK** falling (normal mode)
Tisdsel	**DSEL** input setup to **BCLK** falling
Tisnres	**BnRES** input setup to **BCLK** rising
Tisresp	**BWAIT/BERROR/BLAST** setup to **BCLK** rising (normal mode)

Table 12-1 ARM940T timing parameters (continued)

Parameter	Description
Toha	Sequential **BA** output hold from **BCLK** rising
Tohareq	**AREQ** output hold from **BCLK** rising
Tohastb	**ASTB** output hold from **BCLK** rising
Tohctl	Sequential **BLOK/BPROT[1:0]/BSIZE[1:0]/BWRITE** output hold from **BCLK** rising
Tohdt	**BD** output hold from **BCLK** transition (AMBA test mode)
Tohdws	Sequential **BD** output hold from **BCLK** falling
Tohlok	**BLOK** output hold from **BCLK** rising
Tohresp	**BWAIT/BERROR/BLAST** hold from **BCLK** rising (AMBA test mode)
Tohtr	**BTRAN[1:0]** output hold from **BCLK** falling
Tova	Sequential **BA** output delay from **BCLK** rising
Tovaa	Bus handover Address-only **BA** output delay from **BCLK** falling
Tovareq	**AREQ** output delay from **BCLK** rising
Tovastb	**ASTB** output delay from **BCLK** rising
Tovctl	Sequential **BLOK/BPROT[1:0]/BSIZE[1:0]/BWRITE** output delay from **BCLK** rising
Tovctla	Bus handover Address-only **BLOK/BPROT[1:0]/BSIZE[1:0]/BWRITE** output delay from **BCLK** falling
Tovdt	**BD** output delay from **BCLK** transition (AMBA test mode)
Tovdws	Sequential **BD** output delay from **BCLK** falling
Tovlok	**BLOK** output delay from **BCLK** rising
Tovresp	**BWAIT/BERROR/BLAST** output delay from **BCLK** falling (AMBA test mode)
Tovtr	**BTRAN** output delay from **BCLK** rising
Trgen	**RANGEOUT0/RANGEOUT1** falling output delay from **DBGEN** falling
Tsdnd	**SDIN** output delay from **TCK** falling
Tsdnh	**SDIN** output hold from **TCK** falling

Table 12-1 ARM940T timing parameters (continued)

Parameter	Description
Tsdt	**BD** setup to **BCLK** falling (AMBA test mode)
Tshkf	**SHCLK1BS/SHCLK2BS** falling output from **TCK** changing
Tshkr	**SHCLK1BS/SHCLK2BS** rising output from **TCK** changing
Ttckf	**TCK1/TCK2** falling output from **TCK** changing
Ttckh	**TCK** minimum width high phase
Ttckl	**TCK** minimum width low phase
Ttckr	**TCK1/TCK2** rising output from **TCK** changing
Ttdod	**TDO** output delay from **TCK** falling
Ttdoh	**TDO** output hold from **TCK** falling
Ttdsd	**TDO** output delay from **SDOUTBS** changing
Ttdsh	**TDO** output hold from **SDOUTBS** changing
Ttekf	**ECLK** falling output delay from **TCK** falling
Ttekr	**ECLK** rising output delay from **TCK** rising
Ttoed	**nTDOEN** output delay from **TCK** falling
Ttoeh	**nTDOEN** output hold from **TCK** falling
Ttpmd	**TAPSM[3:0]** output delay from **TCK** falling
Ttpmh	**TAPSM[3:0]** output hold from **TCK** falling

Appendix A
ARM940T Signal Descriptions

This appendix lists and describes the ARM940T signals. It contains the following sections:

A.1 AMBA signals

Table A-1 on page A-2 describes the AMBA signals used by the ARM940T.

Table A-1 AMBA signals

Name	Direction	Description
AGNT	Input	Bus grant. A signal from the bus arbiter to a bus master that indicates the bus master is granted the bus when **BWAIT** next goes LOW.
AREQ	Output	Bus request. A signal from the bus master to the bus arbiter that indicates that the ARM940T requires the bus.
ASTB	Output	Indicates a non-idle A-TRAN cycle.
BA[31:0]	Output	Address bus. The processor address bus driven by the active bus master.
BCLK	Input	Bus clock. This clock times all bus transfers. Both the LOW phase and HIGH phase of **BCLK** are used to control transfers on the bus.
BD[31:0]	Input/Output	Data bus. This is a bidirectional system data bus.
BERROR	Input/Output	Error response. A transfer error is indicated by the selected bus slave using the **BERROR** signal. When **BERROR** is HIGH, a transfer error has occurred, when **BERROR** is LOW, the transfer is successful. This signal is also used in combination with the **BLAST** signal to indicate a bus retract operation.
BLAST	Input/Output	Last response. This signal is driven by the selected bus slave to indicate if the current transfer is the last of a burst sequence. When **BLAST** is HIGH, the decoder must allow sufficient time for address decoding. When **BLAST** is LOW, the next transfer can continue a burst sequence.
BLOK	Output	Locked transfers. When HIGH, this signal indicates that the current transfer, and the next transfer, are to be indivisible, and that no other bus master must be given access to the bus. This signal is used by the bus arbiter.
BnRES	Input	Reset. The bus reset signal is active LOW, and is used to reset the system and the bus. This is the only active LOW AMBA signal.
BUFFSTRAHB	Output	When HIGH, indicates that a buffered write is in progress and that **NCMAHB** is not valid.
BPROT[1:0]	Output	Protection control.These signals provide additional information about a bus access and are primarily intended for use by a bus decoder when acting as a basic protection unit. The signals indicate if the transfer is an opcode fetch or data access. The signals also indicate if it is a privileged mode or User mode transfer.

Table A-1 AMBA signals (continued)

Name	Direction	Description
BSIZE[1:0]	Output	Transfer size. These signals indicate the size of the transfer: 00 = byte access 01 = halfword access 10 = word access 11 = reserved.
BTRAN[1:0]	Output	Transfer type. These signals indicate the type of the next transaction: 00 = an address-only transfer 01 = a nonsequential transfer 10 = reserved 11 = a sequential transfer.
BURST[1:0]	Output	Burst access. These signals indicate the length of a burst transfer: 00 = no sequential information available (default) 01 = reserved 10 = current access is part of a four-word transfer 11 = reserved.
BWAIT	Input/Output	Wait response. This signal is driven by the selected bus slave to indicate if the current transfer can complete. If **BWAIT** is HIGH, another bus cycle is required, if **BWAIT** is LOW, the transfer completes in the current bus cycle.
BWRITE	Input/Output	Transfer direction. When HIGH, this signal indicates a write transfer, when LOW, a read transfer.
DSEL	Input	Slave select. This signal is used during test within the AMBA system and allows the ARM940T to be selected and to have test vectors applied to it.
NCMAHB	Output	Early indication of further S-cycles in the current transfer.

A.1.1 AMBA bus specification

ARM940T has an AMBA-compatible bus interface. See the *AMBA Specification (Rev 2.0)* for full details.

See also Chapter 6 *Bus Interface Unit* for details of the subset of AMBA bus transactions that the ARM940T can initiate.

A.2 Coprocessor interface signals

Table A-2 on page A-4 describes the coprocessor interface signals.

Table A-2 Coprocessor interface signals

Name	Direction	Description
CHSDE[1:0]	Input	Coprocessor handshake decode. The handshake signals from the Decode stage of the coprocessor pipeline follower.
CHSEX[1:0]	Input	Coprocessor handshake execute. The handshake signals from the Execute stage of the coprocessor pipeline follower.
CPCLK	Output	Coprocessor clock. This clock controls the operation of the coprocessor interface.
CPDOUT[31:0]	Output	Coprocessor data out. The coprocessor data bus for transferring `MCR` and `LDC` data to the coprocessor.
CPDIN[31:0]	Input	Coprocessor data in. The coprocessor data bus for transferring `MRC` and `STC` data from the coprocessor to the ARM940T.
CPID[31:0]	Output	Coprocessor instruction data. This is the coprocessor instruction data bus that instructions are transferred over to the pipeline follower in the coprocessor.
CPLATECANCEL	Output	Coprocessor late cancel. When a coprocessor instruction is being executed, if this signal is HIGH during the first memory cycle, the coprocessor instruction is canceled without updating the coprocessor state.
nCPMREQ	Output	Not coprocessor memory request. When LOW on a rising **CPCLK** edge and **nCPWAIT** LOW, the instruction on **CPID** enters the coprocessor pipeline follower Decode stage. The second instruction previously in the pipeline follower Decode stage enters its Execute stage.
CPPASS	Output	Coprocessor pass. This signal indicates that there is a coprocessor instruction in the Execute stage of the pipeline, and it must be executed.
CPTBIT	Output	Coprocessor thumb bit. If HIGH, the coprocessor interface is in Thumb state.
nCPTRANS	Output	Not coprocessor translate. When HIGH, the coprocessor interface is in a non-privileged mode. When LOW, the coprocessor interface is in a privileged mode. The coprocessor samples this signal on every cycle when determining the coprocessor response.
nCPWAIT	Output	Not coprocessor wait. The coprocessor clock **CPCLK** is qualified by **nCPWAIT** to allow the ARM940T to control the transfer of data on the coprocessor interface. **nCPWAIT** changes while **CPCLK** is HIGH.

For more information on the coprocessor interface see Chapter 7 *Coprocessor Interface*.

A.3 JTAG and TAP controller signals

Table A-3 on page A-5 describes the JTAG and TAP controller signals.

Table A-3 JTAG and TAP controller signals

Name	Direction	Description
DRIVEOUTBS	Output	Boundary scan cell enable. This signal is used to control the multiplexors in the scan cells of an external boundary scan chain. This signal changes in the UPDATE-IR state when scan chain 3 is selected, and either the INTEST, EXTEST, CLAMP, or CLAMPZ instruction is loaded. When an external boundary scan chain is not connected, this output must be left unconnected.
ECAPCLKBS	Output	EXTEST capture clock for boundary scan. This is a **TCK2** wide pulse generated when the TAP controller state machine is in the CAPTURE-DR state, the current instruction is EXTEST, and scan chain 3 is selected. This signal is used to capture the chip level inputs during EXTEST. When an external boundary scan chain is not connected, this output must be left unconnected.
ICAPCLKBS	Output	INTEST capture clock. This is a **TCK2** wide pulse generated when the TAP controller state machine is in the CAPTURE-DR state, the current instruction is INTEST, and scan chain 3 is selected. This signal is used to capture the chip level outputs during INTEST. When an external boundary scan chain is not connected, this output must be left unconnected.
IR[3:0]	Output	Tap controller instruction register. These four bits reflect the current instruction loaded into the TAP controller instruction register. The bits change on the falling edge of **TCK** when the state machine is in the UPDATE-IR state.
PCLKBS	Output	Boundary scan update clock. This is a **TCK2** wide pulse generated when the TAP controller state machine is in the UPDATE-DR state, and scan chain 3 is selected. This signal is used by an external boundary scan chain as the update clock. When an external boundary scan chain is not connected, this output must be left unconnected.
RSTCLKBS	Output	Boundary scan reset clock. This signal denotes that either the TAP controller state machine is in the RESET state, or that **nTRST** has been asserted. This can be used to reset external boundary scan cells.
SCREG[4:0]	Output	Scan chain register. These four bits reflect the ID number of the scan chain currently selected by the TAP controller. These bits change on the falling edge of **TCK** when the TAP state machine is in the UPDATE-DR state.
SDIN	Output	Boundary scan serial input data. This signal contains the serial data to be applied to an external scan chain, and is valid around the falling edge of **TCK**.

Table A-3 JTAG and TAP controller signals (continued)

Name	Direction	Description
SDOUTBS	Input	Boundary scan serial output data. This is the serial data out of the boundary scan chain (or other external scan chain). It must be set up to the rising edge of **TCK**. When an external boundary scan chain is not connected, this input must be tied LOW.
SHCLK1BS	Output	Boundary scan shift clock phase 1. This control signal is provided to ease the connection of an external boundary scan chain. **SHCLK1BS** is used to clock the master half of the external scan cells. When in the SHIFT-DR state of the state machine and scan chain 3 is selected, **SHCLK1BS** follows **TCK1**. When not in the SHIFT-DR state, or when scan chain 3 is not selected, this clock is LOW. When an external boundary scan chain is not connected, this output must be left unconnected.
SHCLK2BS	Output	Boundary scan shift clock phase 2. This control signal is provided to ease the connection of an external boundary scan chain. **SHCLK2BS** is used to clock the slave half of the external scan cells. When in the SHIFT-DR state of the state machine and scan chain 3 is selected, **SHCLK2BS** follows **TCK2**. When not in the SHIFT-DR state, or when scan chain 3 is not selected, this clock is LOW. When an external boundary scan chain is not connected, this output must be left unconnected.
TAPID[31:0]	Input	This is the ARM940T device identification (ID) code test data register, accessible from the scan chains. It must be tied to an appropriate value when the device is instantiated.
TAPSM[3:0]	Output	TAP controller state machine. This bus reflects the current state of the TAP controller state machine. These bits change off the rising edge of **TCK**.
TCK	Input	Test clock. The JTAG clock (the test clock).
TCK1	Output	**TCK**, phase 1. **TCK1** is HIGH when **TCK** is HIGH, although there is a slight phase lag because of the internal clock non-overlap.
TCK2	Output	**TCK**, phase 2. **TCK2** is HIGH when **TCK** is LOW, although there is a slight phase lag because of the internal clock non-overlap.
TDI	Input	Test data input. JTAG serial input.
TDO	Output	Test data output. JTAG serial output.

Table A-3 JTAG and TAP controller signals (continued)

Name	Direction	Description
nTDOEN	Output	Not **TDO** Enable. When HIGH, this signal denotes that serial data is being driven out on the **TDO** output. **nTDOEN** would normally be used as an output enable for a **TDO** pin in a packaged part.
TMS	Input	Test mode select. **TMS** selects the next state that the TAP controller state machine must change to.
nTRST	Input	Not test reset. Active-low reset signal for the boundary scan logic. This pin must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset (**BnRES**).

A.4 Debug signals

Table A-4 on page A-8 describes the debug signals.

Table A-4 Debug signals

Name	Direction	Description
COMMRX	Output	Communications channel receive. When HIGH, this signal denotes that the comms channel receive buffer contains data waiting to be read by the processor core.
COMMTX	Output	Communications channel transmit. When HIGH, this signal denotes that the comms channel transmit buffer is empty.
DBGACK	Output	Debug acknowledge. When HIGH, this signal indicates the ARM is in debug state.
DBGEN	Input	Debug enable. This input signal allows the debug features of the ARM940T to be disabled. This signal must be HIGH unless debugging is not required.
DBGRQI	Output	Internal debug request. This signal represents the debug request signal presented to the processor core. This is a combination of **EDBGRQ**, as presented to the ARM940T, and bit 1 of the debug control register.
DEWPT	Input	External watchpoint. This signal allows external data watchpoints to be implemented.
ECLK	Output	External clock output.
EDBGRQ	Input	External debug request. When driven HIGH, this causes the processor to enter debug state when execution of the current instruction has completed.
EXTERN0	Input	External input 0. This is an input to watchpoint unit 0 of the EmbeddedICE unit in the processor, and allows breakpoints/watchpoints to be dependent on an external condition.
EXTERN1	Input	External input 1. This is an input to watchpoint unit 1 of the EmbeddedICE unit in the processor, and allows breakpoints/watchpoints to be dependent on an external condition.
IEBKPT	Input	External breakpoint. This signal allows an external instruction breakpoints to be implemented.
INSTREXEC	Output	Instruction executed. Indicates that in the previous cycle, the instruction in the Execute stage of the pipeline passed its condition codes, and executed.

Table A-4 Debug signals (continued)

Name	Direction	Description
RANGEOUT0	Output	EmbeddedICE rangeout 0. This signal indicates that the EmbeddedICE unit watchpoint unit 0 has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint unit enable control bit.
RANGEOUT1	Output	EmbeddedICE rangeout 1. This signal indicates that the EmbeddedICE unit watchpoint unit 1 has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint unit enable control bit.
TRACK	Input	Enable tracking ICE mode. Driving this signal HIGH places the ARM940T into tracking mode for debugging purposes.

A.5 Miscellaneous signals

Table A-5 on page A-10 describes the miscellaneous signals.

Table A-5 Miscellaneous signals

Name	Direction	Description
BIGENDOUT	Output	Big-endian output. When HIGH, the ARM940T is operating in big-endian configuration. When LOW, it is in little-endian configuration.
ENAMBADRV	Output	AMBA signal enabled. This signal is driven HIGH when the AMBA signals, **BA[31:0]**, **BLOK**, **BPROT**, **BWRITE**, and **BSIZE** are driven out of the ARM940T macrocell. When this signal is driven LOW, these outputs are in the high-impedance state.
ENBTRAN	Output	**BTRAN** enable. This signal is driven HIGH when the AMBA signal **BTRAN[1:0]** is driven out of the ARM940T macrocell. When this signal is driven LOW, **BTRAN[1:0]** is in the high-impedance state.
FCLK	Input	Fast clock. The fast clock input is used when the ARM940T is in the synchronous or asynchronous clocking mode.
FCLKOUT	Output	The fast clock signal after the multiplexing within the AMBA test wrapper, as used by the ARM940T clock generator.
GATEDBDDRV	Output	**BD** direction. This signal is driven HIGH when the bidirectional AMBA data bus, **BD[31:0]**, is driven as an output. When this signal is LOW, **BD[31:0]** is in its input state.
ISYNC	Input	Synchronous interrupts. When HIGH, interrupts must be applied synchronously.
nFIQ	Input	Fast interrupt request, active LOW. When the **nFIQ** signal goes LOW the core is being requested to deal with an interrupt.
nIRQ	Input	Interrupt request, active LOW. When the **nIRQ** signal goes LOW the core is being requested to deal with an interrupt.

Glossary

This glossary describes some of the terms used in this manual. Where terms can have several meanings, the meaning presented here is intended.

Abort A mechanism that indicates to a core that it should halt execution of an attempted illegal memory access. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a prefetch abort, a data abort, or an external abort. See also *Data abort*, *External abort* and *Prefetch abort*.

Abort model An abort model is the defined behavior of an ARM processor in response to a Data Abort exception. Different abort models behave differently with regard to load and store instructions that specify base register writeback.

ALU See *Arithmetic Logic Unit*.

Application Specific Integrated Circuit An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.

Arithmetic Logic Unit The part of a processor core that performs arithmetic and logic operations.

ARM state A processor that is executing ARM (32-bit) word-aligned instructions is operating in ARM state.

ASIC See *Application Specific Integrated Circuit*.

Banked registers Those physical registers whose use is defined by the current processor mode. The banked registers are R8 to R14.

Base register A register specified by a load or store instruction that is used to hold the base value for the instruction's address calculation.

Big-endian Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory. See also *Little-endian* and *Endianness*.

Breakpoint A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to allow inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested. See also *Watchpoint*.

Byte An 8-bit data item.

Cache A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retreiving copies of often used instructions and/or data. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.

Cache contention When the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity increases and performance decreases.

Cache hit A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.

Cache line index The number associated with each cache line in a cache set. Within each cache set, the cache lines are numbered from 0 to (set associativity) -1.

Cache lockdown To fix a line in cache memory so that it cannot be overwritten. Cache lockdown allows critical instructions and/or data to be loaded into the cache so that the cache lines containing them will not subsequently be reallocated. This ensures that all subsequent accesses to the instructions/data concerned are cache hits, and therefore complete as quickly as possible.

Cache miss A memory access that cannot be processed at high speed because the instruction/data it addresses is not in the cache and a main memory access is required.

CAM See *Content addressable memory*.

Central Processing Unit The part of a processor that contains the ALU, the registers, and the instruction decode logic and control circuitry. Also commonly known as the processor core.

Clock gating Gating a clock signal for a macrocell with a control signal (such as **PWRDOWN**) and using the modified clock that results to control the operating state of the macrocell.

Condition field A 4-bit field in an instruction that is used to specify a codition under which the instruction can execute.

Content addressable memory Memory that is identified by its contents. Content addressable memory is used in CAM-RAM architecture caches to store the tags for cache entries.

Coprocessor A processor that supplements the main CPU. It carries out additional functions that the main CPU cannot perform. Usually used for floating-point math calculations, signal processing, or memory management.

CPU See *Central Processing Unit*.

Data Abort An indication from a memory system to a core that it should halt execution of an attempted illegal memory access. A data abort is attempting to access invalid data memory. See also *Abort*, *External abort* and *Prefetch abort*.

Data cache See *DCache*.

DCache A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retreiving copies of often used data. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.

Debugger A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Domain A collection of sections, large pages and small pages of memory, which can have their access permissions switched rapidly by writing to the Domain Access Control Register (CP15 register 3).

Double word A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.

EmbeddedICE The additional JTAG-based hardware provided by debuggable ARM processors to aid debugging.

Endianness Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory. See also *Little-endian* and *Big-endian*.

Exception vector One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt service routine.

External abort An indication from an external memory system to a core that it should halt execution of an attempted illegal memory access. An external abort is caused by the external memory system as a result of attempting to access invalid memory. See also *Abort*, *Data abort* and *Prefetch abort*

Halfword A 16-bit data item.

ICache A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retreiving copies of often used instructions. This is done to greatly reduce the average speed of memory accesses and so to increase processor performance.

Instruction cache See *ICache*.

Joint Test Action Group The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.

JTAG See *Joint Test Action Group*.

Little-endian Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory. See also *Big-endian* and *Endianness*.

Macrocell A complex logic block with a defined interface and behavior. A typical VLSI system will comprise several macrocells (such as an ARM9E-S, an ETM9, and a memory block) plus application-specific logic.

Prefetch abort An indication from a memory system to a core that it should halt execution of an attempted illegal memory access. A prefetch abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory. See also *Data abort*, *External abort* and *Abort*

Processor A contraction of microprocessor. A processor includes the CPU or core, plus additional components such as memory, and interfaces. These are combined as a single macrocell, that can be fabricated on an integrated circuit.

Region A partition of instruction or data memory space.

Register A temporary storage location used to hold binary data until it is ready to be used.

SBO See *Should be one*.

SBZ See *Should be zero*.

SCREG The currently selected scan chain number in an ARM TAP controller.

Should be one Should be written as 1 (or all 1s for bit fields) by software. Writing a 0 will produce UNPREDICTABLE results.

Should be zero Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 will produce UNPREDICTABLE results.

Tag bits The index or key field of a CAM entry.

TAP See *Test access port*.

Test Access Port The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**.

Thumb state A processor that is executing Thumb (16-bit) half-word aligned instructions is operating in Thumb state

UNDEFINED An instruction that generates an undefined instruction exception.

UNPREDICTABLE For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. UNPREDICTABLE instructions must not halt or hang the processor, or any part of the system.

Watchpoint A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to allow inspection of register contents, memory locations, and variable values when memory is written to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested. See also *Breakpoint*.

Word A 32-bit data item.

Index

The items in this index are listed in alphabetical order. The references given are to page numbers.

E

F

I

J

L

M

N

O

P

R

S

T

W

