ARM940T

(Rev 0)

Technical Reference Manual

ARM

ARMDDI 0092B

ARM940T
Technical Reference Manual

Release information

Change history

Description	Issue	Change
February 1998	A	Technical amendments and reformatting
September 1998	B	First full release.

Proprietary notice

Confidentiality status

This document is Open Access (no restriction on distribution).

Product status

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

Web address

`http:\\www.arm.com`

Preface

About this document

This document is a reference manual for the ARM940T microprocessor.

The document describes silicon revisions 0, 0b and 0c. Apart from bug fixes, these revisions have the same specification except that revision 0b and 0c include the **BURST[1:0]** signal, which is described within. The **BURST[1:0]** signal does not appear on revision 0 silicon.

Intended audience

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

Typographical conventions

The following typographical conventions are used in this document:

bold	highlights signal names and menu options within text, internal signals are further identified by italics.
italic	highlights ARM-specific terminology, cross references and references to other publications.
`typewriter`	identifies file and program names, source code, and text (such as commands) that may be entered at the keyboard.
`typewriter italic`	identifies arguments to commands or functions which should be replaced by a specific value.
`typewriter bold`	identifies language keywords when used outside example code.

Related publications

ARM Architecture Reference Manual (ARM DDI 0100).
ARM9TDMI Technical Reference Manual (ARM DDI 0091).
AMBA Specification (ARM IHI 0001).

Further reading

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

Feedback on this manual

If you have any comments or suggestions about this document, please send an 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.

Feedback on this product

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

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

Contents
ARM940T Technical Reference Manual

Chapter 1
Overview

1.1 Introduction

This chapter introduces the ARM940T processor.

1.2 The ARM940T

The ARM940T is a member of the ARM9 Thumb family of general-purpose microprocessors. The ARM940T is targeted at embedded control applications where high performance, low die size, and low power are all important. The ARM940T supports both the 32-bit ARM and 16-bit Thumb instruction sets, allowing the user to trade off between high performance and high code density. 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.

The ARM940T is a Harvard cache architecture processor. The separate instruction and data caches in this design are 4KB each in size, with a 4-word line length. A protection unit allows the memory to be segmented and protected in a simple manner, and is ideal for embedded control applications. There is no virtual to physical address mapping. A writeback cache scheme and write buffer are used to optimize performance and minimize bus traffic, thus reducing system power consumption.

The processor core within ARM940T is an ARM9TDMI. This processor core is a Harvard architecture device implemented using a five-stage pipeline consisting of fetch, decode, execute, memory and write stages, and can be provided as a stand-alone core which may be embedded into more complex devices.

The ARM940T interface to the rest of the system is via unified address and data buses. This interface is compatible with the *Advanced Microcontroller Bus Architecture* (*AMBA*) bus scheme. For coprocessor support, the instruction and data buses are exported along with simple handshaking signals. The ARM940T also has a 'TrackingICE' mode which allows an approach similar to a conventional ICE mode of operation.

1.3 Processor block diagram

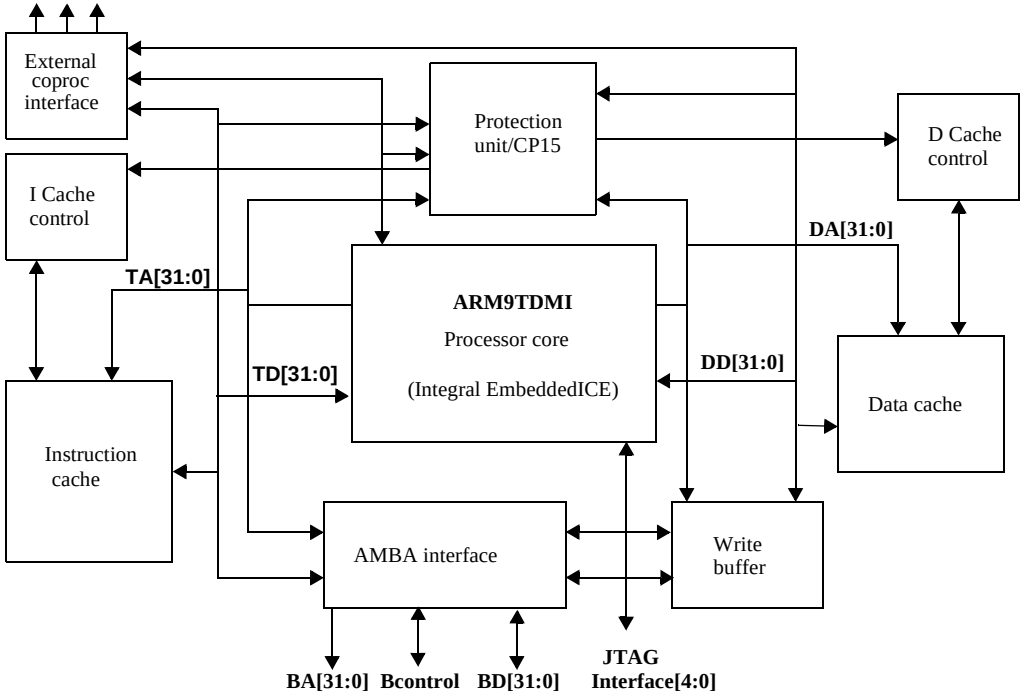

Figure 1-1 ARM940T block diagram

Chapter 2
Programmer's Model

This chapter describes the programmer's model for the ARM940T.

2.1 Introduction

The ARM940T cache processor macrocell is built around the ARM9TDMI processor core. The ARM9TDMI processor core implements ARM Architecture V4T, which includes the 32-bit ARM instruction set and the 16-bit Thumb instruction set. The ARM Architecture V4T programmer's model is described in the *ARM Architecture Reference Manual*, and implementation-specific information is described in the *ARM9TDMI Technical Reference Manual.*

The ARM940T has two coprocessors, CP14 and CP15, which extend the programmer's model. A coprocessor interface allows additional coprocessors to be attached to add floating point, DSP, graphics acceleration, or other application-specific functionality.

CP14 is described in *8.16 Debug communications channel* on page 8-45; CP15 is described below, in section *2.2 ARM940T CP15 registers*.

2.2 ARM940T CP15 registers

2.2.1 CP15 register map summary

As with all cached ARM processors, the ARM940T includes coprocessor 15 (CP15) for system control. The structure of CP15 is very similar to that of other cached ARM processors such as the ARM720T, and the ARM710T:

- Register 0 is read only. All writes to this register are ignored.
- Register 7 is write only. Reads of this register are unpredictable.
- All other registers are read/write.
- A read from or write to a reserved register is UNDEFINED.

A summary of the register map is given in *Table 2-1 CP15 register map*:

Table 2-1 CP15 register map

Register	Functions
0	ID code
1	Control
2	Cacheable
3	Write buffer control
5	Protection region access permissions
6	Protection region base / size control
7	Cache operations
9	Cache lock down
15	Test
4,8,10–14	Reserved

2.2.2 Register 0

This is a read-only register which returns a 32-bit ID code.

31 24	23 16	15 4	3 0
Implementor	Architecture version	Part number	Revision
0x41 = A= ARM	0x2 = Architecture 4T	0x940	0x0

2.2.3 Register 1: Control register

This contains the global control bits of the ARM940T. All reserved bits should either be written with zero or one, as indicated, or written using read-modify-write. The reserved bits have an UNPREDICTABLE value when read.

All defined bits in the control register are set to zero at reset.

Table 2-2 CP15 register 1

Register Bit	Functions
0	Protection unit enable (P)
1	Reserved (should be zero)
2	D Cache enable bit (D)
3:6	Reserved (should be one)
7	Big-end bit (E)
8:11	Reserved (should be zero)
12	I Cache enable bit (1)
13	Alternate vectors select (V)
14:29	Reserved (should be zero)
30	**nFastBus** select (nF)
31	Asynchronous clocking select (iA)

- Bit 0 enables the protection unit (see *Chapter 4 Caches and Write Buffer*)
- Bits 2 and 12 enable the caches (see *Chapter 4 Caches and Write Buffer*)

- Bit 7 selects the endian configuration of the ARM940T. Setting bit 7 selects a big-endian configuration. Clearing bit 7 selects a little–endian configuration. Bit 7 is cleared during reset.

Bit 13 selects the location of the vector table. During reset, the bit is cleared and the vector table is located at address 0x00000000. When bit 13 is set, the vector table is relocated to address 0xffff0000.

Bits 30 and 31 determine the clocking mode of the processor.

Table 2-3 Clocking modes

Clocking mode	nFASTBUS	ASYNC
FastBus mode	0	0
Reserved	0	1
Synchronous	1	0
Asynchronous	1	1

Clocking modes are discussed in *Chapter 7 ARM940T Coprocessor Interface*.

2.2.4 Register 2: Cacheable registers

These registers contain the cacheable attributes for the eight areas of memory. Individual control is provided for the I and D caches.

- If the opcode_2 field = 0, the data-cacheable bits are programmed:
  `MCR/MRC p15,0,Rd,c2,c0,0 Write/Read data-cacheable bits`
- If the opcode_2 field = 1 the instruction-cacheable bits are programmed:
  `MCR/MRC p15,0,Rd,c2,c0,1 Write/Read instruction cacheable bits`

The format for the cacheable bits in data and instruction regions is the same, and is given in *Table 2-4 Cacheable bits register format*. Setting a bit makes an area cacheable, clearing it makes it non-cacheable. See also *4.5.1 Write buffer operation*. All defined bits in the cacheable registers are set to zero at reset.

Table 2-4 Cacheable bits register format

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

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

2.2.5 Register 3: Write buffer control register

This register contains the write buffer control (bufferable) attribute for the eight areas of memory. Setting a bit makes an area bufferable, clearing a bit makes an area unbuffered. For cacheable regions, this determines the type of cache operations. See *4.5.1 Write buffer operation*.

```
MCR/MRC p15,0,Rd,c3,c0,0 Write/Read data-cacheable bits
```

The opcode_2 field should be 0 as the write buffer only operates on data regions.

This only applies to the D Cache.

All defined bits in the write buffer control register are set to zero at reset.

Table 2-5 CP15 register map

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 discussed in *Chapter 3 Protection Unit*.

2.2.6 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 of a MRC/MCR determines whether the instruction or data access permissions are to be programmed:

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

Each register contains the access permission bits, **apn[1:0]**, for the eight areas of instruction or data memory.

All defined bits in the protection registers are set to zero at reset.

Table 2-6 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 encoding is shown in *Table 2-7 Permission encoding*.

Table 2-7 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 discussed in *Chapter 3 Protection Unit.*

2.2.7 Register 6: Protection region base / size registers

This register can define 16 programmable regions (eight instruction, eight data) in memory. These registers define the base and size of each of the eight areas of memory. Individual control is provided for the instruction and data memory regions. 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. At least one instruction and data memory region must be programmed before the protection unit is enabled including its size, base address, access permissions, cache and write buffer enables.

The opcode_2 field defines whether the data or instruction protection regions are to be programmed. The CRm field selects the region number.

Table 2-8 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
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 2-9 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

Table 2-9 CP15 instruction protection region registers

ARM instruction	Protection region register
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 2-10 CP15 protection region register format*.

Table 2-10 CP15 protection region register format

Register bit	Function
31:12	Base address
11:6	Unused
5:1	Area size
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.

Example base setting

An 8KB size region must be aligned to an 8KB boundary—bits [31:12] = 0x00002. Area sizes are given in *Table 2-11*. Register 6 is discussed in *Chapter 3 Protection Unit*.

Table 2-11 Area size encoding

Bit encoding	Area size
00000 to 01010	Reserved
01011	4KB
01100	8KB
01101	16KB
01110	32KB
01111	64KB
10000	128KB
10001	256KB
10010	512KB
10011	1MB
10100	2MB
10101	4MB
10110	8MB
10111	16MB
11000	32MB
11001	64MB
11010	128MB
11011	256MB
11100	512MB
11101	1GB
11110	2GB
11111	4GB

2.2.8 Register 7

A write to this register allows the caches to be flushed, and an I Cache line to be prefetched. A read from this register returns an UNPREDICTABLE value. A subset of the Architecture V4 functions, as defined in the *ARM Architecture Reference Manual*, is implemented, see *Table 2-12 Cache operations through register 7*. "Data" means the value transferred in the Rd.

Table 2-12 Cache operations through register 7

Function	Data	ARM instruction
Flush I Cache	Should be zero	MCR p15, 0, Rd, c7, c5, 0
Flush I Cache single entry	Index/segment	MCR p15, 0, Rd, c7, c5, 1
Flush D Cache	Should be zero	MCR p15, 0, Rd, c7, c6, 0
Flush D Cache single entry	Index/segment	MCR p15, 0, Rd, c7, c6, 1
Clean D Cache entry	Index/segment	MCR p15, 0, Rd, c7, c10, 1
Prefetch I Cache line	Address	MCR p15, 0, Rd, c7, c13, 1
Clean and Flush D Cache entry	Index/segment	MCR p15, 0, Rd, c7, c14, 1

Where the required value is an Index/Segment, the format is:

Table 2-13 CP15 register 7 index/segment data format

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

For the I Cache prefetch operation, the data format is:

Table 2-14 CP15 Register 7 prefetch address format

Rd bit position	Function
31:6	Address bits 31:6
5:4	Cache segment
3:0	Should be zero

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

2.2.9 Register 9: Programming lockdown registers

These registers allow regions of the cache to be locked down. The format is:

Table 2-15 Programming the lockdown registers

ARM instructions	Lockdown register
MCR/MRC p15, 0, Rd, c9, c0, 0	Data lockdown control
MCR/MRC p15, 0, Rd, c9, c0, 1	Instruction lockdown control

The format of the registers, Rd, transferred during this operation, is shown below:

All defined bits in the lockdown registers are set to zero at reset.

Table 2-16 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 discussed in *Chapter 4 Caches and Write Buffer*.

2.2.10 Register 15: Test register

This register controls features intended for use during silicon production testing only. The DTRRobin and ITRRobin bits set the respective caches into a pseudo round-robin replacement mode.

All defined bits in the test register are set to zero at reset.

Table 2-17 CP15 register 15

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

This register is for production test purposes only, and should not be used for any other purpose.

2.2.11 Reserved registers

Accessing a reserved register is UNPREDICTABLE.

Chapter 3
Protection Unit

3.1 Introduction

This chapter describes the ARM940T protection unit. This unit allows memory to be partitioned, and individual attributes to be set for each protection region. Both the instruction address space and the data address space may be divided into eight regions of variable size. The protection unit is programmed via CP15 registers 1, 2, 3, 5 and 6.

The information in this chapter is organized as follows:

- Enabling the protection unit
- Memory regions
- Overlapping regions.

3.2 Enabling the protection unit

Before the protection unit is enabled, valid protection regions must be programmed. If they are not programmed, the ARM940T can enter a state that is recoverable only by reset. Setting bit 0 of the CP15 register 1, the control register, enables the protection unit.

When the protection unit is disabled, all instruction fetches are non-cacheable and all data accesses are non-cacheable and non-bufferable. This results in very poor system performance, so software should define memory regions and enable the protection unit soon after reset.

3.3 Memory regions

Both the instruction and data address spaces may be partitioned into a maximum of eight regions. Each region is specified by:

- a base address
- a size field
- cache and write buffer configuration
- read/write access permissions

The ARM architecture uses constants with code to do address calculations. These are called *inline literals*. For correct operation, any area of memory from which code will be executed should be defined for both the instruction and data address spaces.

The base address and size properties are programmed via CP15 register 6, the format of which is shown in *Table 3-1 Protection register format*:

Table 3-1 Protection register format

Register bit	Function
31:12	Base address
11:6	Unused
5:1	Area size
0	Region enable, reset to disable (0)

3.3.1 Area size

The area size is specified as a 5-bit value, encoding a range of values from 4KB to 4GB. The encoding is shown below in *Table 3-2 Region size encoding*:

Table 3-2 Region size encoding

Bit encoding	Area size
00000 to 01010	Reserved
01011	4KB
01100	8KB
01101	16KB
01110	32KB
01111	64KB
10000	128KB
10001	256KB
10010	512KB
10011	1MB
10100	2MB
10101	4MB
10110	8MB
10111	16MB
11000	32MB
11001	64MB
11010	128MB
11011	256MB
11100	512MB
11101	1GB
11110	2GB
11111	4GB

Note

Any value less than ‘01011’ programmed in **Rd[5:1]** will result in unpredictable behavior.

3.3.2 Base address

The base address defines the start of the memory region. This must be aligned to the region size. If not, this results in UNPREDICTABLE behavior. For example, if a region size of 8KB is programmed for a given region, the base address must be a multiple of 8KB.

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 which falls within a given region.

3.3.3 Region attributes

The attributes are:

- cacheable
- 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 cacheable 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 *Chapter 4* and specifically in *4.5.1 Write buffer operation*.

3.4 Overlapping regions

The protection unit may be programmed such that two, or more, regions overlap. In this case, a fixed priority scheme applies to determine which region's attributes should be applied to the memory access. Attributes for region 7 take highest priority, and region 0 take lowest priority.

A block diagram showing the protection unit is given in *Figure 3-1*.

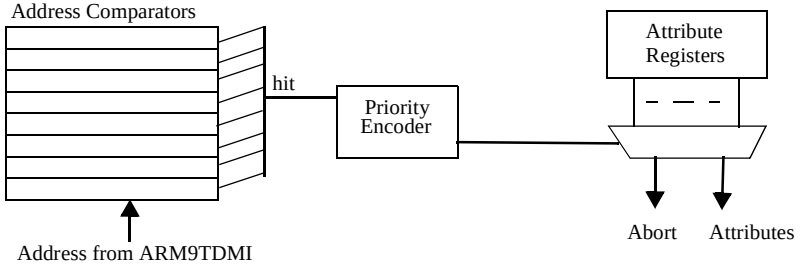

Figure 3-1 ARM940T protection unit

Consider the following:

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).

If the processor attempts to perform a data store to address 0x3010 while in user mode, the address falls into both region 1 and region 2, as shown in *Figure 3-2*. As there is a clash, the attributes associated with region 2 are applied, because of the fixed priority scheme. In this case, the user is only allowed to perform reads from this region, and so a data abort occurs.

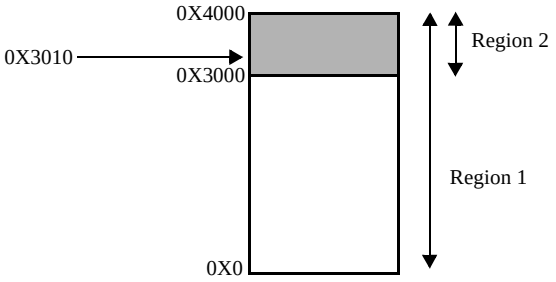

Figure 3-2 Overlapping memory regions

3.4.1 Background regions

Overlapping regions increase the flexibility of how the eight regions may be mapped onto physical memory devices in the system. The overlapping properties may also be used to specify a background region. For example, there may be a number of physical memory areas sparsely distributed across the 4GB address space. If a programing error occurs therefore, it may be possible for the processor to issue an address which does not fall into any defined region.

If the address issued by the processor does not fall into any of the defined regions, the ARM940T protection unit is hardwired to abort the access. You may override this behavior 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 the user has specified for region 0.

Chapter 4
Caches and Write Buffer

4.1 Introduction

To reduce the effective memory access time, the ARM940T employs an *Instruction Cache* (*I Cache*), a *Data Cache* (*D Cache*) and a Write Buffer. The following sections describe the features and behavior of each of these blocks. The information in this chapter is organized as follows:

- Cache architecture
- Instruction cache
- Data cache
- The write buffer
- Cache lock down.

4.2 Cache architecture

The ARM940T uses:

- a 4KB Instruction cache
- a 4KB Data cache
- an 8-word write buffer.

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

Each cache segment consists of 64 CAM rows to select one of 64 RAM lines of four words in length. On an I Cache or D Cache access, a segment is selected and the access address is compared with the 64 TAGs in the CAM. If a match occurs, the cache has ‘hit’. The row line corresponding to the match is then enabled so the data can be accessed. If none of the row TAGs match, the access has missed. External memory must be accessed unless the access is a buffered write, in which case the write buffer is used.

If a read access from a cacheable memory region misses in the cache, one of the 64 segment row lines is selected as a target into which to load new data (allocate on read-miss replacement policy). This selection is performed by a randomly clocked target row counter. Critical or frequently accessed instructions and/or data may be locked down in the I Cache and D Cache respectively, by restricting the range of the target counter. Locked down lines are immune to replacement and remain in the cache until they are unlocked, or flushed.

Figure 4-2 4KB cache used for ARM940T instruction and data caches shows the 4KB Instruction Cache or Data Cache architecture:

- Address bits 5 to 4 select one of the four cache segments
- Bits 3 to 2 select a word in the cache line.

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 will be required again in a later iteration of the loop. The overhead for this high associativity is the need to store a larger TAG, in this case 26 bits per line. *Figure 4-1 ARM940T Instruction/Data cache address mapping* shows how the address space accesses the 4KB I Cache and 4KB D Cache.

Two additional bits are used on each segment row line:

- The *valid bit* is set once 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.
- The *dirty bit* is associated with write operations in the D Cache 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).

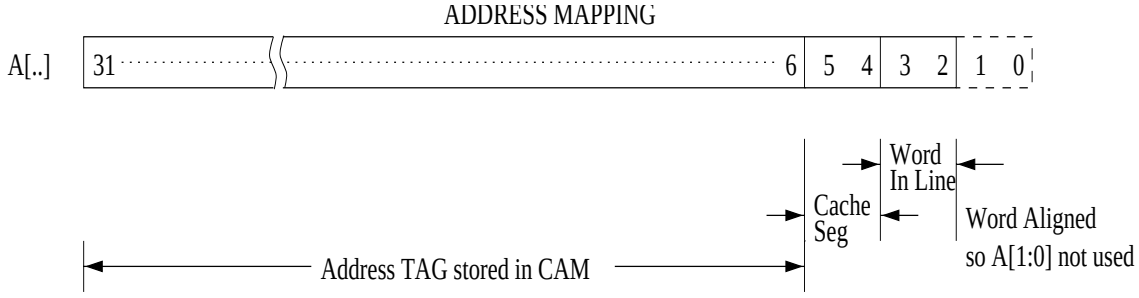

Figure 4-1 ARM940T Instruction/Data cache address mapping

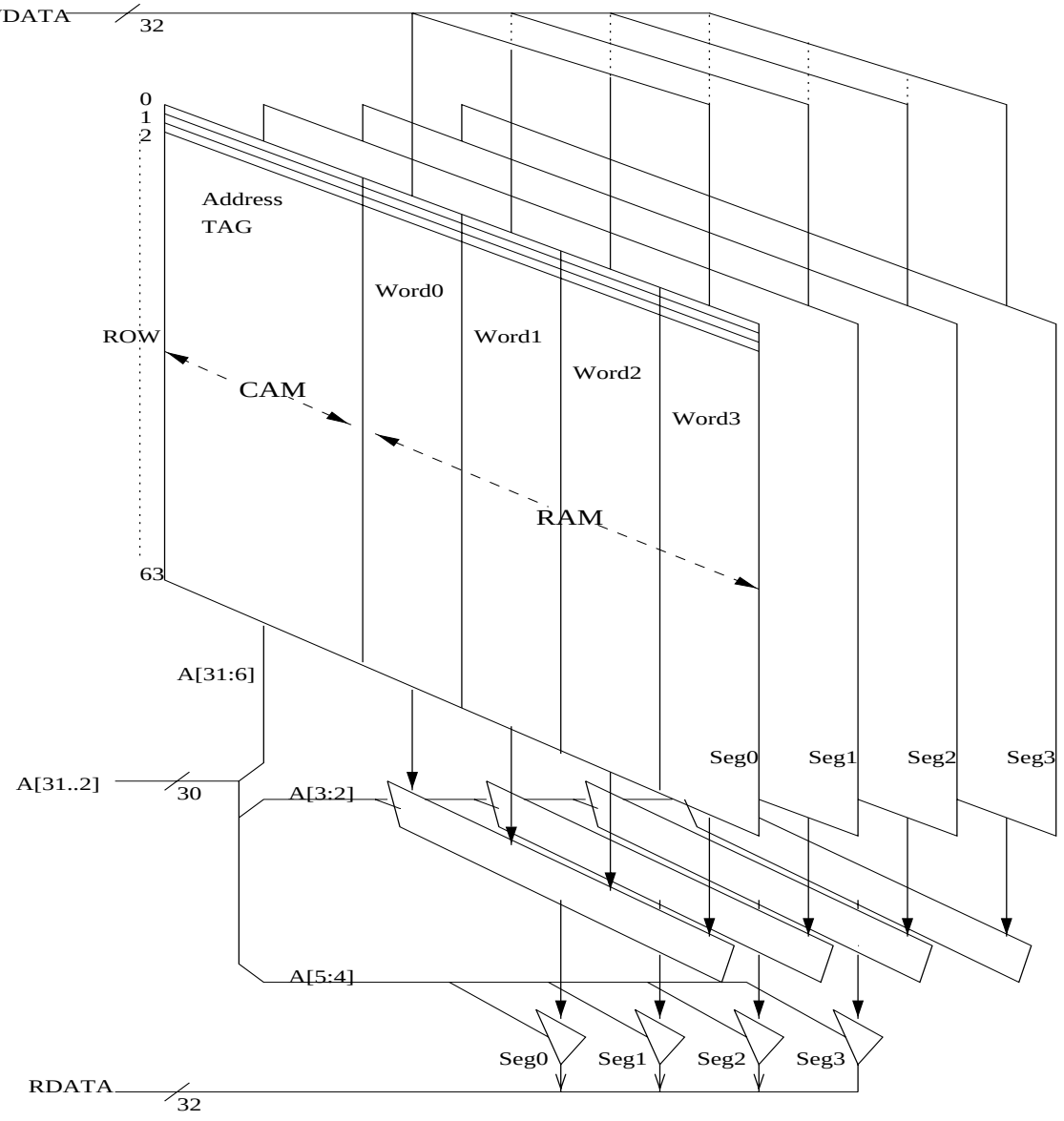

Figure 4-2 4KB cache used for ARM940T instruction and data caches

4.3 Instruction cache

The ARM940T has a 4KB *Instruction Cache* (*I Cache*) comprising 16 bytes (four words) arranged as four 64-way associative segments.

The I Cache uses the physical address generated by the processor core. It employs a policy of ‘allocate on read-miss’ and is always reloaded one cache line (four words) at a time, through the external interface.

The I Cache operation may be enabled or disabled by the CP15 control register, and is always disabled on reset. When enabled, the I Cache operation is further controlled by the (GCi) *Gated Cacheable data* bit stored in the protection unit, which selectively enables/disables caching for different memory regions. The GCi bits have the protection unit enable factored into them such that GCi = 1 only when a cacheable region is accessed AND the protection unit is enabled.

The I Cache and protection unit can be enabled with a single write to the CP15 control register, although at least one protection region should be programmed before the protection unit is enabled. Critical or frequently accessed instructions can be locked down into the I Cache with a granularity of 64 bytes.

Note

Instructions in this lockdown region are immune to replacement, and remain in the I Cache, although they are not immune to being flushed.

4.3.1 Instruction cache operation

When the I Cache is enabled, it is searched when the processor requests an instruction:

Successful cache read:

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

Unsuccessful cache read:

The GCi bit is examined:

If this bit is 1, a cacheable code area and protection unit enabled — a linefetch of four words is performed. The data is written into a randomly chosen line in the I Cache.

If this 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 I Cache searches. Lines containing locked down code cannot be selected for replacement during a linefetch.

4.3.2 Instruction cache validity

The ARM940T does not support external memory snooping. If, therefore, self-modifying code is written, the instructions in the I Cache may become invalid. Similarly, if the instruction protection regions are reprogrammed, code may exist in the cache which should now be in a non-cacheable region. In either of these cases, the I Cache must be flushed by the programmer.

The entire I Cache can be invalidated (flushed) by software in one operation, or flushed one line at a time by writing to the CP15 cache operations register (register 7). The I Cache is automatically flushed in hardware during reset. The I Cache never needs to be cleaned as its only source of data is from external memory (the processor only ever performs reads from the I Cache).

4.3.3 Flushing the entire cache

As shown in *Table 2-12 Cache operations through register 7* on page 2-11, the entire I Cache can be flushed through the use of an MCR instruction. In this case, the contents of the ARM register transferred to CP15 should be zero. The code segment shown below may be used. Note that the use of R0 is arbitrary:

```
M0V     RO, #0; Clear R0
MCR     p15, R0, c7,c5, 0; Flush entire I Cache
```

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

4.3.4 Flushing a single cache line

A single cache line may be flushed. To do this, the cache line must be specified in Rd. As the ARM940T I Cache comprises four segments, each with 64 lines, both the segment and line number index must be specified. The format of Rd for this operation is shown in *Table 4-1 CP15 Register 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 may be used to flush line 25 of segment 2 in the I Cache.

```
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 I Cache line
```

4.3.5 Instruction cache enable/disable and reset

The I Cache is enabled by setting bit 12 of the CP 15 control register. The cache is only enabled if the protection unit is already enabled, or is enabled simultaneously. When the I Cache is enabled, a cacheable read-miss causes lines to be placed in the I Cache.

The I Cache can be disabled by clearing bit 12 of the CP15 control register. This has the effect of preventing all I Cache look-ups and line fills, and forces all instruction fetches to be performed by single external accesses.

4.4 Data cache

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

The D Cache supports both *Write-back* (*WB*) and *Write-Through* (*WT*) modes. For data stores that hit in the D Cache, 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 D Cache causes the cache line to be updated but not marked as dirty, as 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 D Cache, 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 line fill replaces it.

The *Gated Cacheable Data* (*GCd*) bit and the *Gated Write Buffer Control* (*GBd*) bit control the D Cache behavior. For this reason the protection unit must be enabled when the D Cache is enabled.

4.4.1 Gated cacheable data bit

The GCd bit determines whether data being read should be placed in the D Cache and used for subsequent reads. Typically, main memory is marked as cacheable to reduce memory access time and therefore increase system performance. Input/output space is usually marked as non-cacheable. For example, if a processor is polling a hardware flag in input/output 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 D Cache.

4.4.2 Gated write buffer control bit

The GBd and GCd bits affect writes that both hit and miss in the D Cache. For details of the ways these bits are decoded to perform different types of writes, see *4.5 The write buffer* on page 4-12.

4.4.3 Data cache operation

When the D Cache 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:

Cacheable data area and protection unit enabled. A line fill of four words is performed, and the data is written into a randomly chosen line in the D Cache.

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 whether the write is buffered (see *4.5 The write buffer* on page 4-12).

Non-cacheable load multiples and *non-cacheable non-bufferable* (NCNB) store multiples are broken up on 4KB boundaries (the minimum protection region size), allowing a protection check to be performed in case the LDM or STM crosses into a region with different protection properties.

D Cache lock down is supported with 16-word granularity. Data that is locked down always hits on D Cache searches, and lines containing locked down data cannot be selected for replacement during a line fill.

Back-to-back stores from adjacent store instructions to the same segment within the D Cache 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 cycle to be performed. Single back-to-back stores to different segments are also performed without a stall, allowing one write cycle.

4.4.4 Data cache validity

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

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

4.4.5 Data cache clean and/or flush

The D Cache has flexible cleaning and flushing utilities. The whole D Cache can be invalidated (Flush D Cache) in one operation without writing back dirty data. Individual D Cache lines can also be invalidated without writing back any dirty data (Flush D Cache Single Entry). Cleaning is performed on a line-by-line basis where 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 D Cache Single Entry). Lastly, a line may be cleaned and flushed in one operation (Clean and Flush D Cache Single Entry).

Note

Flushing the entire D Cache will also flush 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 previously in *4.3 Instruction cache* on page 4-5 for I Cache. The format of Rd transferred to CP15 is as shown in *Table 4-1 CP15 Register 7* on page 4-6 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.

The code segment below shows how the entire cache can be cleaned and flushed:

```
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, 1; 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_loopIf not, branch back to outer_loop
```

4.4.6 Data cache enable/disable and reset

The D Cache is automatically disabled and flushed on reset. If the D Cache is subsequently disabled, further D Cache searches are prevented. This has the effect of making all data accesses non-cacheable and forcing the ARM940T to perform external accesses. The write buffer control is still decoded from the GBd and the GCd bit, the latter being forced to 0 (non-cacheable) when the D Cache is disabled.

Writing to the CP15 control register bit 2 enables the D Cache. This should only be done if bit 0 is already set, enabling the protection unit. These two bits can be written to at the same time, enabling the D Cache and protection unit. The D Cache can be disabled by clearing bit 2 of the CP15 control register.

4.5 The write buffer

The ARM940T provides a write buffer to increase system performance. The write buffer can buffer up to eight words of data and four separate non-sequential addresses. On reset, the buffer is flushed.

Write buffer behavior is controlled by the protection region attributes of the store being performed, and the D Cache and protection unit enable status. This control is represented by the following bits:

GCd bit — Gated Cacheable Data (GCd) bit. The GCd bit is generated from the cacheable attribute of the protection region AND the D Cache enable AND the protection unit enable.

GBd bit — Gated Write Buffer Control (GBd) bit. The GBd bit is generated from the bufferable attribute of the protection region AND the protection unit enable.

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

4.5.1 Write buffer operation

The write buffer is used when the D Cache hits and/or misses, depending on the mode of operation. *Table 4-2 Data write modes* shows how the GCd and GBd bits determine the behavior of the write buffer:

Table 4-2 Data write modes

GCd	GBd	Access mode	
0	0	NCNB	Non-cacheable, non-bufferable
0	1	NCB	Non-cacheable, bufferable
1	0	WT	Write-through
1	1	WB	Write-back

NCNB
: Data reads and writes are not cached, and may be externally aborted. Writes are not buffered; the processor is stalled during the external access.

NCB
: Data reads and writes are not cached. Writes are buffered, and so cannot be externally aborted. Reads can be externally aborted.

  If the D Cache hits for this type of access, there has been a programming error. This error is treated like a write-through, in that the D Cache line is updated and the data is buffered.

  Swap instructions operation on data in an NCB region are made to perform NCNB type accesses and are *not* buffered.

WT
: Searches the D Cache for reads and writes. Reads which miss in the D Cache cause a line fill. Reads which hit in the D Cache do not perform an external access. All writes are buffered, regardless of whether they hit or miss in the D Cache. Writes which hit in the D Cache update the cache but do not mark the cache line as dirty, as the write is also sent to the Write Buffer. Writes cannot be externally aborted.

WB
: Searches the D Cache for reads and writes. Reads which miss in the D Cache cause a line fill. Reads which hit in the D Cache do not perform an external access. Writes which miss in the D Cache are buffered. Writes which hit in the D Cache update the cache line, mark it as dirty, and do not send the data to the write buffer. D Cache write-backs are buffered. Writes (write-miss and write-back) may not be externally aborted.

4.5.2 Enabling/disabling the write buffer

The write buffer cannot be directly enabled or disabled. However, setting the properties of a memory region to be NCNB or disabling the protection unit prevents the write buffer being used.

4.6 Cache lock down

To provide a predictable code behavior in embedded systems, a mechanism for locking code and data into the Instruction and Data caches respectively is provided. This feature may be used, for example, to hold high-priority interrupt routines where there is a hard real-time constraint, or to hold the co-efficients of a DSP filter routine in order to reduce external bus traffic.

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

- the program should be held in a non-cached area of memory
- the cache should be enabled and interrupts should 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 since the last reset, the software must ensure that the cache in question is cleaned, if appropriate, and then flushed.

Lock down in the D Cache is achieved through use of CP15 register 9. I Cache lock down uses both CP15 registers 7 and 9.

As described in *4.2 Cache architecture* on page 4-2, the ARM940T I and D Caches comprise four segments, each with 64 lines of four words each. Each segment is 1KB in size. Lock down can be performed with a granularity of one line across each of the four segments; the smallest space which may be locked down is 16 words. Lock down starts at line zero, and can continue until 63 of the 64 lines are locked.

4.6.1 Locking down the caches

The procedure for locking down a line in the I Cache and the D Cache are slightly different. In both cases:

1. The cache must be put into lock down mode by programming register 9.
2. A line fill 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 lock down

For the D Cache, 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 bank 1.
5. Execute an LDR from that location. The resulting linefill is captured in cache bank 2.
6. Repeat steps 1 to 5 for cache banks 3 and 4.
7. Write to CP15 register 9, setting DL=0 and Dindex=1.

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

Instruction cache lock down

For the I Cache, 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 lock down.
3. Force a line fill from that location by writing to CP15 register 7.
4. Increment the pointer by 16 to select cache segment 1.
5. Force a line fill from that location by writing to CP15 register 7. The resulting line fill 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 were more data to lock down, at the final step 7, the IL bit should be left HIGH, Iindex increment by 1 line and the process repeated. The IL bit should be set LOW when all the lock down data had been loaded.

Performing lock down in the I Cache involves a similar sequence of operations, except that the IL and Iindex of CP15 register 9 are accessed.

The only significant difference in the sequence of operations is that an MCR instruction must be used to force the line fill in the I Cache, instead of an LDR, This is due to 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. As this misses in the cache (due to earlier flushing), a line fill occurs.

The rest of the sequence of operations is exactly the same as for D Cache lock down.

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

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

A macro used to lock down code in the instruction cache is given below:

```
;       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 non-cacheable
;       region of memory in order to work, or these instructions
;       themselves will be locked into the cache. Interrupts should also
;       be disabled.
;       The subroutine should be called via 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
        BICR1, R1, #0x3f; align address to cache line
        MRCp15, 0, R3, c9, c0, 1 ; get current instruction cache index
        ANDR2, R2, #0x3f; mask off unwanted bits
        ADDR3, 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
        ORR2, R2, #0x80000000; set lock bit, r2 contains the cache
        ; line number to lock down
lock_loop
        MCR p15, 0, R2, c9, c0, 1 ; write lock down register
        MCR p15, 0, R1, c7, c13, 1; force line fetch from external
memory
        ADD R1, R1, #16; add 4 words to address
        MCRp15, 0, R1, c7, c13, 1; force line fetch from external memory
        ADDR1, R1, #16; add 4 words to address
        MCRp15, 0, R1, c7, c13, 1; force line fetch from external memory
        ADDR1, R1, #16; add 4 words to address
        MCRp15, 0, R1, c7, c13, 1; force line fetch from external memory
        ADDR1, R1, #16; add 4 words to address

        ADDR2, R2, #0x1; increment cache line in lock down
        ; register
        SUBSR0, R0, #0x1; decrement line count and set flags
        BNElock_loop; if r0! = 0 then branch round

        BICR0, R2, #0x80000000; clear lock bit in lockdown register
```



```
        MCRp15, 0, R0, c9, c0, 1; restrict victim counter to lines
        ; r0 to 63

        LDMFD R13!, {R1-R3}; restore corrupted registers and return
        MOVPC, LR; R0 contains the first free cache line
        ; number
error
        LDRR0, =0; make r0 = 0 to indicate error
        LDMFD R13!, {R1-R3}; restore corrupted registers and return
        MOVPC, LR
```


Chapter 5
Clock Modes

5.1 Introduction

This chapter describes the different clock modes available on the ARM940T. The information in this chapter is organized as follows:

- Overview
- FastBus mode
- Sychronous mode
- Asynchronous mode.

5.2 Overview

The ARM940T has two main clock inputs **BCLK** and **FCLK,** which allow flexible clocking configurations. There are three different modes of operation, selected using bits 30 and 31 of CP15 register 1, the control register. The three modes are FastBus, Synchronous and Asynchronous. **ECLK** and **CPCLK** reflect which clock is currently selected.

The ARM940T is a pseudo-static design and both clocks can be stopped. Typically when accessing slow memory systems or peripherals, wait states will be applied using the **BWAIT** signal, refer to the *AMBA Specification* for more details.

5.3 FastBus mode

In this mode of operation the **BCLK** input is used for controlling the internal ARM9TDMI, cache operations and the external memory interface. The **FCLK** input is ignored. This mode is typically used in systems with high speed memory.

5.4 Sychronous mode

This mode is typically used in systems with low speed memory. In this mode both the **BCLK** and **FCLK** inputs are used. **BCLK** is used to control the AMBA memory interface. **FCLK** is used to control the internal ARM9TDMI processor core and any cache operations. **FCLK** must have a higher frequency and must also be an integer multiple of **BCLK,** with a **BCLK** transition only when **FCLK** is HIGH. An example is shown in *Figure 5-1 Sychronous clocking mode:*

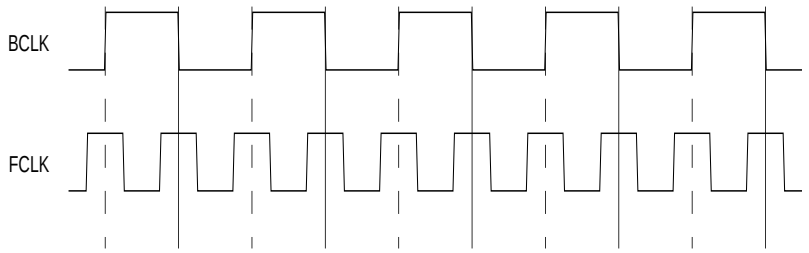

Figure 5-1 Sychronous clocking mode

If the ARM940T performs an external access, for example, a cache miss or a cache line fill, the ARM940T will switch to **BCLK** to perform the access. The delay when switching from **FCLK** to **BCLK** is a minimum of one **FCLK** phase and a maximum of one **BCLK** cycle. An example of the clock switching is shown in *Figure 5-2 Switching from FCLK to BCLK in sychronous mode*. The delay when switching from **BCLK** to **FCLK** is a maximum of one **FCLK** phase.

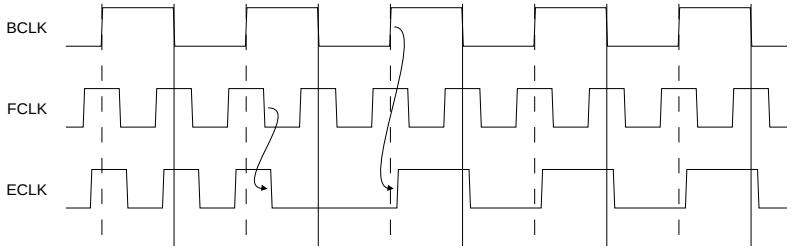

Figure 5-2 Switching from FCLK to BCLK in sychronous mode

Care must be taken if **BCLK** is stopped by the system so that when **BCLK** is restarted it does not violate any of the above restrictions.

5.5 Asynchronous mode

This mode is typically used in systems with low speed memory. In this mode of operation both the **BCLK** and **FCLK** inputs are used. **BCLK** is used to control the AMBA memory interface. **FCLK** is used to control the internal ARM9TDMI processor core and any cache operations. The one restriction is that **FCLK** must have a higher frequency than **BCLK**. An example is shown in *Figure 5-3 Asynchronous clocking mode*:

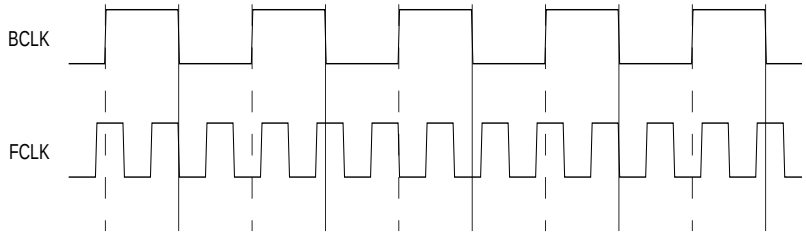

Figure 5-3 Asynchronous clocking mode

If the ARM940T performs an external access, for example, a cache miss or a cache line fill, ARM940T will switch to **BCLK** to perform the access. The delay when switching from **FCLK** and **BCLK** is a minimum of one **BCLK** cycle, and a maximum of one and a half **BCLK** cycles. An example of the clock switching is shown in *Figure 5-3 Asynchronous clocking mode*. When switching from **BCLK** to **FCLK** the minimum delay is one **FCLK** cycle and the maximum delay is one and a half **FCLK** cycles. An example of the clock switching is shown in *Figure 5-4 Switching from FCLK to BCLK in asynchronous mode*:

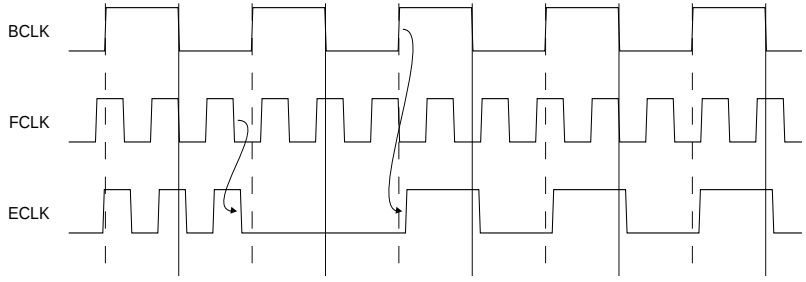

Figure 5-4 Switching from FCLK to BCLK in asynchronous mode

Chapter 6
Bus Interface Unit

6.1 Introduction

The ARM940T has an *Advanced Microprocessor Bus Architecture* (*AMBA*) interface. This chapter describes the different type of behavior on this interface.

6.2 ASB transfers

When accessing the *Advanced System bus* (*ASB*), the ARM940T does not use the non-sequential transfer. Instead, an address-only transfer, followed by a sequential transfer, is used. This eases the AMBA decoder design considerably, particularly for high speed designs.

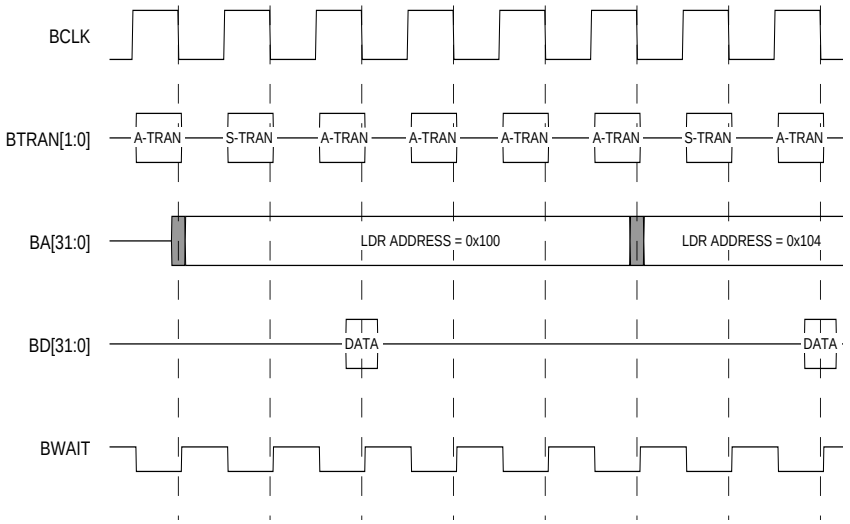

Figure 6-1 Sequential LDR accesses

Figure 6-1 Sequential LDR accesses shows the ARM940T ASB activity when two LDR instructions are executed. In this example, the LDR instructions are accessing a non-cacheable region of memory. As can be seen, there are two sets of address-only transfers, followed by sequential transfers, even though the two addresses are sequentially related.

6.3 Burst accesses

To help implement an efficient memory system, the ARM940T supports *burst transfers*. Burst transfers are used for cache line fills, and for buffered writes caused by cache lines that have been evicted or cleaned. In each case, a transfer of four words will take place.

The bus **BURST[1:0]** indicates when a transfer of four words is going to take place. **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.

The code of **BURST[1:0]** is shown below:

Table 6-1 BURST[1:0] encoding

BURST [1]	[0]	Transfer
0	0	No sequential information available (default)
0	1	Reserved
1	0	Current access is part of a 4-word transfer
1	1	Reserved

Figure 6-2 shows a cache line fill followed by a buffered write where a cache line has been evicted.

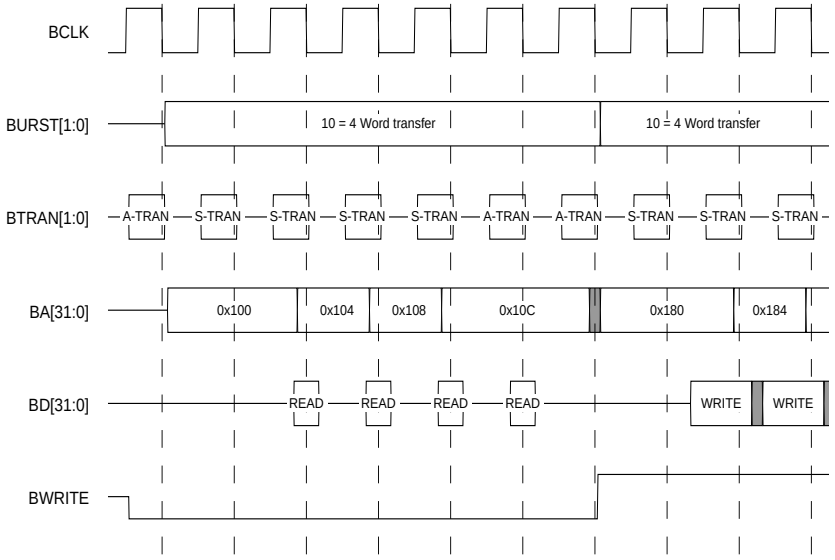

Figure 6-2 Cache line fill

Note

A cache line can only be evicted from the D Cache when a protection region is marked as a writeback area, and the dirty bit of the line has been set.

BURST[1:0] only indicates a four-word transfer when either a cache line fill takes place, or when a line within a writeback 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.

Cache line fills are performed by reading four words of data aligned to a 4-word boundary. The word of data aligned onto the 4-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 may also be streamed to the ARM9TDMI.

6.4 Buffered writes

The write buffer buffers eight words of data at up to four non-related addresses. The write buffer is used for memory which is marked as one of the following:

- A non-cacheable, buffered region (NCB)
- A writeback region (WB)
- A write-through region (WT).

Refer to section *4.5.1 Write buffer operation* on page 4-13.

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

The write buffer will split any accesses caused by a STM instruction on 4-word boundaries. Each set of words will use one address location within the write buffer. This mechanism allows privileges to be rechecked in the instance where the access crosses a memory region and the memory region privileges may change, therefore protecting any regions of reserved memory.

Figure 6-3 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
```


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. A further store to address 0x20c is then executed using the remaining address location. The internal ARM9TDMI is then stalled until an address register becomes free.

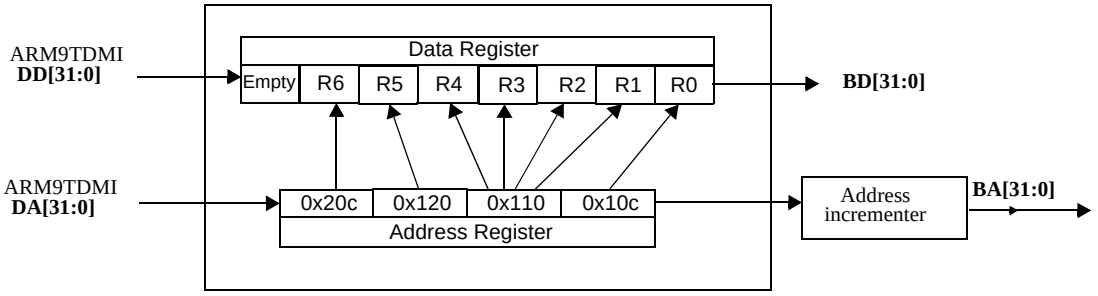

Figure 6-3 Write buffer allocation

Note

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

6.5 LDM operations from a non-cached region

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 non-cacheable region of memory, a 16-word sequential load will take place on the AMBA interface. If the access crosses a 4KB boundary, the access will be split. This allows the region properties to be checked in the instance where there is a transition between memory protection regions. *Figure 6-4 LDM operation* on page 6-6 shows a LDM operation crossing a 4KB boundary.

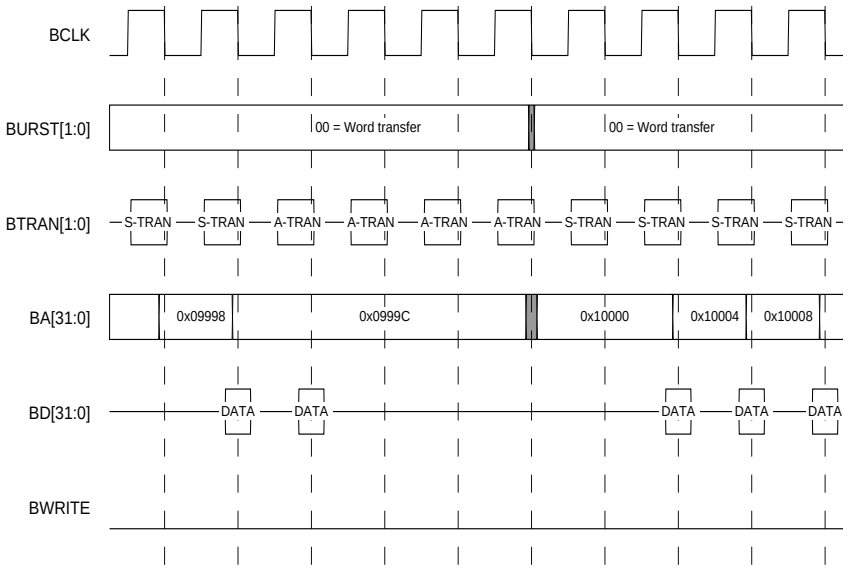

Figure 6-4 LDM operation

Note

In *Figure 6-4***,** 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.

6.6 STM operation to a non-cached region

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 a non-cacheable region of memory, a 16-word sequential write will take place on the AMBA interface. If the access crosses a 4KB boundary, the access will be split. This allows the region properties to be checked in the case where there is a transition between memory regions. *Figure 6-5* show an STM operation crossing a 4KB boundary:

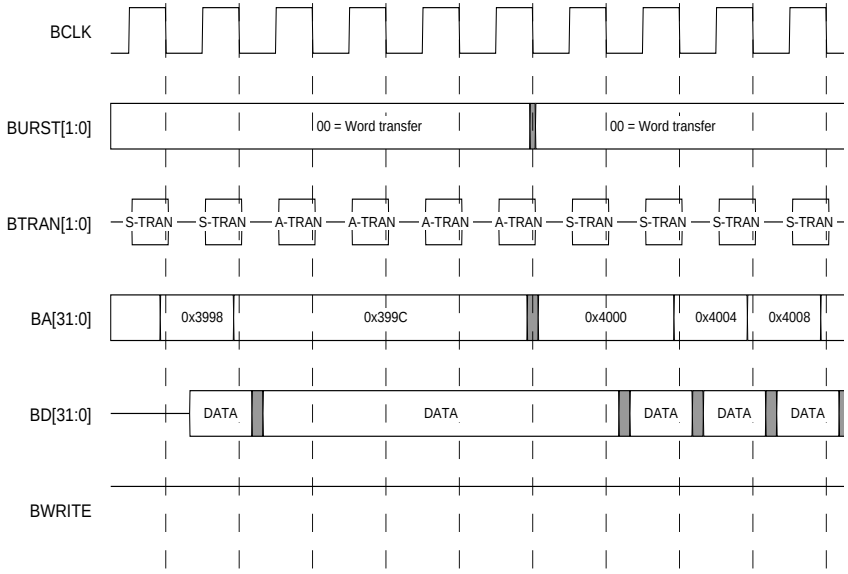

Figure 6-5 STM operation

Note

In *Figure 6-5*, **BURST[1:0]** bus is only indicating word transfers during the STM operation.

6.7 External aborts

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

6.8 SWP instruction

The *Swap* (*SWP*) instruction results in a read operation being followed by a write 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 a *non-cacheable non-bufferable* (*NCNB*) or *non-cacheable bufferable* (*NCB*) region of memory. When an SWP instruction is executed, any cache line fills will complete and the write buffer will drain before the SWP instruction memory accesses take place. During the SWP access, the **BLOK** signal will go HIGH to indicate that the two memory accesses are indivisible.

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

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

If a region of memory is changed from being cacheable to non-cacheable 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 and should be avoided.

6.9 Memory access order

If a simultaneous data access and instruction fetch both cause cache misses, the data access will take precedence and be completed first. Typically, instructions tend to require frequent sequential accesses and data requires infrequent non-sequential accesses. This type of behavior results in more efficient ASB usage, and improves the chances at streaming line fill words to the ARM9TDMI core.

Figure 6-6 shows how misses in both the I Cache and D Cache result in external access, with the data access taking place first, followed by the instruction fetch.

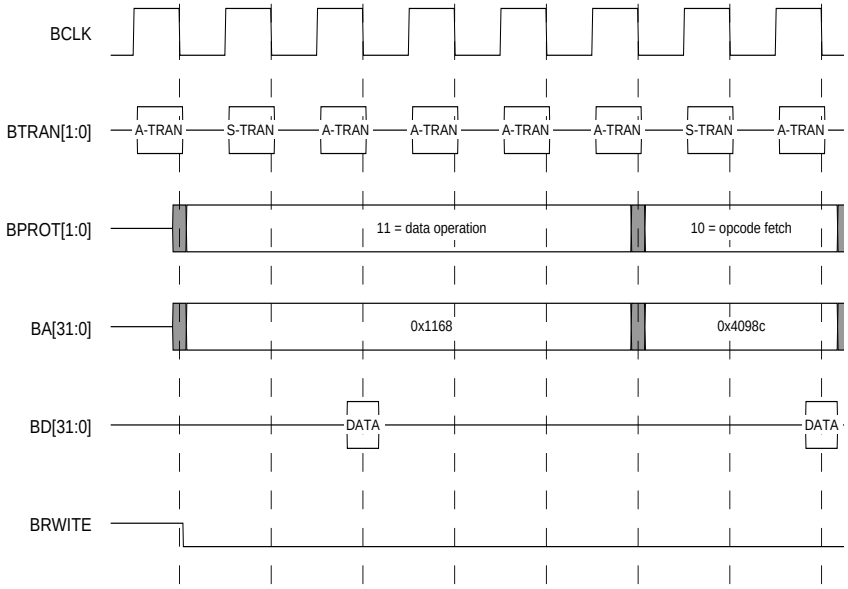

Figure 6-6 Simultaneous cache misses

Chapter 7
ARM940T Coprocessor Interface

The ARM940T supports the connection of on-chip coprocessors through an external coprocessor interface. All types of coprocessor instruction are supported. This chapter describes the ARM940T coprocessor interface:

- Overview
- LDC/STC
- MCR/MRC
- Interlocked MCR
- CDP
- Privileged instructions
- Busy-waiting and interrupts.

7.1 Overview

The ARM940T coprocessor interface allows specially designed coprocessor hardware to be attached to the ARM940T. Example uses include:

- Attachment of accelerators for floating point math, DSP, 3-D graphics, encryption or decryption
- The ARM instruction set supports the connection of 16 coprocessors, numbered 0 to 15, to an ARM processor.

7.1.1 Internal coprocessors

The ARM940T contains two internal coprocessors; CP14 for debug control and CP15 for cache and protection unit control. This means that coprocessors attached externally to the ARM940T cannot be assigned coprocessor numbers 15 or 14. Some other coprocessor numbers have been allocated by ARM for internal usage. Please contact ARM for a full list of reserved coprocessor numbers.

The register map of CP15 is described in *2.2 ARM940T CP15 registers* on page 2-2. The functionality of CP14 is described in *8.16 Debug communications channel* on page 8-45.

7.1.2 External coprocessors

Coprocessors determine which instructions they need to execute by using a *pipeline follower* in the coprocessor. As each instruction arrives from memory, it enters both the ARM pipeline and the coprocessor's pipeline. To avoid a critical path for the instruction being latched by the coprocessor, the coprocessor pipeline should operate one clock phase behind the ARM940T pipeline. The ARM940T then informs the coprocessor when instructions move from decode into execute, and whether the instruction needs to be executed.

To enable coprocessors to continue execution of coprocessor data operations while the ARM940T pipeline is stalled (for instance waiting for a cache line fill to occur), the coprocessor should 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 should not advance.

Figure 7-1 ARM940T coprocessor clocking indicates the timing for these signals and when the coprocessor pipeline should 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 which reflects the ARM9TDMI pipeline advancing.

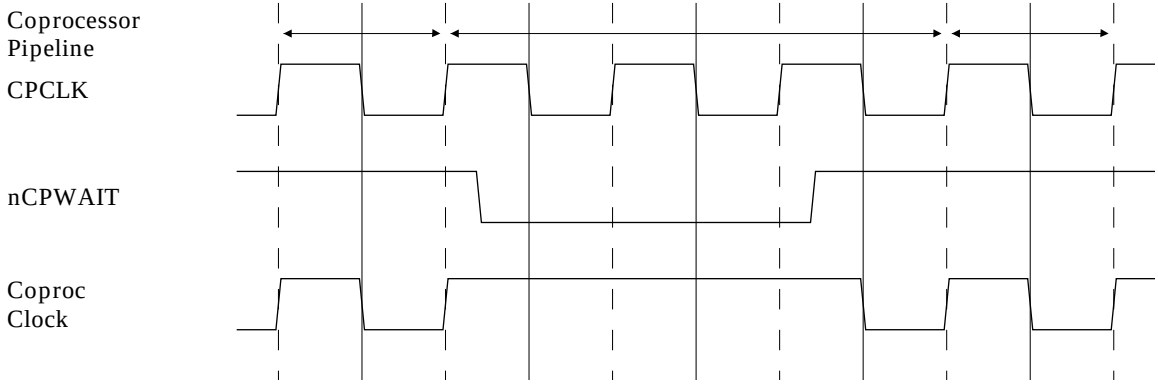

Figure 7-1 ARM940T coprocessor clocking

Coprocessor instructions

These are three classes of coprocessor instructions:

LDC/STC	Load/Store from/to coprocessor register to memory.
MCR/MRC	Register transfer between coprocessor and ARM processor core.
CDP	Coprocessor data operation.

The remainder of this chapter gives examples of how a coprocessor should execute these instruction classes.

7.2 LDC/STC

The cycle timing for this operation is shown in *Figure 7-2*.

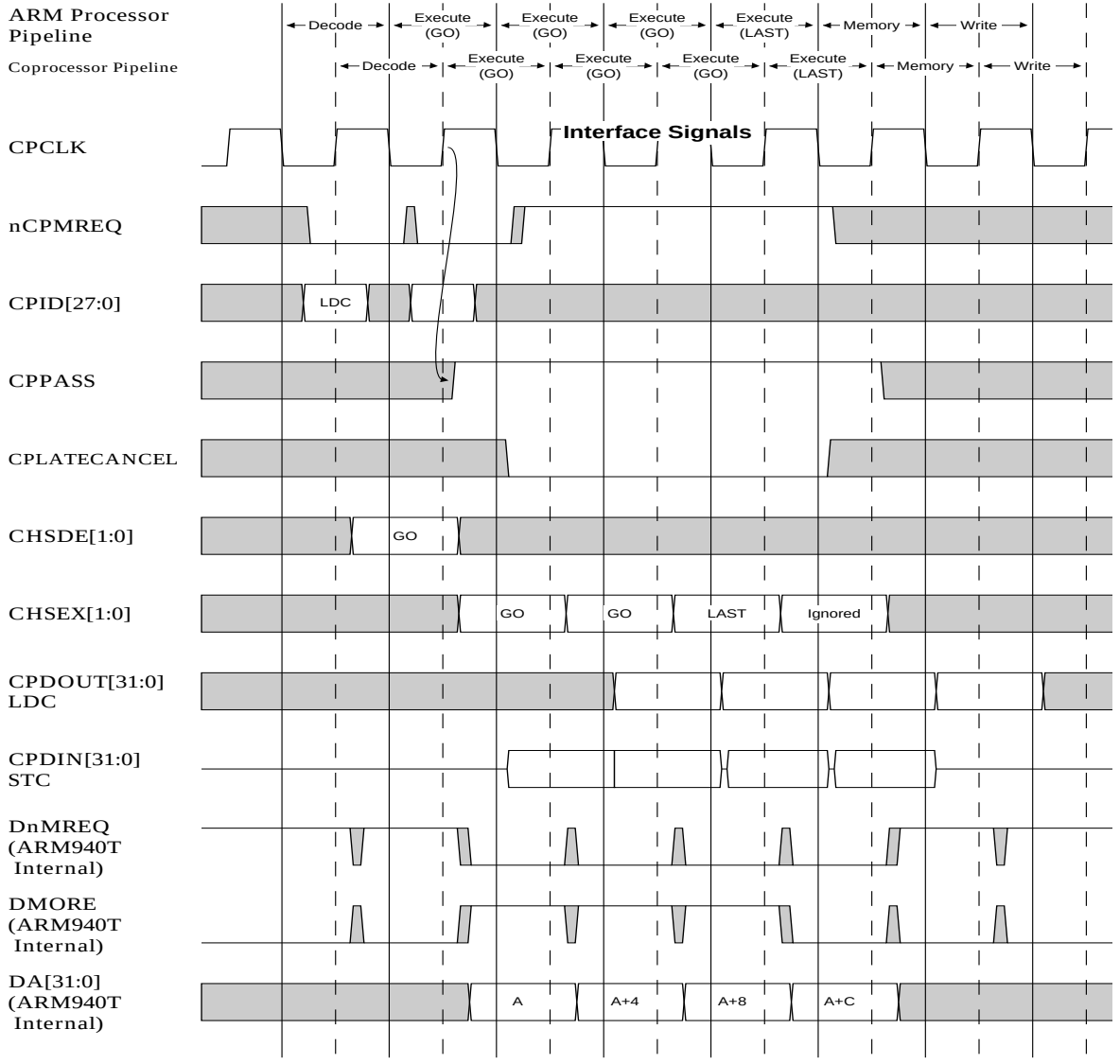

Figure 7-2 ARM940T LDC / STC cycle timing

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's instruction pipeline should keep in step with the ARM940T by monitoring **CPMREQ**, a latched copy of the ARM9TDMI instruction memory request signal **nIMREQ**. Whenever **nCPMREQ** is LOW, an instruction fetch is occurring and **CPID** will be updated with fetched instruction in the next cycle. This means that the instruction currently on **CPID** should enter the decode stage of the coprocessor pipeline, and that the instruction in the decode stage of the coprocessor's pipeline should enter its execute stage.

During the execute stage, the condition codes are combined with the flags to determine whether the instruction should 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 should stop execution of the coprocessor instruction.

A further 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** will only be asserted during the first memory cycle of a coprocessor instruction's 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 which can execute the coprocessor instruction, the handshake signals indicate the ABSENT state. In this case, the ARM9TDMI 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 should be driven to indicate that the ARM9TDMI processor core should stall until the coprocessor can catch up. This is known as the '*busy-wait*' condition.

In this case, the ARM9TDMI 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 will be taken.
If **CHSEX[1:0]** changes to GO or LAST, the instruction will proceed as described below.

If an interrupt occurs, the ARM9TDMI processor core is forced out of the busy-wait state. This is indicated to the coprocessor by the **CPPASS** signal going LOW. The instruction will be restarted at a later date and so the coprocessor must not commit to the instruction (change any of the coprocessor states) until it has seen **CPPASS** HIGH and when the handshake signals indicate the GO or LAST condition.

GO

The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires another cycle of execution. Both the ARM9TDMI 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 should drive the handshake signals with GO when two or more words still need to be transferred. When only one further word is required, the coprocessor should drive the handshake signals with the LAST condition.

In phase 2 of the execute stage, the ARM9TDMI 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 ARM940T LDC / STC cycle timing* on page 7-4.

LAST

An LDC or STC can be used for more than one item of data. If this is the case, possibly after busy waiting, the coprocessor should 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 was only one transfer, the sequence would be [WAIT,[WAIT,...]],LAST.

7.2.1 Coprocessor Handshake Encoding

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

Table 7-1 Handshake encoding

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

If a coprocessor is not attached to the ARM940T, then the handshake signals must be driven with “10” ABSENT.

If multiple coprocessors are to be attached to the interface, the handshaking signals can be combined by ANDing bit 1, and ORing bit 0. In the case of two coprocessors which have handshaking signals **CHSDE1**, **CHSEX1** and **CHSDE2**, **CHSEX2** respectively:

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

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

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

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

7.3 MCR/MRC

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

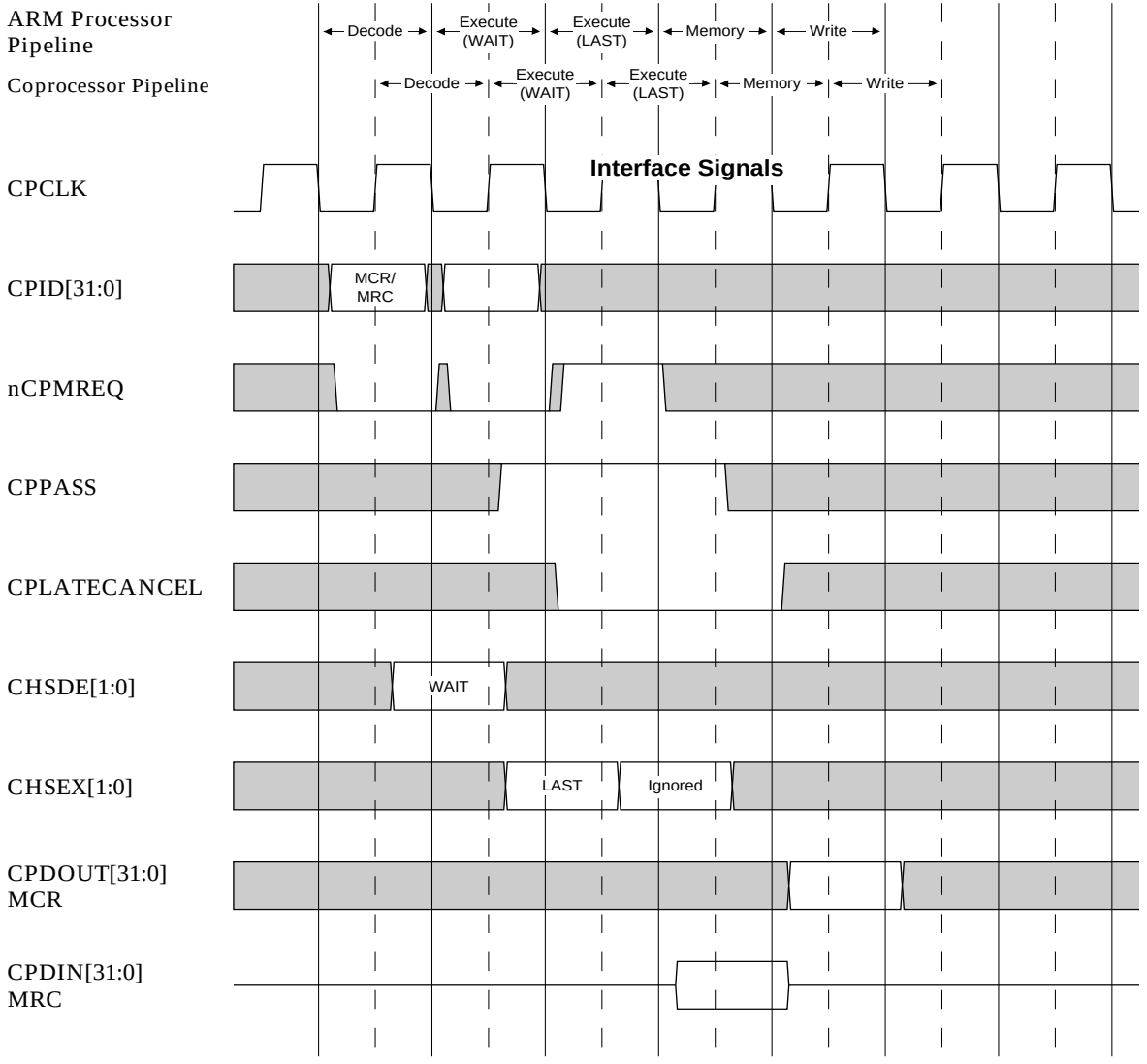

Figure 7-3 ARM940T MCR / MRC transfer timing

First **nCPMREQ** is driven LOW to denote that the instruction on **CPID** is entering the decode stage of the pipeline. This should cause 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.

7.4 Interlocked MCR

If the data for an MCR operation is not available inside the ARM9TDMI pipeline during its first decode cycle, the ARM940T pipeline interlock 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 will enter the decode stage of the coprocessor pipeline, and remain there for a number of cycles before entering the execute stage. *Figure 7-4* gives an example of an interlocked MCR.

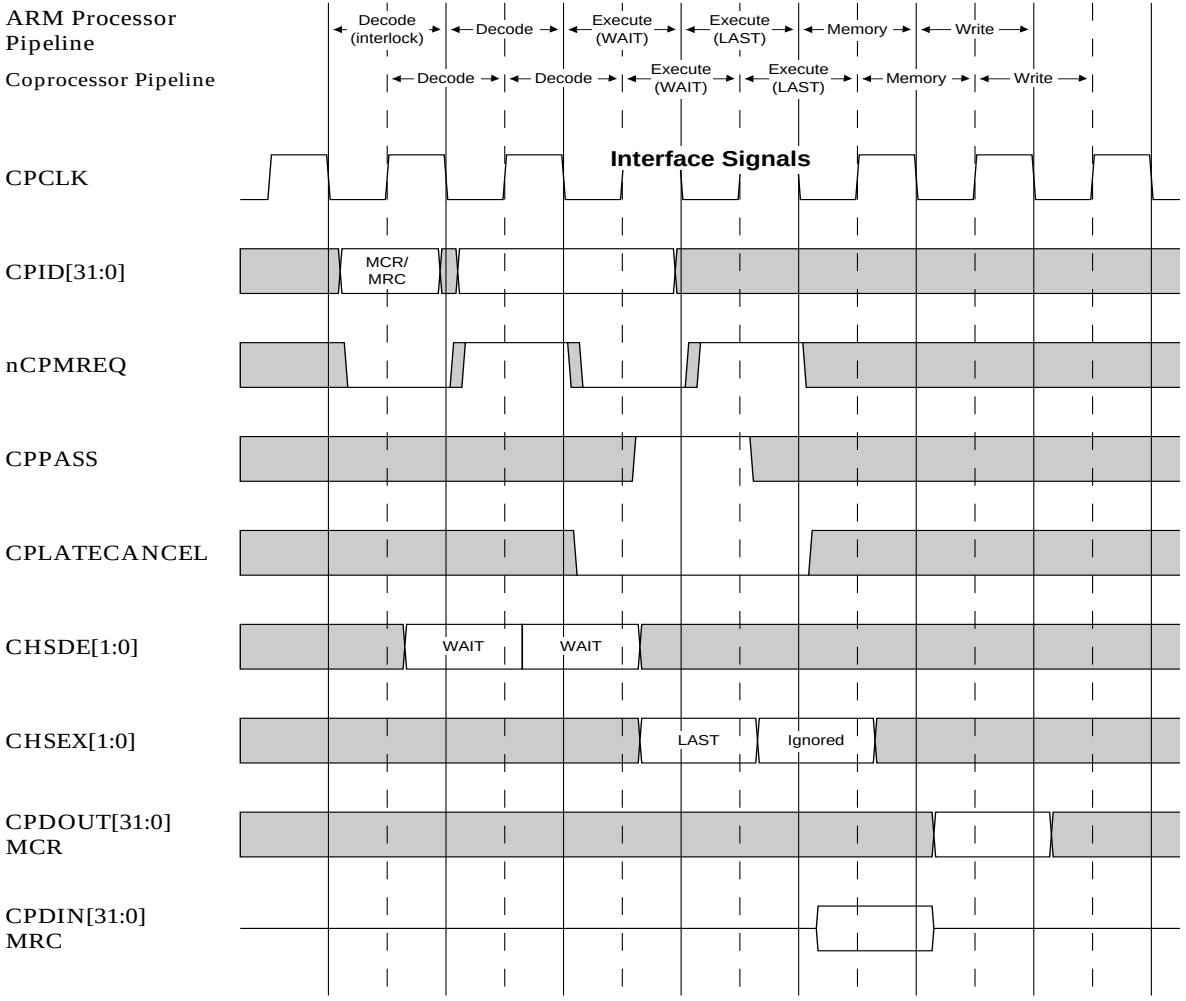

Figure 7-4 ARM940T interlocked MCR

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 ARM940T late cancelled CDP on page 7-12 shows a CDP which is cancelled due to the previous instruction causing 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 for it to cause no state changes to the coprocessor.

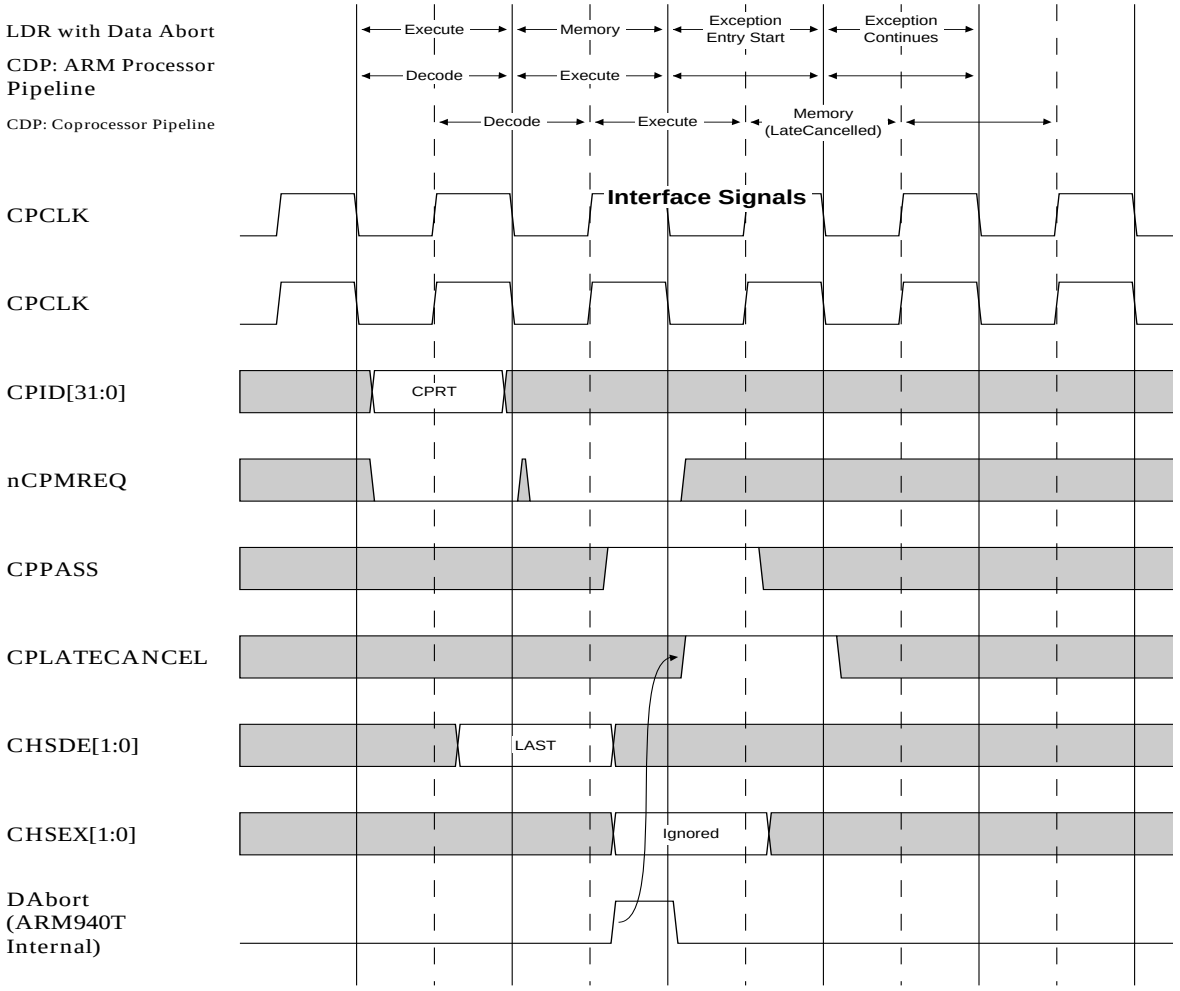

Figure 7-5 ARM940T late cancelled CDP

7.6 Privileged instructions

The coprocessor may restrict certain instructions for use in privileged modes only. To do this, the coprocessor should track the **nCPTRANS** output. *Figure 7-6* shows how **nCPTRANS** changes after a mode change.

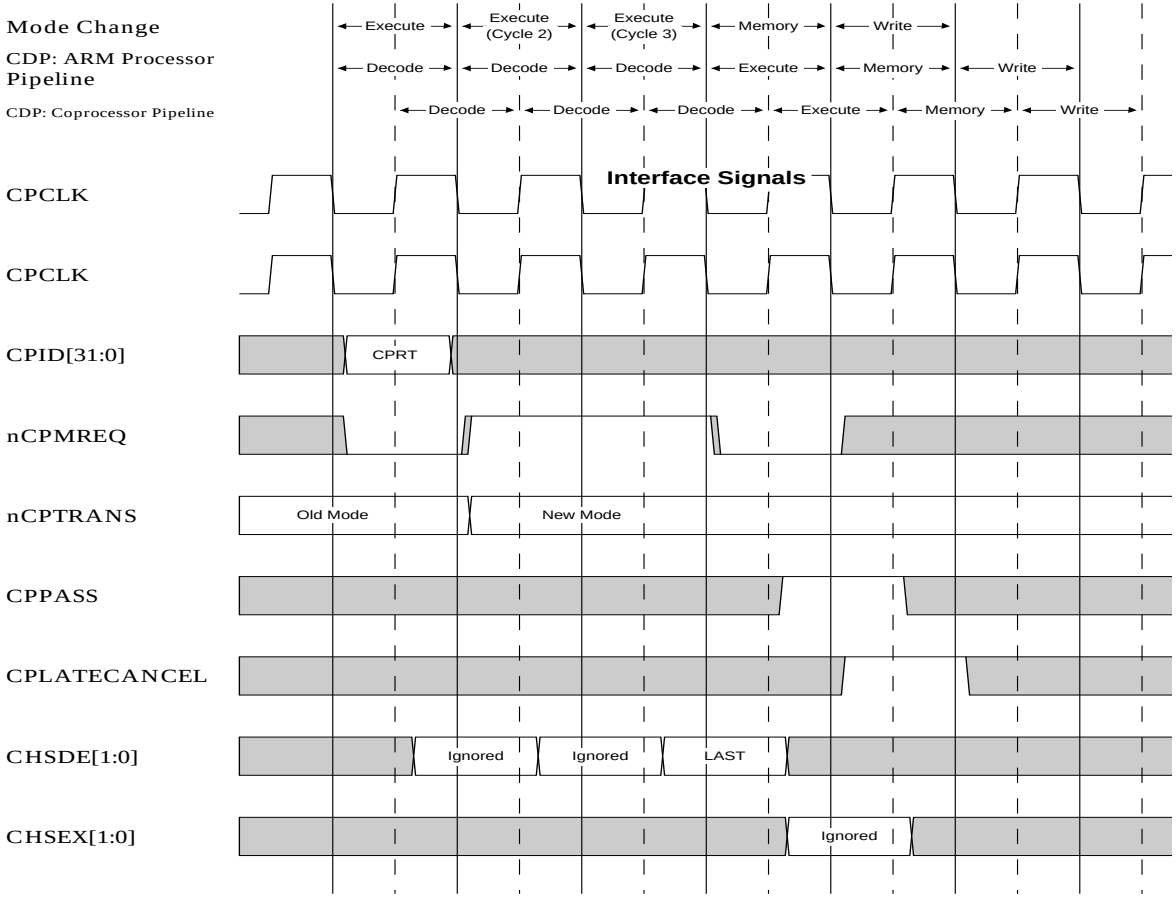

Figure 7-6 ARM940T privileged instructions

Note

The first two **CHSDE** responses are ignored by the ARM940T since 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 should drive WAIT in **CHSDE[1:0]**. When the instruction concerned enters the execute stage of the pipeline, the coprocessor may 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 may be interrupted while busy-waiting, causing the instruction to be abandoned. Abandoning execution is done through **CPPASS**. The coprocessor should monitor the state of **CPPASS** during every busy-wait cycle. If it is HIGH, the instruction should still be executed. If it is LOW, the instruction should be abandoned. *Figure 7-7* shows a busy-waited coprocessor instruction being abandoned due to an interrupt.

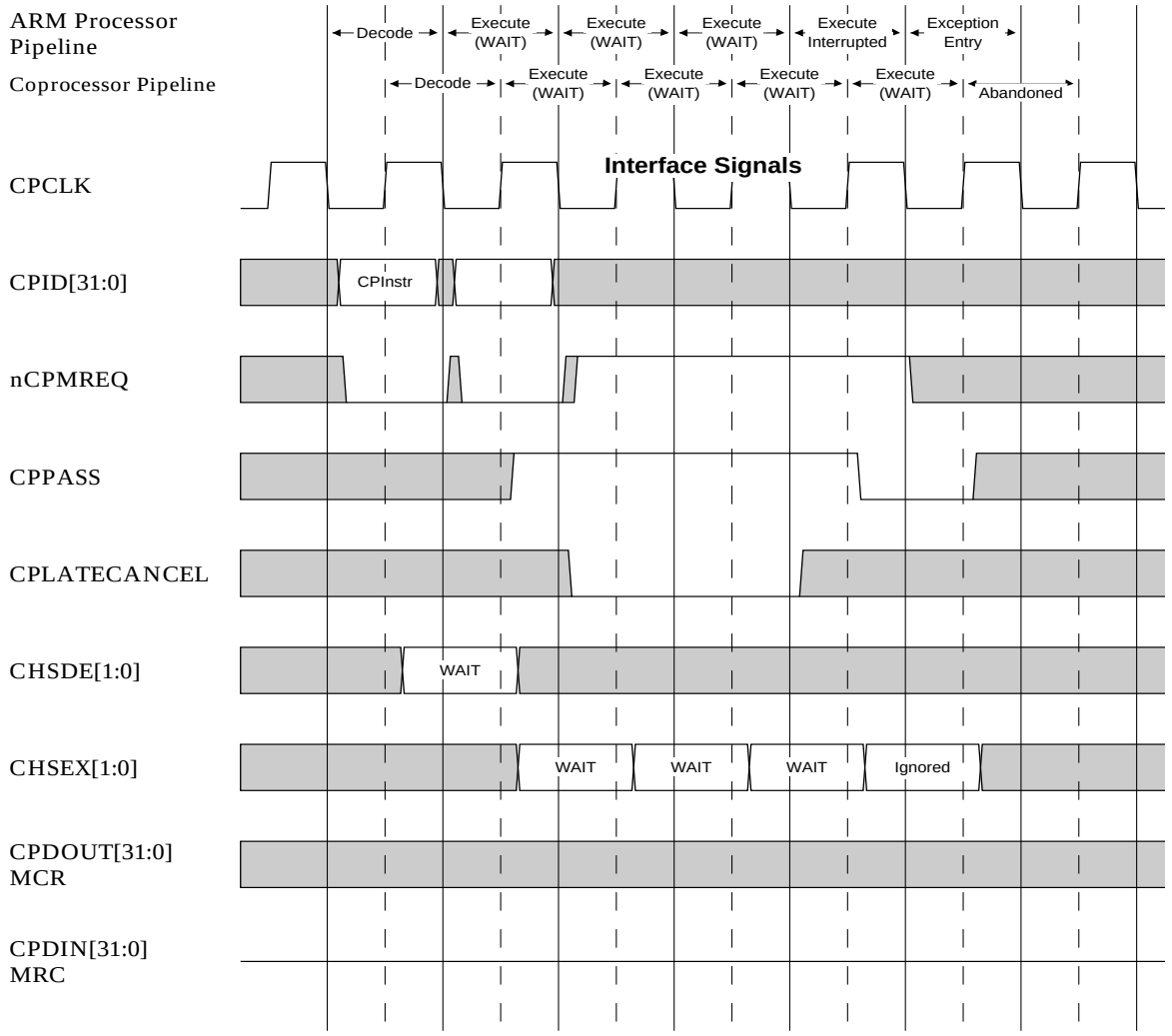

Figure 7-7 ARM940T busy waiting and interrupts

Chapter 8
Debug Support

This chapter describes the debug support for the ARM940T, including the EmbeddedICE hardware:

- Overview
- Debug systems
- Debug interface signals
- Scan chains and JTAG interface
- The JTAG state machine
- Test data registers
- ARM940T core clocks
- Clock switching during debug
- Clock switching during test
- Determining the core and system state
- Exit from debug state
- The PC’s behavior during debug
- EmbeddedICE
- Vector catching

- Single stepping
- Debug communications channel
- The debugger's view of the cache.

8.1 Overview

Debug support is implemented by using the ARM9TDMI processor core embedded within the ARM940T. Throughout this chapter therefore, ARM9TDMI refers to this core.

The ARM940T debug interface is based on IEEE Std. 1149.1- 1990, "*Standard Test Access Port and Boundary-Scan Architecture"*. Please refer to 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. These are intended to ease the user's development of application software, operating systems, and the hardware itself.

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 core's internal state and the system's external state may be examined. Once examination is complete, the core and system state may 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 macrocell*. Once 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.

The ARM940T internal state is examined via a JTAG-style serial interface, which allows instructions to be serially inserted into the core's pipeline without using the external data bus. Thus, when in debug state, a store-multiple (STM) could be inserted into the instruction pipeline, and this would 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. Such a system typically has three parts:

- the debug host

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

- the protocol converter

  The debug host is connected to the ARM940T development system via 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).

- the ARM940T

  The ARM940T, with hardware extensions to ease debugging, is the lowest level of the system. The debug extensions allow the user to stall the core from program execution, examine its internal state and the state of the memory system, and then resume program execution.

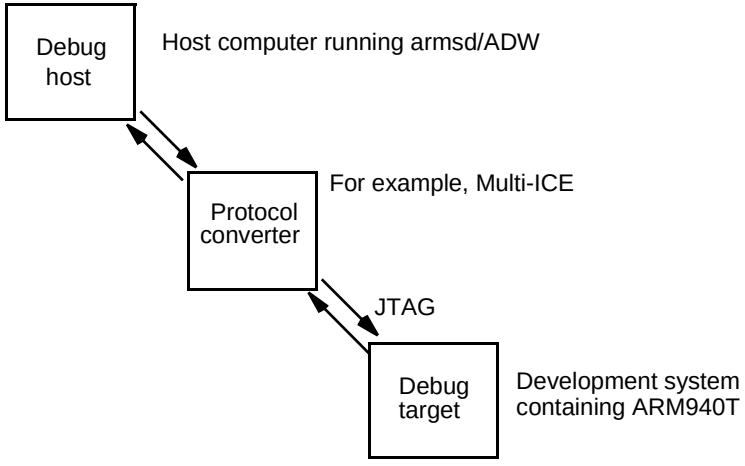

Figure 8-1 Typical debug system

The debug host and the protocol converter are system dependent. The remainder of this chapter describes the hardware debug extensions of ARM940T.

8.3 Debug interface signals

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

- **IEBKPT**, **DEWPT**, and **EDBGRQ**, with which the system requests the ARM9TDMI to enter debug state
- **DBGACK**, which the ARM9TDMI uses to flag back to the system when it is in debug state.

8.3.1 Entry into debug state on breakpoint

Any instruction being fetched for memory is latched at the end of phase 2. 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 hardware an entire phase in which to perform the comparison. This is shown in *Figure 8-2 Breakpoint timing.*

External logic, such as additional breakpoint comparators, may be built to extend the functionality of the EmbeddedICE macrocell. Their output should be applied 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.

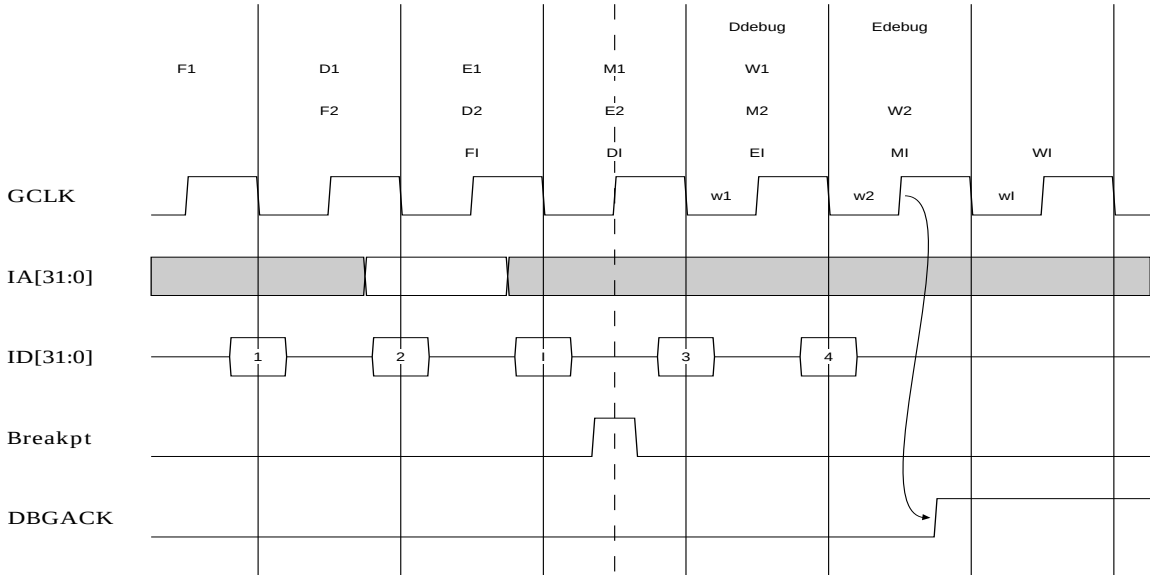

Figure 8-2 Breakpoint timing

8.3.2 Breakpoints and exceptions

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

SWI and undefined instruction are treated in the same way as any other instruction which may 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. Once the interrupt has been serviced, the execution flow is returned to the original program. This means that the instruction which was previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters the debug state once it reaches the execute stage of the pipeline.

Once the processor has entered debug state, it is important that further 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 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 while the instruction completes. The timing of debug entry following a watchpointed load in this case is shown in *Figure 8-3 Watchpoint entry with data processing instruction*:

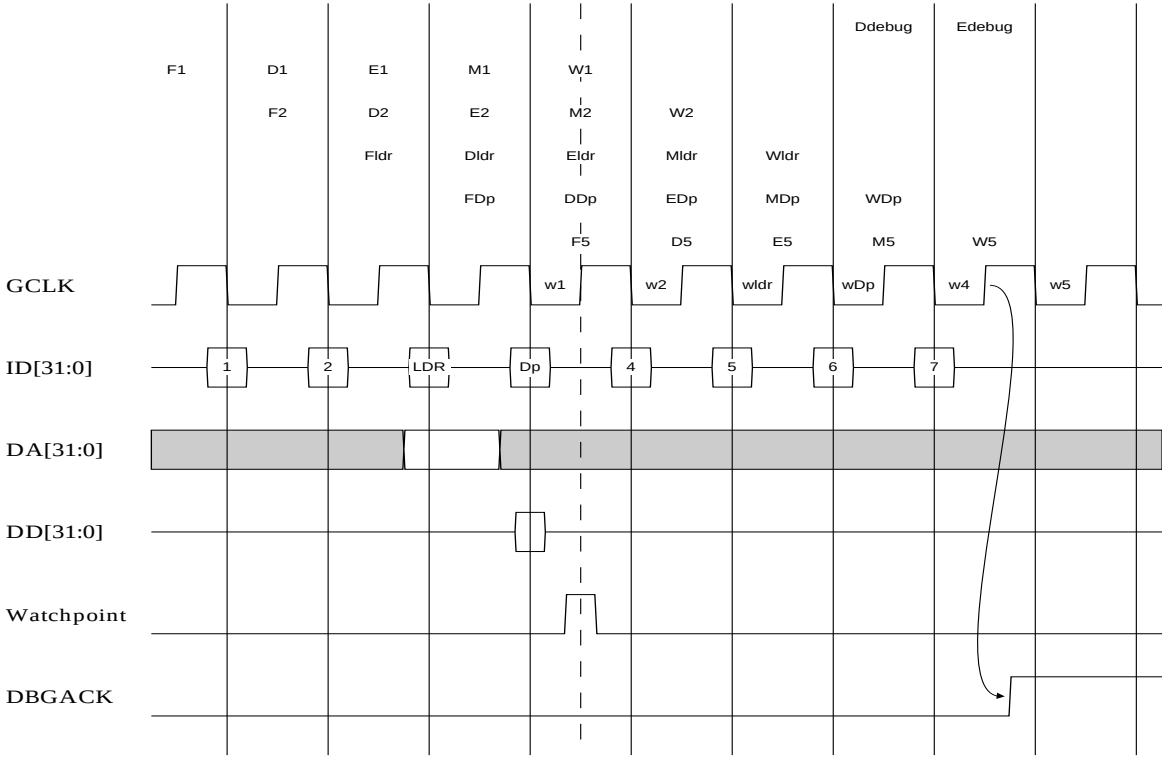

Figure 8-3 Watchpoint entry with data processing instruction

Note

Although instruction 4 enters the execute state, it is not executed, and there is no state update as a result of this instruction. Once the debugging session is complete, normal continuation would involve a return to instruction 4, the next instruction in the code sequence which has not yet been executed.

The instruction following the instruction which generated the watchpoint could have modified the PC. If this has happened, it will not be possible to determine the instruction which 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 Watchpoint entry with branch* on page 8-7. However, it is always possible to restart the processor.

Once the processor has entered debug state, the ARM940T core may be interrogated to determine its state. In the case of a watchpoint, the PC contains a value that is six 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, #0x20
```

is scanned in and the processor is restarted. Execution flow would then return to the next instruction in the code sequence.

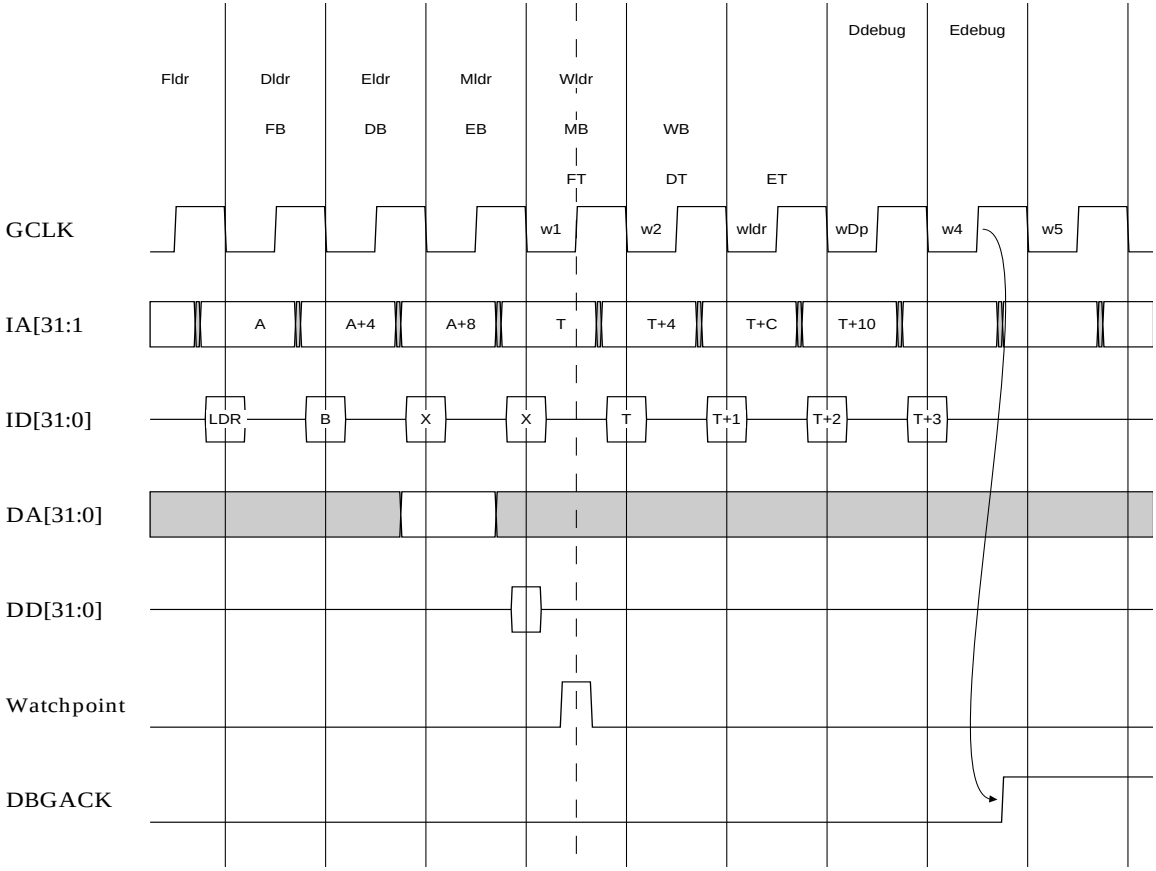

Figure 8-4 Watchpoint entry with branch

8.3.4 Watchpoints and exceptions

If there is an abort with the data access as well as a watchpoint, the watchpoint condition is latched, the exception entry sequence performed, and then the processor enters debug state. If there is an interrupt pending, again the 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 hardware 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 will enter debug state when the instruction at the execution stage of the pipeline has completely finished executing (once memory and write stages of the pipeline have completed). While waiting for the instruction to finish executing, no more instructions will be issued to the execute stage of the pipeline.

8.3.6 Actions of the ARM940T in debug state

Once the ARM940T is in debug state, internally both memory interfaces will indicate internal cycles. Since the rest of the system continues operation, the ARM940T will ignore aborts and interrupts.

8.4 Scan chains and JTAG interface

There are six scan chains inside the ARM940T. These allow testing, debugging and programming of the EmbeddedICE watchpoint units. The scan chains are controlled by a JTAG-style *Test Access Port* (*TAP*) controller. In addition, support is provided for an optional seventh scan chain. This is intended to be used for an external boundary scan chain around the pads of a packaged device. The signals provided for this scan chain are described later.

The seven scan chains of the ARM940T are referred to as scan chain 0, 1, 2, 3, 4, 5 and 15.

Note

The ARM940T TAP controller supports 32 scan chains. Scan chains 0 to 15 have been reserved for use by ARM. Any extension scan chains should be implemented in the remaining space. The **SCREG[4:0]** signals indicate which scan chain is being accessed.

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 Test access port (TAP) controller state transitions* shows the state transitions that occur in the TAP controller.

The state numbers are also shown on the diagram. These are output from the ARM940T on the **TAPSM[3:0]** bits.

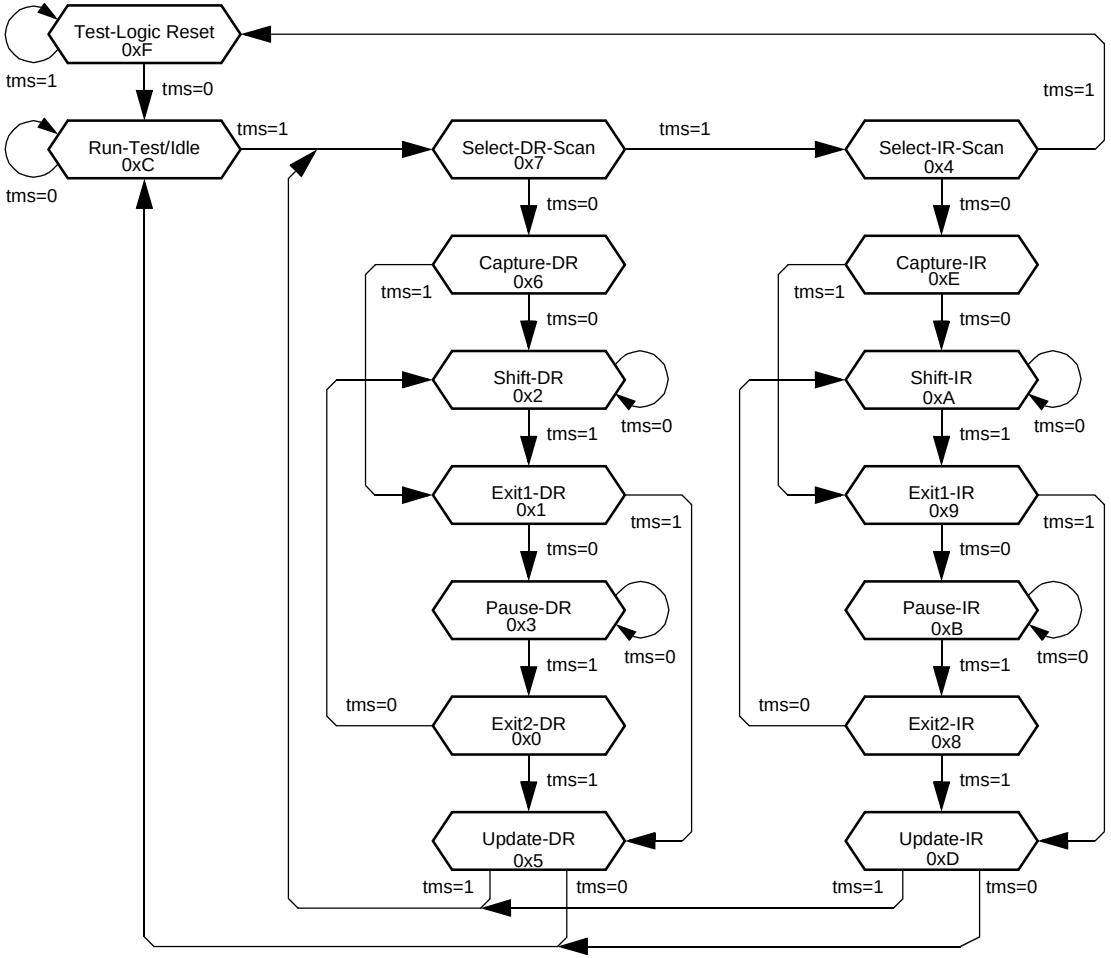

Figure 8-5 Test access port (TAP) controller state transitions

8.5.1 Reset

The JTAG interface includes a state-machine controller (the TAP controller). In order 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 may 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 Pullup resistors

The IEEE 1149.1 standard effectively requires **TDI** and **TMS** to have internal pullup resistors. In order 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

The following public instructions are supported:

Table 8-1 Public instructions

Instruction	Binary code
EXTEST	0000
SCAN_N	0010
INTEST	1100
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 should 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 via **TDO**, while new test data is shifted in via 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 via the **TDO** pin, while new test data is shifted in via 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 via the **TDO** pin, while data is shifted in via 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 via **TDI** and out via **TDO** after a delay of one **TCK** cycle. The first bit shifted out will be 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 should 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 via **TDI** and out via **TDO** after a delay of one **TCK** cycle. The first bit shifted out will be 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 is 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 via **TDI** and out via **TDO** after a delay of one **TCK** cycle. The first bit shifted out will be 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 3-state 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 via **TDI** and out via **TDO** after a delay of one **TCK** cycle. The first bit shifted out will be 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 via the **TDO** pin, while new data is shifted in via the **TDI** pin to preload the boundary scan parallel input latch. Note that this data is not applied to the system logic or system pins while the SAMPLE/PRELOAD instruction is active.

This instruction should 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 will resynchronize back to the memory system once the RUN-TEST/ IDLE state is entered.

8.6 Test data registers

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

- Bypass register
- ID code resister
- Instruction register
- Scan chain select register
- Scan chains 0, 1, 2, 4 and 5.

These are described in turn below.

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 during the CAPTURE-DR state.

The IEEE format of the ID register is as follows:

Table 8-2 ID code register

Bits	Contents
31–28	Version number
27–12	Part number
11–1	Manufacturer identity
0	1

The ARM940T ID code is 0x0f0f0f0f.

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 0001 binary 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 10000 binary 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 further 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 may 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 are shown in *Table 8-3 Scan chain number allocation*. An external scan chain may 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** will be 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 macrocell programming
3	External boundary scan
4	I Cache CAM
5	D Cache 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 hardware 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 via the SCAN_N instruction.

Length — 184 bits

INTEST allows serial testing of the 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’s 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, thus applying known stimuli to the inputs.

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

EXTEST allows inter-device testing, useful for verifying the connections between devices in the design. The TAP controller must be placed in EXTEST mode after scan chain 0 has been selected.

During CAPTURE-DR, the current inputs to the core's logic from the system are captured in the input cells.

During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known values on the core's outputs.

During RUN-TEST/IDLE, the core is not clocked.

The operation may then be repeated.

Scan chain 1

Purpose	Primarily for debugging. Scan chain 1 is selected via 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.
- While debugging, the value placed in the SYSSPEED control bit determines whether 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 whether the core has entered debug state due to a breakpoint (SYSSPEED LOW), or a watchpoint (SYSSPEED HIGH). It is possible to have a watchpoint and breakpoint condition occur simultaneously. When a watchpoint condition occurs the WPTANDBKPT bit must be examined by the debugger to determine whether the instruction currently in the execute stage of the pipeline is breakpointed. If so, WPTANDBKPT will be HIGH, otherwise it will be LOW.

Scan chain 2

Purpose — Allows access to the EmbeddedICE hardware registers. The order of the scan chain from **TDI** to **TDO** is:
read/write,
register address bits 4 to 0,
data values bits 31 to 0.

Length — 38 bits

To access this serial register, scan chain 2 must first be selected via 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 hardware register to be accessed.

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

Scan chain 3

Purpose — Allows the ARM9TDMI to control an external boundary scan chain.

Length — User defined.

Scan chain 3 is provided so that an optional external boundary scan chain may be controlled via the ARM9TDMI. Typically this would be 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.

DRIVEBS — This is used to switch the scan cells from system mode to test mode. This signal is asserted whenever either the 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 will be 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.

nHIGHZ
: This signal may be used to drive the outputs of the scan cells to the high impedance state. This signal is driven LOW when the HIGHZ instruction is loaded into the instruction register, and HIGH at all other times.

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

Scan chain 4

Purpose
: Allows access to the I Cache CAM array. The scan chain has two modes of operation.

  In addressing mode, the order of the scan chain **TDI** to **TDO** is:
  CAM index bits 5 to 0,
  segment select bits 1 to 0,
  20 bits which should be LOW.

  In reading mode, the order of the scan chain **TDI** to **TDO** is:
  valid bit,
  dirty bit,
  address bits 25 to 0.

Length
: 28 bits

To access this serial register, scan chain 4 must first be selected via 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 27 to 20 define the cache segment and CAM index to be accessed.

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 D cache CAM array. The scan chain has two modes of operation,

In addressing mode, the order of the scan chain **TDI** to **TDO** is:
CAM index bits 5 to 0,
segment select bits 1 to 0,
20 bits which should be LOW.

In reading mode, the order of the scan chain **TDI** to **TDO** is:
valid bit,
dirty bit,
address bits 25 to 0.

Length — 28 bits

To access this serial register, scan chain 4 must first be selected via 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 27 to 20 define the cache segment and CAM index to be accessed.

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 registers. The order of the scan chain **TDI** to **TDO** is:
read/write,
address bits 5 to 0,
32 bits register value.

Length — 39 bits

To access this serial register, scan chain 15 must first be selected via 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 EmbeddedICE hardware register to be accessed.

During SHIFT-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 bus is dependent on the current selected clock mode and the operation being performed. Refer to *Chapter 5 Clock Modes* for further 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 core is clocked by **DCLK** under control of the TAP state machine, and **GCLK** may 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 in which the clocks switch—during debugging and during testing.

8.8 Clock switching during debug

When the ARM9TDMI 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 asserts **DBGACK** in the HIGH phase of **GCLK**. The switch between the two clocks occurs on the next falling edge of **GCLK**.

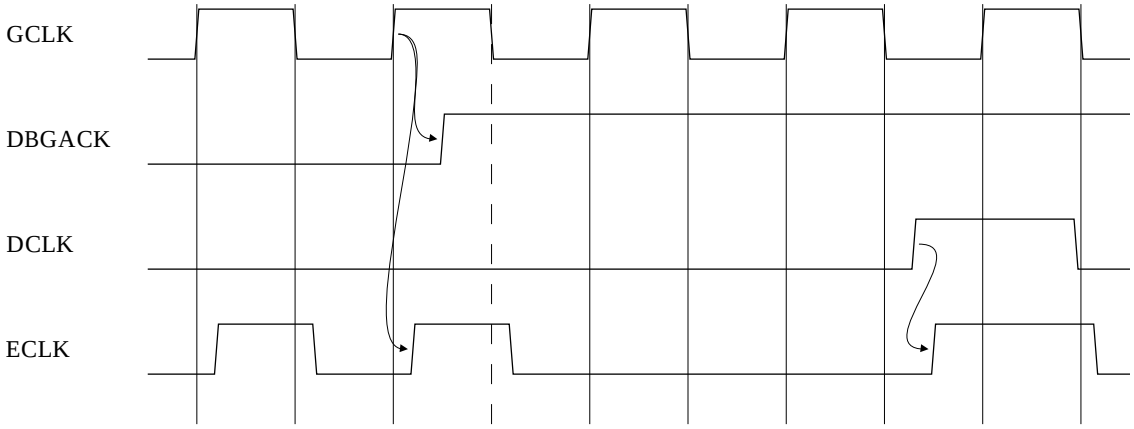

Figure 8-6 Clock switching on entry to debug state

The ARM9TDMI 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. 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.

The ARM9TDMI will now automatically resynchronize back to **GCLK** when the TAP controller enters to the RUN-TEST/IDLE mode and start fetching instructions from memory at **GCLK** speed. For more information, refer to *8.11 Exit from debug state*.

8.9 Clock switching during test

When under serial test conditions - when test patterns are being applied to the 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 the way 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 should 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.

8.10 Determining the core and system state

When the ARM9TDMI is in debug state, the core and system state may be examined. This is done by forcing load and store multiples into the pipeline.

Before the core and system state can be examined, the debugger must first determine whether the processor was in Thumb or ARM state when it entered debug. This is achieved by examining bit 4 of the EmbeddedICE hardware's Debug Status register. If this is HIGH, the core was in Thumb state when it entered debug.

8.10.1 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. Once this is done, the debugger can always execute the same sequence of instructions to determine the processor's state.

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

```
STR R0, [R1] ; Save R0 before use
MOV R0,  PC  ; Copy PC into R1
STR R0, [R1] ; Save the PC in R1
BX PC        ; Jump into ARM state
MOV R8, R8   ; NOP
MOV R8, R8   ; NOP
```

The above use of R0 as the base register for the stores is for illustration only—any register could be used.

Since 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 from which half of the bus the processor expects to read instructions.

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

Once the processor is in ARM state, typically the first instruction executed would be:

```
STMIA R0, {R0-R15}
```

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

After determining the values in the current bank of registers, it may be desirable to access banked registers. This can only be done by changing mode. Normally, a mode change may 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 may occur. Note that 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 was entered in supervisor mode.

The instruction sequence could be:

```
STMIA R0,  {R0-R15} ; Save current registers
MRS   R0, CPSR
STR   R0, R1        ; Save CPSR to determine current mode
BIC   R0, 0x1F      ; Clear mode bits
ORR   R0, 0x10      ; Select USER mode
MSR   CPSR, R0      ; Enter USER mode
STMIA R0, {R13-R14} ; Save registers not previously visible
ORR   R0, 0x01      ; Select FIQ mode
MSR   CPSR, R0      ; Enter FIQ mode
STMIA R1, {R8-R14}  ; Save banked FIQ registers
```

All these instructions are said to execute at *debug speed*. Debug speed is much slower than system speed since between each core clock, 67 scan clocks occur in order to shift in an instruction, or shift out data. Executing instructions more slowly than usual is fine for accessing the core's state since the ARM9TDMI is fully static. However, this same method cannot be used for determining the state of the rest of the system.

While in debug state, only the following instructions may be inserted into the instruction pipeline for execution:

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

8.10.2 Determining system state

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

A legal debug instruction may be placed in the instruction data bus of scan chain 1 with bit 33 (the SYSSPEED bit) LOW. This instruction will then be 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 will cause the ARM9TDMI to automatically resynchronize back to **GCLK** when the TAP controller enters RUN-TEST/IDLE state, and execute the instruction at system speed. Debug state will be re-entered once the instruction completes execution, when the processor will switch itself back to the internally generated **DCLK**. When the instruction has completed, **DBGACK** will be HIGH. At this point INTEST can be selected in the TAP controller, and debugging can resume.

Note

When preforming system-speed accesses, the caches will operate as usual, for example, performing cache lookups, line fills and evicting lines. To prevent the contents of the caches being altered, it is necessary to disable them first.

To determine whether a system speed instruction has completed, the debugger must look at SYSCOMP (bit 3 of the debug status register). To access memory, the ARM9TDMI must access memory through the data bus interface, as this access may be stalled indefinitely by **nWAIT**. The only way to determine whether 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’s memory can be passed to the debug host.

8.10.3 Instructions which may have the SYSSPEED bit set

The only valid instructions on which to set this bit 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.

8.11 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 *8.12 The PC's behavior during debug* on page 8-34.

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 at which 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 an 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 will revert back to system mode and clock resynchronization to **GCLK** will occur within the ARM940T. Normal operation will then resume, 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 which 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 would normally be expected. Any system peripheral that may be sensitive to the number of memory accesses can be inhibited through the use of **DBGACK**.

For example, consider a peripheral that simply counts the number of instruction fetches. This device should return the same answer after a program has run both with and without debugging. *Figure 8-7 Debug exit sequence* shows the behavior of the ARM940T on exit from debug state.

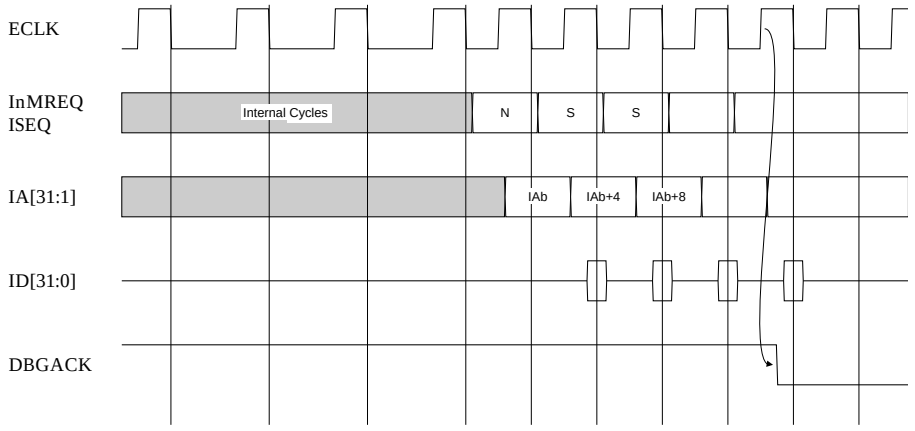

Figure 8-7 Debug exit sequence

It can be seen in *Figure 8-8 Debug state entry* that the final instruction fetch occurs in the cycle after **DBGACK** goes HIGH, and this is the point at which the instruction counter should be disabled. *Figure 8-7 Debug exit sequence* shows that the first memory access that the instruction fetch that the counter has not seen before occurs in the cycle after **DBGACK** goes LOW, and so this is the point at which the counter should be re-enabled.

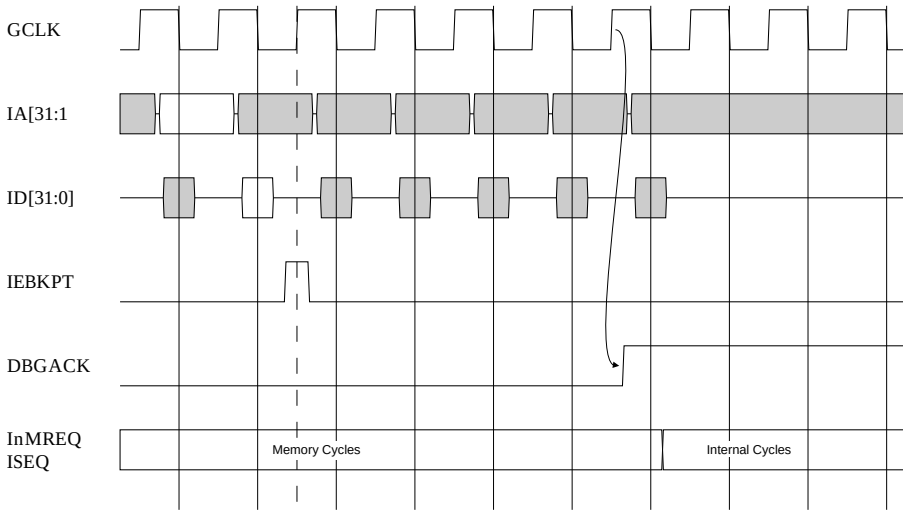

Figure 8-8 Debug state entry

Note that when a system speed access from debug state occurs, the ARM940T temporarily drops out of debug state, and **DBGACK** goes LOW. If there are peripherals that are sensitive to the number of memory accesses, they must be led to believe that the ARM940T is still in debug state. By programming the EmbeddedICE hardware control register, the value of **DBGACK** can be forced HIGH.

8.12 The PC's behavior during debug

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

- breakpoint
- watchpoint
- watchpoint when another exception occurs
- watchpoint when the next instruction is breakpoint
- debug request
- system speed accesses.

These are explained below:

8.12.1 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 –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 could be replaced with a subtract with the PC as the destination (SUB PC, PC, #28 for ARM code in the above example).

8.12.2 Watchpoint

Returning to the program execution after entering debug state from a watchpoint is done in the same way as the procedure described in *8.12.1 Breakpoint* above. Debug entry adds four addresses to the PC, and every instruction adds one address. The difference is that the instruction after that which caused the watchpoint has executed. The next instruction after that is the return instruction.

8.12.3 Watchpoint with another exception

If a watchpoint access simultaneously causes a data abort, the ARM940T will enter 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 will enter debug state in the exception's mode, and the debugger must check to see whether this happened. The debugger can deduce whether an exception occurred by looking at the current and previous mode (in the CPSR and SPSR), and the value of the PC. If an exception did take place, the user should be given the choice of whether 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 could 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 will force a branch back to the abort vector, causing the instructions at that location to be refetched and executed. Note that after the abort service routine, the instruction that caused the abort and watchpoint will be re-executed. This will cause the watchpoint to be generated and the ARM940T will enter debug state again.

8.12.4 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. In this instance, the PC will have been advanced by four addresses, but due to the breakpointed instruction not being executed, the instruction to be executed upon exit from debug state is at PC -5 addresses.

8.12.5 Debug request

Entry into debug state via a debug request is similar to a breakpoint. However, unlike a breakpoint, the last instruction in the execution stage of the pipeline will have completed execution and so must not be refetched on exit from debug state. Therefore, entry to debug state adds three addresses to the PC, and every instruction executed in debug state adds one.

For example, the following sequence handles a situation in which the user has invoked a debug request, and decides 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.

8.12.6 System speed accesses

If a system speed access is performed during debug state, the value of the PC is increased by five addresses. Since 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 was 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 hence the abort address. In this case, the value of the PC is invalid, but the debugger will know the address of the location that was being accessed. Thus the debugger can be written to help the abort handler fix the memory system.

8.12.7 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), and S is the number of system speed instructions executed.

8.13 EmbeddedICE

The EmbeddedICE hardware is integral to the ARM9TDMI processor core. It has two hardware breakpoint/watchpoint units that may be configured to monitor either the instruction memory interface or the data memory interface. Each watchpoint unit set has a value and mask register, with an address, data and control field.

Because the ARM9TDMI processor core has a Harvard Architecture, the user needs to specify whether the watchpoint registers examine the instruction interface or the data interface. This is specified by bit 3 in the control field of the watchpoint register. When bit 3 is set, the data interface is examined. When it is clear, the instruction interface is examined. There can be no “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 whether the **IBREAKPT** or **DBREAKPT** signal should be driven by the result of the comparison, as shown in *Figure 8-9 ARM940T EmbeddedICE overview* on page 8-39.

The ARM940T EmbeddedICE unit has logic that allows single stepping through code. This reduces the work required by an external debugger, and removes the need 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.

The general arrangement of the EmbeddedICE hardware is shown in *Figure 8-9 ARM940T EmbeddedICE overview* on page 8-39.

8.13.1 Register map

The EmbeddedICE register map is shown below:

Table 8-4 ARM940T EmbeddedICE register map

Address	Width	Function
00000	4	Debug control
00001	5	Debug status
00010	8	Vector catch control
00100	6	Debug comms control
00101	32	Debug comms data
01000	32	Watchpoint 0 address value
01001	32	Watchpoint 0 address mask

Table 8-4 ARM940T EmbeddedICE register map (continued)

Address	Width	Function
01010	32	Watchpoint 0 data value
01011	32	Watchpoint 0 data mask
01100	9	Watchpoint 0 control value
01101	8	Watchpoint 0 control mask
10000	32	Watchpoint 1 address value
10001	32	Watchpoint 1 address mask
10010	32	Watchpoint 1 data value
10011	32	Watchpoint 1 data mask
10100	9	Watchpoint 1 control value
10101	8	Watchpoint 1 control mask

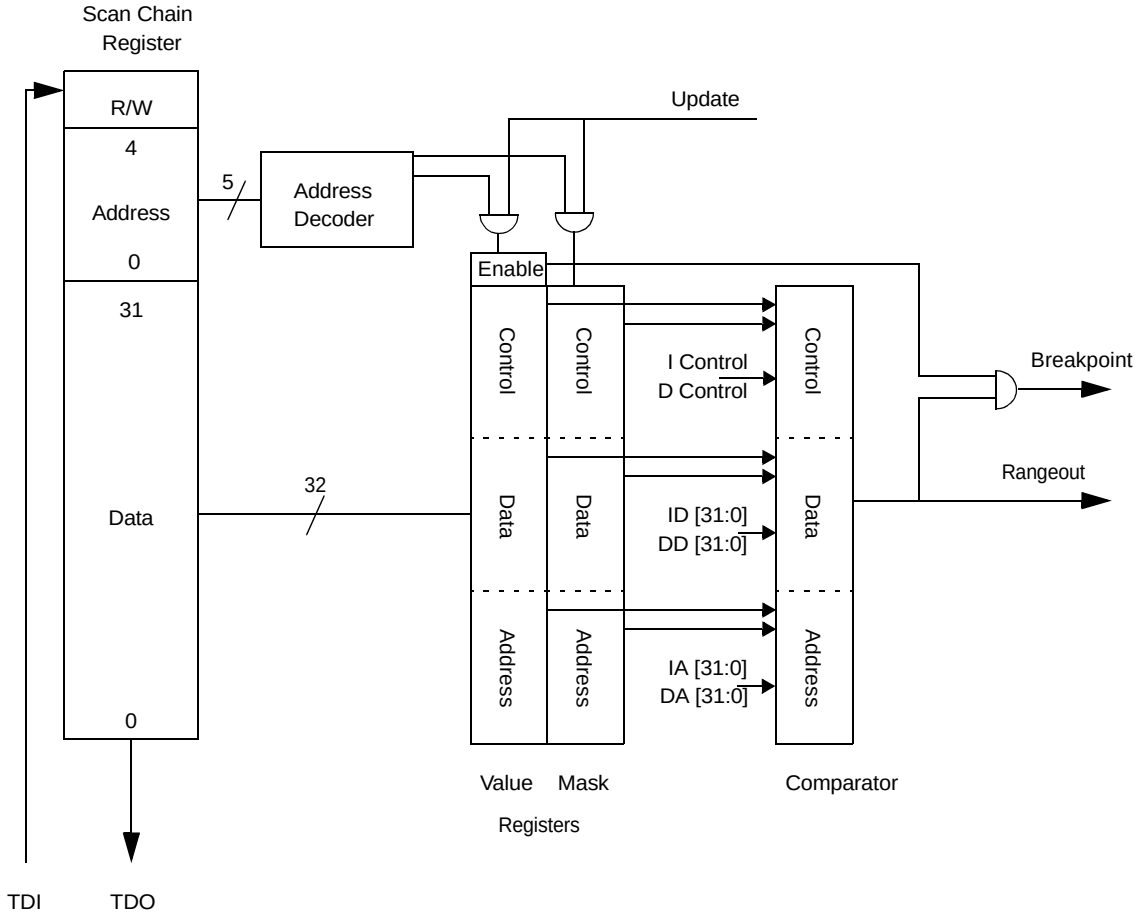

Figure 8-9 ARM940T EmbeddedICE overview

8.13.2 Control registers

The format of the control registers depends on how bit 3 is programmed. If bit 3 is programmed to be 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 Watchpoint control register for data comparison.*

Note

Bit 8 and bit 3 cannot be masked.

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	EXTERN	DnTRANS	1	DMAS[1]	DMAS[0]	DnRW

Figure 8-10 Watchpoint control register for data comparison

The bits have the following functions:

Table 8-5 Watchpoint control register for data comparison bit functions

Bit	Function
DnRW	Compares against the data not read/write signal from the core in order to detect the direction of the data bus activity. **nRW** is 0 for a read, and 1 for a write
DMAS[1:0]	Compares against the **DMAS[1:0]** signal from the core in order to detect the size of the data bus activity.
DnTRANS	Compares against the data not translate signal from the core in order 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 hardware that allows the watchpoint to be dependent upon some external condition. The **EXTERN** input for watchpoint 0 is labelled **EXTERN0**, and the **EXTERN** input for watchpoint 1 is labelled **EXTERN1**.

Table 8-5 Watchpoint control register for data comparison bit functions

Bit	Function
CHAIN	Can be connected to chain output of another watchpoint in order to implement, for example, debugger requests of the form “breakpoint on address YYY only when in process XXX”. In the ARM940T EmbeddedICE hardware, 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 hardware, 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** of **DBREAKPT** signal will only be asserted when the ENABLE bit is set. This bit only exists in the value register; it cannot be masked.

If bit 3 of the control register is programmed to 0, the comparators will examine the instruction address, instruction data and instruction control buses. In this case 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 Watchpoint control register for instruction comparison*.

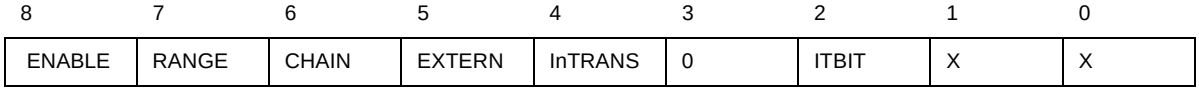

Figure 8-11 Watchpoint control register for instruction comparison

Table 8-6 Watchpoint control register for instruction comparison bit functions

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 in order 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 hardware that allows the watchpoint to be dependent upon some external condition. The **EXTERN** input for watchpoint 0 is labelled **EXTERN0**, and the **EXTERN** input for watchpoint 1 is labelled **EXTERN1**.
CHAIN	Can be connected to chain output of another watchpoint in order to implement, for example, debugger requests of the form “breakpoint on address YYY only when in process XXX”. In the ARM940T EmbeddedICE hardware, 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 hardware, 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** of **DBREAKPT** signal will only be asserted when the ENABLE bit is set. This bit only exists in the value register; it cannot be masked.

8.13.3 Debug control register

The ARM940T debug control register is four bits wide and is shown in *Figure 8-12 Debug control register*. Bit 3 controls the single-step hardware. This is explained in more detail in *8.15 Single stepping* on page 8-44.

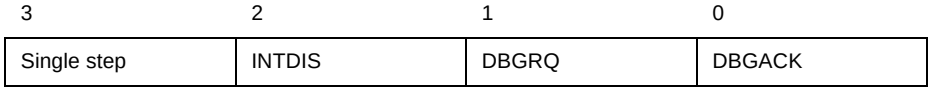

Figure 8-12 Debug control register

8.13.4 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.

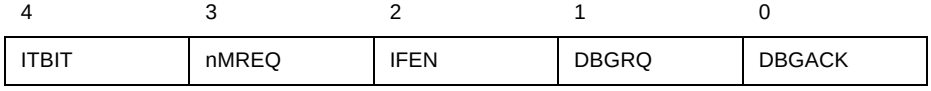

Figure 8-13 Debug status register

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. Since the capture clock for the scan chain may 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 **nMREQ** 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 I**TBIT** to be read. This enables the debugger to determine what state the processor is in, and hence which instructions to execute.

8.13.5 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 Vector catch register*. The functionality is described in *8.14 Vector catching*, below.

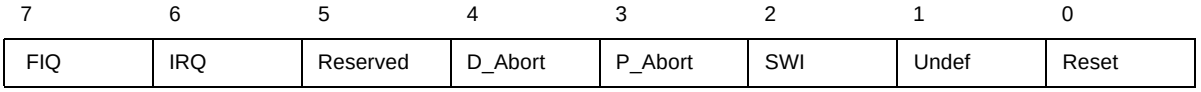

Figure 8-14 Vector catch register

8.14 Vector catching

The ARM940T EmbeddedICE macrocell 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.

8.15 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 should 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.

8.16 Debug communications channel

The ARM940T EmbeddedICE hardware 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 wide Comms Data Read register, a 32-bit wide Comms Data Write register and a 6-bit wide Comms Control register for synchronized handshaking between the processor and the asynchronous debugger. These registers are located in fixed locations in the EmbeddedICE register map (as shown in *Figure 8-9 ARM940T EmbeddedICE overview* on page 8-39) and are accessed from the processor via MCR and MRC instructions to coprocessor 14.

8.16.1 Debug comms channel registers

The Debug Comms Control register is read only, and allows synchronized handshaking between the processor and the debugger.

31	30	29	28	···	1	0
0	0	1	0	···	W	R

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 hardware version number, in this case 0010.

Bits 27:2 Unused.

Bit 1 Denotes from the processor's point of view, whether the Comms Data Write register is free.
If, from the processor's point of view, the Comms Data Write register is free (W=0), new data may be written.
If it is not free (W=1), the processor must poll until W=0.
If, from the debugger's point of view, W=1, some new data has been written which may then be scanned out.

Bit 0 Denotes whether there is some new data in the Comms Data Read register. If, from the processor's point of view, R=1, there is some new data which may be read via an MRC instruction.
If, from the debugger's point of view, R=0, the Comms Data Read register is free and new data may be placed there through the scan chain.
If R=1, this denotes that data previously placed there through the scan chain has not been collected by the processor, and so the debugger must wait.

From the debugger's point of view, the registers are accessed via the scan chain in the usual way. From the processor, these registers are accessed via coprocessor register transfer instructions. The following instructions should be used:

```
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.

The Thumb instruction set does not support coprocessor instructions (must be in ARM state).

8.16.2 Communications via 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 will only be 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 will the Debug Comms Write or Read register be accessed.

8.16.3 Software polling communication

Sending a message to EmbeddedICE

When the processor wishes to send a message to EmbeddedICE hardware, it must check that the Comms Data Write register is free for use by finding out whether the W bit of the Debug Comms Control register is clear.

It reads the Debug Comms Control register to check status of the W bit:

- 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 clear.

When the W bit is clear, a message is written by a register transfer to coprocessor 14. As 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 may begin again.

Receiving a message from EmbeddedICE

Message transfer from the debugger to the processor is similar to sending a message to EmbeddedICE. 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 via 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 an MCLK synchronized version. If the R bit is set, there is data waiting to be collected; this can be read via 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 may now be repeated.

8.16.4 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 was not the source of the interrupt.

8.17 The debugger's view of the cache

When in debug state, the debugger is able to see the state of the memory system, including the caches. The debugger needs 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 38 bits long, and has a structure similar to the EmbeddedICE macrocell scan chain 2. The format of scan chain 15 is shown below in *Table 8-7*. An access via this scan chain allows any of CP15 registers to be read or written.

Table 8-7 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 should extract and save the state of CP15. It is advisable then 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 8-8*. For CP register 6, CRm corresponds to the region number.

Table 8-8 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)
0	0101	1	5 (Instruction)
0	1001	0	9 (Data)
0	1001	1	9 (Instruction)

Table 8-8 Scan access mapping to CP15 register

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

The contents of the caches is determined by:

1. Extracting the contents of the CAMs
2. Determining the contents of the RAMs.

The CAM arrays are read via scan chain 4 for the I Cache, and scan chain 5 for the D Cache. 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 8-9 Scan chain 4 and 5 addressing mode*.

Reading — The contents of the CAM entry are read back. The format of the data read back is shown in *Table 8-10 Scan chains 4 and 5 reading mode*. When the I Cache CAM is read, the dirty bit will always be read as zero.

The addressing mode format 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.

Table 8-9 Scan chain 4 and 5 addressing mode

Scan chain bit	Write function
27:22	CAM index
21:20	Segment select
19:0	Should be zero

Table 8-10 Scan chains 4 and 5 reading mode

Scan chain bit	Write function
27	Valid
26	Dirty
25:0	Address

The debugger must index through all the entries in the CAM (0-63) to determine the 26-bit TAG addresses. When this information is extracted, the contents of the cache RAM array can be determined.

For the Data cache

This is achieved by taking each TAG address, padding the bottom 4-bits with zeros, setting bits 5 and 6 to indicate the same segment that the TAG was scanned from, and performing a system-speed 4-word LDM to that address, with the cache switched on. As 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 which 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.

For the Instruction cache

For the instruction cache, the system-speed LDM should be performed with the cache switched off. This ensures that the external memory system is accessed. As it is impossible to change the data in the instruction cache, the I Cache and external memory are guaranteed coherent.

Chapter 9
TrackingICE

This chapter describes how TrackingICE mode is used by the ARM940T:

- Timing requirements
- TrackingICE outputs.

9.1 Overview

When in TrackingICE mode, a number of the ARM940T outputs track the inputs to the ARM9TDMI processor core embedded within the ARM940T. An ARM9TDMI test chip can then be connected to the outputs, which will precisely track the ARM9TDMI processor core inside the ARM940T. This enables all outputs of the ARM9TDMI to be observed.

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

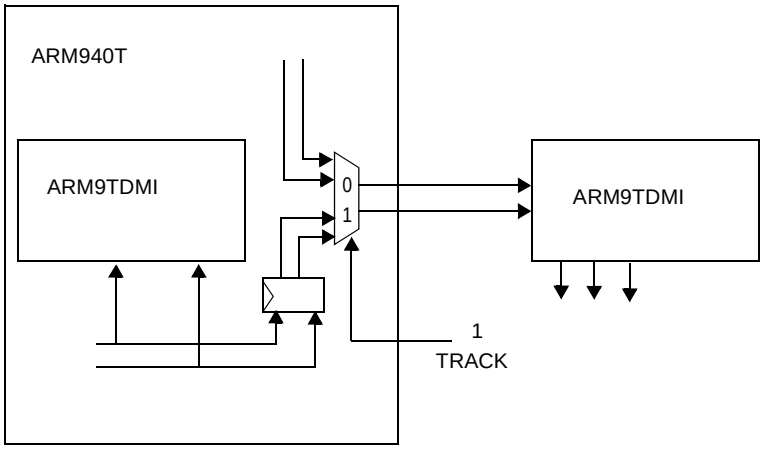

Figure 9-1 Using TrackingICE

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 outputs. 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 following ARM940T outputs 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[3][4]	**nIRQ**
SCREG[2][3]	**nFIQ**
SCREG[1][2]	**DABORT**
SCREG[0][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 just for coprocessor data transfers. The following conditions apply:

- The **ISYNC** and **nTRST** inputs should be common between the ARM940T and the tracking ARM9TDMI
- **IABE** and **DABE** should be HIGH so that the address outputs of the tracking ARM9TDMI can be observed
- **DDBE** should 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 since **CPDOUT** is driven with the data from all memory access cycles.

Chapter 10
Test Issues

This chapter examines the test issues for the ARM940T and lists the scan chain 0 bit order.

10.1 Introduction

The ARM940T processor core supports parallel and serial test methodologies. The parallel test patterns are derived from assembler ARM code programs written to achieve a high fault location coverage.

The ARM940T processor core has a fully JTAG-compatible scan chain which intersects all the inputs and outputs. This allows the test patterns to be serialized and injected to the processor via the JTAG interface. Both the parallel and serial test patterns are supplied to ARM940T processor core licensees. The scan chain also supports EXTEST, allowing the connections between the ARM940T processor core and other JTAG-compatible peripherals to be tested.

The ARM940T supports parallel and AMBA test. The ARM940T parallel patterns are generated in a similar way to those for the ARM9TDMI processor core. The AMBA test methodology involves using the main system data bus to apply vectors to the device under test. Each test vector has to be built up in 32-bit multiples on the inputs of the device (because the AMBA data bus from the ARM940T is 32 bits wide). This means that a number of latches are required in the AMBA veneer.

The ARM940T AMBA test wrapper also provides a high level of controllability over the caches. This allows the caches to be tested independently of the ARM9TDMI processor core. The addresses in the CAM array may be read and written. The CAM hit bits may be read and the data in the RAM may be read and written.

10.2 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
38:67	**DD[29:1]**	Bidirectional
68	**DD[0]**	Bidirectional
69	**DA[31]**	Output
70	**DA[30]**	Output
71:99	**DA[29:1]**	Output
100	**DA[0]**	Output
101	**IA[31]**	Output
102	**IA[30]**	Output
103:131	**IA[29:2]**	Output
132	**IA[1]**	Output
133	**IEBKPT**	Input
134	**DEWPT**	Input
135	**EDBGRQ**	Input
136	**EXTERN0**	Input
137	**EXTERN1**	Input

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

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

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

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

Chapter 11
Instruction Cycle Summary and Interlocks

11.1 Introduction

This chapter gives the instruction cycle times and shows the timing diagrams for interlock timing. All signals quoted are ARM9TDMI signals, and are internal to the ARM940T. In all cases it is assumed that all accessess are from cached regions of memory.

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

11.2 Instruction cycle times

Key to tables

Table 11-1 Symbols used in tables

Symbol	Meaning
b	The number of busy-wait states during coprocessor accesses
m	Is in the range 0 to 3, depending on early termination (see *11.2.1 Multiplier cycle counts* on page 11-5)
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 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
Data Op	2	1S+1I	2I	With register controlled shift
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)

Table 11-2 Instruction cycle bus times (continued)

Instruction	Cycles	Instruction bus	Data bus	Comment
LDR	3	1S+2I	1N+2I	Loaded byte, halfword, 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
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 byte used by following instruction
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

Table 11-2 Instruction cycle bus times (continued)

Instruction	Cycles	Instruction bus	Data bus	Comment
MRC	b+2	1S+(b+1)I	(b+I)I+1C	Following instruction uses transferred data
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 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 due to the incomplete execution of an earlier instruction. When an interlock occurs, instruction fetches stop on the instruction memory interface of the ARM940T. Four examples of this are given below.

Example 1

In this first 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*.

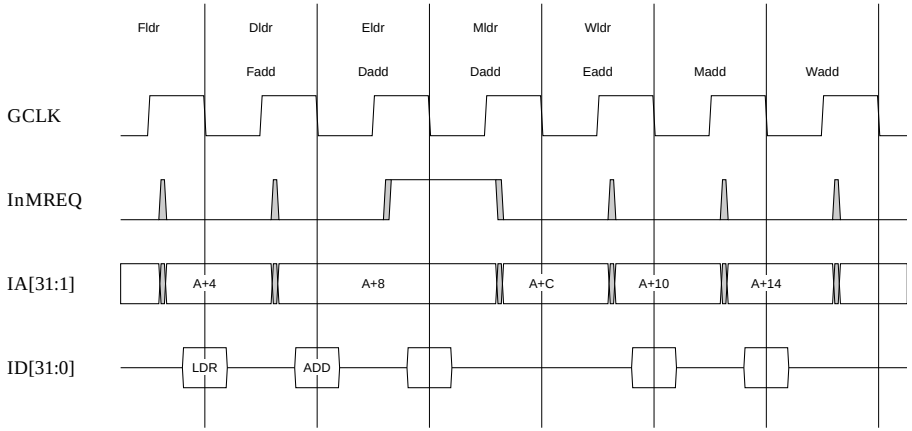

Figure 11-1 Single load interlock timing

Example 2

In this second 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*.

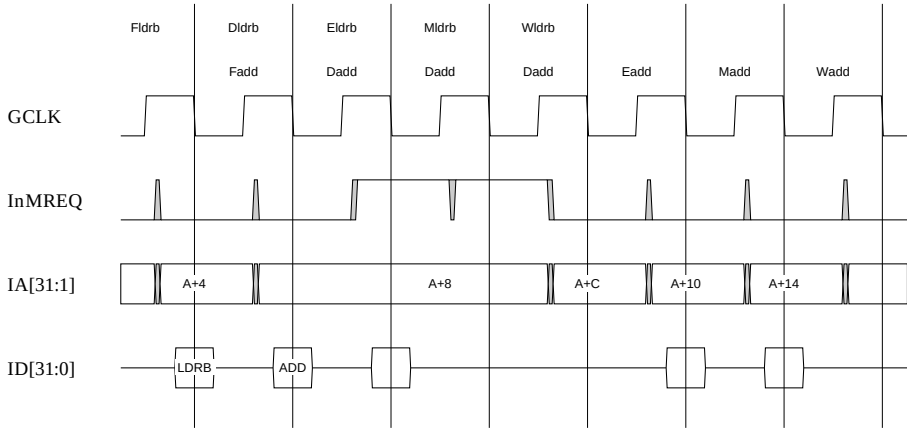

Figure 11-2 Two cycle load interlock

Example 3

In this third example, the following code sequence is executed:

```
LDM 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*.

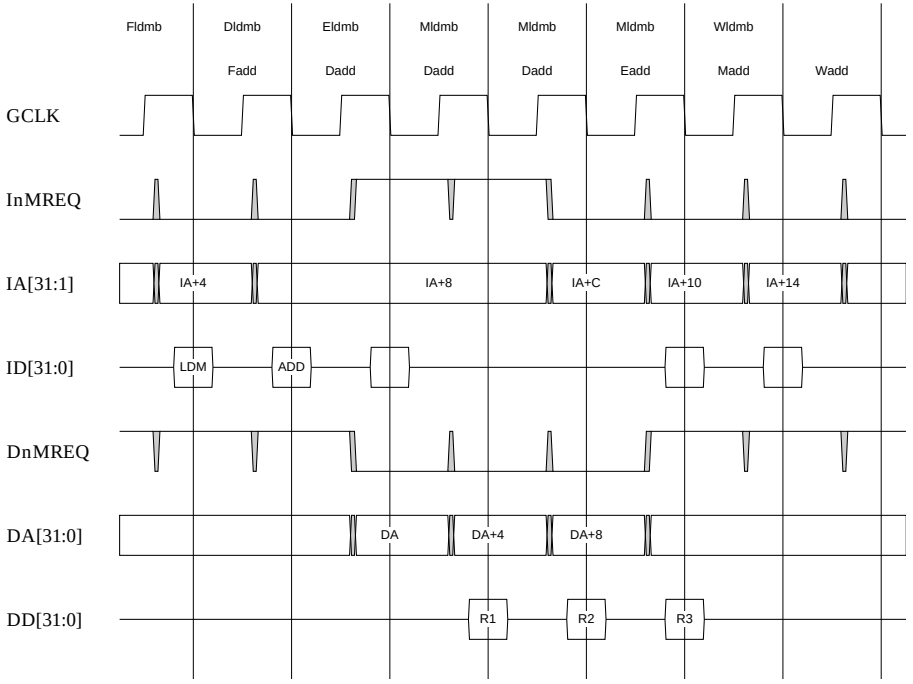

Figure 11-3 LDM interlock

Example 4

In the fourth example, the following code sequence is executed:

```
LDM 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. Due to 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 a further cycle of interlock while R3 is loaded. The behavior on the instruction and data memory interface is shown in *Figure 11-4*.

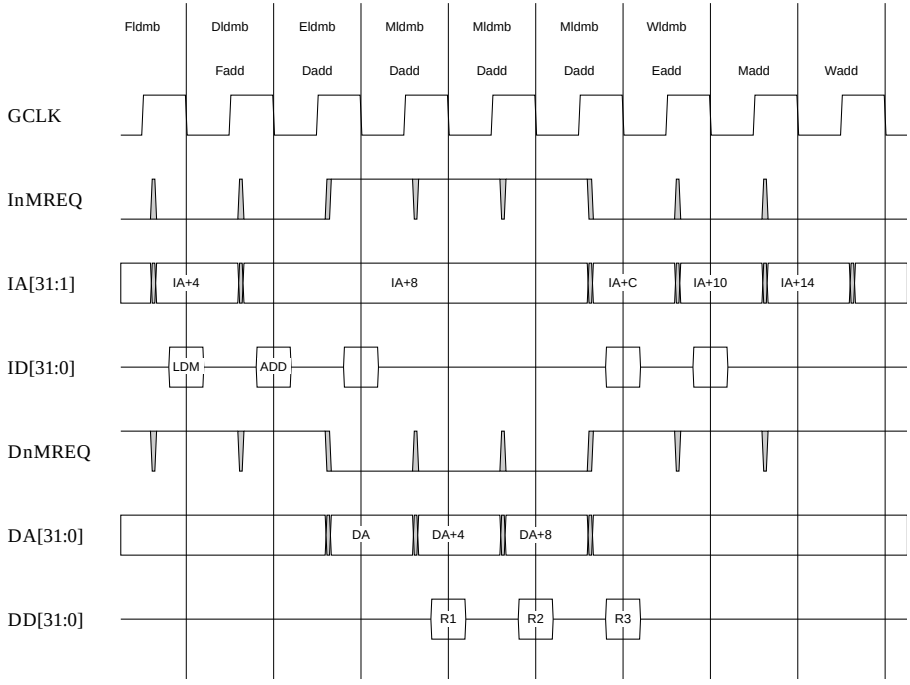

Figure 11-4 LDM dependent interlock

Chapter 12
ARM940T AC Characteristics

12.1 Introduction

This chapter gives the timing diagrams and timing parameters for the ARM940T. The information in this chapter is organized as follows:

- ARM940T timing diagrams
- ARM940T timing parameters.

12.2 ARM940T timing diagrams

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

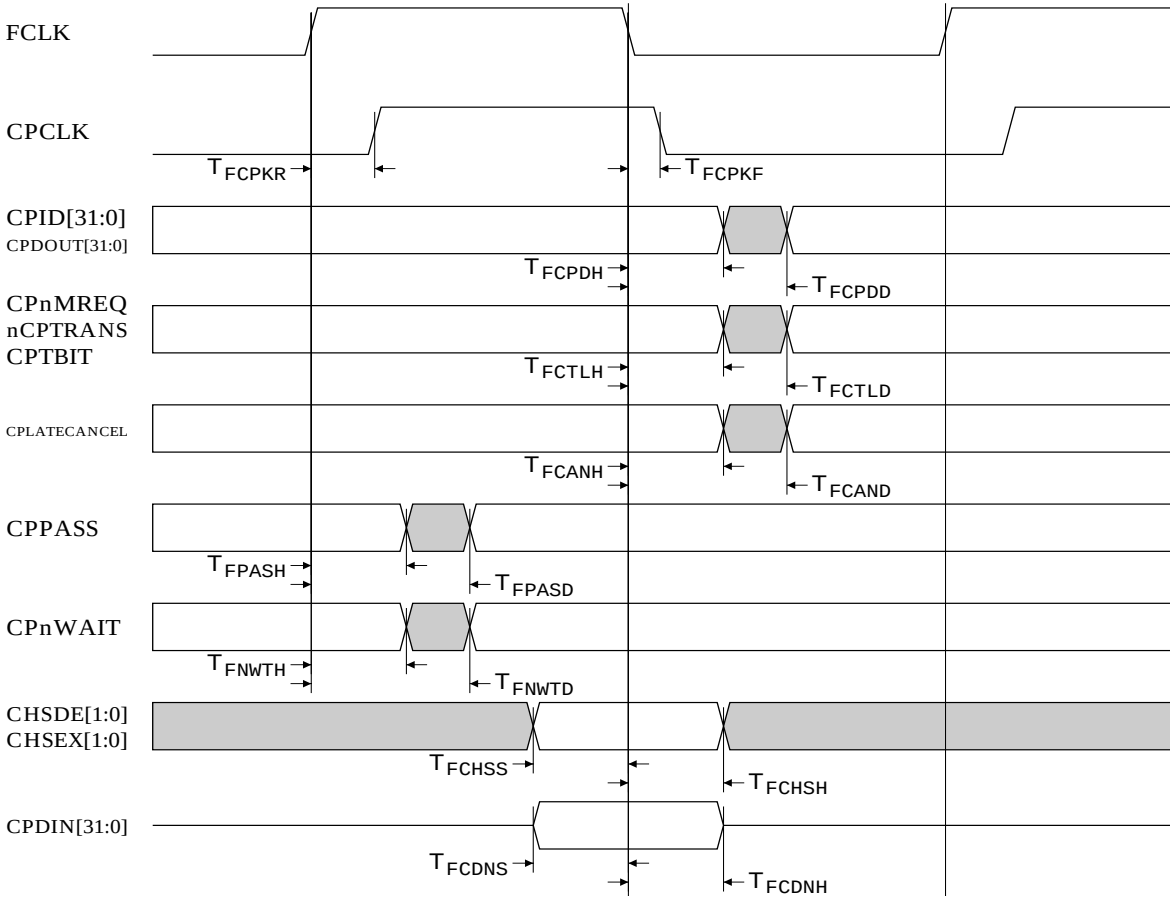

Figure 12-1 ARM940T FCLK timed coprocessor interface

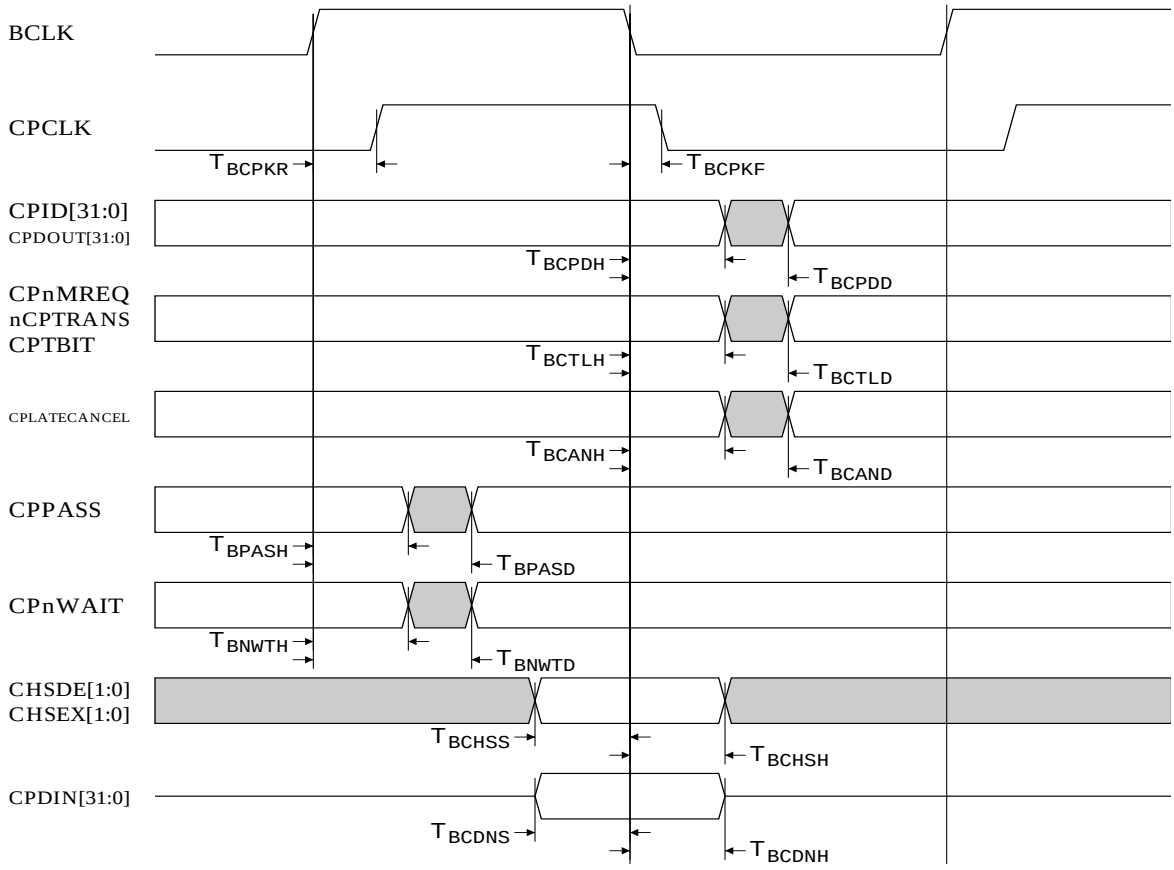

Figure 12-2 ARM940T BCLK timed coprocessor interface

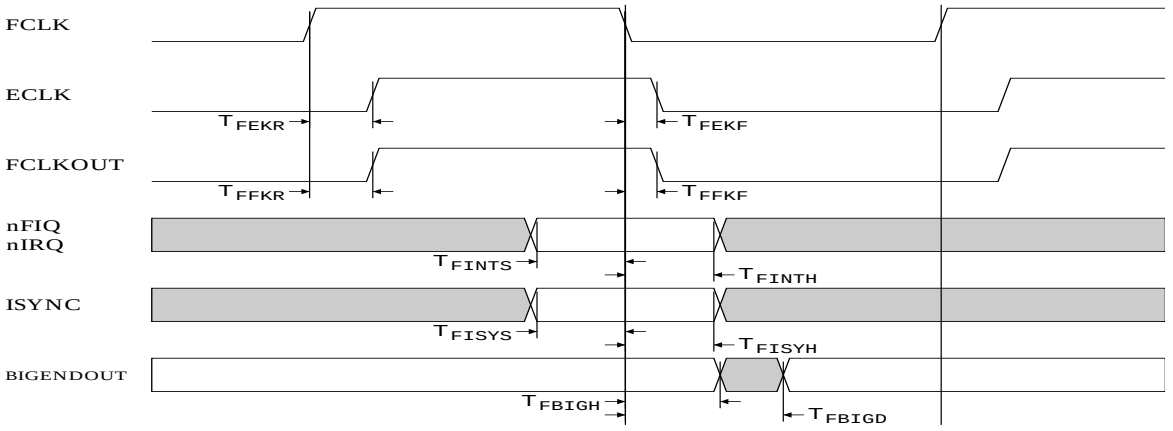

Figure 12-3 ARM940T FCLK related signal timing

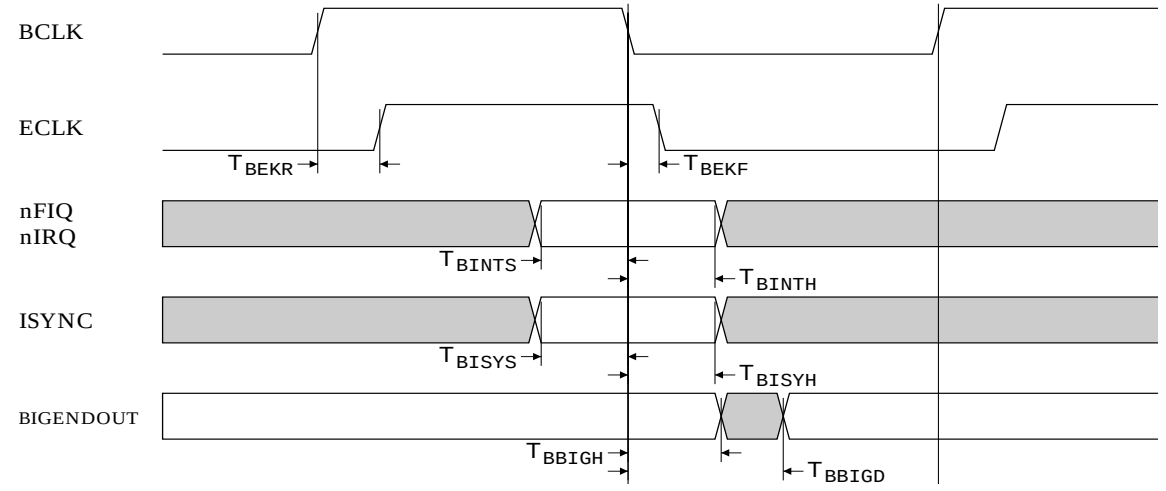

Figure 12-4 ARM940T BCLK related signal timing

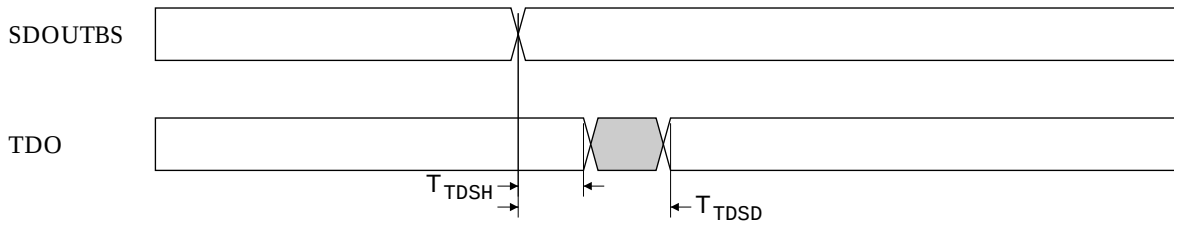

Figure 12-5 ARM940T SDOUTBS to TDO relationship

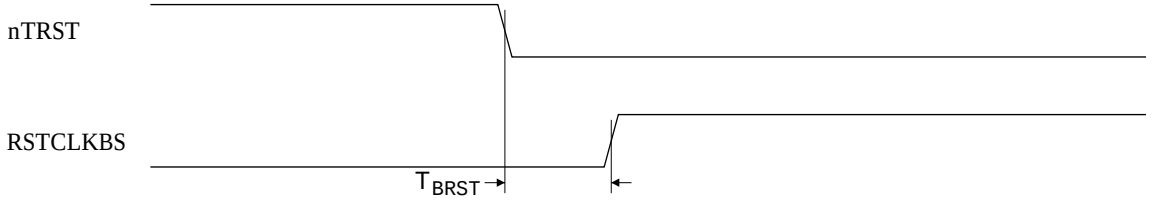

Figure 12-6 ARM940T nTRST to RSTCLKBS relationship

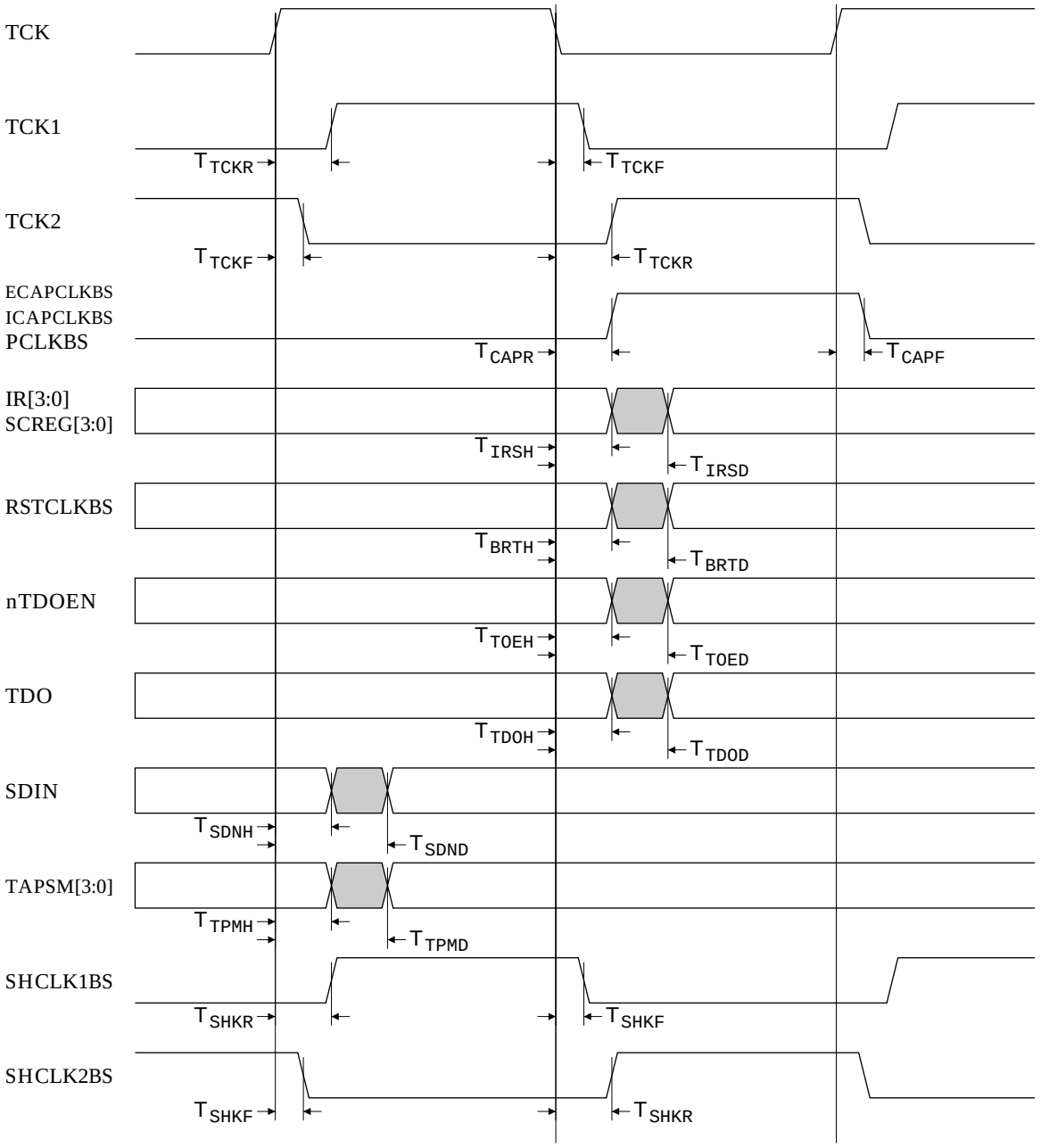

Figure 12-7 ARM940T JTAG output signal

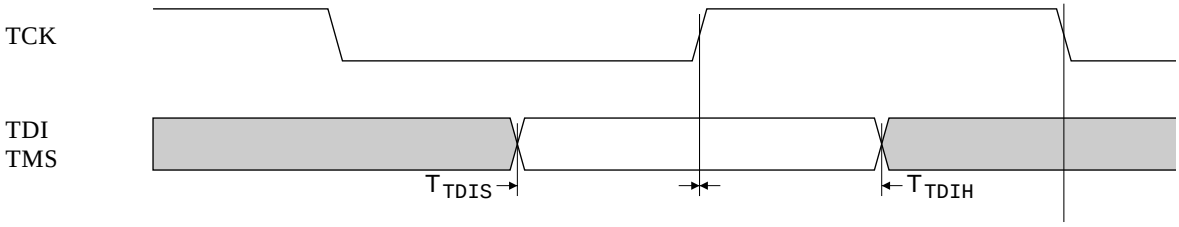

Figure 12-8 ARM940T JTAG input signal timing

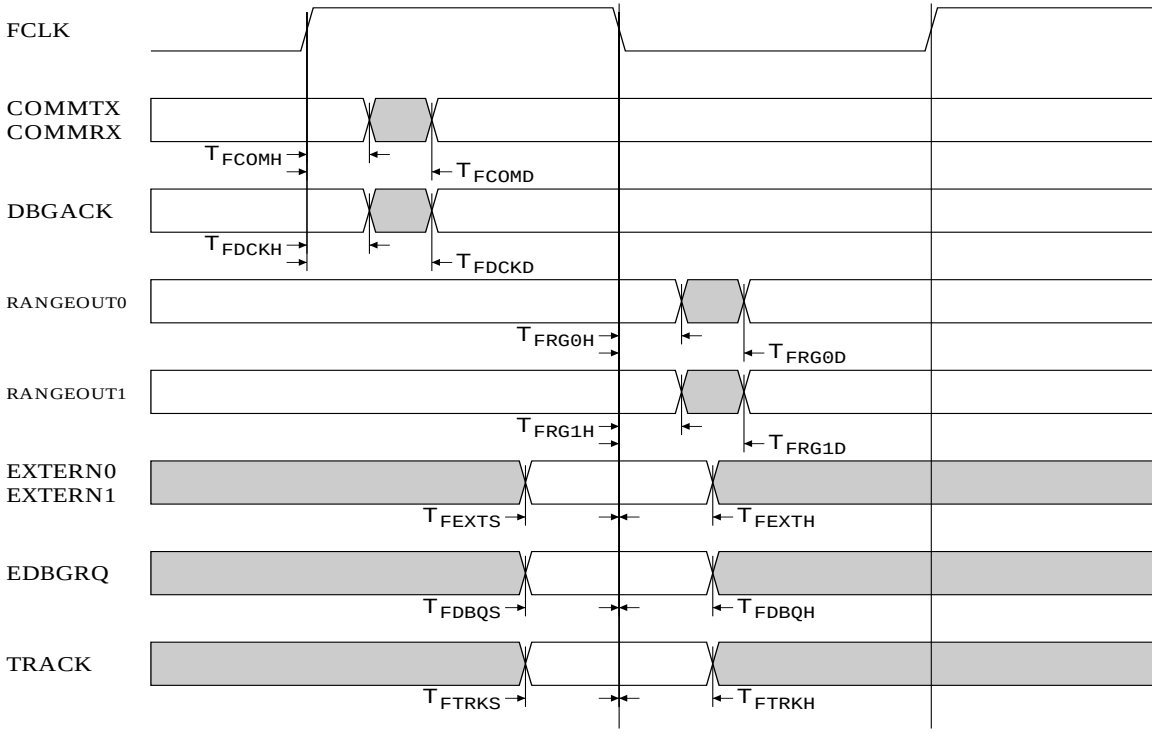

Figure 12-9 ARM940T FCLK related debug output timings

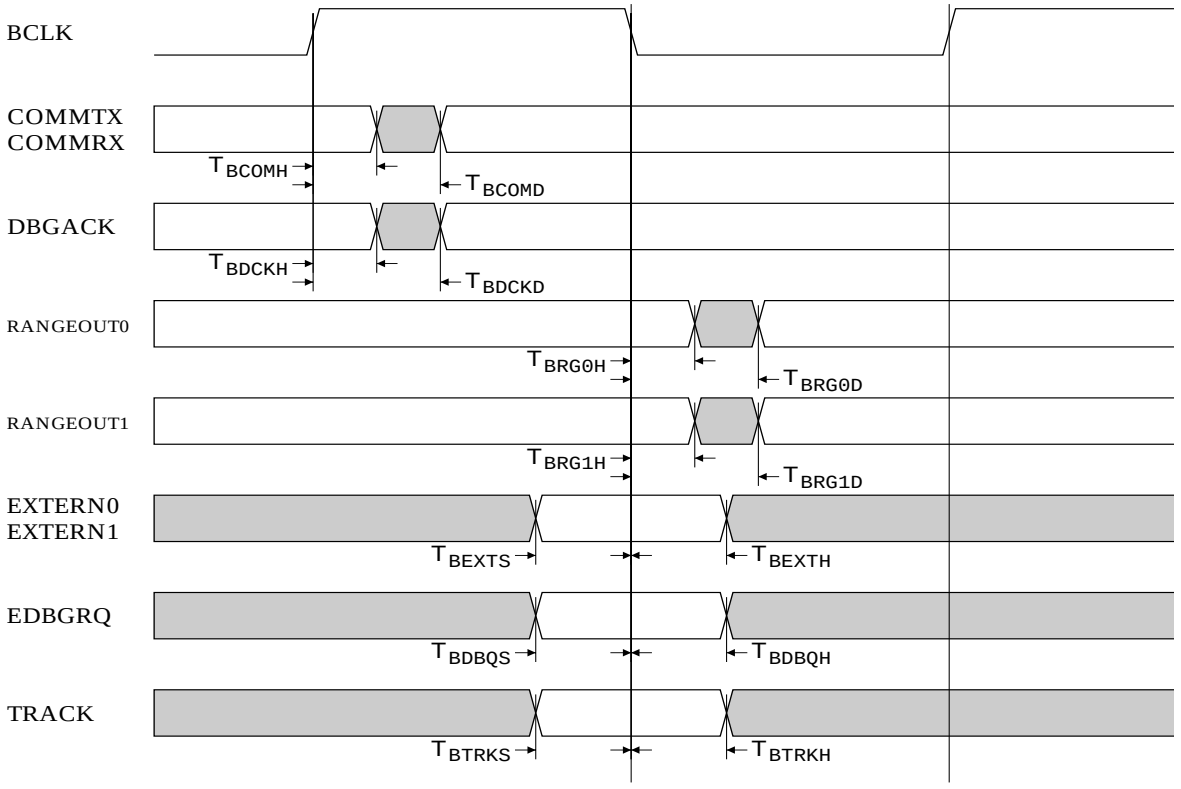

Figure 12-10 ARM940T BCLK related debug output timings

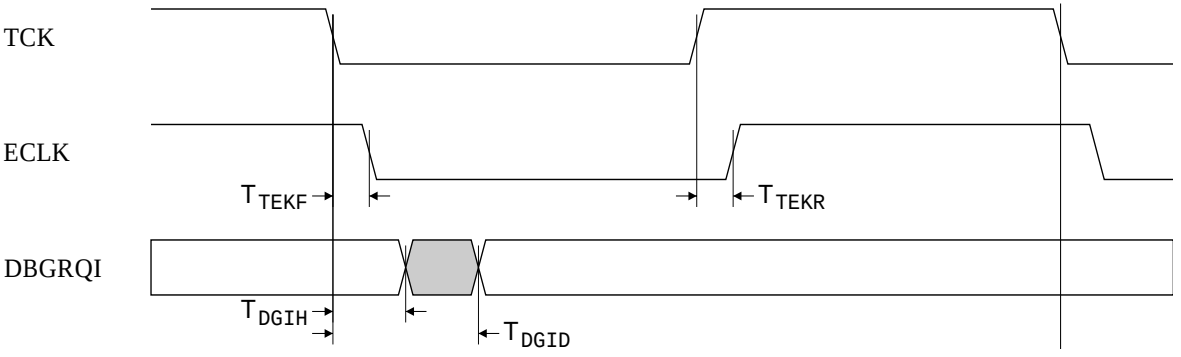

Figure 12-11 ARM940T TCK related debug output timings

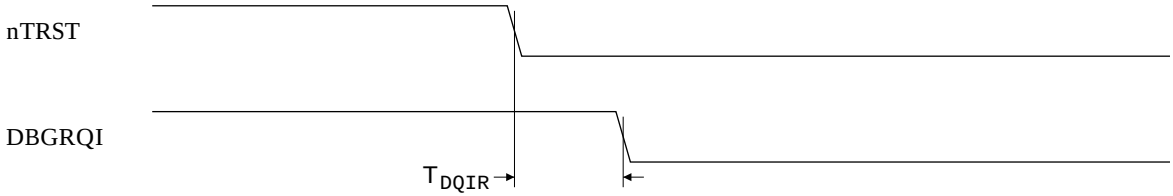

Figure 12-12 nTRST to DBGRQI relationship

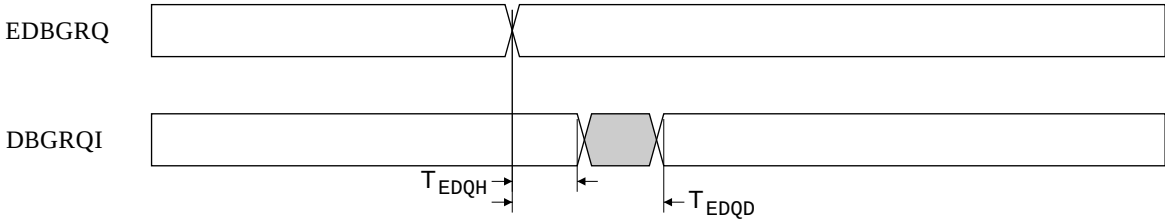

Figure 12-13 ARM940T EDBGRQ to DBGRQI relationship

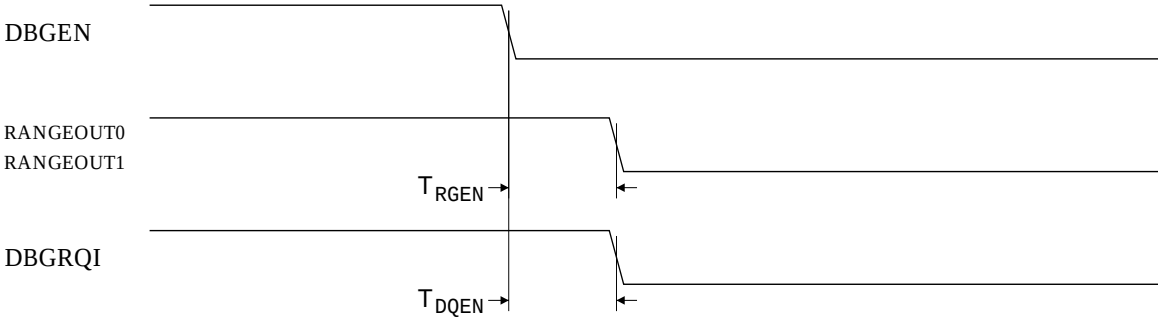

Figure 12-14 ARM940T DBGEN to Output relationship

12.3 ARM940T timing parameters

Table 12-1 ARM940T timing parameters

Timing parameter	Description
Tbbigd	**BIGENOUT** output delay from **BCLK** falling
Tbbigh	**BIGENOUT** output hold time from **BCLK** falling
Tbcand	**CPLATECANCEL** output delay from **BCLK** falling
Tbcanh	**CPLATECANCEL** output hold time from **BCLK** falling
Tbcdnh	**CPDIN[31:0]** set up time to **BCLK** falling
Tbcdns	**CPDIN[31:0]** set up time to **BCLK** falling
Tbchsh	**CHSDE[1:0]**/**CHSEX[1:0]** hold time to **BCLK** falling
Tbchss	**CHSDE[1:0]**/**CHSEX[1:0]** setup time to **BCLK** falling
Tbcomd	**COMMTX**/**COMMRX** output delay
Tbcomh	**COMMTX**/**COMMRX** output hold time
Tbcpdd	**CPID[31:0]**/**CPDOUT[31:0]** output delay from **BCLK** falling
Tbcpdh	**CPID[31:0]**/**CPDOUT[31:0]** output hold time from **BCLK** falling
Tbcpkf	Delay from **BCLK** falling to **CPCLK** falling
Tbcpkr	Delay from **BCLK** rising to **CPCLK** rising
Tbctld	**CPnMREQ**/**nCPTRANS**/**CPTBIT** output delay from **BCLK** falling
Tbctlh	**CPnMREQ**/**nCPTRANS**/**CPTBIT** output hold time from **BCLK** falling
Tbdbqh	**EDBGRQ** input hold time from **BCLK** falling
Tbdbqs	**EDBGRQ** input setup time to **BCLK** falling
Tbdckd	**DBGACK** output delay
Tbdckh	**DBGACK** output hold time
Tbekf	Delay from **BCLK** falling to **ECLK** falling
Tbekr	Delay from **BCLK** rising to **ECLK** rising
Tbexth	**EXTERN0**/**EXTERN1** input hold time from **BCLK** falling

Table 12-1 ARM940T timing parameters (continued)

Timing parameter	Description
Tbexts	**EXTERN0**/**EXTERN1** input setup time to **BCLK** falling
Tbinth	**nFIQ/nIRQ** hold time from **BCLK** falling
Tbints	**nFIQ/nIRQ** setup time to **BCLK** falling
Tbisyh	**ISYNC** hold time from **BCLK** falling
Tbisys	**ISYNC** setup time to **BCLK** falling
Tbnwtd	**CPnWAIT** output delay from **BCLK** rising
Tbnwth	**CPnWAIT** output hold time from **BCLK** rising
Tbpasd	**CPPASS** output delay from **BCLK** rising
Tbpash	**CPPASS** output hold time from **BCLK** rising
Tbrg0d	**RANGEOUT0** output delay
Tbrg0h	**RANGEOUT0** output hold time
Tbrg1h	**RANGEOUT1** output hold time
Tbrgqd	**RANGEOUT1** output delay
Tbrst	Delay from **nTRST** falling to **RSTCLKBS** rising
Tbrtd	**RSTCLKBS** output delay from **TCK** falling
Tbrth	**RSTCLKBS** hold time from **TCK** falling
Tbtrks	**TRACK** input setup time to **BCLK** falling
Tbtrsh	**TRACK** input hold time from **BCLK** falling
Tcapf	**ECAPCLKBS**/**ICAPCLKBS**/**PCLKBS** falling from **TCK** rising
Tcapr	**ECAPCLKBS**/**ICAPCLKBS**/**PCLKBS** rising from **TCK** rising
Tdgid	**DBGRQI** output delay from **TCK** falling
Tdgih	**DBGRQI** output hold time from **TCK** falling
Tdih	**TDI** and **TMS** hold time from **TCK** rising
Tdis	**TDI** and **TMS** setup time to **TCK** rising
Tdqen	Delay from **DBGEN** falling to **DBGRQI** falling

Table 12-1 ARM940T timing parameters (continued)

Timing parameter	Description
Tdqir	Delay from **nTRST** falling to **DBGRQI**
Tedqd	**DBGRQI** output delay from **EDBGRQ** falling
Tedqh	**DBGRQI** output hold time from **EDBGRQ** falling
Tfbigd	**BIGENOUT** output delay from **FCLK** falling
Tfbigh	**BIGENOUT** output hold time from **FCLK** falling
Tfcand	**CPLATECANCEL** output delay from **FCLK** falling
Tfcanh	**CPLATECANCEL** output hold time from **FCLK** falling
Tfcdnh	**CPDIN[31:0]** set up time to **FCLK** falling
Tfcdns	**CPDIN[31:0]** set up time to **FCLK** falling
Tfchsh	**CHSDE[1:0]/CHSEX[1:0]** hold time to **FCLK** falling
Tfchss	**CHSDE[1:0]/CHSEX[1:0]** setup time to **FCLK** falling
Tfcpdh	**CPID[31:0]/CPOUT[31:0]** output delay from **FCLK** falling
Tfcpkf	Delay from **FCLK** falling to **CPCLK** falling
Tfcpkr	Delay from **FCLK** rising to **CPCLK** rising
Tfctld	**CPnMREQ**/**nCPTRANS**/**CPTBIT** output delay from **FCLK** falling
Tfctlh	**CPnMREQ**/**nCPTRANS**/**CPTBIT** output hold time from **FCLK** falling
Tfekf	Delay from **FCLK** falling to **ECLK** falling
Tfekr	Delay from **FCLK** rising to **ECLK** rising
Tffkf	Delay from **FCLK** falling to **FCLKOUT** falling
Tffkr	Delay from **FCLK** rising to **FCLKOUT** rising
Tfinth	**nFIQ/nIRQ** hold time from **FCLK** falling
Tfints	**nFIQ/nIRQ** setup time to **FCLK** falling
Tfisyh	**ISYNC** hold time from **FCLK** falling
Tfisys	**ISYNC** setup time to **FCLK** falling

Table 12-1 ARM940T timing parameters (continued)

Timing parameter	Description
Tfnwtd	**CPnWAIT** output delay from **FCLK** rising
Tfnwth	**CPnWAIT** output hold time from **FCLK** rising
Tfpasd	**CPPASS** output delay from **FCLK** rising
Tfpash	**CPPASS** output hold time from **FCLK** rising
Tirsd	**IREG[3:0]/SCREG[3:0]** output delay from **TCK** falling
Tirsh	**IREG[3:0]/SCREG[3:0]** hold time from **TCK** falling
Trgen	Delay from **DBGEN** falling to **RANGEOUT0/RANGEOUT1** falling
Tsdnd	**SDIN** output delay from **TCK** falling
Tsdnh	**SDIN** hold time from **TCK** falling
Tshkf	**SHCLK1BS/SHCLK2BS** falling from **TCK** changing
Tshkr	**SHCLK1BS/SHCLK2BS** rising from **TCK** changing
Ttckf	**TCK1/TCK2** falling from **TCK** changing
Ttckr	**TCK1/TCK2** rising from **TCK** changing
Ttdod	**TDO** output delay from **TCK** falling
Ttdoh	**TDO** hold time from **TCK** falling
Ttdsd	**TDO** output delay from **SDOUTBS** changing
Ttdsh	**TDO** output hold time from **SDOUTBS** changing
Ttekf	Delay from **TCK** falling to **ECLK** falling
Ttekr	Delay from **TCK** rising to **ECLK** rising
Tteod	**nTDOEN** output delay from **TCK** falling
Tteoh	**nTDOEN** hold time from **TCK** falling
Ttpmd	**TAPSM[3:]** output delay from **TCK** falling
Ttpmh	**TAPSM[3:0]** hold time from **TCK** falling

Appendix A
ARM940T Signal Descriptions

This appendix lists and describes the ARM940T signals under the following headings:

- AMBA signals
- Coprocessor interface signals
- JTAG and TAP controller signals
- Debug signals
- Miscellaneous signals.

A.1 AMBA signals

Table A-1 AMBA signals

Name	Direction	Description
AGNT	Input	Bus Grant. A signal from the bus arbiter to a bus master which indicates that the bus master will be granted the bus when **BWAIT** is LOW.
AREQ	Output	Bus Request. A signal from the bus master to the bus arbiter which indicates that the ARM940T requires the bus.
BA[31:0]	Input/ Output	Address Bus. The processor address bus, which is 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 whether the current transfer should be the last of a burst sequence. When **BLAST** is HIGH, the decoder must allow sufficient time for address decoding. When **BLAST** is LOW, the next transfer may continue a burst sequence.
BLOK	Input/ 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 should 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.
BPROT[1:0]	Input/ 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 whether the transfer is an opcode fetch or data access, as well as whether the transfer is a privileged mode or user mode.
BSIZE[1:0]	Input/ Output	Transfer Size. These signals indicate the size of the transfer: 10encodes word access 01encodes a half word 00encodes a byte access 11reserved

Table A-1 AMBA signals

Name	Direction	Description
BTRAN[1:0]	Input/ Output	Transfer Type. These signals indicate the type of the next transaction: 00encodes an address-only transfer 01encodes a non-sequential transfer 11encodes a sequential transfer 01reserved
BWAIT	Input/ Output	Wait Response. This signal is driven by the selected bus slave to indicate whether the current transfer may complete. If **BWAIT** is HIGH, a further bus cycle is required, if **BWAIT** is LOW, the transfer will complete 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.

A.1.1 AMBA Bus Specification

ARM940T has an AMBA-compatible bus interface. Please refer to the *AMBA Specification* for full details.

A.2 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.
CDPID[31:0]	Output	Coprocessor Instruction Data. This is the coprocessor instruction data bus over which instructions are transferred 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 should be cancelled without having updated the coprocessor state.
nCPMREQ	Output	Not Coprocessor Memory Request. When LOW on a rising **CPCLK** edge and **nCPWAIT** LOW, the instruction on **CPID** should enter the coprocessor pipeline follower's decode stage. The second instruction previously in the pipeline followers decode stage should enter 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 should 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 whilst **CPCLK** is HIGH.

For further information on the coprocessor interface refer to *Chapter 7 ARM940T Coprocessor Interface*.

A.3 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 multiplexers 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 should 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 should 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 should 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 should 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 may 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**.
SDOUTBS	Input	Boundary Scan Serial Output Data. This is the serial data out of the boundary scan chain (or other external scan chain). It should be set up to the rising edge of **TCK**. When an external boundary scan chain is not connected, this input should be tied LOW.

Table A-3 JTAG and TAP controller signals (continued)

Name	Direction	Description
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.
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 due to the internal clock non-overlap.
TCK2	Output	TCK, Phase 2. **TCK2** is HIGH when **TCK** is LOW, although there is a slight phase lag due to the internal clock non-overlap.
TDI	Input	Test Data Input. JTAG serial input.
TDO	Output	Test Data Output. JTAG serial output.
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 to which state the TAP controller state machine should change.
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 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 should be LOW only when debugging will not be required.
DBGRQI	Output	Internal Debug Request. This signal represents the debug request signal which is 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 once execution of the current instruction has completed.
EXTERN0	Input	External Input 0. This is an input to watchpoint unit 0 of the EmbeddedICE logic 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 logic 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 was executed.
RANGEOUT0	Output	EmbeddedICE Rangeout 0. This signal indicates that the EmbeddedICE 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 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 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.
BURST[1:0]	Output	Burst Access. These signals indicate the length of a burst transfer: 00No sequential information available (default) 01Reserved 10Current access is part of a 4-word transfer 11Reserved
ENAMADRV	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.
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 should be applied synchronously.
nFIQ	Input	Not Fast Interrupt request. This is the not Fast Interrupt Request (**nFIQ)** signal**.**
nIRQ	Input	Not Interrupt Request. This is the not Interrupt Request (**nIRQ)** signal**.**

Index
ARM940T Technical Reference Manual

The items in this index are listed in alphabetic order, with symbols and numerics appearing at the front. The references given are to page numbers.

A

B

C

D

E

I

J

L

M

O

P

S

T

V

W

