ARM720T

(Rev 3)
Technical Reference Manual

ARM

ARM DDI 0192A

ARM720T
Technical Reference Manual

Release information

Change history

Date	Issue	Change
September 2000	A	First release

Proprietary notice

Document confidentiality status

This document is Open Access. This means there is no restriction on the distribution of the information.

Product status

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

ARM web address

`http://www.arm.com`

Contents
ARM720T Technical Reference Manual

List of Tables vii

List of Figures ix

Preface
About this document xii
Further reading xv
Feedback xvi

Chapter 1 **Introduction**
1.1 About the ARM720T 1-2
1.2 Coprocessors 1-4
1.3 About the instruction set 1-5

Chapter 2 **Programmer's Model**
2.1 Processor operating states 2-2
2.2 Memory formats 2-3
2.3 Instruction length 2-5
2.4 Data types 2-6
2.5 Operating modes 2-7
2.6 Registers 2-8
2.7 The program status registers 2-13

2.8 Exceptions 2-16
2.9 Relocation of low virtual addresses by the FCSE PID 2-22
2.10 Reset 2-23
2.11 Implementation-defined behavior of instructions 2-24

Chapter 3 Configuration
3.1 About configuration 3-2
3.2 Internal coprocessor instructions 3-3
3.3 Registers 3-4

Chapter 4 Instruction and Data Cache
4.1 About the instruction and data cache 4-2
4.2 IDC validity 4-4
4.3 IDC enable, disable, and reset 4-5
4.4 IDC disable for secure applications 4-6

Chapter 5 Write Buffer
5.1 About the write buffer 5-2
5.2 Write buffer operation 5-3

Chapter 6 Memory Management Unit
6.1 About the MMU 6-2
6.2 MMU program accessible registers 6-4
6.3 Address translation process 6-5
6.4 Level 1 descriptor 6-7
6.5 Page table descriptor 6-8
6.6 Section descriptor 6-9
6.7 Translating section references 6-11
6.8 Level 2 descriptor 6-12
6.9 Translating small page references 6-14
6.10 Translating large page references 6-16
6.11 MMU faults and CPU aborts 6-18
6.12 Fault address and fault status registers 6-19
6.13 Domain access control 6-21
6.14 Fault checking sequence 6-22
6.15 External aborts 6-25
6.16 Interaction of the MMU, IDC, and write buffer 6-26

Chapter 7 Debug Interface
7.1 About the debug interface 7-2
7.2 Debug systems 7-4
7.3 Entering debug state 7-7
7.4 Scan chains and JTAG interface 7-9
7.5 Reset 7-11
7.6 Public instructions 7-12
7.7 Test data registers 7-16
7.8 ARM7TDM core clocks 7-23

7.9 Determining the core and system state 7-25
7.10 The PC during debug 7-30
7.11 Priorities and exceptions 7-34
7.12 Scan interface timing 7-35
7.13 Scan and debug signals used by the embedded trace logic 7-42

Chapter 8 EmbeddedICE Logic
8.1 About EmbeddedICE Logic 8-2
8.2 The watchpoint registers 8-4
8.3 Programming breakpoints 8-9
8.4 Programming watchpoints 8-11
8.5 The debug control register 8-13
8.6 Debug status register 8-15
8.7 Coupling breakpoints and watchpoints 8-17
8.8 Debug communications channel 8-19

Chapter 9 Bus Clocking
9.1 About the ARM720T bus interface 9-2
9.2 Fastbus extension 9-3
9.3 Standard mode 9-5

Chapter 10 AMBA Interface
10.1 About the AMBA interface 10-2
10.2 ASB bus interface signals 10-3
10.3 Cycle types 10-4
10.4 Addressing signals 10-7
10.5 Memory request signals 10-8
10.6 Data signal timing 10-9
10.7 Slave response signals 10-10
10.8 Maximum sequential length 10-12
10.9 Read-lock-write 10-13
10.10 Little-endian and big-endian operation 10-14
10.11 Multi-master operation 10-17
10.12 Bus master handover 10-19
10.13 Default bus master 10-21

Chapter 11 AMBA Test
11.1 Slave operation, test mode 11-2
11.2 ARM720T test mode 11-3
11.3 ARM7TDM core test mode 11-5
11.4 RAM test mode 11-6
11.5 TAG test mode 11-8
11.6 MMU test mode 11-10
11.7 Test register mapping 11-11

Chapter 12 Trace Interface Port
12.1 About the ETM 12-2

12.2 ETM interface 12-3

Appendix A Signal Descriptions

A.1 AMBA interface signals A-2
A.2 Coprocessor interface signals A-5
A.3 JTAG signals A-7
A.4 Debugger signals A-9
A.5 Embedded trace macrocell interface signals A-10
A.6 Miscellaneous signals A-12
A.7 Additional signal outputs A-13

Index ..... Index-i

List of Tables
ARM720T Technical Reference Manual

Table 1-1 Key to tables 1-5
Table 1-2 ARM instruction summary 1-8
Table 1-3 Addressing mode 2 1-11
Table 1-4 Addressing mode 2 (privileged) 1-12
Table 1-5 Addressing mode 3 1-12
Table 1-6 Addressing mode 4 (load) 1-13
Table 1-7 Addressing mode 4 (store) 1-13
Table 1-8 Addressing mode 5 1-14
Table 1-9 Operand 2 1-14
Table 1-10 Fields 1-14
Table 1-11 Condition fields 1-15
Table 1-12 Thumb instruction summary 1-17
Table 2-1 ARM720T modes of operation 2-7
Table 2-2 PSR mode bit values 2-14
Table 2-3 Exception entry and exit 2-17
Table 2-4 Exception vector addresses 2-20
Table 3-1 Cache and MMU control register 3-4
Table 3-2 Cache operation 3-9
Table 3-3 TLB operations 3-10
Table 6-1 MMU program accessible registers 6-4
Table 6-2 Interpreting level 1 descriptor bits [1:0] 6-7
Table 6-3 Interpreting access permission (AP) bits 6-10
Table 6-4 Interpreting page table entry bits 1:0 6-12

Table 6-5 Priority encoding of fault status 6-19
Table 6-6 Interpreting access bits in domain access control register 6-21
Table 6-7 Valid MMU, IDC and write buffer combinations 6-26
Table 7-1 Scan chain number allocation 7-18
Table 7-2 ARM720T scan interface timing 7-35
Table 7-3 Scan chain 0, signals and positions 7-37
Table 7-4 Scan and debug signals used by the ETM 7-42
Table 8-1 Function and mapping of EmbeddedICE registers 8-4
Table 8-2 MAS[1:0] signal encoding 8-7
Table 8-3 IFEN signal control 8-14
Table 10-1 BTRAN[1:0] encoding 10-8
Table 11-1 RAM test mode address packet bit positions 11-6
Table 11-2 TAG test mode TAG CTL packet bit positions 11-9
Table 11-3 Status packet bit positions bits [31:0] 11-11
Table 11-4 Control packet bit positions bits [31:0] 11-13
Table A-1 AMBA signal descriptions A-2
Table A-2 Coprocessor interface signal descriptions A-5
Table A-3 JTAG signal descriptions A-7
Table A-4 Debugger signal descriptions A-9
Table A-5 ETM interface signal descriptions A-10
Table A-6 Miscellaneous signal descriptions A-12

List of Figures
ARM720T Technical Reference Manual

Figure 1-1 Block diagram 1-3
Figure 1-2 ARM instruction set formats 1-7
Figure 1-3 Thumb instruction set formats 1-16
Figure 2-1 Big-endian addresses of bytes with words 2-3
Figure 2-2 Little-endian addresses of bytes with words 2-4
Figure 2-3 Register organization in ARM state 2-9
Figure 2-4 Register organization in Thumb state 2-10
Figure 2-5 Mapping of Thumb state registers onto ARM state registers 2-11
Figure 2-6 Program status register format 2-13
Figure 3-1 MRC and MCR bit pattern 3-3
Figure 3-2 ID register read 3-5
Figure 3-3 ID register write 3-5
Figure 3-4 Register 1 read 3-5
Figure 3-5 Register 1 write 3-5
Figure 3-6 Register 2 3-7
Figure 3-7 Register 3 3-7
Figure 3-8 Register 4 3-8
Figure 3-9 Register 5 3-8
Figure 3-10 Register 6 3-9
Figure 3-11 Register 13 with opcode_2=0 3-11
Figure 3-12 Register 13 with opcode_2=1 3-11
Figure 6-1 Translation table base register 6-5
Figure 6-2 Accessing the translation table first level descriptors 6-6

Figure 6-3 Level 1 descriptors 6-7
Figure 6-4 Section translation 6-11
Figure 6-5 Page table entry, level 2 descriptor 6-12
Figure 6-6 Small page translation 6-15
Figure 6-7 Large page translation 6-17
Figure 6-8 Domain access control register format 6-21
Figure 6-9 Sequence for checking faults 6-22
Figure 7-1 Typical debug system 7-4
Figure 7-2 ARM7TDM scan chain arrangement 7-6
Figure 7-3 Test access port (TAP) controller state transitions 7-10
Figure 7-4 ID code register format 7-16
Figure 7-5 Input scan cell 7-19
Figure 7-6 Clock switching on entry to debug state 7-24
Figure 7-7 Scan general timing 7-35
Figure 7-8 Reset period timing 7-36
Figure 7-9 Output enable and disable times due to HIGHZ TAP instruction 7-36
Figure 7-10 Output enable and disable times due to data scanning 7-37
Figure 8-1 ARM7TDMI TAP controller and EmbeddedICE 8-2
Figure 8-2 EmbeddedICE block diagram 8-5
Figure 8-3 Watchpoint control value and mask format 8-6
Figure 8-4 Debug control register format 8-13
Figure 8-5 Debug status register format 8-15
Figure 8-6 Debug control and status register structure 8-16
Figure 8-7 Debug comms control register 8-19
Figure 9-1 Conceptual device clocking using the fastbus extension 9-3
Figure 9-2 Conceptual device clocking in standard mode 9-5
Figure 9-3 Relationship of FCLK and BCLK in synchronous mode 9-7
Figure 10-1 Simple single-cycle access 10-4
Figure 10-2 Simple sequential access 10-5
Figure 10-3 Minimum interval between bus accesses 10-6
Figure 10-4 Use of the BWAIT pin to stop ARM720T for 1 BCLK cycle 10-11
Figure 10-5 Little-endian addresses of bytes within words 10-14
Figure 10-6 Big-endian addresses of bytes within words 10-15
Figure 10-7 Bus master handover 10-19
Figure 11-1 Running a test vector on the processor core 11-2
Figure 11-2 State machine for ARM720T and ARM7TDMI test 11-3
Figure 11-3 .State machine for RAM test mode 11-6
Figure 11-4 State machine for TAG test mode 11-8
Figure 11-5 State machine for MMU test mode 11-10
Figure 12-1 ETM interface signal timing 12-3
Figure 12-2 ETMCLK power saving 12-4

Preface

This preface introduces the ARM720T and its reference documentation. It contains the following sections:

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

About this document

This document is a technical reference manual for the ARM720T.

Intended audience

This document has been written for experienced hardware and software engineers who might or might not have experience of the architecture, configuration, integration, and instruction sets with reference to the ARM product range.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM720T.

Chapter 2 *Programmer's Model*

Read this chapter for a description of the 32-bit ARM and 16-bit Thumb instruction sets.

Chapter 3 *Configuration*

Read this chapter for a description of how the operation and configuration of the ARM720T is controlled.

Chapter 4 *Instruction and Data Cache*

Read this chapter for an overview of the mixed instruction and data cache.

Chapter 5 *Write Buffer*

Read this chapter for a description of how you can enhance the system performance of the ARM720T by using the write buffer.

Chapter 6 *Memory Management Unit*

Read this chapter for a description of the functions and use of the memory management unit.

Chapter 7 *Debug Interface*

Read this chapter for a description of the hardware extensions used for advanced debugging.

Chapter 8 *EmbeddedICE Logic*

Read this chapter for a description of the integrated on-chip debug support for the ARM720T core.

Chapter 9 *Bus Clocking*

Read this chapter for a description of the ARM720T bus interface.

Chapter 10 *AMBA Interface*

Read this chapter for a description of the functions and operation of the ARM720T bus master.

Chapter 11 *AMBA Test*

Read this chapter for a description of the ARM720T test features.

Chapter 12 *Trace Interface Port*

Read this chapter for a description of the Embedded Trace Macrocell support for the ARM720T.

Appendix A *Signal Descriptions*

Read this appendix for a list of all ARM720T interface signals.

Typographical conventions

The following typographical conventions are used in this document:

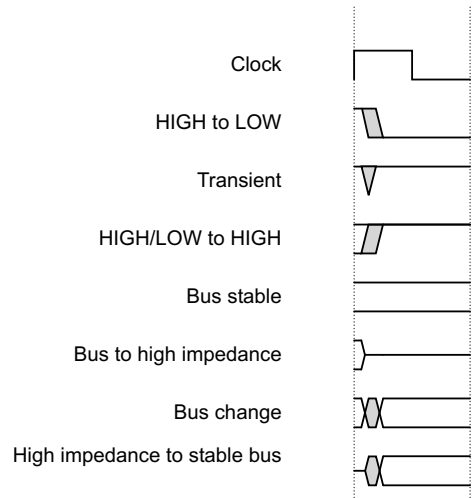
bold	Highlights ARM processor signal names, and interface elements such as menu names. Also used for terms in descriptive lists, where appropriate.
italic	Highlights special terminology, cross-references, and citations.
`typewriter`	Denotes text that can be entered at the keyboard, such as commands, file names and program names, and source code.
`typewriter`	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`typewriter italic`
: Denotes arguments to commands or functions where the argument is to be replaced by a specific value.

`typewriter bold`
: Denotes language keywords when used outside example code.

Timing diagram conventions

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



Key to timing diagram conventions

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

Further reading

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

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

This document contains information that is specific to the ARM720T. Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *AMBA Specification* (ARM IHI 0001)
- *ETM7 Technical Reference Manual* (ARM DDI 0158)
- *ARM7TDMI Technical Reference Manual* (ARM DDI 0029).

Other publications

This section lists relevant documents published by third parties.

- *Standard Test Access Port and Boundary Scan Architecture* (IEEE Std. 1149.1.1990).

Feedback

ARM Limited welcomes feedback both on the ARM720T, and on the documentation.

Feedback on the ARM720T

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

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

Feedback on the ARM720T documentation

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

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

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter provides an introduction to the ARM720T. It contains the following sections:

- *About the ARM720T* on page 1-2
- *Coprocessors* on page 1-4
- *About the instruction set* on page 1-5.

1.1 About the ARM720T

The ARM720T is a general-purpose 32-bit microprocessor with 8KB cache, enlarged write buffer, and *Memory Management Unit* (MMU) combined in a single chip. The CPU within the ARM720T is the ARM7TDMI. The ARM720T is software-compatible with the ARM processor family.

The on-chip mixed data and instruction cache, together with the write buffer, substantially raise the average execution speed and reduce the average amount of memory bandwidth required by the processor. This allows the external memory to support additional processors or *Direct Memory Access* (DMA) channels with minimal performance loss.

The allocation of virtual addresses with different task IDs improve performance in task switching operations with the cache enabled. These relocated virtual addresses are monitored by the EmbeddedICE block.

The MMU supports a conventional two-level, page-table structure and a number of extensions that make it ideal for embedded control, UNIX, and object-oriented systems.

The memory interface is designed to allow the performance potential to be realized without incurring high costs in the memory system. Speed-critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic, and these control signals permit the exploitation of paged mode access offered by industry-standard DRAMs.

The ARM720T is provided with an *Embedded Trace Macrocell* (ETM) interface that brings out the required signals from the ARM core to the periphery of the ARM720T. This allows you to connect a standard ETM7 macrocell.

ARM720T is a fully static part and has been designed to minimize power requirements. This makes it ideal for portable applications where both features are essential.

The ARM720T architecture is based on *Reduced Instruction Set Computer* (RISC) principles. The instruction set and related decode mechanism are greatly simplified compared with microprogrammed *Complex Instruction Set Computers* (CISCs).

A block diagram of the ARM720T is shown in Figure 1-1 on page 1-3.

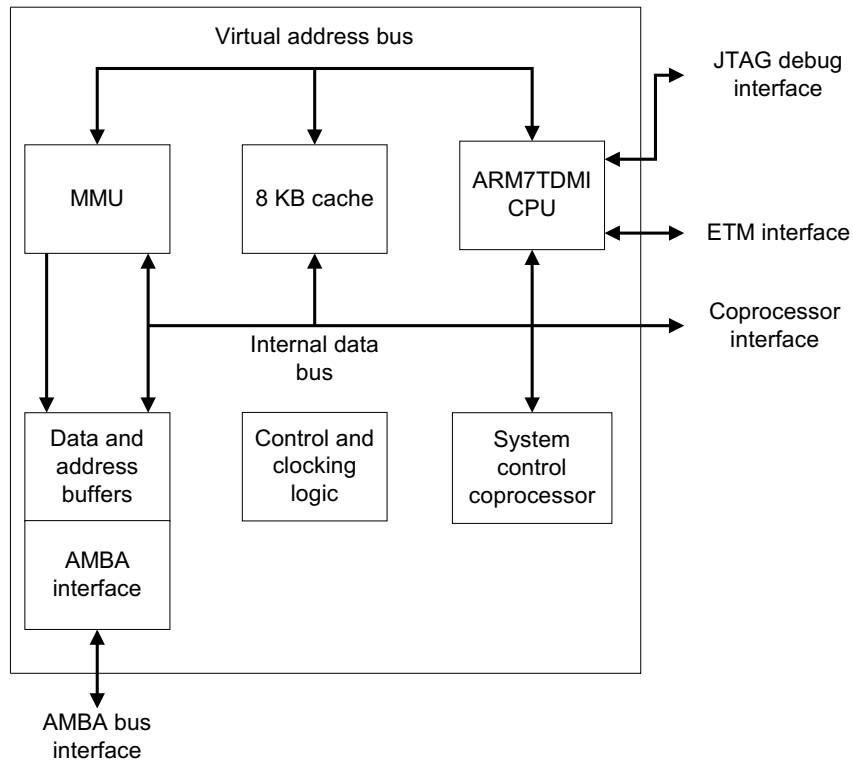


Figure 1-1 Block diagram

1.2 Coprocessors

The ARM720T has an internal coprocessor designated CP15 for internal control of the device (see *Registers* on page 3-4).

The ARM720T also includes a port for the connection of on-chip coprocessors. These allow extension of the ARM720T functionality in an architecturally consistent manner.

1.3 About the instruction set

The instruction set comprises ten basic instruction types:

- Two types use the on-chip arithmetic logic unit, barrel shifter, and multiplier to perform high-speed operations on the data in a bank of 31 registers, each 32 bits wide.
- Three types of instruction control the data transfer between memory and the registers:
  - one optimized for flexibility of addressing
  - one for rapid context switching
  - one for swapping data.
- Two instructions control the flow and privilege level of execution.
- Three types are dedicated to the control of external coprocessors. These allow you to extend the functionality of the instruction set off-chip in an open and uniform way.

The ARM instruction set is a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors that depend on sophisticated compiler technology to manage complicated instruction interdependencies.

1.3.1 Format summary

This section provides a summary of the ARM and Thumb instruction sets:

- *ARM instruction set* on page 1-6
- *Thumb instruction set* on page 1-15.

A key to the instruction set tables is listed in Table 1-1.

The ARM7TDMI is an implementation of the ARMv4T architecture. For a complete description of both instruction sets, see the *ARM Architecture Reference Manual*.

Table 1-1 Key to tables

	Description
`{cond}`	Refer to Table 1-11 on page 1-15.
`<Oprnd2>`	Refer to Table 1-9 on page 1-14.
`{field}`	Refer to Table 1-10 on page 1-14.

Table 1-1 Key to tables (continued)

S	Sets condition codes (optional).
B	Byte operation (optional).
H	Halfword operation (optional).
T	Forces address translation. Cannot be used with pre-indexed addresses.
<a_mode2>	Refer to Table 1-3 on page 1-11.
<a_mode2P>	Refer to Table 1-4 on page 1-12.
<a_mode3>	Refer to Table 1-5 on page 1-12.
<a_mode4L>	Refer to Table 1-6 on page 1-13.
<a_mode4S>	Refer to Table 1-7 on page 1-13.
<a_mode5>	Refer to Table 1-8 on page 1-14.
#32bit_Imm	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.
<reglist>	A comma-separated list of registers, enclosed in braces ({ and }).

1.3.2 ARM instruction set

This section gives an overview of the ARM instructions available. For full details of these instructions, refer to the *ARM Architecture Reference Manual.*

The ARM instruction set formats are shown at Figure 1-2 on page 1-7.

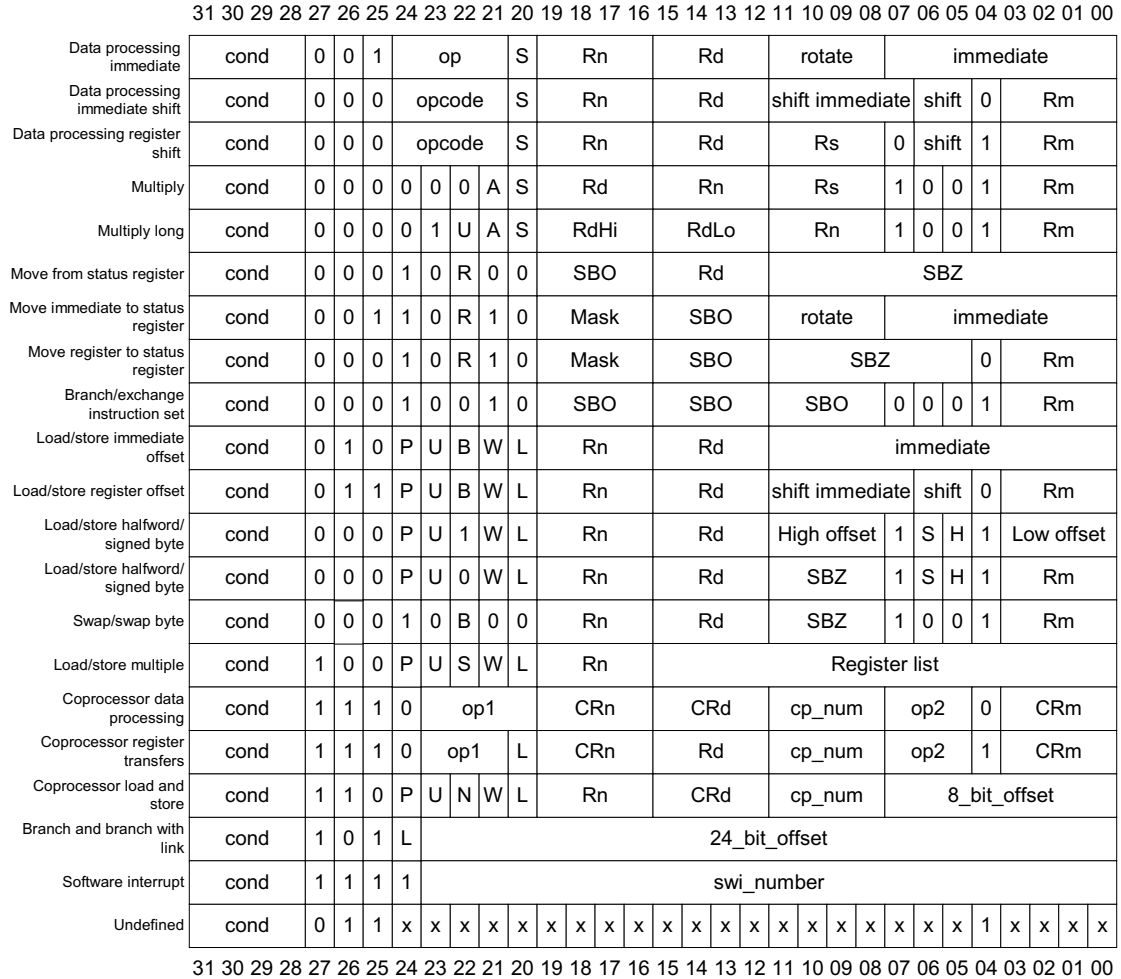


Figure 1-2 ARM instruction set formats

Note

Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken. For example, a multiply instruction with bit 6 changed to a 1. You must not use these instructions, as their action might change in future ARM implementations.

The ARM instruction set summary is listed in Table 1-2.

Table 1-2 ARM instruction summary

Operation		Assembler
Move	Move	`MOV{cond}{S} Rd, <Oprnd2>`
	Move NOT	`MVN{cond}{S} Rd, <Oprnd2>`
	Move SPSR to register	`MRS{cond} Rd, SPSR`
	Move CPSR to register	`MRS{cond} Rd, CPSR`
	Move register to SPSR	`MSR{cond} SPSR{field}, Rm`
	Move register to CPSR	`MSR{cond} CPSR{field}, Rm`
	Move immediate to SPSR flags	`MSR{cond} SPSR_f, #32bit_Imm`
	Move immediate to CPSR flags	`MSR{cond} CPSR_f, #32bit_Imm`
Arithmetic	Add	`ADD{cond}{S} Rd, Rn, <Oprnd2>`
	Add with carry	`ADC{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract	`SUB{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract with carry	`SBC{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract reverse subtract	`RSB{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract reverse subtract with carry	`RSC{cond}{S} Rd, Rn, <Oprnd2>`
	Multiply	`MUL{cond}{S} Rd, Rm, Rs`
	Multiply accumulate	`MLA{cond}{S} Rd, Rm, Rs, Rn`
	Multiply unsigned long	`UMULL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply unsigned accumulate long	`UMLAL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply signed long	`SMULL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply signed accumulate long	`SMLAL{cond}{S} RdLo, RdHi, Rm, Rs`
	Compare	`CMP{cond} Rd, <Oprnd2>`
	Compare negative	`CMN{cond} Rd, <Oprnd2>`

Table 1-2 ARM instruction summary (continued)

Operation		Assembler
Logical	Test	`TST{cond} Rn, <Oprnd2>`
	Test equivalence	`TEQ{cond} Rn, <Oprnd2>`
	AND	`AND{cond}{S} Rd, Rn, <Oprnd2>`
	EOR	`EOR{cond}{S} Rd, Rn, <Oprnd2>`
	ORR	`ORR{cond}{S} Rd, Rn, <Oprnd2>`
	Bit clear	`BIC{cond}{S} Rd, Rn, <Oprnd2>`
Branch	Branch	`B{cond} label`
	Branch with link	`BL{cond} label`
	Branch, and exchange instruction set	`BX{cond} Rn`
Load	Word	`LDR{cond} Rd, <a_mode2>`
	Word with User Mode privilege	`LDR{cond}T Rd, <a_mode2P>`
	Byte	`LDR{cond}B Rd, <a_mode2>`
	Byte with User Mode privilege	`LDR{cond}BT Rd, <a_mode2P>`
	Byte signed	`LDR{cond}SB Rd, <a_mode3>`
	Halfword	`LDR{cond}H Rd, <a_mode3>`
	Halfword signed	`LDR{cond}SH Rd, <a_mode3>`
	Multiple	
	Block data operations	
	Increment before	`LDM{cond}IB Rd{!}, <reglist>{^}`
	Increment after	`LDM{cond}IA Rd{!}, <reglist>{^}`
	Decrement before	`LDM{cond}DB Rd{!}, <reglist>{^}`
	Decrement after	`LDM{cond}DA Rd{!}, <reglist>{^}`
	Stack operations	`LDM{cond}<a_mode4L> Rd{!}, <reglist>`
	Stack operations, and restore CPSR	`LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^`

Table 1-2 ARM instruction summary (continued)

Operation		Assembler
	User registers	`LDM{cond}<a_mode4L> Rd{!}, <reglist>^`
Store	Word	`STR{cond} Rd, <a_mode2>`
	Word with User Mode privilege	`STR{cond}T Rd, <a_mode2P>`
	Byte	`STR{cond}B Rd, <a_mode2>`
	Byte with User Mode privilege	`STR{cond}BT Rd, <a_mode2P>`
	Halfword	`STR{cond}H Rd, <a_mode3>`
	Multiple	
	Block data operations	
	Increment before	`STM{cond}IB Rd{!}, <reglist>{^}`
	Increment after	`STM{cond}IA Rd{!}, <reglist>{^}`
	Decrement before	`STM{cond}DB Rd{!}, <reglist>{^}`
	Decrement after	`STM{cond}DA Rd{!}, <reglist>{^}`
	Stack operations	`STM{cond}<a_mode4S> Rd{!}, <reglist>`
	User registers	`STM{cond}<a_mode4S> Rd{!}, <reglist>^`
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`
	Byte	`SWP{cond}B Rd, Rm, [Rn]`
Coprocessors	Data operations	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`
	Move to ARM reg from coproc	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Move to coproc from ARM reg	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode5>`
	Store	`STC{cond} p<cpnum>, CRd, <a_mode5>`
Software Interrupt		`SWI 24bit_Imm`

Addressing mode 2 is listed in Table 1-3.

Table 1-3 Addressing mode 2

Addressing mode 2 `<a_mode2>`	
Immediate offset	`[Rn, #+/-12bit_Offset]`
Register offset	`[Rn, +/-Rm]`
Scaled register offset	`[Rn, +/-Rm, LSL #5bit_shift_imm]`
	`[Rn, +/-Rm, LSR #5bit_shift_imm]`
	`[Rn, +/-Rm, ASR #5bit_shift_imm]`
	`[Rn, +/-Rm, ROR #5bit_shift_imm]`
	`[Rn, +/-Rm, RRX]`
Pre-indexed offset	
Immediate	`[Rn, #+/-12bit_Offset]!`
Register	`[Rn, +/-Rm]!`
Scaled register	`[Rn, +/-Rm, LSL #5bit_shift_imm]!`
	`[Rn, +/-Rm, LSR #5bit_shift_imm]!`
	`[Rn, +/-Rm, ASR #5bit_shift_imm]!`
	`[Rn, +/-Rm, ROR #5bit_shift_imm]!`
	`[Rn, +/-Rm, RRX]!`
Post-indexed offset	
Immediate	`[Rn], #+/-12bit_Offset`
Register	`[Rn], +/-Rm`
Scaled register	`[Rn], +/-Rm, LSL #5bit_shift_imm`
	`[Rn], +/-Rm, LSR #5bit_shift_imm`
	`[Rn], +/-Rm, ASR #5bit_shift_imm`
	`[Rn], +/-Rm, ROR #5bit_shift_imm`
	`[Rn, +/-Rm, RRX]`

Addressing mode 2 (privileged) is listed in Table 1-4.

Table 1-4 Addressing mode 2 (privileged)

Addressing mode 2 (privileged) `<a_mode2P>`	
Immediate offset	`[Rn, #+/-12bit_Offset]`
Register offset	`[Rn, +/-Rm]`
Scaled register offset	`[Rn, +/-Rm, LSL #5bit_shift_imm]`
	`[Rn, +/-Rm, LSR #5bit_shift_imm]`
	`[Rn, +/-Rm, ASR #5bit_shift_imm]`
	`[Rn, +/-Rm, ROR #5bit_shift_imm]`
	`[Rn, +/-Rm, RRX]`
Post-indexed offset	
Immediate	`[Rn], #+/-12bit_Offset`
Register	`[Rn], +/-Rm`
Scaled register	`[Rn], +/-Rm, LSL #5bit_shift_imm`
	`[Rn], +/-Rm, LSR #5bit_shift_imm`
	`[Rn], +/-Rm, ASR #5bit_shift_imm`
	`[Rn], +/-Rm, ROR #5bit_shift_imm`
	`[Rn, +/-Rm, RRX]`

Addressing mode 3 is listed in Table 1-5.

Table 1-5 Addressing mode 3

Addressing mode 3 - signed byte, and halfword data transfer `<a_mode3>`	
Immediate offset	`[Rn, #+/-8bit_Offset]`
Pre-indexed	`[Rn, #+/-8bit_Offset]!`
Post-indexed	`[Rn], #+/-8bit_Offset`

Table 1-5 Addressing mode 3 (continued)

Register	`[Rn, +/-Rm]`
Pre-indexed	`[Rn, +/-Rm]!`
Post-indexed	`[Rn], +/-Rm`

Addressing mode 4 (load) is listed in Table 1-6.

Table 1-6 Addressing mode 4 (load)

Addressing mode 4 (Load) `<a_mode4L>`			
Addressing mode		Stack type	
`IA`	Increment after	`FD`	Full descending
`IB`	Increment before	`ED`	Empty descending
`DA`	Decrement after	`FA`	Full ascending
`DB`	Decrement before	`EA`	Empty ascending

Addressing mode 4 (store) is listed in Table 1-7.

Table 1-7 Addressing mode 4 (store)

Addressing mode 4 (Store) `<a_mode4S>`			
Addressing mode		Stack type	
`IA`	Increment after	`EA`	Empty ascending
`IB`	Increment before	`FA`	Full ascending
`DA`	Decrement after	`ED`	Empty descending
`DB`	Decrement before	`FD`	Full descending

Addressing mode 5 is listed in Table 1-8.

Table 1-8 Addressing mode 5

Addressing mode 5 - coprocessor data transfer `<a_mode5>`	
Immediate offset	`[Rn, #+/-(8bit_Offset*4)]`
Pre-indexed	`[Rn, #+/-(8bit_Offset*4)]!`
Post-indexed	`[Rn], #+/-(8bit_Offset*4)`

Operand 2 is listed in Table 1-9.

Table 1-9 Operand 2

Operand 2 `<Oprnd2>`	
Immediate value	`#32bit_Imm`
Logical shift left	`Rm LSL #5bit_Imm`
Logical shift right	`Rm LSR #5bit_Imm`
Arithmetic shift right	`Rm ASR #5bit_Imm`
Rotate right	`Rm ROR #5bit_Imm`
Register	`Rm`
Logical shift left	`Rm LSL Rs`
Logical shift right	`Rm LSR Rs`
Arithmetic shift right	`Rm ASR Rs`
Rotate right	`Rm ROR Rs`
Rotate right extended	`Rm RRX`

Fields are listed in Table 1-10.

Table 1-10 Fields

Field `{field}`	
Suffix	Sets
_c	Control field mask bit (bit 3)

Table 1-10 Fields (continued)

_f	Flags field mask bit (bit 0)
_s	Status field mask bit (bit 1)
_x	Extension field mask bit (bit 2)

Condition fields are listed in Table 1-11.

Table 1-11 Condition fields

Condition field {cond}		
Suffix	**Description**	**Condition(s)**
EQ	Equal	Z set
NE	Not equal	Z clear
CS	Unsigned higher, or same	C set
CC	Unsigned lower	C clear
MI	Negative	N set
PL	Positive, or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set, Z clear
LS	Unsigned lower, or same	C clear, Z set
GE	Greater, or equal	N=V (N and V set or N and V clear)
LT	Less than	N<>V (N set and V clear) or (N clear and V set)
GT	Greater than	Z clear, N=V (N and V set or N and V clear)
LE	Less than, or equal	Z set or N<>V (N set and V clear) or (N clear and V set)
AL	Always	Always

1.3.3 Thumb instruction set

This section gives an overview of the Thumb instructions available. For full details of these instructions, see the *ARM Architecture Reference Manual*.

The Thumb instruction set formats are shown in Figure 1-3.

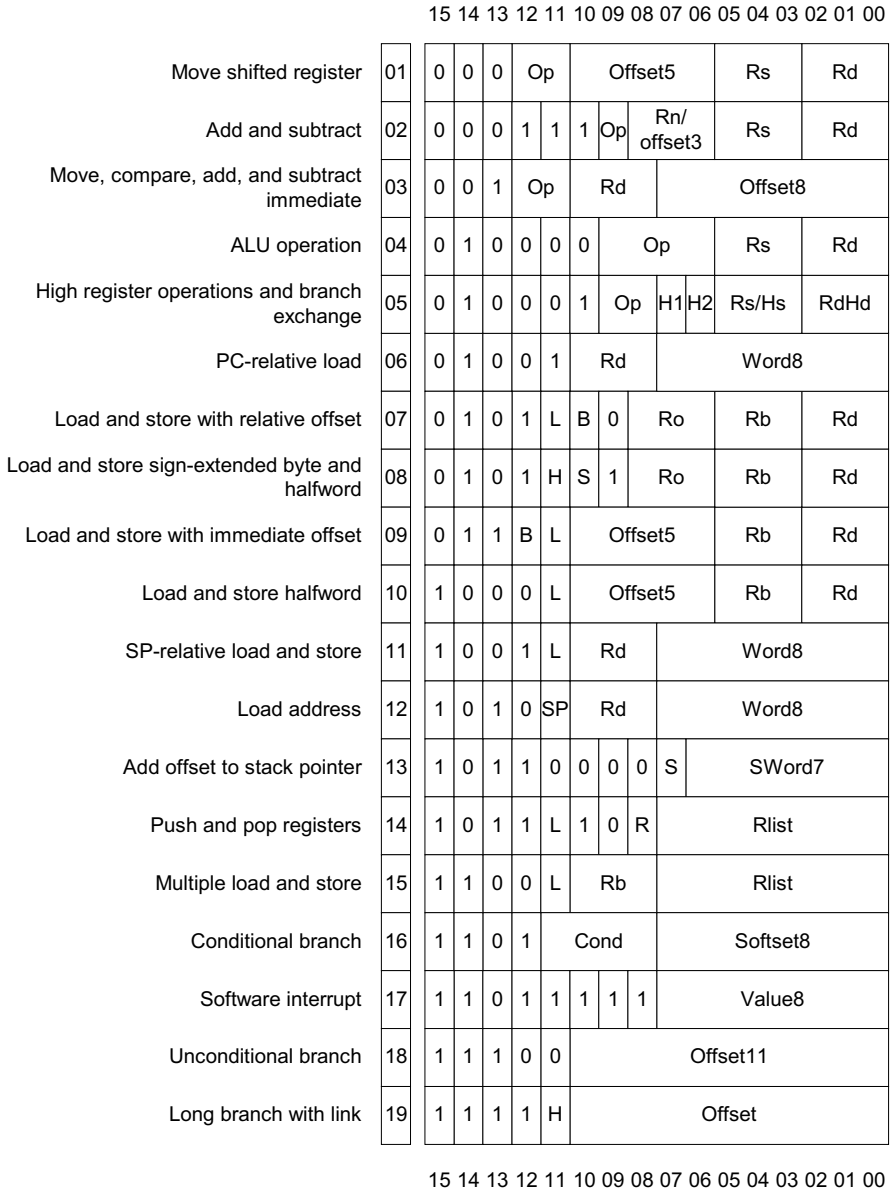


Figure 1-3 Thumb instruction set formats

The Thumb instruction set summary is listed in Table 1-12.

Table 1-12 Thumb instruction summary

Operation		Assembler
Move	Immediate	`MOV Rd, #8bit_Imm`
	High to Low	`MOV Rd, Hs`
	Low to High	`MOV Hd, Rs`
	High to High	`MOV Hd, Hs`
Arithmetic	Add	`ADD Rd, Rs, #3bit_Imm`
	Add Low, and Low	`ADD Rd, Rs, Rn`
	Add High to Low	`ADD Rd, Hs`
	Add Low to High	`ADD Hd, Rs`
	Add High to High	`ADD Hd, Hs`
	Add Immediate	`ADD Rd, #8bit_Imm`
	Add Value to SP	`ADD SP, #7bit_Imm` `ADD SP, #-7bit_Imm`
	Add with carry	`ADC Rd, Rs`
	Subtract	`SUB Rd, Rs, Rn` `SUB Rd, Rs, #3bit_Imm`
	Subtract Immediate	`SUB Rd, #8bit_Imm`
	Subtract with carry	`SBC Rd, Rs`
	Negate	`NEG Rd, Rs`
	Multiply	`MUL Rd, Rs`
	Compare Low, and Low	`CMP Rd, Rs`
	Compare Low, and High	`CMP Rd, Hs`
	Compare High, and Low	`CMP Hd, Rs`
	Compare High, and High	`CMP Hd, Hs`
	Compare Negative	`CMN Rd, Rs`
	Compare Immediate	`CMP Rd, #8bit_Imm`
Logical	AND	`AND Rd, Rs`
	EOR	`EOR Rd, Rs`
	OR	`ORR Rd, Rs`
	Bit clear	`BIC Rd, Rs`
	Move NOT	`MVN Rd, Rs`
	Test bits	`TST Rd, Rs`

Table 1-12 Thumb instruction summary (continued)

Operation		Assembler
Shift/Rotate	Logical shift left	`LSL Rd, Rs, #5bit_shift_imm` `LSL Rd, Rs`
	Logical shift right	`LSR Rd, Rs, #5bit_shift_imm` `LSR Rd, Rs`
	Arithmetic shift right	`ASR Rd, Rs, #5bit_shift_imm` `ASR Rd, Rs`
	Rotate right	`ROR Rd, Rs`
Branch	Conditional	
	if Z set	`BEQ label`
	if Z clear	`BNE label`
	if C set	`BCS label`
	if C clear	`BCC label`
	if N set	`BMI label`
	if N clear	`BPL label`
	if V set	`BVS label`
	if V clear	`BVC label`
	if C set, and Z clear	`BHI label`
	if C clear, and Z set	`BLS label`
	if N set, and V set, or if N clear, and V clear	`BGE label`
	if N set, and V clear, or if N clear, and V set	`BLT label`
	if Z clear, and N, or V set, or if Z clear, and N, or V clear	`BGT label`
	if Z set, or N set, and V clear, or N clear, and V set	`BLE label`
	Unconditional	`B label`
	Long branch with link	`BL label`
	Optional state change	
	to address held in Lo reg	`BX Rs`
	to address held in Hi reg	`BX Hs`

Table 1-12 Thumb instruction summary (continued)

Operation		Assembler
Load	With immediate offset	
	word	`LDR Rd, [Rb, #7bit_offset]`
	halfword	`LDRH Rd, [Rb, #6bit_offset]`
	byte	`LDRB Rd, [Rb, #5bit_offset]`
	With register offset	
	word	`LDR Rd, [Rb, Ro]`
	halfword	`LDRH Rd, [Rb, Ro]`
	signed halfword	`LDRSH Rd, [Rb, Ro]`
	byte	`LDRB Rd, [Rb, Ro]`
	signed byte	`LDRSB Rd, [Rb, Ro]`
	PC-relative	`LDR Rd, [PC, #10bit_Offset]`
	SP-relative	`LDR Rd, [SP, #10bit_Offset]`
	Address	
	using PC	`ADD Rd, PC, #10bit_Offset`
	using SP	`ADD Rd, SP, #10bit_Offset`
	Multiple	`LDMIA Rb!, <reglist>`
Store	With immediate offset	
	word	`STR Rd, [Rb, #7bit_offset]`
	halfword	`STRH Rd, [Rb, #6bit_offset]`
	byte	`STRB Rd, [Rb, #5bit_offset]`
	With register offset	
	word	`STR Rd, [Rb, Ro]`
	halfword	`STRH Rd, [Rb, Ro]`
	byte	`STRB Rd, [Rb, Ro]`
	SP-relative	`STR Rd, [SP, #10bit_offset]`
	Multiple	`STMIA Rb!, <reglist>`
Push/Pop	Push registers onto stack	`PUSH <reglist>`
	Push LR, and registers onto stack	`PUSH <reglist, LR>`

Table 1-12 Thumb instruction summary (continued)

Operation		Assembler
	Pop registers from stack	`POP <reglist>`
	Pop registers, and PC from stack	`POP <reglist, PC>`
Software Interrupt		`SWI 8bit_Imm`

Chapter 2
Programmer's Model

This chapter describes the ARM720T programmer's model. It contains the following sections:

- *Processor operating states* on page 2-2
- *Memory formats* on page 2-3
- *Instruction length* on page 2-5
- *Data types* on page 2-6
- *Operating modes* on page 2-7
- *Registers* on page 2-8
- *The Thumb state register set is a subset of the ARM state set. You have direct access to:* on page 2-10
- *The program status registers* on page 2-13
- *Exceptions* on page 2-16
- *Reset* on page 2-23
- *Relocation of low virtual addresses by the FCSE PID* on page 2-22
- *Implementation-defined behavior of instructions* on page 2-24.

2.1 Processor operating states

From the programmer point of view, the ARM720T can be in one of two states:

ARM state This executes 32-bit, word-aligned ARM instructions.

Thumb state This operates with 16-bit, halfword-aligned Thumb instructions. In this state, the PC uses bit 1 to select between alternate halfwords.

Note

Transition between these two states does not affect the processor mode or the contents of the registers.

2.1.1 Switching state

Entering Thumb state

Entry into Thumb state can be achieved by executing a BX instruction with the state bit (bit 0) set in the operand register.

Transition to Thumb state also occurs automatically on return from an exception, for example, *Interrupt ReQuest* (IRQ), *Fast Interrupt reQuest* (FIQ), UNDEF, ABORT, and *SoftWare Interrupt* (SWI) if the exception was entered with the processor in Thumb state.

Entering ARM state

Entry into ARM state happens:

- On execution of the BX instruction with the state bit clear in the operand register.
- On the processor taking an exception, for example, IRQ, FIQ, RESET, UNDEF, ABORT, and SWI. In this case, the PC is placed in the link register of the exception mode, and execution starts at the vector address of the exception.

2.2 Memory formats

The bigend bit in the Control Register selects whether the ARM720T treats words in memory as being stored in big-endian or little-endian format. See Chapter 3 *Configuration* for more information on the Control Register.

ARM720T views memory as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second. and bytes 8 to 11 the third. ARM720T can treat words in memory as being stored as follows:

- *Big-endian format*
- *Little-endian format* on page 2-4.

2.2.1 Big-endian format

In big-endian format, the most significant byte of a word is stored at the lowest numbered byte and the least significant byte at the highest numbered byte. Byte 0 of the memory system is therefore connected to data lines 31 to 24. This is shown in Figure 2-1.

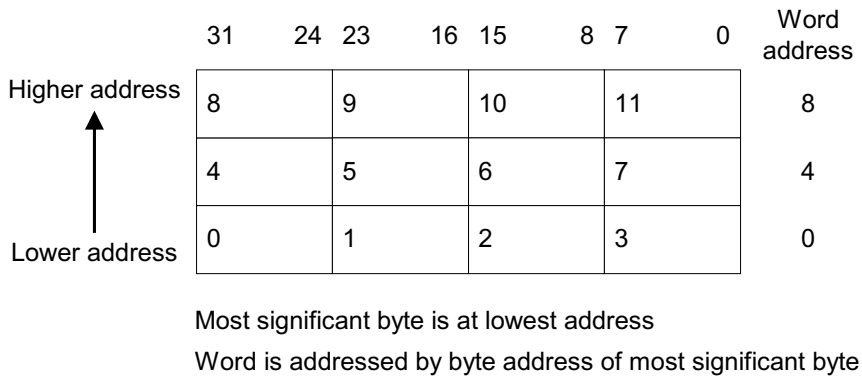


Figure 2-1 Big-endian addresses of bytes with words

2.2.2 Little-endian format

In little-endian format, the lowest numbered byte in a word is considered the least significant byte of the word, and the highest numbered byte the most significant. Byte 0 of the memory system is therefore connected to data lines 7 to 0. This is shown in Figure 2-2.

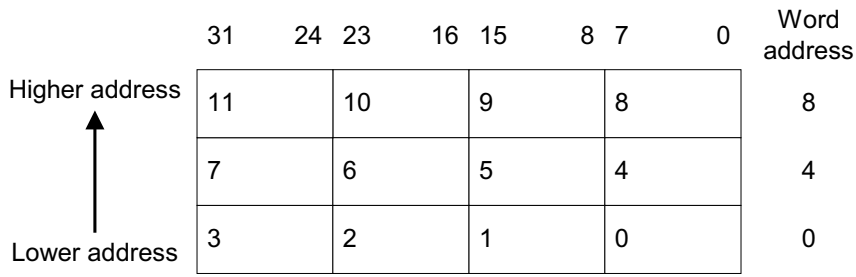


Figure 2-2 Little-endian addresses of bytes with words

2.3 Instruction length

Instructions are:

- 32 bits long in ARM state
- 16 bits long in Thumb state.

2.4 Data types

The ARM720T supports the following data types:

- byte (8-bit)
- halfword (16-bit)
- word (32-bit).

You must align these as follows:

- word quantities to 4-byte boundaries
- halfwords quantities to 2-byte boundaries
- byte quantities can be placed on any byte boundary.

2.5 Operating modes

The ARM720T supports seven modes of operation as listed in Table 2-1.

Table 2-1 ARM720T modes of operation

Mode	Type	Description
User	usr	The normal ARM program execution state
FIQ	fiq	Designed to support a data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	Protected mode for the operating system
Abort mode	abt	Entered after a Data Abort or instruction Prefetch Abort
System	sys	A privileged User mode for the operating system
Undefined	und	Entered when an Undefined Instruction is executed

Changing modes

Mode changes can be made under software control, by external interrupts or during exception processing. Most application programs execute in User mode. The non-User modes, known as privileged modes, are entered in order to service interrupts or exceptions, or to access protected resources.

2.6 Registers

ARM720T has a total of 37 registers:

- 31 general-purpose 32-bit registers
- six status registers.

These registers cannot all be seen at once. The processor state and operating mode dictate which registers are available to the programmer.

2.6.1 The ARM state register set

In ARM state, 16 general registers and one or two status registers are visible at any one time. In privileged (non-User) modes, mode-specific banked registers are switched in. Figure 2-3 on page 2-9 shows which registers are available in each mode. The banked registers are marked with a shaded triangle.

The ARM state register set contains 16 directly accessible registers, R0 to R15. All of these, except R15, are general-purpose, and can be used to hold either data or address values. In addition to these, R16 is used to store status information:

Register 14 R14 is used as the subroutine link register. This receives a copy of R15 when a *Branch and Link* (BL) code instruction is executed. At all other times it can be treated as a general-purpose register. The corresponding banked registers R14_svc, R14_irq, R14_fiq, R14_abt, and R14_und are similarly used to hold the return values of R15 when interrupts and exceptions arise, or when BL instructions are executed within interrupt or exception routines.

Register 15 R15 holds the *Program Counter* (PC). In ARM state, bits [1:0] of R15 are zero and bits [31:2] contain the PC. In Thumb state, bit [0] is zero and bits [31:1] contain the PC.

Register 16 R16 is the *Current Program Status Register* (CPSR). This contains condition code flags and the current mode bits.

Interrupt modes

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). In ARM state, many FIQ handlers do not have to save any registers. User, IRQ, Supervisor, Abort, and Undefined modes each have two banked registers mapped to R13 and R14, allowing each of these modes to have a private stack pointer and link registers.

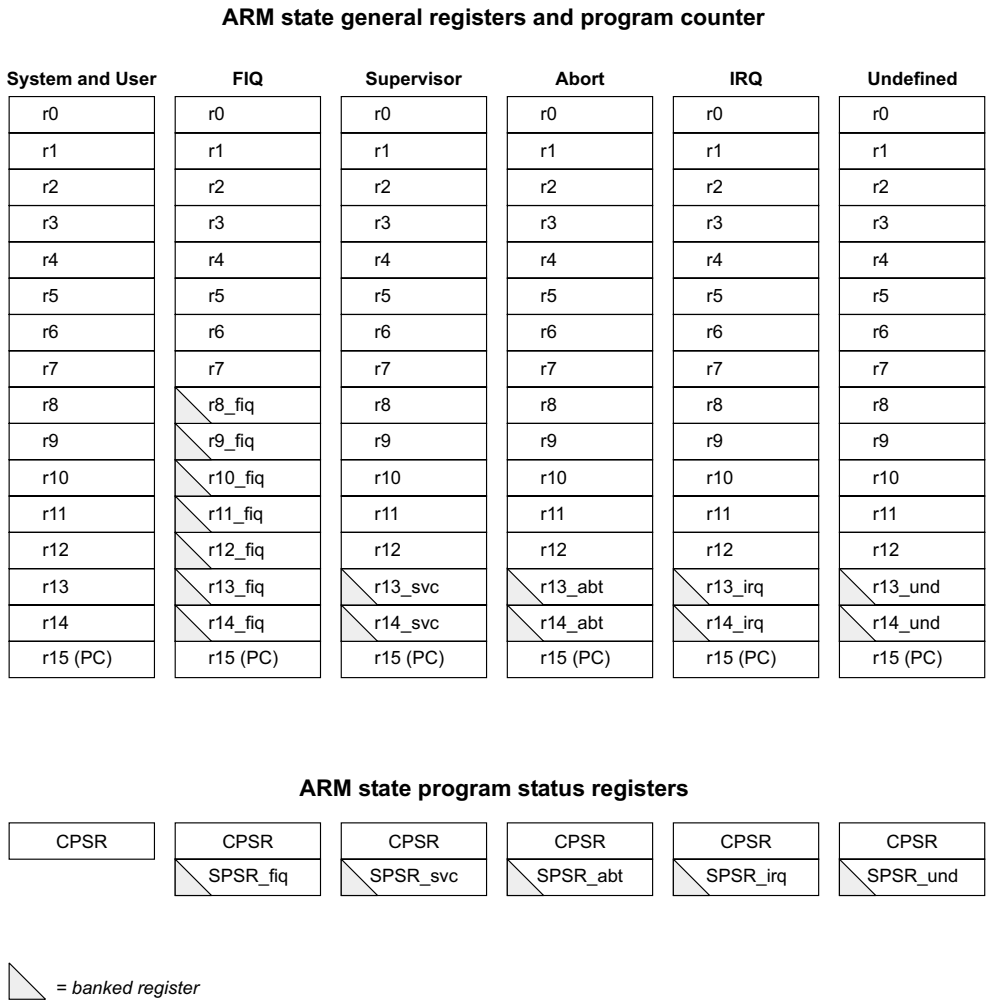


Figure 2-3 Register organization in ARM state

2.6.2 The Thumb state register set

The Thumb state register set is a subset of the ARM state set. You have direct access to:

- eight general registers, (R0–R7)
- the PC
- a *Stack Pointer* (SP) *register*
- a *Link Register* (LR)
- the CPSR.

There are banked SPs, LRs, and *Saved Process Status Registers* (SPSRs) for each privileged mode. This is shown in Figure 2-4.

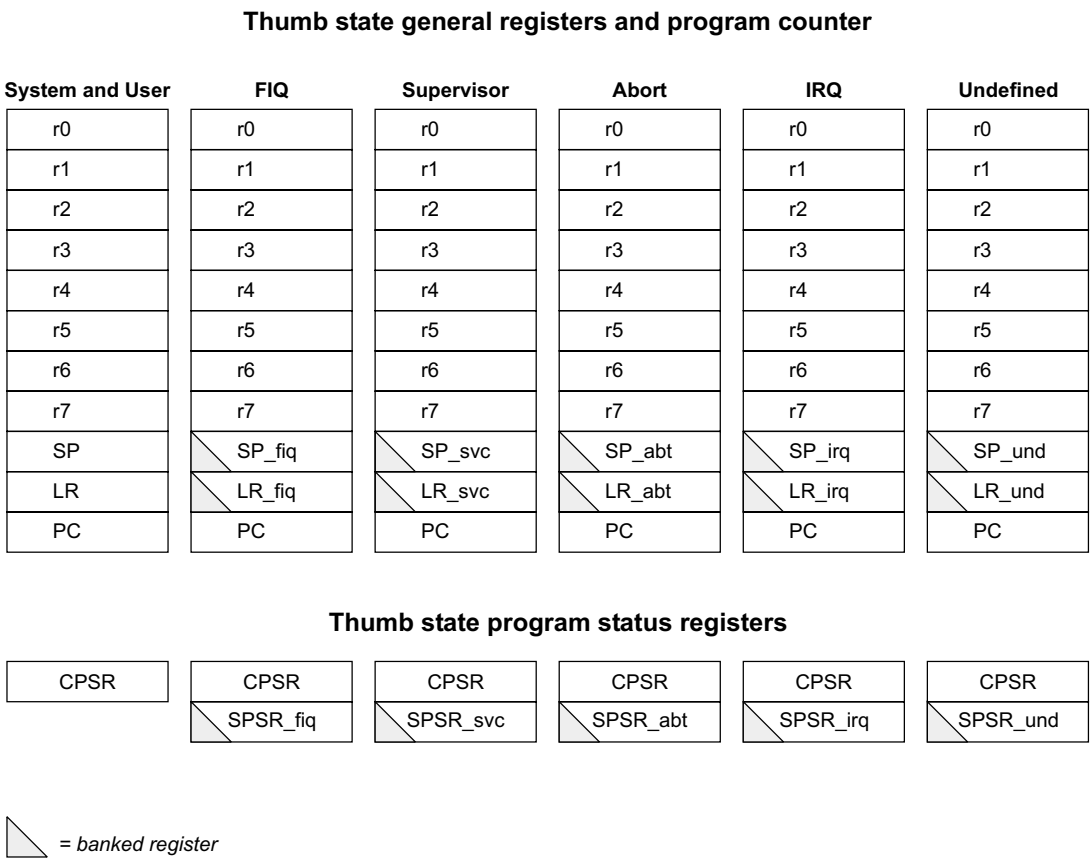


Figure 2-4 Register organization in Thumb state

2.6.3 The relationship between ARM and Thumb state registers

The Thumb state registers relate to the ARM state registers in the following ways:

- Thumb state R0–R7, and ARM state R0–R7 are identical
- Thumb state CPSR and SPSRs, and ARM state CPSR and SPSRs are identical
- Thumb state SP maps onto ARM state R13
- Thumb state LR maps onto ARM state R14
- Thumb state PC maps onto ARM state PC (R15).

This relationship is shown in Figure 2-5.

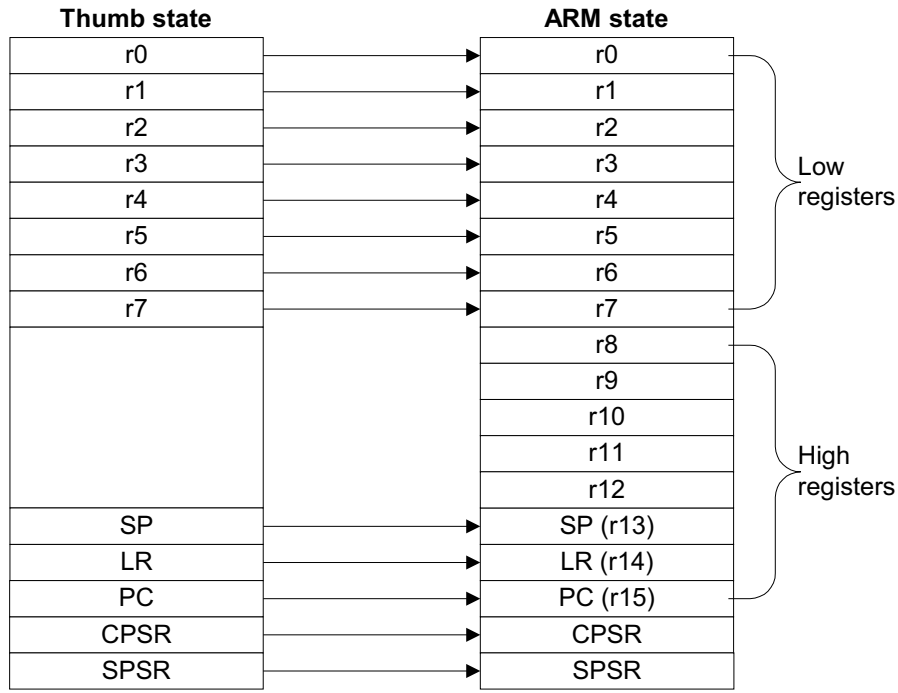


Figure 2-5 Mapping of Thumb state registers onto ARM state registers

2.6.4 Accessing high registers in Thumb state

In Thumb state, registers R8–R15 (the high registers) are not part of the standard register set. However, the assembly language programmer has limited access to them, and can use them for fast temporary storage.

A value can be transferred from a register in the range R0 – R7 (a low register) to a high register, and from a high register to a low register, using special variants of the `MOV` instruction. High register values can also be compared against or added to low register values with the `CMP` and `ADD` instructions. See the *ARM Architecture Reference Manual* for details on high register operations.

2.7 The program status registers

The ARM720T contains a CPSR, and five SPSRs for use by exception handlers. These registers:

- hold information about the most recently performed ALU operation
- control the enabling and disabling of interrupts
- set the processor operating mode.

The arrangement of bits is shown in Figure 2-6.

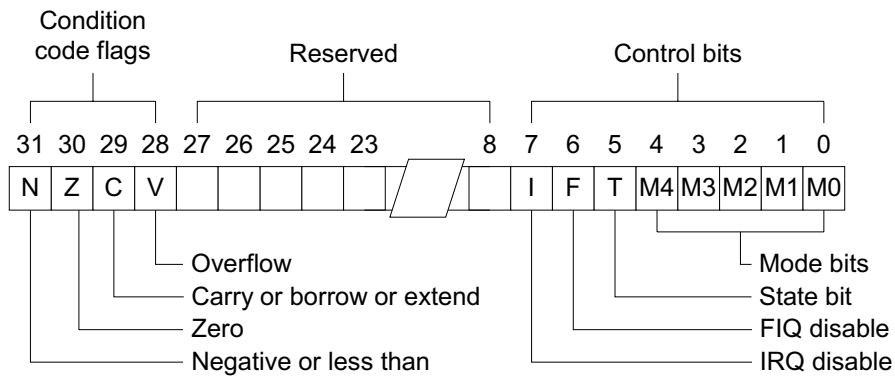


Figure 2-6 Program status register format

2.7.1 The condition code flags

The N, Z, C, and V bits are the condition code flags. These can be changed as a result of arithmetic and logical operations, and tested to determine whether an instruction executes.

In ARM state, all instructions can be executed conditionally. In Thumb state, only the Branch instruction is capable of conditional execution. See the *ARM Architecture Reference Manual* for details.

2.7.2 The control bits

The bottom eight bits of a PSR (incorporating I, F, T, and M[4:0]) are known collectively as the control bits. These change when an exception arises. If the processor is operating in a privileged mode, they can also be manipulated by software.

I and F bits These are the interrupt disable bits. When set, these disable the IRQ and FIQ interrupts respectively.

The T bit This reflects the operating state. When this bit is set, the processor is executing in Thumb state, otherwise it is executing in ARM state. This is reflected on the **TBIT** external signal. Software must never change the state of the **TBIT** in the CPSR. If this happens, the processor then enters an unpredictable state.

M[4:0] bits These are the mode bits. These determine the processor operating mode, as shown in Table 2-2. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described can be used.

Note

If you program any illegal value into the mode bits, M[4:0], then the processor enters an unrecoverable state. If this occurs, apply reset.

2.7.3 Reserved bits

The remaining bits in the PSRs are *reserved*. When changing flag or control bits of a PSR, you must ensure that these unused bits are not altered. Also, your program must not rely on them containing specific values, because in future processors they might read as one or zero.

Table 2-2 PSR mode bit values

M[4:0]	Mode	Visible Thumb state registers	Visible ARM state registers
10000	User	R7 to R0, LR, SP PC, CPSR	R14 to R0, PC, CPSR
10001	FIQ	R7 to R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7 to R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7 to R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12 to R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7 to R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12 to R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc

Table 2-2 PSR mode bit values (continued)

M[4:0]	Mode	Visible Thumb state registers	Visible ARM state registers
10111	Abort	R7 to R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12 to R0, R14_abt..R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7 to R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12 to R0, R14_und, R13_und, PC, CPSR, SPSR_und
11111	System	R7 to R0, LR, SP PC, CPSR	R14 to R0, PC, CPSR

2.8 Exceptions

Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example to service an interrupt from a peripheral. Before an exception can be handled, the current processor state must be preserved so that the original program can resume when the handler routine has finished.

Several exceptions can arise at the same time. If this happens, they are dealt with in a fixed order. See *Exception priorities* on page 2-21.

2.8.1 Action on entering an exception

When handling an exception, the ARM720T:

1. Preserves the address of the next instruction in the appropriate LR.
   a. If the exception has been entered from ARM state, the address of the next instruction is copied into the LR (that is, current PC+4 or PC+8 depending on the exception, See Table 2-3 on page 2-17 for details).
   b. If the exception has been entered from Thumb state, the value written into the LR is the current PC, offset by a value so that the program resumes from the correct place on return from the exception. This means that the exception handler does not have to determine which state the exception was entered from.

   For example, in the case of SWI:

   ```
   MOVS PC, R14_svc
   ```

   always returns to the next instruction regardless of whether the SWI was executed in ARM or Thumb state.
2. Copies the CPSR into the appropriate SPSR.
3. Forces the CPSR mode bits to a value which depends on the exception.
4. Forces the PC to fetch the next instruction from the relevant exception vector.

It can also set the interrupt disable flags to prevent otherwise unmanageable nestings of exceptions.

If the processor is in Thumb state when an exception occurs, it automatically switches into ARM state when the PC is loaded with the exception vector address.

2.8.2 Action on leaving an exception

On completion, the exception handler:

1. Moves the LR, minus an offset where appropriate, to the PC. The offset varies depending on the type of exception.
2. Copies the SPSR back to the CPSR.
3. Clears the interrupt disable flags, if they were set on entry.

Note

An explicit switch back to Thumb state is never necessary, because restoring the CPSR from the SPSR automatically sets the T bit to the value it held immediately prior to the exception.

2.8.3 Exception entry and exit summary

Table 2-3 summarizes the PC value preserved in the relevant R14 on exception entry, and the recommended instruction for exiting the exception handler.

Table 2-3 Exception entry and exit

Exception	Return Instruction	Previous State	
		ARM R14_x	Thumb R14_x
BL[a]	`MOV PC, R14`	PC + 4	PC + 2
SWI[a]	`MOVS PC, R14_svc`	PC + 4	PC + 2
UDEF[a]	`MOVS PC, R14_und`	PC + 4	PC + 2
FIQ[b]	`SUBS PC, R14_fiq, #4`	PC + 4	PC + 4
IRQ[b]	`SUBS PC, R14_irq, #4`	PC + 4	PC + 4
PABT[a]	`SUBS PC, R14_abt, #4`	PC + 4	PC + 4
DABT[c]	`SUBS PC, R14_abt, #8`	PC + 8	PC + 8
RESET[d]	NA	-	-

a.Where PC is the address of the BL/SWI/Undefined Instruction fetch that had the Prefetch Abort.
b.Where PC is the address of the instruction that was not executed because the FIQ or IRQ took priority.

c.Where PC is the address of the Load or Store instruction that generated the Data Abort.
d.The value saved in R14_svc upon reset is unpredictable.

2.8.4 Fast interrupt request

The FIQ exception is designed to support a data transfer or channel process. In ARM state it has sufficient private registers to remove the necessity for register saving, minimizing the overhead of context switching.

FIQ is externally generated by taking the **nFIQ** input LOW. **nFIQ** and **nIRQ** are considered asynchronous, and a cycle delay for synchronization is incurred before the interrupt can affect the processor flow.

Irrespective of whether the exception was entered from ARM or Thumb state, a FIQ handler must leave the interrupt by executing:

```
SUBS PC, R14_fiq, #4
```

FIQ can be disabled by setting the CPSR F flag.

Note

This is not possible from User mode. If the F flag is clear, ARM720T checks for a LOW level on the output of the FIQ synchronizer at the end of each instruction.

2.8.5 Interrupt request

The IRQ exception is a normal interrupt caused by a LOW level on the **nIRQ** input. IRQ has a lower priority than FIQ and is masked out when a FIQ sequence is entered. It can be disabled at any time by setting the I bit in the CPSR, though this can only be done from a privileged (non-User) mode.

Irrespective of whether the exception was entered from ARM or Thumb state, an IRQ handler must return from the interrupt by executing:

```
SUBS PC, R14_irq, #4
```

2.8.6 Abort

An abort indicates that the current memory access cannot be completed. It can be signaled either by the protection unit, or by the external **BERROR** input. The ARM720T checks for the abort exception during memory access cycles.

There are two types of abort:

Prefetch Abort This occurs during an instruction prefetch.

Data Abort This occurs during a data access.

If a Prefetch Abort occurs, the prefetched instruction is marked as invalid, but the exception is not taken until the instruction reaches the head of the pipeline. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, the abort does not take place.

If a Data Abort occurs, the action taken depends on the instruction type:

1. Single data transfer instructions (`LDR, STR`) write-back modified base registers, the Abort handler must be aware of this.
2. The swap instruction (`SWP`) is aborted as though it had not been executed.
3. Block data transfer instructions (`LDM, STM`) complete. If write-back is set, the base is updated. If the instruction attempts to overwrites the base with data (that is, it has the base in the transfer list), the overwriting is prevented. All register overwriting is prevented after an abort is indicated. This means, in particular, that R15 (always the last register to be transferred) is preserved in an aborted `LDM` instruction.

After fixing the reason for the abort, the handler must execute the following irrespective of the state (ARM or Thumb):

```
SUBS PC, R14_abt, #4 for a Prefetch Abort, or
SUBS PC, R14_abt, #8 for a Data Abort
```

This restores both the PC and the CPSR, and retries the aborted instruction.

Note

There are restrictions on the use of the external abort signal. See *External aborts on page 6-25*.

2.8.7 Software interrupt

The SWI instruction is used for entering Supervisor mode, usually to request a particular supervisor function. A SWI handler must return by executing the following irrespective of the state (ARM or Thumb):

```
MOV PC, R14_svc
```

This restores the PC and CPSR, and returns to the instruction following the SWI.

2.8.8 Undefined instruction

When ARM720T comes across an instruction that it cannot handle, it takes the undefined instruction trap. This mechanism can be used to extend either the Thumb or ARM instruction set by software emulation.

After emulating the failed instruction, the trap handler must execute the following irrespective of the state (ARM or Thumb):

```
MOVS PC, R14_und
```

This restores the CPSR and returns to the instruction following the Undefined Instruction.

2.8.9 Exception vectors

The ARM720T can have exception vectors mapped to either low or high addresses, controlled by the V bit in the control register (See *Register 1, control register* on page 3-5). Table 2-4 lists the exception vector addresses.

Table 2-4 Exception vector addresses

High address	Low address	Exception	Mode on entry
0xFFFF0000	0x00000000	Reset	Supervisor
0xFFFF0004	0x00000004	Undefined instruction	Undefined
0xFFFF0008	0x00000008	Software interrupt	Supervisor
0xFFFF000C	0x0000000C	Abort (prefetch)	Abort
0xFFFF0010	0x00000010	Abort (data)	Abort
0xFFFF0014	0x00000014	Reserved	Reserved
0xFFFF0018	0x00000018	IRQ	IRQ
0xFFFF001C	0x0000001C	FIQ	FIQ

Note

The low addresses are those generated by the processor core before relocation.

2.8.10 Exception priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they are handled:

1. Reset (highest priority).
2. Data Abort.
3. FIQ.
4. IRQ.
5. Prefetch Abort.
6. Undefined Instruction, SWI (lowest priority).

2.8.11 Exception restrictions

Undefined Instruction and SWI are mutually exclusive, because they each correspond to particular (non-overlapping) decodings of the current instruction.

If a Data Abort occurs at the same time as a FIQ, and FIQs are enabled, the CPSR F flag is clear, ARM720T enters the Data Abort handler and then immediately proceeds to the FIQ vector. A normal return from FIQ causes the Data Abort handler to resume execution. Placing Data Abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection. The time for this exception entry must be added to worst-case FIQ latency calculations.

2.9 Relocation of low virtual addresses by the FCSE PID

The ARM720T provides a mechanism, *Fast Context Switch Extension* (FCSE), to translate virtual addresses to physical addresses based on the current value of the FCSE *Process IDentifier* (PID).

The virtual address produced by the processor core going to the IDC and MMU can be relocated if it lies in the bottom 32MB of the virtual address. That is, virtual address bits [31:25] = b0000000 by the substitution of the seven bits [31:25] of the FCSE PID register in the CP15 coprocessor.

A change to the FCSE PID exhibits similar behavior to a delayed branch if:

- the two instructions fetched immediately following an instruction to change the FCSE PID are fetched with a relocation to the previous FCSE PID
- the addresses of the instructions being fetched lie within the range of addresses to be relocated.

On reset, the FCSE PID register bits [31:25] are set to b`0000000`, disabling all relocation. For this reason, the low address reset exception vector is effectively never relocated by this mechanism.

Note

All addresses produced by the processor core undergo this translation if they lie in the appropriate address range. This includes the exception vectors if they are configured to lie in the bottom of the virtual memory map. This configuration is determined by the V bit in the CP15 control register.

2.10 Reset

When the **BnRES** signal goes LOW, ARM720T:

1. Abandons the executing instruction.
2. Flushes the cache and *Translation Lookaside Buffer* (TLB).
3. Disables the *Write Buffer* (WB), cache, and MMU.
4. Resets the FCSE PID.
5. Continues to fetch instructions from incrementing word addresses.

When **BnRES** goes HIGH again, the ARM720T:

1. Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and SPSR is not defined.
2. Forces M[4:0] to 10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR T bit.
3. Forces the PC to fetch the next instruction from the low reset exception vector.
4. Resumes execution in ARM state.

2.11 Implementation-defined behavior of instructions

The *ARM Architectural Reference Manual* defines the instruction set of the ARM720T:

- See *Indexed Addressing on a Data Abort* for the behavior of the ARM720T instructions for those features which are denoted as being implementation-defined in that manual.
- See *Early termination* for those features that define signed and unsigned early termination on the ARM720T.

2.11.1 Indexed Addressing on a Data Abort

In the event of a Data Abort with pre-indexed or post-indexed addressing, the value left in Rn is defined to be the updated base register value for the following instructions:

- LDC
- LDM
- LDR
- LDRB
- LDRBT
- LDRH
- LDRSB
- LDRSH
- LDRT
- STC
- STM
- STR
- STRB
- STRBT
- STRH
- STRT.

2.11.2 Early termination

On the ARM720T, early termination is defined as:

MLA, MUL Signed early termination.
SMULL, SMLAL Signed early termination.
UMULL, UMLAL Unsigned early termination.

Chapter 3
Configuration

This chapter describes the configuration of the ARM720T. It contains the following sections.

- *About configuration* on page 3-2
- *Internal coprocessor instructions* on page 3-3
- *Registers* on page 3-4.

3.1 About configuration

The operation and configuration of ARM720T is controlled:

- directly using coprocessor instructions
- indirectly using the MMU page tables.

The coprocessor instructions manipulate a number of on-chip registers which control the configuration of the following:

- cache
- write buffer
- MMU
- other configuration options.

3.1.1 Compatibility

To ensure backwards compatibility of future CPUs:

- all reserved or unused bits in registers and coprocessor instructions must be programmed to 0
- invalid registers must not be read or written
- the following bits must be programmed to 0:
  - Register 1, bits[31:14] and bits [12:10]
  - Register 2, bits[13:0]
  - Register 5, bits[31:9]
  - Register 7, bits[31:0]
  - Register 13 FCSE PID, bits[24:0].

3.1.2 Notation

Throughout this section, the following terms and abbreviations are used:

Unpredictable (UNP)
: If specified for reads, the data returned when reading from this location is unpredictable. It can have any value.
If specified for writes, writing to this location causes unpredictable behavior or change in device configuration.

Should Be Zero (SBZ)
: When writing to this location, all bits of this field should be zero.

3.2 Internal coprocessor instructions

The ARM720T instruction set allows specialized additional instructions to be implemented using coprocessors. These are separate processing units that are coupled to the ARM720T processor.

Note

The CP15 register map might change in future ARM processors. You are strongly recommended to structure software so that any code accessing CP15 is contained in a single module. It can then be updated easily.

CP15 registers can only be accessed with MRC and MCR instructions in a privileged mode. The instruction bit pattern of the MCR and MRC instructions is shown in Figure 3-1.

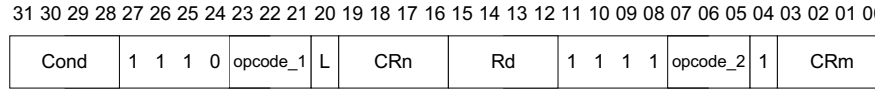


Figure 3-1 MRC and MCR bit pattern

CDP, LDC, and STC instructions, as well as unprivileged MRC and MCR instructions to CP15 cause the Undefined Instruction trap to be taken.

The CRn field of MRC and MCR instructions specifies the coprocessor register to access. The CRm field and opcode_2 fields specify a particular action when addressing some registers.

In all instructions accessing CP15:

- the opcode_1 field should be zero (SBZ).
- the opcode_2 and CRm fields should be zero except when accessing registers 7, 8, and 13 when the specified values must be used to select the desired cache, TLB, or process identifier operations.

3.3 Registers

ARM720T contains registers that control the cache and MMU operation. These registers are accessed using CPRT instructions to CP15 with the processor in a privileged mode.

Only some of registers R0 to R15 are valid. An access to an invalid register causes neither the access nor an undefined instruction trap, and therefore must never be carried out.

Table 3-1 Cache and MMU control register

Register	Register reads	Register writes
0	ID register	Reserved
1	Control	Control
2	Translation table base	Translation table base
3	Domain access control	Domain access control
4	Reserved	Reserved
5	Fault status	Fault status
6	Fault address	Fault address
7	Reserved	Cache operations
8	Reserved	TLB operations
9 – 12	Reserved	Reserved
13	Process identifier	Process identifier
14 – 15	Reserved	Reserved

3.3.1 Register 0, ID register

Reading from CP15 register 0 returns the value:

`0x41807203`

Note

The final nibble represents the core revision.

The CRm and opcode_2 fields should be zero when reading CP15 register 0. This is shown in Figure 3-2.

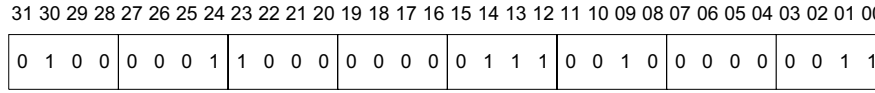


Figure 3-2 ID register read

Writing to CP15 register 0 is unpredictable. ID register write is shown in Figure 3-3.

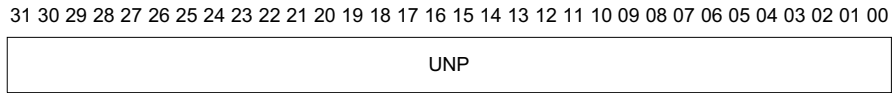


Figure 3-3 ID register write

3.3.2 Register 1, control register

Reading from CP15 register 1 reads the control bits. The CRm and opcode_2 fields should be zero when reading CP15 register 1. Register 1 read is shown in Figure 3-4.

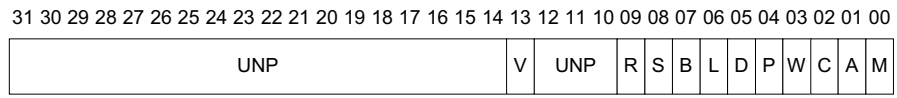


Figure 3-4 Register 1 read

Writing to CP15 register 1 sets the control bits. The CRm and opcode_2 fields must be zero when writing CP15 register 1. Register 1 write is shown in Figure 3-5.

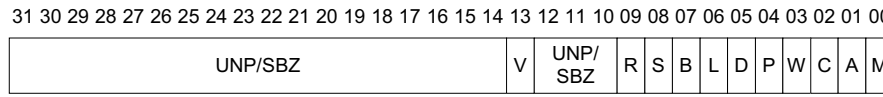


Figure 3-5 Register 1 write

All defined control bits are set to zero on reset. The control bits have the following functions:

M Bit 0 MMU enable/disable:
0 = MMU disabled
1 = MMU enabled.

A Bit 1 Alignment fault enable/disable:
0 = Address Alignment Fault Checking disabled
1 = Address Alignment Fault Checking enabled.

C Bit 2 Cache enable/disable:
0 = Instruction and/or Data Cache (IDC) disabled
1 = Instruction and/or Data Cache (IDC) enabled.

W Bit 3 Write buffer enable/disable:
0 = Write Buffer disabled
1 = Write Buffer enabled.

P Bit 4 When read, returns 1. When written, is ignored.

D Bit 5 When read, returns 1. When written, is ignored.

L Bit 6 When read, returns 1. When written, is ignored.

B Bit 7 Big-endian/little-endian:
0 = Little-endian operation
1 = Big-endian operation.

S Bit 8 System protection:
Modifies the MMU protection system.

R Bit 9 ROM protection:
Modifies the MMU protection system.

Bits 12:10 When read, this returns an unpredictable value. When written, it should be zero, or a value read from these bits on the same processor.

Note

Using a read-write-modify sequence when modifying this register provides the greatest future compatibility.

V Bit 13 Location of exception vectors:
0 = low addresses
1 = high addresses.

Bits 31:14 When read, this returns an unpredictable value. When written, it should be zero, or a value read from these bits on the same processor.

Enabling the MMU

You must take care if the translated address differs from the untranslated address, because the instructions following the enabling of the MMU are fetched using no address translation. Enabling the MMU can be considered as a branch with delayed execution.

A similar situation occurs when the MMU is disabled. The correct code sequence for enabling and disabling the MMU is given in *Interaction of the MMU, IDC, and write buffer* on page 6-26.

If the cache and write buffer are enabled when the MMU is not enabled, the results are unpredictable.

3.3.3 Register 2, translation table base register

Reading from CP15 register 2 returns the pointer to the currently active first-level translation table in bits [31:14] and an unpredictable value in bits [13:0]. The CRm and opcode_2 fields should be zero when reading CP15 register 2.

Writing to CP15 register 2 updates the pointer to the currently active first-level translation table from the value in bits [31:14] of the written value. Bits [13:0] should be zero. The CRm and opcode_2 fields should be zero when writing CP15 register 2. Register 2 is shown in Figure 3-6.

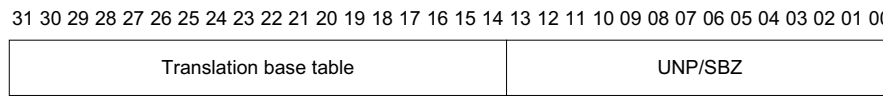


Figure 3-6 Register 2

3.3.4 Register 3, domain access control register

Reading from CP15 register 3 returns the value of the domain access control register.

Writing to CP15 register 3 writes the value of the domain access control register.

The domain access control register consists of 16 2-bit fields, each of which defines the access permissions for one of the 16 domains (D15-D0).

The CRm and opcode_2 fields should be zero when reading or writing CP15 register 3. This is shown in Figure 3-7.

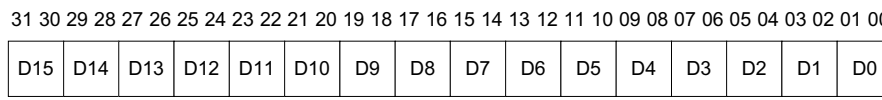


Figure 3-7 Register 3

3.3.5 Register 4, reserved

Register 4 is reserved. Reading CP15 register 4 is unpredictable. Writing CP15 register 4 is unpredictable. This is shown in Figure 3-8.

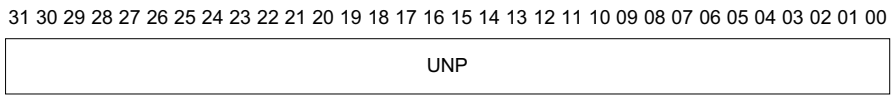


Figure 3-8 Register 4

3.3.6 Register 5, fault status register

Reading CP15 register 5 returns the value of the *Fault Status Register* (FSR). The FSR contains the source of the last data fault.

Note

Only the bottom 9 bits are returned. The upper 23 bits are unpredictable.

The FSR indicates the domain and type of access being attempted when an abort occurred:

Bit 8 This is always read as zero. Bit 8 is ignored on writes.

Bits [7:4] These specify which of the 16 domains (D15-D0) was being accessed when a fault occurred.

Bits [3:1] Theses indicate the type of access being attempted.

The encoding of these bits is shown in *Fault address and fault status registers* on page 6-19. The FSR is only updated for data faults, not for prefetch faults.

Writing CP15 register 5 sets the FSR to the value of the data written. This is useful when a debugger has to restore the value of the FSR. The upper 24 bits written should be zero.

The CRm and opcode_2 fields should be zero when reading or writing CP15 register 5. Register 5 is shown in Figure 3-9.

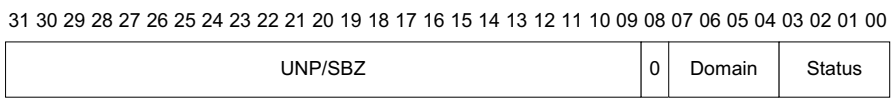


Figure 3-9 Register 5

3.3.7 Register 6, Fault Address Register

Reading CP15 register 6 returns the value of the *Fault Address Register* (FAR). The FAR holds the virtual address of the access that was attempted when a fault occurred. The FAR is only updated for data faults, not for prefetch faults.

Writing CP15 register 6 sets the FAR to the value of the data written. This is useful when a debugger has to restore the value of the FAR.

The CRm and opcode_2 fields should be zero when reading or writing CP15 register 6. Register 6 is shown in Figure 3-10.

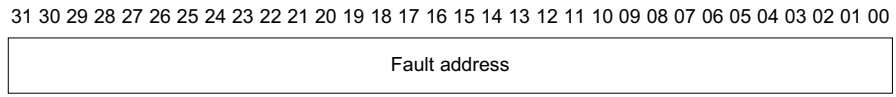


Figure 3-10 Register 6

Note

Register 6 contains a modified virtual address if the FCSE PID register is nonzero.

3.3.8 Register 7, cache operations

Writing to CP15 register 7 manages the unified instruction and data cache of the ARM720T. Only one cache operation is defined using the following opcode_2 and CRm fields in the MCR instruction that writes the CP15 register 7.

Caution

The Invalidate ID cache function invalidates all cache data. Use this with caution.

Register 7 is shown in Table 3-2.

Table 3-2 Cache operation

Function	opcode_2 value	CRm value	Data	Instruction
Invalidate ID cache	0b000	0b0111	SBZ	`MCR p15, 0, Rd, c7, c7, 0`

Reading from CP15 register 7 is undefined.

3.3.9 Register 8, TLB operations

Writing to CP15 register 8 controls the *Translation Lookaside Buffer* (TLB). The ARM720T implements a unified instruction and data TLB.

Two TLB operations are defined, and the function to be performed selected by the opcode_2 and CRm fields in the MCR instruction used to write CP15 register 8. This is listed in Table 3-3.

Table 3-3 TLB operations

Function	opcode_2 value	CRm value	Data	Instruction
Invalidate TLB	0b000	0b0111	SBZ	`MCR p15, 0, Rd, c8, c7, 0`
Invalidate TLB single entry	0b001	0b0111	Virtual Address	`MCR p15, 0, Rd, c8, c7, 1`

Reading from CP15 register 8 is undefined.

The Invalidate TLB function invalidates all of the unlocked entries in the TLB.

The Invalidate TLB single entry function invalidates any TLB entry corresponding to the Virtual Address given in `Rd`.

Note

Register 8 contains a modified virtual address if the FCSE PID register is nonzero.

3.3.10 Registers 9 to 12, reserved

Accessing any of these registers is undefined. Writing to any of these registers is undefined.

3.3.11 Register 13, process identifier

Two independent process identifier registers can be accessed using register 13:

- *Fast context switch extension process identifier* on page 3-11
- *Trace process identifier* on page 3-11.

Fast context switch extension process identifier

Reading from CP15 register 13 with opcode_2=0 returns the value of the FCSE PID. This is shown in Figure 3-11.

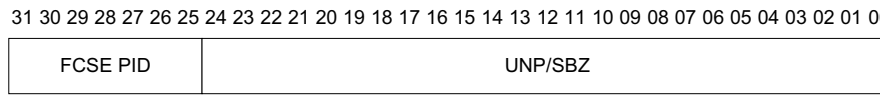


Figure 3-11 Register 13 with opcode_2=0

Note

Only bits [31:25] are returned. The remaining 25 bits are unpredictable.

Writing to CP15 register 13 with opcode_2=0 updates the FCSE PID from the value in bits [31:25]. Bits [24:0] should be zero. The FCSE PID is set to b`0000000` on Reset.

The CRm and opcode_2 should be zero when reading or writing the FCSE PID.

Changing FCSE PID

You must take care when changing the FCSE PID because the following instructions have been fetched with the previous FCSE PID. In this way, changing the FCSE PID has similarities with a branch with delayed execution. See *Relocation of low virtual addresses by the FCSE PID* on page 2-22.

Trace process identifier

A 32-bit read/write register is provided to hold a Trace *PROCess IDentifier* (PROCID) up to 32-bits in length visible to the ETM7. This is achieved by reading from or writing to the CP15 register 13 with opcode_2 = 1 as shown in Figure 3-12.

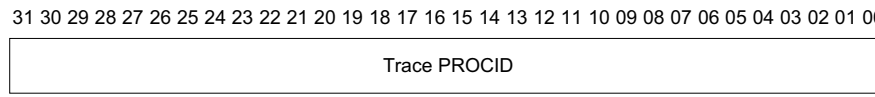


Figure 3-12 Register 13 with opcode_2=1

Signal **PROCIDWR** is exported to notify the ETM7 that the Trace PROCID has been written.

3.3.12 Registers 14-15, reserved

Accessing any of these registers is undefined. Writing to any of these registers is undefined.

Chapter 4
Instruction and Data Cache

This chapter describes the instruction and data cache. It contains the following sections:

- *About the instruction and data cache* on page 4-2
- *IDC validity* on page 4-4
- *IDC enable, disable, and reset* on page 4-5
- *IDC disable for secure applications* on page 4-6.

4.1 About the instruction and data cache

The cache only operates on a write-through basis with a read-miss allocation policy and a random replacement algorithm.

4.1.1 IDC operation

The ARM720T contains an 8KB mixed *Instruction and Data Cache* (IDC).

The C bit in the ARM720T control register and the cachable bit in the MMU page tables only affect loading data into the cache. The cache is always searched regardless of these two bits. If the data is found then it is used, so when the cache is disabled, it must also be flushed.

The IDC has 512 lines of 16 bytes (four words), arranged as a 4-way set-associative cache, and uses the virtual addresses generated by the processor core after relocation by the FCSE PID as appropriate. The IDC is always reloaded a line at a time (four words). It can be enabled or disabled using the ARM720T control register and is disabled on **BnRES**.

The operation of the cache is further controlled by the *Cachable bit* (C bit) stored in the MMU page table (see Chapter 6 *Memory Management Unit*). For this reason, the MMU must be enabled in order to use the IDC. However, the two functions can be enabled simultaneously, with a single write to the control register.

4.1.2 Cachable bit

The C bit determines whether data being read can be placed in the IDC and used for subsequent read operations. Typically, main memory is marked as cachable to improve system performance, and I/O space is marked as noncachable to stop the data being stored in the ARM720T cache.

For example, if the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of the initial data held in the cache. The cachable bit can be configured for both pages and sections.

Cachable reads (C=1)

A line fetch of four words is performed when a cache miss occurs in a cachable area of memory, and it is randomly placed in a cache bank.

Uncachable reads (C=0)

An external memory access is performed and the cache is not written.

4.1.3 Read-lock-write

The IDC treats the read-lock-write instruction as a special case:

Read phase Always forces a read of external memory, regardless of whether the data is contained in the cache.

Write phase Is treated as a normal write operation. If the data is already in the cache, the cache is updated.

Externally, the two phases are flagged as indivisible by asserting the **BLOK** signal.

4.2 IDC validity

The IDC operates with virtual addresses, so you must ensure that its contents remain consistent with the virtual to physical mappings performed by the MMU. If the memory mappings are changed, the IDC validity must be ensured.

4.2.1 Software IDC flush

The entire IDC can be marked as invalid by writing to the cache operations register R7. The cache is flushed immediately the register is written, but the following two instruction fetches can come from the cache before the register is written.

4.2.2 Doubly-mapped space

Because the cache works with virtual addresses, it is assumed that every virtual address maps to a different physical address. If the same physical location is accessed by more than one virtual address, the cache cannot maintain consistency. Each virtual address has a separate entry in the cache, and only one entry can be updated on a processor write operation.

To avoid any cache inconsistencies, both doubly-mapped virtual addresses must be marked as uncachable.

4.3 IDC enable, disable, and reset

The IDC is automatically disabled and flushed on **BnRES**. Once enabled, cachable read accesses cause lines to be placed in the cache.

To enable the IDC:

1. Make sure that the MMU is enabled first by setting bit 0 in the control register.
2. Enable the IDC by setting bit 2 in the control register. The MMU and IDC can be enabled simultaneously with a single write to the control register.

To disable the IDC:

1. Clear bit 2 in the control register.
2. Perform a flush by writing to the cache operations register.

4.4 IDC disable for secure applications

You can disable the IDC in certain secure applications. This is achieved by forcing the IDC to miss without triggering a line fill.

Caution

You are strongly advised not to use this feature in normal applications. When the **CACHEDIS** signal is not being used then it must be held LOW.

To disable the IDC:

1. Disable the MMU by writing to CP15 register 1 using an MCR and setting bit 0 LOW.
2. Input the special signal, **CACHEDIS**.

When **CACHEDIS** is asserted, held HIGH, it masks out some cache signals to disable the cache RAM banks and stop a cache hit being generated as a consequence.

Note

- You must disable the MMU before **CACHEDIS** is asserted.
- You must not enable the MMU until after **CACHEDIS** is deasserted.
- ARM does not support the use of this feature.

Chapter 5
Write Buffer

This chapter describes the write buffer. It contains the following sections:

- *About the write buffer* on page 5-2
- *Write buffer operation* on page 5-3.

5.1 About the write buffer

The ARM720T write buffer is provided to improve system performance. It can buffer up to eight words of data, and four independent addresses. It can be enabled or disabled using the W bit, bit 3, in the ARM720T control register. The buffer is disabled and flushed on reset.

The operation of the write buffer is further controlled by the *Bufferable* (B) bit, which is stored in the MMU page tables. For this reason, the MMU must be enabled before using the write buffer. The two functions can, however, be enabled simultaneously, with a single write to the control register.

For a write to use the write buffer, both the W bit in the control register and the B bit in the corresponding page table must be set.

Note

It is not possible to abort buffered writes externally. The **BERROR** pin is ignored. Areas of memory that can generate aborts must be marked as unbufferable in the MMU page tables.

5.1.1 Bufferable bit

This bit controls whether a write operation uses or does not use the write buffer. Typically, main memory is bufferable and I/O space unbufferable. The B bit can be configured for both pages and sections.

5.2 Write buffer operation

When the CPU performs a write operation, the translation entry for that address is inspected and the state of the B bit determines the subsequent action. If the write buffer is disabled using the ARM720T control register, buffered writes are treated in the same way as unbuffered writes.

To enable the write buffer:

1. Ensure the MMU is enabled by setting bit 0 in the control register.
2. Enable the write buffer by setting bit 3 in the control register. The MMU and write buffer can be enabled simultaneously with a single write to the control register.

To disable the write buffer, clear bit 3 in the control register.

Note

- Any writes already in the write buffer complete normally.
- The write buffer will attempt a write operation as long as there is data present.

5.2.1 Bufferable write

If the write buffer is enabled and the processor performs a write to a bufferable area, the data is placed in the write buffer at **FCLK** speeds, or **BCLK** speeds if running with fastbus extension, and the CPU continues execution. The write buffer then performs the external write in parallel.

If the write buffer is full (either because there are already eight words of data in the buffer, or because there is no slot for the new address), the processor is stalled until there is sufficient space in the buffer.

5.2.2 Unbufferable write

If the write buffer is disabled or the CPU performs a write to an unbufferable area, the processor is stalled until the write buffer empties and the write completes externally. This might require synchronization and several external clock cycles.

5.2.3 Read-lock-write

The write phase of a read-lock-write sequence is treated as an unbuffered write, even if it is marked as buffered.

Note

A single write requires one address slot and one data slot in the write buffer. A sequential write of n words requires one address slot and n data slots. The total of eight data slots in the buffer can be used as required. For example, there can be three nonsequential writes and one sequential write of five words in the buffer, and the processor could continue as normal, A fifth write or a sixth word in the fourth write stalls the processor until the first write has completed.

Chapter 6
Memory Management Unit

This chapter describes the *Memory Management Unit* (MMU). It contains the following sections:

- *About the MMU* on page 6-2
- *MMU program accessible registers* on page 6-4
- *Address translation process* on page 6-5
- *Level 1 descriptor* on page 6-7
- *Page table descriptor* on page 6-8
- *Section descriptor* on page 6-9
- *Translating section references* on page 6-11
- *Level 2 descriptor* on page 6-12
- *Translating small page references* on page 6-14
- *Translating large page references* on page 6-16
- *MMU faults and CPU aborts* on page 6-18
- *Fault address and fault status registers* on page 6-19
- *Domain access control* on page 6-21
- *Fault checking sequence* on page 6-22
- *External aborts on page 6-25*
- *Interaction of the MMU, IDC, and write buffer* on page 6-26.

6.1 About the MMU

The MMU performs two primary functions:

- translates virtual addresses into physical addresses
- controls memory access permissions.

The MMU hardware required to perform these functions consists of:

- a TLB
- access control logic
- translation table walking logic.

When the MMU is turned off, as happens on reset, the virtual address is output directly onto the physical address bus.

Note

The MMU works with virtual addresses after any relocation by the FCSE PID.

6.1.1 Memory accesses

The MMU supports memory accesses based on Sections or Pages:

Sections Are 1MB blocks of memory.

Pages Two different page sizes are supported:

- Small pages consist of 4KB blocks of memory. Additional access control mechanisms are extended to 1KB subpages.
- Large pages consist of 64KB blocks of memory. Large pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB. Additional access control mechanisms are extended to 16KB subpages.

6.1.2 Domains

The MMU also supports the concept of *domains*. These are areas of memory that can be defined to possess individual access rights. The domain access control register specifies access rights for up to 16 separate domains.

6.1.3 TLB

The TLB caches 64 translated entries. During most memory accesses, the TLB provides the translation information to the access control logic:

- If the TLB contains a translated entry for the virtual address, the access control logic determines if access is permitted.
- If access is permitted, the MMU outputs the appropriate physical address corresponding to the virtual address.
- If access is not permitted, the MMU signals the CPU to abort.

If the TLB misses (it does not contain a translated entry for the virtual address), the translation table walking hardware is invoked to retrieve the translation information from a translation table in physical memory. Once retrieved, the translation information is placed into the TLB, possibly overwriting an existing value. The entry to be overwritten is chosen by cycling sequentially through the TLB locations.

Note

The TLB must be flushed whenever the virtual to physical address mappings are changed.

6.1.4 Effect of reset

For information on the effect of reset, see *Reset* on page 2-23.

6.2 MMU program accessible registers

The ARM720T processor provides several 32-bit registers that determine the operation of the MMU.

Data is written to and read from the MMU registers using the ARM CPU `MRC` and `MCR` coprocessor instructions.

A brief description of the registers is given in Table 6-1. Each register is discussed in more detail in its relevant section.

Table 6-1 MMU program accessible registers

Register	Description
Translation table base	Holds the physical address of the base of the translation table maintained in main memory. This base must reside on a 16KB boundary.
Domain access control	Consists of 16 2-bit fields, each of which defines the access permissions for one of the 16 domains (D15–D0).
TLB operations	Allows individual or all TLB entries to be marked as invalid.
Fault status	Indicates the domain and type of access being attempted when an abort occurred. Bits [7:4] specify which of the 16 domains (D15-D0) was being accessed when a fault occurred. Bits [3:1] indicate the type of access being attempted. The encoding of these bits is different for internal and external faults (as indicated by bit 0 in the register) and is shown in Table 6-5 on page 6-19.
Fault address	Holds the virtual address of the access which was attempted when a fault occurred.

The FSR and FAR are only updated for data faults, not for prefetch faults.

6.3 Address translation process

The MMU translates virtual addresses generated by the CPU after relocation by the FCSE PID into physical addresses to access external memory. It also derives and checks the access permission. Translation information, that consists of both the address translation data and the access permission data, resides in a translation table located in physical memory.

The MMU provides the logic required to:

- traverse this translation table
- obtain the translated address
- check the access permission.

There are three routes by which the address translation, and therefore permission check, takes place. The route taken depends on whether the address has been marked as a section-mapped access or a page-mapped access. There are two sizes of page-mapped access, large pages and small pages. However, the translation process always starts out in the same way, as described in *Translation table base*, with a level one fetch. A section-mapped access only requires a level one fetch, but a page-mapped access also requires a level two fetch.

6.3.1 Translation table base

The translation process is initiated when the TLB does not contain an entry for the requested virtual address. The *Translation Table Base* (TTB) register points to the base of a table in physical memory that contains:

- section and page descriptors
- section or page descriptors.

The 14 low-order bits of the TTB Register should be zero as illustrated in Figure 6-1.

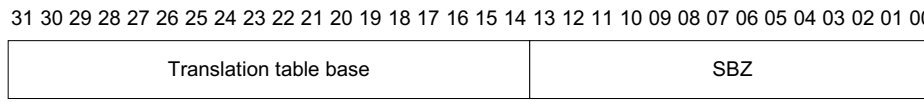


Figure 6-1 Translation table base register

Note

The table must reside on a 16KB boundary.

6.3.2 Level 1 fetch

Bits [31:14] of the TTB register are concatenated with bits [31:20] of the virtual address to produce a 30-bit address. This address selects a 4-byte translation table entry that is a first level descriptor for either a section or a page. Bit 1 of the returned descriptor specifies whether it is for a section or page. This is shown in Figure 6-2.

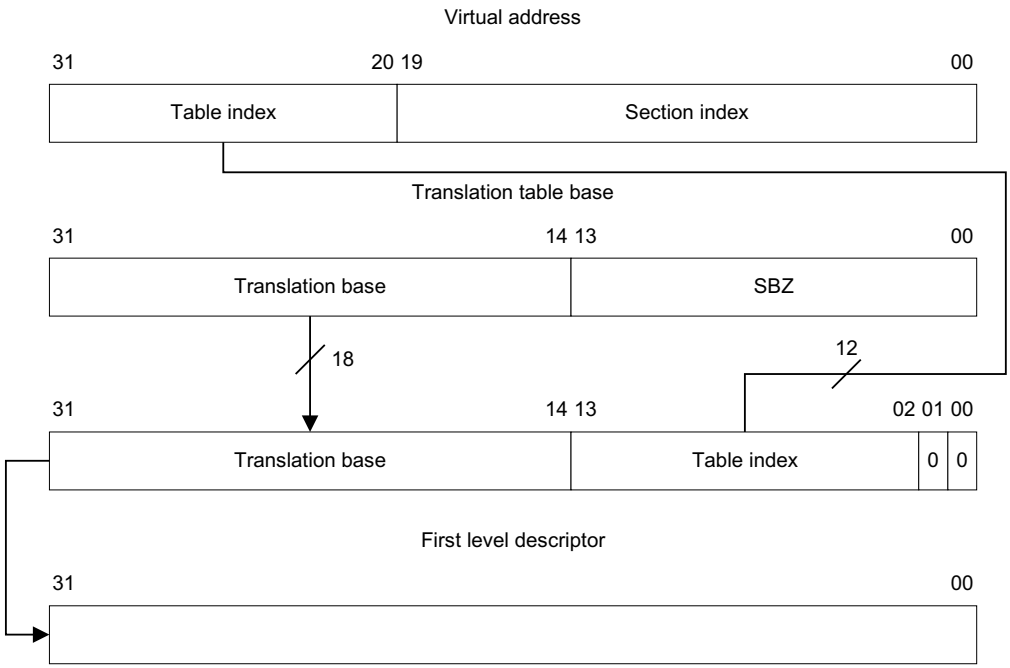


Figure 6-2 Accessing the translation table first level descriptors

6.4 Level 1 descriptor

The level 1 descriptor returned is either a page table descriptor or a section descriptor, and its format varies accordingly. Figure 6-3 illustrates the format of level 1 descriptors.

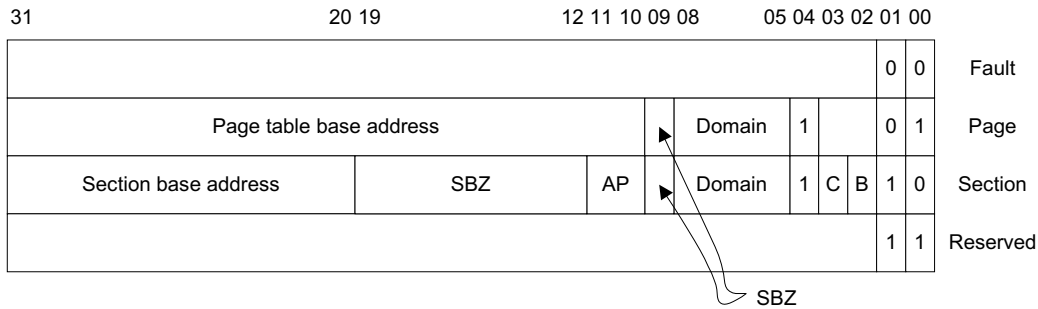


Figure 6-3 Level 1 descriptors

The two least significant bits indicate the descriptor type and validity, and are interpreted as listed in Table 6-2.

Table 6-2 Interpreting level 1 descriptor bits [1:0]

Value	Meaning	Notes
0 0	Invalid	Generates a section translation fault
0 1	Page	Indicates that this is a page descriptor
1 0	Section	Indicates that this is a section descriptor
1 1	Reserved	Reserved for future use

6.5 Page table descriptor

The bits used for the page table descriptor are as follows:

Bits [3:2] Are always written as 0.

Bit [4] Must be written to 1 for backward compatibility.

Bits [8:5] Specify one of the 16 possible domains, held in the domain access control register, that contain the primary access controls.

Bit [9] Is always written as 0.

Bits [31:10] Form the base for referencing the page table entry. The page table index for the entry is derived from the virtual address as illustrated in Figure 6-6 on page 6-15.

If a page table descriptor is returned from the level one fetch, a level two fetch is initiated as described in *Section descriptor* on page 6-9.

6.6 Section descriptor

Address data is described as:

C (Cachable) Indicates that data at this address is placed in the cache if the cache is enabled.

B (Bufferable) Indicates that data at this address is written through the write buffer if the write buffer is enabled.

Note

The meaning of the C and B bits might change in later ARM processors. You are strongly recommend to structure software so that code that manipulates the MMU page tables is contained in a single module. It can then be updated easily when you port it to a different ARM processor.

The bits used for the page table descriptor are as follows:

Bits [3:2] (C, B) Control the cache and write buffer related functions.

Bit [4] Must be written to 1 for backward compatibility.

Bits [8:5] Specify one of the 16 possible domains held in the domain access control register that contain the primary access controls.

Bit [9] Is always written as 0.

Bits [11:10] (AP) Specify the access permissions for this section and are interpreted as listed in Table 6-3 on page 6-10. Their interpretation depends on the setting of the S and R bits, control register bits 8 and 9. The domain access control specifies the primary access control. The AP bits only have an effect in client mode. Refer to *Domain access control* on page 6-21.

Bits [19:12] Are always written as 0.

Bits [31:20] Form the corresponding bits of the physical address for the 1MB section.

Table 6-3 Interpreting access permission (AP) bits

AP	S	R	Supervisor Permission	User Permission	Notes
00	0	0	No access	No access	Any access generates a permission fault
00	1	0	Read only	No access	Supervisor read only permitted
00	0	1	Read only	Read only	Any write generates a permission fault
00	1	1	Reserved	Reserved	Reserved
01	x	x	Read/write	No access	Access allowed only in Supervisor mode
10	x	x	Read/write	Read only	Writes in User mode cause permission fault
11	x	x	Read/write	Read/write	All access types permitted in both modes
xx	1	1	Reserved	Reserved	Reserved

6.7 Translating section references

Figure 6-4 shows the complete section translation sequence.

Note

The access permissions contained in the level 1 descriptor must be checked before the physical address is generated.

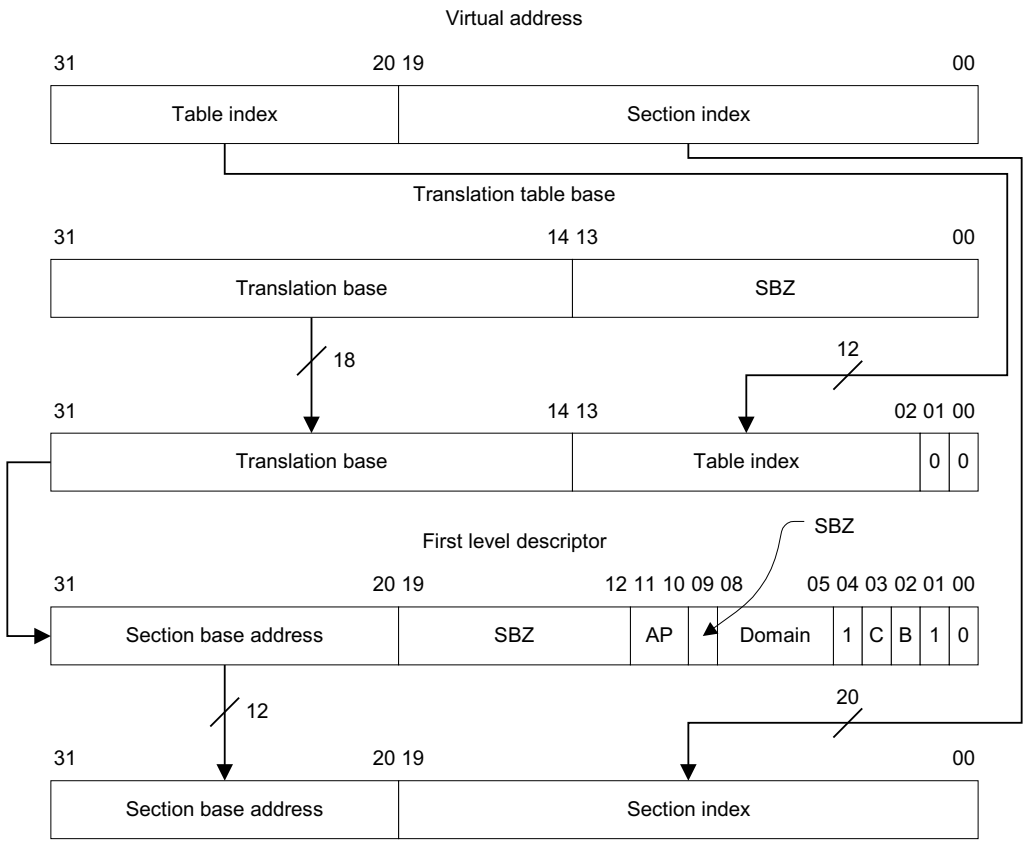


Figure 6-4 Section translation

6.8 Level 2 descriptor

If the level one fetch returns a page table descriptor, this provides the base address of the page table to be used. The page table is then accessed as described in Figure 6-6 on page 6-15, and a page table entry, or level 2 descriptor, is returned. This in turn can define either a small page or a large page access. Figure 6-5 shows the format of level 2 descriptors.

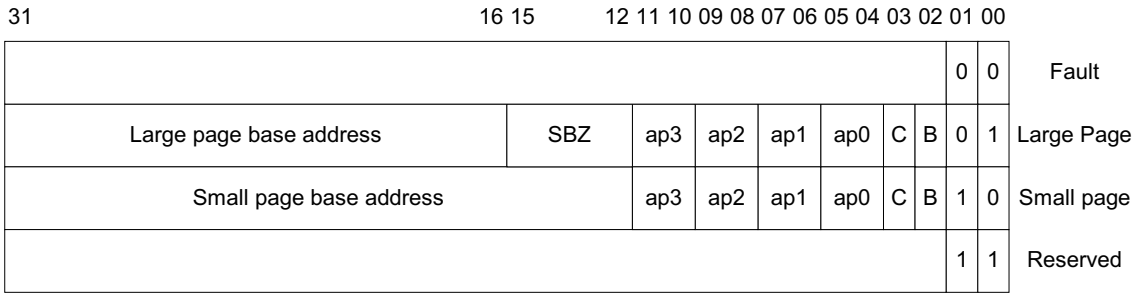


Figure 6-5 Page table entry, level 2 descriptor

The two least significant bits indicate the page size and validity, and are interpreted as listed in Table 6-4.

Table 6-4 Interpreting page table entry bits 1:0

Value	Meaning	Notes
0 0	Invalid	Generates a page translation fault
0 1	Large page	Indicates that this is a 64KB page
1 0	Small page	Indicates that this is a 4KB page
1 1	Reserved	Reserved for future use

The remaining bits are interpreted as follows:

Bit [2] B, bufferable, indicates that data at this address is written through the write buffer if the write buffer is enabled.

Bit [3] C, cacheable, indicates that data at this address is placed in the IDC if the cache is enabled.

Bits [11:4] Specify the access permissions (ap3–ap0) for the four subpages. Interpretation of these bits is listed in Table 6-2 on page 6-7.

Bits [15:12] Are programmed as 0 for large pages.

Bits [31:12] Small pages.

Bits [31:16] Large pages.

Note

Small and large pages form the corresponding bits of the physical address, that is the physical page number. The page index is derived from the virtual address as illustrated in Figure 6-6 on page 6-15 and Figure 6-7 on page 6-17.

6.9 Translating small page references

Figure 6-6 illustrates the complete translation sequence for a 4KB small page. Page translation involves one additional step beyond that of a section translation. The level 1 descriptor is the page table descriptor, and this points to the level 2 descriptor, or page table entry. As the access permissions are now contained in the level 2 descriptor they must be checked before the physical address is generated. The sequence for checking access permissions is described in *Fault checking sequence* on page 6-22.

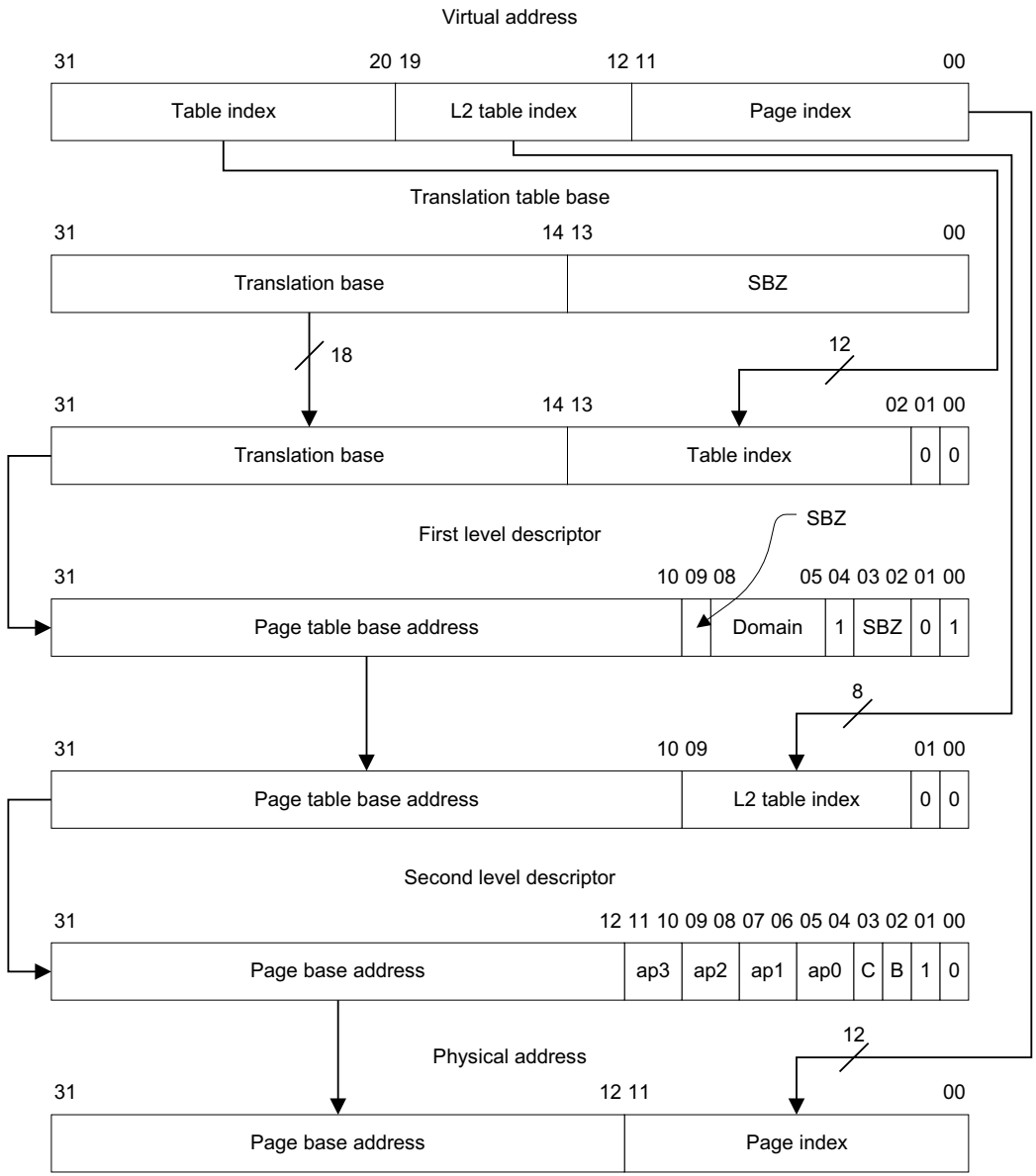


Figure 6-6 Small page translation

6.10 Translating large page references

Figure 6-7 illustrates the complete translation sequence for a 64KB large page. As the upper four bits of the page index and low-order four bits of the page table index overlap, each page table entry for a large page must be duplicated 16 times, in consecutive memory locations, in the page table.

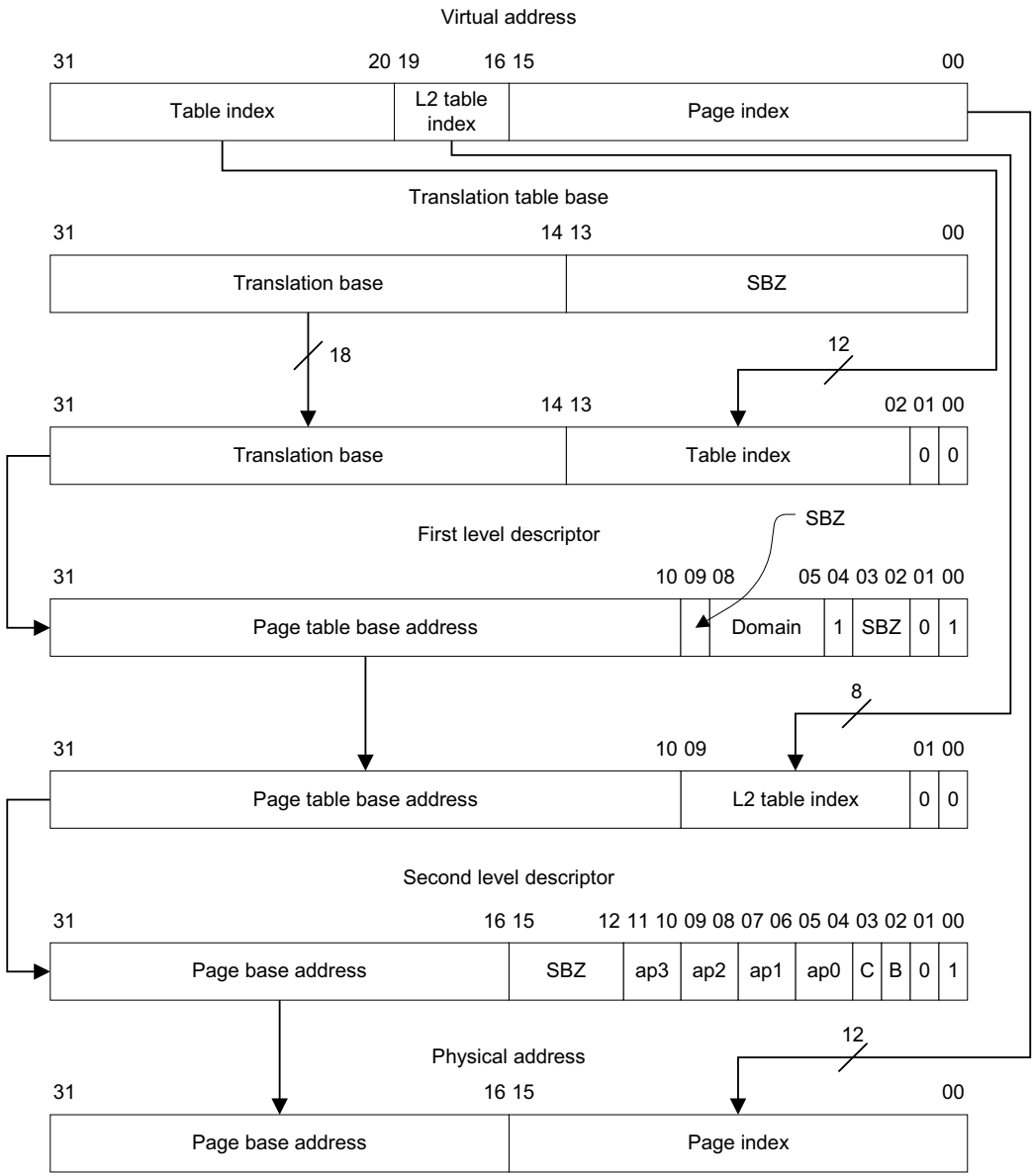


Figure 6-7 Large page translation

6.11 MMU faults and CPU aborts

The MMU generates four types of faults:

- alignment fault
- translation fault
- domain fault
- permission fault.

In addition, an external abort can be raised on external data access.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as the result of a memory access, the MMU aborts the access and signals the fault condition to the CPU. The MMU is also capable of retaining status and address information about the abort. The CPU recognizes two types of abort that are treated differently by the MMU:

- Data Aborts
- Prefetch Aborts.

If the MMU detects an access violation, it does so before the external memory access takes place, and it therefore inhibits the access. External aborts do not necessarily inhibit the external access, as described in *External aborts on page 6-25*.

If the ARM720T is operating in fastbus mode an internally aborting access can cause the address on the external address bus to change, even though the external bus cycle has been canceled. The address that is placed on the bus is the translation of the address that caused the abort, though in the case of a translation fault the value of this address is undefined. No memory access is performed to this address.

6.12 Fault address and fault status registers

Aborts resulting from data accesses, Data Aborts are acted upon by the CPU immediately, and the MMU places an encoded 4-bit value FS[3:0], along with the 4-bit encoded domain number, in the FSR.

In addition, the virtual processor address which caused the data abort is latched into the FAR. If an access violation simultaneously generates more than one source of abort, they are encoded in the priority listed in Table 6-5.

Table 6-5 Priority encoding of fault status

Priority	Source	FS[3:0]	Domain [3:0]	FAR
Highest	Alignment	00x1[a]	Invalid	Valid
	Bus error (translation) level 1	1100	Invalid	Valid
	Level 2	1110	Valid	Valid
	Translation section	0101	Invalid	Valid
	Page	0111	Valid	Valid
	Domain section	1001	Valid	Valid
	Page	1011	Valid	Valid
	Permission section	1101	Valid	Valid
	Page	1111	Valid	Valid
	Bus error (linefetch) section	0100	Valid	Contains the address of the start of the linefetch
	Page	0110	Valid	Contains the address of the start of the linefetch
Lowest	Bus error (other) section	1000	Valid	Valid
	Page	1010	Valid	Valid

a. x is undefined, and can be read as zero or one.

Note

Any abort masked by the priority encoding can be regenerated by fixing the primary abort and restarting the instruction.

CPU instructions are prefetched, so a Prefetch Abort simply flags the instruction as it enters the instruction pipeline. Only when, and if, the instruction is executed does it cause an abort. An abort is not acted upon if the instruction is not used, that is, it is

branched around. Because instruction Prefetch Aborts might not be acted upon, the MMU status information is not preserved for the resulting CPU abort. For a Prefetch Abort, the MMU does not update the FSR or FAR.

The sections that follow describe the various access permissions and controls supported by the MMU and describe how these are interpreted to generate faults.

Note

The FAR will contain a modified virtual address if the process identifier register is nonzero.

6.13 Domain access control

MMU accesses are primarily controlled through domains. There are 16 domains, and each has a 2-bit field to define it.

Two basic kinds of users are supported:

Clients Use a domain.

Managers Control the behavior of the domain.

The domains are defined in the domain access control register. Figure 6-8 illustrates how the 32 bits of the register are allocated to define the 16 2-bit domains.

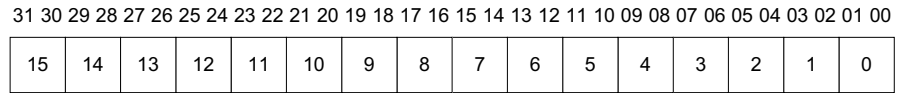


Figure 6-8 Domain access control register format

Table 6-6 lists how the bits within each domain are interpreted to specify the access permissions.

Table 6-6 Interpreting access bits in domain access control register

Value	Meaning	Notes
00	No access	Any access generates a domain fault.
01	Client	Accesses are checked against the access permission bits in the section or page descriptor.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are *not* checked against the access permission bits so a permission fault cannot be generated.

6.14 Fault checking sequence

The sequence the MMU uses to check for access faults is slightly different for sections and pages. Figure 6-9 illustrates the sequence for both types of access.

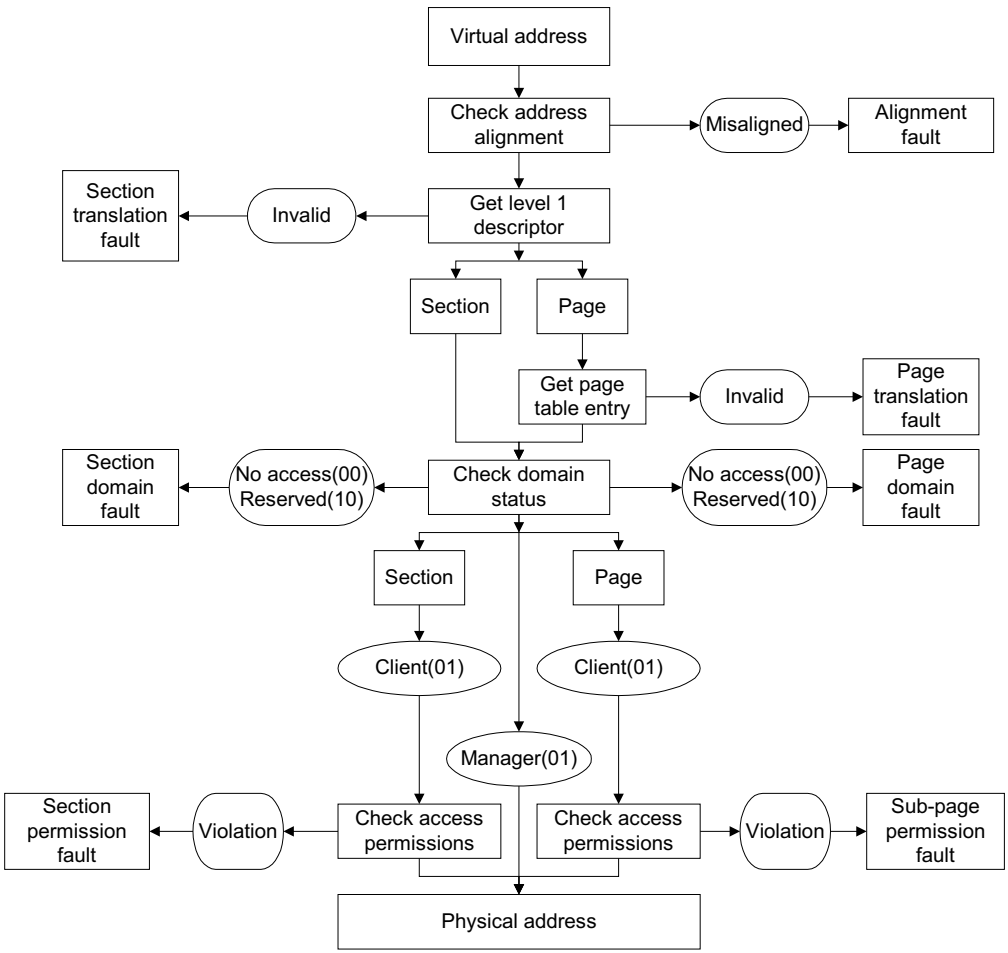


Figure 6-9 Sequence for checking faults

Descriptions of the conditions that generate each of the faults are provided as follows:

- *Alignment fault*
- *Translation fault*
- *Domain fault*
- *Permission fault* on page 6-24.

6.14.1 Alignment fault

If alignment fault is enabled (bit 1 in the control register set), the MMU generates an alignment fault on any data word access without a word-aligned address, irrespective of whether the MMU is enabled or not. In other words, if either of virtual address bits [1:0] are not 0, the alignment fault is enabled.

An alignment fault is not generated on any instruction fetch, nor on any byte access.

Note

If the access generates an alignment fault, the access sequence aborts without reference to further permission checks.

6.14.2 Translation fault

There are two types of translation fault:

Section Is generated if the level 1 descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0, or both 1.

Page Is generated if the page table entry is marked as invalid. This happens if bits[1:0] of the entry are both 0, or both 1.

6.14.3 Domain fault

There are two types of domain fault:

Section The domain is checked when the level 1 descriptor is returned.

Page. The domain is checked when the page table entry is returned.

In both cases, the level 1 descriptor holds the 4-bit domain field that selects one of the 16 2-bit domains in the domain access control register. The two bits of the specified domain are then checked for access permissions as listed in Table 6-3 on page 6-10.

If the specified access is either no access (00) or reserved (10), either a section domain fault or page domain fault occurs.

6.14.4 Permission fault

Permission fault is checked at the same time as domain fault. If the 2-bit domain field returns Client (01), the permission access check is invoked as follows:

There are two types of permission fault:

- section
- subpage.

Section

If the level 1 descriptor defines a section-mapped access, the AP bits of the descriptor define whether or not the access is allowed according to Table 6-3 on page 6-10. Interpretation depends on the setting of the S bit (control register bit 8). If the access is not allowed, a section permission fault is generated.

Subpage

If the level 1 descriptor defines a page-mapped access then the level 2 descriptor specifies four access permission fields (ap3 to ap0), each corresponding to one quarter of the page:

- For small pages:
  - ap3 is selected by the top 1KB of the page
  - ap0 is selected by the bottom 1KB of the page.
- For large pages:
  - ap3 is selected by the top 16KB of the page
  - ap0 is selected by the bottom 16KB of the page.

The selected AP bits are then interpreted in exactly the same way as for a section (see Table 6-3 on page 6-10. The only difference is that the fault generated is a subpage permission fault.

6.15 External aborts

In addition to the MMU-generated aborts, ARM720T has an external abort pin, **BERROR**, which can be used to flag an error on an external memory access. However, not all accesses can be aborted in this way, so use this pin with great care. This section describes the restrictions.

The following accesses can be aborted and restarted safely. The external access stops on the next cycle if any of the following are aborted:

- reads
- unbuffered writes
- level 1 descriptor fetch
- level 2 descriptor fetch
- read-lock-write sequence.

In the case of a read-lock-write sequence in which the read aborts, the write does not happen.

6.15.1 Cachable reads (linefetches)

A linefetch can be safely aborted on any word in the transfer.

If an abort occurs during the linefetch, the cache is purged, so it does not contain invalid data.

If the abort happens on a word that has been requested by the ARM720T, it is aborted, otherwise the cache line is purged but program flow is *not* interrupted. The line is therefore purged under all circumstances.

6.15.2 Buffered writes

Buffered writes cannot be externally aborted. Therefore, the system must be configured so that it does not attempt buffered writes to areas of memory that are capable of flagging an external abort.

Note

Areas of memory that can generate an external abort on a location that has previously been read successfully must not be marked as cachable or unbufferable. This applies to both the MMU page tables and the configuration register. If all writes to an area of memory abort, it is recommended that you mark it as read-only in the MMU, otherwise mark it as uncachable and unbufferable.

6.16 Interaction of the MMU, IDC, and write buffer

The MMU, IDC, and WB can be enabled or disabled independently. However, in order for the write buffer or the cache to be enabled the MMU must also be enabled. There are no hardware interlocks on these restrictions, so invalid combinations cause undefined results. Valid buffer combinations are listed in Table 6-7.

Table 6-7 Valid MMU, IDC and write buffer combinations

MMU	IDC	WB
Off	Off	Off
On	Off	Off
On	On	Off
On	Off	On
On	On	On

The procedures described in *Enabling the MMU* and *Disabling the MMU* on page 6-27 must be observed.

6.16.1 Enabling the MMU

To enable the MMU:

1. Program the translation table base and domain access control registers
2. Program level 1 and level 2 page tables as required.
3. Enable the MMU by setting bit 0 in the control register.

Note

You must take care if the translated address differs from the untranslated address because the two instructions following the enabling of the MMU have been fetched using flat translation. Enabling the MMU might be considered as a branch with delayed execution. A similar situation occurs when the MMU is disabled. Consider the following code sequence:

```
MOV R1, #0x1
MCR 15,0,R1,0,0;        Enable MMU
Fetch Flat
Fetch Flat
Fetch Translated
```


6.16.2 Disabling the MMU

To disable the MMU:

1. Disable the WB by clearing bit 3 in the control register.
2. Disable the IDC by clearing bit 2 in the control register.
3. Disable the MMU by clearing bit 0 in the control register.

You can disable all three functions simultaneously.

Note

If the MMU is enabled, then disabled and subsequently re-enabled, the contents of the TLB are preserved. If these are now invalid, you must flush the TLB before re-enabling the MMU.

Chapter 7
Debug Interface

This chapter describes the ARM720T advanced debug interface. It contains the following sections:

- *About the debug interface* on page 7-2
- *Debug systems* on page 7-4
- *Entering debug state* on page 7-7
- *Scan chains and JTAG interface* on page 7-9
- *Reset* on page 7-11
- *Public instructions* on page 7-12
- *Test data registers* on page 7-16
- *ARM7TDM core clocks* on page 7-23
- *Determining the core and system state* on page 7-25
- *The PC during debug* on page 7-30
- *Priorities and exceptions* on page 7-34
- *Scan interface timing* on page 7-35
- *Scan and debug signals used by the embedded trace logic* on page 7-42.

7.1 About the debug interface

In this chapter ARM7TDM refers to the ARM7TDMI core excluding the EmbeddedICE Logic. The ARM7TDM debug interface is based on IEEE Std. 1149.1-1990, *Standard Test Access Port and Boundary-Scan Architecture*. Refer to this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

7.1.1 Debug extensions

ARM7TDM contains hardware extensions for advanced debugging features. These are intended to ease the development of application software, operating systems, and the hardware itself.

The debug extensions allow you to stop the core either on a given instruction fetch (breakpoint) or data access (watchpoint), or asynchronously by a debug-request. When this happens, ARM7TDM is said to be in *debug state*. At this point, the internal state of the core and the external state of the system can be examined. Once examination is complete, the core and system state can be restored and program execution resumed.

Debug state

ARM7TDM 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 EmbeddedICE Logic. 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.

Internal state

The internal state of the ARM7TDM is examined through a JTAG-style serial interface, that allows instructions to be serially inserted into the pipeline of the core without using the external data bus. When in debug state, a *STore Multiple* (`STM`) can be inserted into the instruction pipeline and this dumps the contents of the ARM7TDM registers. This data can be serially shifted out without affecting the rest of the system.

7.1.2 Pullup resistors

The IEEE 1149.1 standard effectively requires that **XTDI**, **XnTRST**, and **XTMS** have internal pullup resistors. In order to minimize static current draw, these resistors are *not* fitted to ARM7TDM. Accordingly, the four inputs to the test interface (the above three signals plus **XTCK**) must all be driven to good logic levels to achieve normal circuit operation.

7.1.3 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 b0001.

7.2 Debug systems

The ARM7TDM 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 ARM7TDM. Figure 7-1 shows a typical debug system.

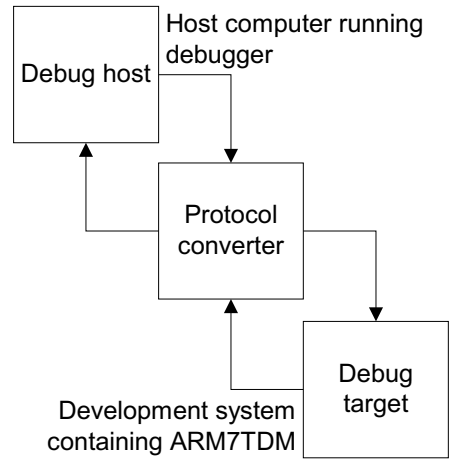


Figure 7-1 Typical debug system

A debug system typically has three parts:

- *Debug host*
- *Protocol converter*
- *resume program execution.* on page 7-5.

7.2.1 Debug host

This is a computer, for example a personal computer, running a software debugger such as *Arm Debugger for Windows* (ADW). The debug host allows you to issue high level commands such as setting breakpoints, or examining the contents of memory.

7.2.2 Protocol converter

The protocol converter interfaces between the high-level commands issued by the debug host and the low-level commands of the ARM720T JTAG interface. Typically it interfaces to the host through an interface such as an enhanced parallel port.

7.2.3 ARM720T

The ARM720T has hardware extensions that ease debugging at the lowest level. The debug extensions:

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

The ARM720T contains the ARM7TDM core. The major blocks of the ARM7TDM core are:

The ARM CPU core
This has hardware support for debug.

The EmbeddedICE Logic
This is a set of registers and comparators used to generate debug exceptions (such as breakpoints). This unit is described in Chapter 8 *EmbeddedICE Logic***.**

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

The anatomy of ARM7TDM is shown in Figure 7-2 on page 7-6 with the ARM720T system control processor.

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

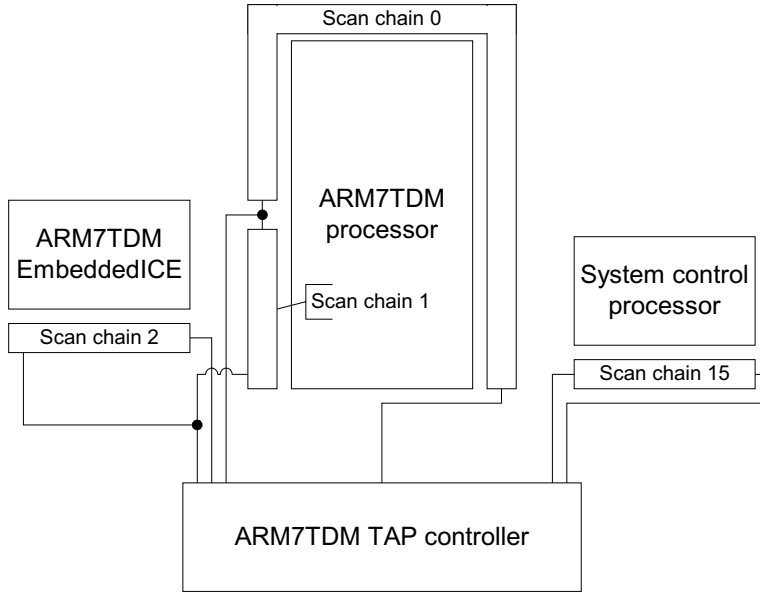


Figure 7-2 ARM7TDM scan chain arrangement

7.3 Entering debug state

ARM7TDM is forced into debug state after a breakpoint, watchpoint, or debug request. You can program the conditions under which a breakpoint or watchpoint occur using EmbeddedICE Logic. Alternatively, external logic can monitor the address and data bus, and flag breakpoints and watchpoints using the **BREAKPOINT** pin.

7.3.1 Entering debug state on breakpoint

After an instruction has been breakpointed, the core does not enter debug state immediately. Instructions are marked as being breakpointed as they enter the ARM7TDM instruction pipeline. Therefore ARM7TDM only enters debug state when and if the instruction reaches the execute stage of the pipeline.

There are two reasons why a breakpointed instruction might not cause ARM7TDM to enter debug state:

- A branch precedes the breakpointed instruction. When the branch is executed, the instruction pipeline is flushed and the breakpoint is canceled.
- An exception has occurred. Again, the instruction pipeline is flushed and the breakpoint is canceled. However, the normal way to exit from an exception is to branch back to the instruction that would have executed next. This involves refilling the pipeline, and so the breakpoint can be re-flagged.

When a breakpointed conditional instruction reaches the Execute stage of the pipeline, the breakpoint is *always* taken and ARM7TDM enters debug state, regardless of whether the condition was met.

Breakpointed instructions *are not* executed. Instead, ARM7TDM enters debug state, so that when the internal state is examined, the state *before* the breakpointed instruction is seen. Once examination is complete, the breakpoint must be removed and program execution restarted from the previously breakpointed instruction.

7.3.2 Entering debug state on watchpoint

Watchpoints occur on data accesses. A watchpoint is always taken, but the core might not enter debug state immediately. In all cases, the current instruction does complete. If this is a multi-word load or store (`LDM` or `STM`), many cycles can elapse before the watchpoint is taken.

Watchpoints are similar to Data Aborts. The difference is that if a Data Abort occurs, although the instruction completes, all subsequent changes to ARM7TDM state are prevented. This allows the cause of the abort to be cured by the abort handler, and the

instruction re-executed. In the case of a watchpoint, the instruction completes and all changes to the core state occur (load data is written into the destination registers, and base writeback occurs). Therefore, the instruction does not have to be restarted.

Watchpoints are *always* taken. If an exception is pending when a watchpoint occurs, the core enters debug state in the mode of that exception.

7.3.3 Entering debug state on debug-request

ARM7TDM can also be forced into debug state on debug request. This can be done either through EmbeddedICE programming (see Chapter 8 *EmbeddedICE Logic*), or by the assertion of the **DBGRQ** pin. This pin is an asynchronous input and is therefore synchronized by logic inside ARM7TDM before it takes effect. Following synchronization, the core normally enters debug state at the end of the current instruction. However, if the current instruction is a busy-waiting access to a coprocessor, the instruction terminates and ARM7TDM enters debug state immediately. This is similar to the action of **nIRQ** and **nFIQ**.

7.4 Scan chains and JTAG interface

There are three JTAG style scan chains inside ARM7TDM and an additional scan chain inside ARM720T. These allow testing, debugging, and EmbeddedICE programming.

In addition, support is provided for further scan chains outside of ARM720T. Unused scan chains can be used for *Application-Specific Integrated Circuit* (ASIC) boundary scan or for ASIC test. The control signals provided for this are described later.

The scan chains are controlled from a JTAG-style *Test Access Port* (TAP) controller. For further details of the JTAG specification, refer to IEEE Standard 1149.1-1990 *Standard Test Access Port and Boundary-Scan Architecture*.

Note

The scan cells are not fully JTAG-compliant, see *Scan limitations* for a description of the limitations on their use.

7.4.1 Scan limitations

The three ARM7TDM scan paths are referred to as scan chain 0, 1, and 2. These are shown in Figure 7-2 on page 7-6. Scan chain functions are described below:

- Scan chain 0 allows access to the entire periphery of the ARM7TDM core, including the data bus. The scan chain functions allow inter-device testing (EXTEST) and serial testing of the core (INTEST). The order of the scan chain (from **SDIN** to **SDOUT**) is:
  - data bus bits 0 to 31
  - the control signals
  - the address bus bits 31 to 0.
- Scan chain 1 is a subset of the signals that are accessible through scan chain 0. Access to the core data bus **D[31:0]**, and the **BREAKPOINT** signal is available serially. There are 33 bits in this scan chain. The order is (from serial data in to out):
  - data bus bits 0 through 31
  - **BREAKPOINT**
- Scan chain 2 allows access to the EmbeddedICE Logic registers. See Chapter 8 *EmbeddedICE Logic* for details.
- Scan chain 15 allows access to the system control coprocessor registers.

7.4.2 The JTAG state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure 7-3 shows the state transitions that occur in the TAP controller. The state numbers are also shown on the diagram.

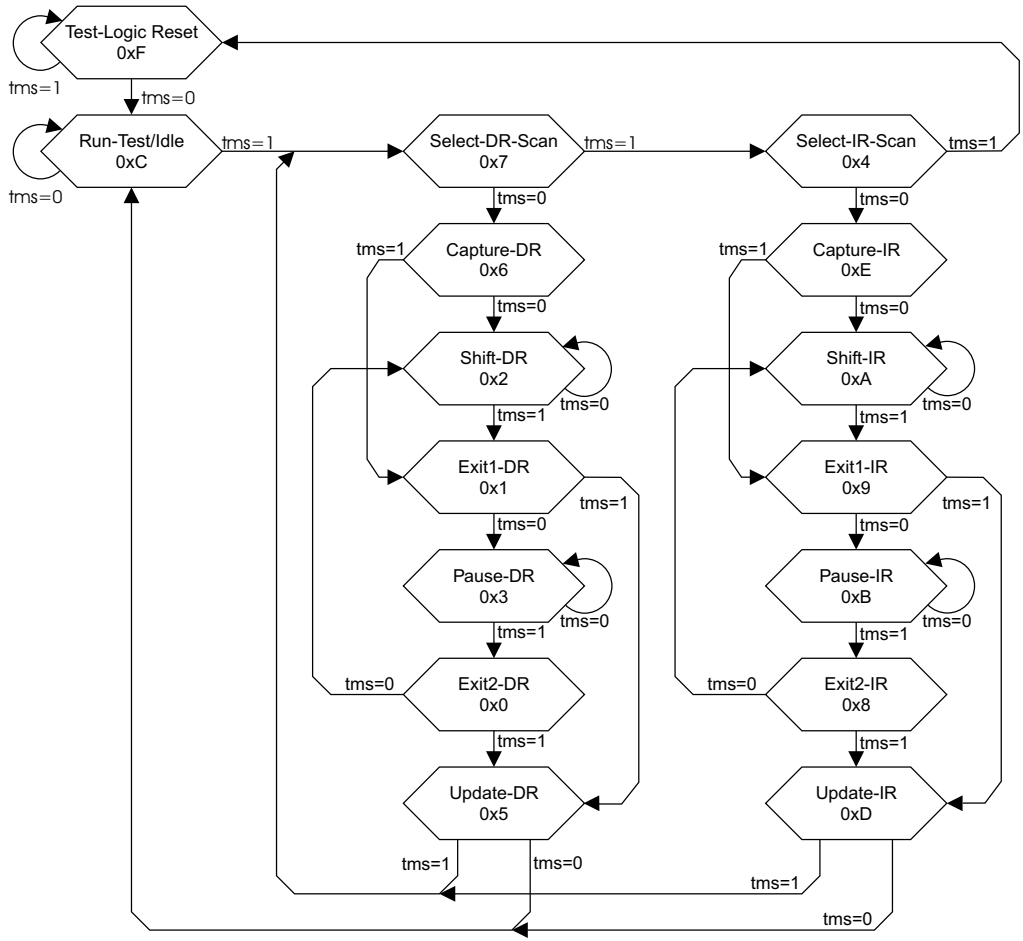


Figure 7-3 Test access port (TAP) controller state transitions

7.5 Reset

The boundary-scan interface includes a state machine controller, the TAP controller. To force the TAP controller into the correct state after power-up of the device, a reset pulse must be applied to the **XnTRST** signal.

If the boundary scan interface is to be used, **XnTRST** must be driven LOW, and then HIGH again. If the boundary scan interface is not to be used, the **XnTRST** input can be tied permanently LOW.

Note

A clock on **XTCK** 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 **XTCK** is pulsed, the contents of the ID register are clocked out of **XTDO**.

7.6 Public instructions

The public instructions are listed in this section. In the descriptions that follow, **XTDI** and **XTMS** are sampled on the rising edge of **XTCK** and all output transitions on **XTDO** occur as a result of the falling edge of **XTCK**.

7.6.1 EXTEST (0000)

This instruction places the selected scan chain in test mode. It connects the selected scan chain between **XTDI** and **XTDO**.

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

CAPTURE-DR Inputs from the system logic and outputs from the output scan cells to the system are captured by the scan cells.

SHIFT-DR The previously captured test data is shifted out of the scan chain using **XTDO**, while new test data is shifted in through the **XTDI** input. This data is applied immediately to the system logic and system pins.

7.6.2 SCAN_N (0010)

This instruction connects the scan path select register between **XTDI** and **XTDO**. On reset, scan chain 3 is selected by default. The scan path select register is four bits long in this implementation, although no finite length is specified.

CAPTURE-DR The fixed value 1000 is loaded into the register.

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

UPDATE-DR The scan register of the selected scan chain is connected between **XTDI** and **XTDO**, and remains connected until a subsequent SCAN_N instruction is issued.

7.6.3 INTEST (1100)

This instruction places the selected scan chain in test mode. It connects the selected scan chain between **XTDI** and **XTDO**.

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

Single-step operation is possible using the INTEST instruction.

CAPTURE-DR 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.

SHIFT-DR The previously captured test data is shifted out of the scan chain using the **XTDO** pin, while new test data is shifted in using the **XTDI** pin.

7.6.4 IDCODE (1110)

This instruction connects the device *IDentification* (ID) register between **XTDI** and **XTDO**. 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. See *ARM7TDM device identification code register* on page 7-16 for details of the ID register format.

When the instruction register is loaded with this instruction, all the scan cells are placed in their normal, System, mode of operation:

CAPTURE-DR The device identification code is captured by the ID register.

SHIFT-DR The previously captured device identification code is shifted out of the ID register using the **XTDO** pin, while data is shifted in through the **XTDI** pin into the ID register.

UPDATE-DR The ID register is unaffected.

7.6.5 BYPASS (1111)

This instruction connects a 1-bit shift register, the bypass register, between **XTDI** and **XTDO**. When this 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.

CAPTURE-DR A logic 0 is captured by the bypass register.

SHIFT-DR Test data is shifted into the bypass register through **XTDI** and out through **XTDO** after a delay of one **XTCK** cycle.

Note

- The first bit shifted out is a zero.
- All unused instruction codes default to the bypass instruction.

UPDATE-DR The bypass register is not affected.

7.6.6 CLAMP (0101)

This instruction connects a 1-bit shift register, the bypass register, between **XTDI** and **XTDO**. When this instruction is loaded into the instruction register, the state of all the output signals is defined by the values previously loaded into the currently loaded scan chain.

Note

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

CAPTURE-DR A logic 0 is captured by the bypass register.

SHIFT-DR Test data is shifted into the bypass register using **XTDI** and out using **XTDO** after a delay of one **XTCK** cycle.

Note

The first bit shifted out is a zero.

UPDATE-DR The bypass register is not affected.

7.6.7 HIGHZ (0111)

This instruction connects a 1-bit shift register, the bypass register, between **XTDI** and **XTDO**. When this instruction is loaded into the instruction register, the address bus, **A[31:0]**, the data bus, **D[31:0]**, plus **nRW**, **nOPC**, **LOCK**, **MAS[1:0]**, and **nTRANS** are all driven to the high impedance state and the external **HIGHZ** signal is driven HIGH. This is as if the signal **TBE** had been driven LOW.

CAPTURE-DR A logic 0 is captured by the bypass register.

SHIFT-DR Test data is shifted into the bypass register using **XTDI** and out using **XTDO** after a delay of one **XTCK** cycle.

Note

The first bit shifted out is a zero.

UPDATE-DR The bypass register is not affected.

7.6.8 CLAMPZ (1001)

This instruction connects a 1-bit shift register, the bypass register, between **XTDI** and **XTDO**. When this instruction is loaded into the instruction register, all the 3-state outputs 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 a logic 1.

CAPTURE-DR A logic 0 is captured by the bypass register.

SHIFT-DR Test data is shifted into the bypass register through **XTDI** and out through **XTDO** after a delay of one **XTCK** cycle.

Note

The first bit shifted out will be a zero.

UPDATE-DR The bypass register is not affected.

7.6.9 RESTART (0100)

This instruction restarts the processor on exit from debug state. It connects the bypass register between **XTDI** and **XTDO,** and the TAP controller behaves as if the bypass instruction had been loaded. The processor resynchronizes back to the memory system once the RUN-TEST/IDLE state is entered.

7.6.10 SAMPLE/PRELOAD (0011)

This instruction is included for production test only, and must never be used.

7.7 Test data registers

You can connect five test data registers between **XTDI** and **XTDO**:

- *This register bypasses the device during scan testing by providing a path between XTDI and XTDO. The bypass register is 1 bit in length.*
- *ARM7TDM device identification code register*
- *This register changes the current TAP instruction. The register is four bits in length.* on page 7-17
- *This register changes the current active scan chain. The register is 4 bits in length.* on page 7-17
- *These allow serial access to the core logic, and to EmbeddedICE Logic for programming purposes. They are described in this section and shown in Figure 7-5 on page 7-19.* on page 7-18.

These are described in the following sections.

7.7.1 Bypass register

This register bypasses the device during scan testing by providing a path between **XTDI** and **XTDO**. The bypass register is 1 bit in length.

Operating mode

When the BYPASS instruction is the current instruction in the instruction register, serial data is transferred from **XTDI** to **XTDO** in the SHIFT-DR state with a delay of one **XTCK** cycle.

There is no parallel output from the bypass register.

A logic 0 is loaded from the parallel input of the bypass register in the CAPTURE-DR state.

7.7.2 ARM7TDM device identification code register

This register reads the 32-bit device ID code. No programmable supplementary identification code is provided. The register is 32 bits in length.

The format of the ID register is shown in Figure 7-4.

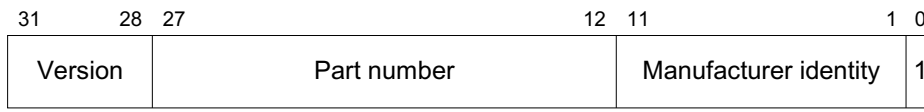


Figure 7-4 ID code register format

Contact your supplier for the correct device identification code.

Operating mode

When the IDCODE instruction is current, the ID register is selected as the serial path between **XTDI** and **XTDO**.

There is no parallel output from the ID register.

The 32-bit device identification code is loaded into the ID register from its parallel inputs during the CAPTURE-DR state.

7.7.3 Instruction register

This register changes the current TAP instruction. The register is four bits in length.

Operating mode

When in the SHIFT-IR state, the instruction register is selected as the serial path between **XTDI** and **XTDO**.

During the CAPTURE-IR state, the value 0001 binary is loaded into this register. This is shifted out during SHIFT-IR *Least Significant Bit* (LSB) first, while a new instruction is shifted in (LSB first).

During the UPDATE-IR state, the value in the instruction register becomes the current instruction.

On reset, IDCODE becomes the current instruction.

7.7.4 Scan chain select register

This register changes the current active scan chain. The register is 4 bits in length.

Operating mode

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

During the CAPTURE-DR state, the value 1000 binary is loaded into this register. This is shifted out during SHIFT-DR (LSB first), while a new value is shifted in (LSB 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[3:0]** outputs. You can use the TAP controller to drive external scan chains in addition to those within the ARM7TDM macrocell. You must assign the external scan chain a number and control signals for it can be derived from **SCREG[3:0]**, **IR[3:0]**, **TAPSM[3:0]**, **TCK1**, and **TCK2**.

The list of scan chain numbers allocated by ARM are listed in Table 7-1. An external scan chain can take any other number. The serial data stream to be applied to the external scan chain is made present on **SDINBS**. The serial data back from the scan chain must be presented to the TAP controller on the **SDOUTBS** input.

The scan chain present between **SDINBS** and **SDOUTBS** is connected between **XTDI** and **XTDO** 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[3:0]**.

Table 7-1 Scan chain number allocation

Scan chain number	Function
0	Macrocell scan test
1	Debug
2	EmbeddedICE programming
3	Reserved (external boundary scan)
4	Reserved
8	Reserved
15	System control coprocessor

7.7.5 Scan chains 0, 1, 2, and 15

These allow serial access to the core logic, and to EmbeddedICE Logic for programming purposes. They are described in this section and shown in Figure 7-5 on page 7-19.

Scan chains 0 and 1 allow access to the processor core for test and debug. They have the following lengths:

- Scan chain 0, 105 bits

- Scan chain 1, 33 bits.

Each scan chain cell consists of a serial register and a multiplexor. The scan cells perform two basic functions:

Capture For input cells, the capture stage involves copying the value of the system input to the core into the serial register.
For output cells, capture involves placing the value of a core output into the serial register.

Shift For input cells, during shift, this value is output serially. The value applied to the core from an input cell is either the system input or the contents of the serial register, and this is controlled by the multiplexor.
For output cells, during shift, this value is serially output as before. The value applied to the system from an output cell is either the core output, or the contents of the serial register.

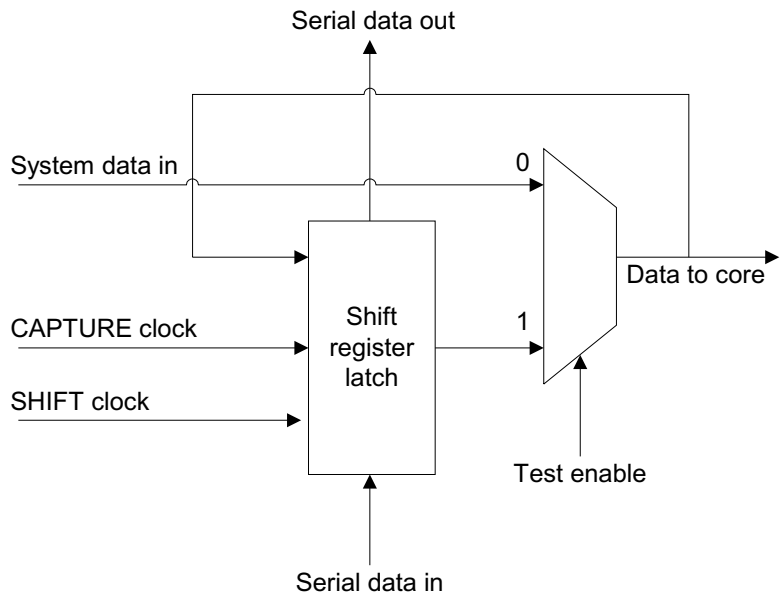


Figure 7-5 Input scan cell

All the control signals for the scan cells are generated internally by the TAP controller. The action of the TAP controller is determined by the current instruction, and the state of the TAP state machine. This is described in *Operating modes* on page 7-20.

Operating modes

The scan chains have three basic modes of operation, selected by the various TAP controller instructions:

SYSTEM mode The scan cells are idle. System data is applied to inputs, and core outputs are applied to the system.

INTEST mode The core is internally tested. The data serially scanned in is applied to the core, and the resulting outputs are captured in the output cells and scanned out.

EXTEST mode Data is scanned onto the core outputs and applied to the external system. System input data is captured in the input cells and then shifted out.

Note

- The scan cells are not fully JTAG-compliant because they do not have an update stage. Therefore, while data is being moved around the scan chain, the contents of the scan cell are not isolated from the output. Therefore the output from the scan cell to the core or to the external system can change on every scan clock.
- This does not affect ARM7TDM because its internal state does not change until it is clocked. However, the rest of the system has to be aware that every output can change asynchronously as data is moved around the scan chain. External logic must ensure that this does not harm the rest of the system.

7.7.6 Scan chain 0

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

Serial testing the core

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

- During CAPTURE-DR, the current outputs from the core logic are captured in the output cells.
- During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, applying known stimuli to the inputs.
- During RUN-TEST-IDLE, the core is clocked. The TAP controller must only spend one cycle in RUN-TEST-IDLE.

The whole operation can then be repeated.

See *ARM7TDM core clocks* on page 7-23 for details of the core clocks during test and debug.

Inter-device testing

EXTEST allows inter-device testing. This is useful for verifying the connections between devices on a circuit board. 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 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, applying known values on the core outputs.
- During UPDATE-DR, the value shifted into the data bus **D[31:0]** scan cells appears on the outputs. For all other outputs, the value appears as the data is shifted round.

Note

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

The operation can then be repeated. The ordering of signals on scan chain 0 is listed in Table 7-3 on page 7-37.

7.7.7 Scan chain 1

The primary use for scan chain 1 is for debugging, although it can be used for EXTEST on the data bus. Scan chain 1 is selected using the SCAN_N TAP controller instruction. Debugging is similar to INTEST, and the procedure described above for scan chain 0 must be followed.

Scan chain length and purpose

This scan chain is 33 bits long.32 bits are for the data value, plus an additional bit for the scan cell on the **BREAKPOINT** core input. This 33rd bit serves four purposes:

1. Under normal INTEST test conditions, it allows a known value to be scanned into the **BREAKPOINT** input.
2. During EXTEST test conditions, the value applied to the **BREAKPOINT** input from the system can be captured.

3. While debugging, the value placed in the 33rd bit determines whether ARM7TDM synchronizes back to system speed before executing the instruction. See *System-speed access* on page 7-32 for more information.

4. After ARM7TDM has entered debug state, the first time this bit is captured and scanned out, its value tells the debugger whether the core entered debug state is due to a breakpoint (bit 33 LOW), or a watchpoint (bit 33 HIGH).

7.7.8 Scan chain 2

This scan chain allows you to access the EmbeddedICE Logic registers. The scan chain is 38 bits in length.

The order of the scan chain from **XTDI** to **XTDO** is:

- read/write
- register address bits 4 to 0
- data value bits 31 to 0

See Figure 8-2 on page 8-5 for more information.

To access this serial register, scan chain 2 must first be selected using the SCAN_N TAP controller instruction. The TAP controller must then be place 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 Logic register to be accessed.
- During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read). Refer to Chapter 8 *EmbeddedICE Logic* for further details.

7.7.9 Scan chain 15

This scan chain allows access to the system control coprocessor registers. Scan chain 15 is selected using the SCAN_N TAP controller instruction. This scan chain is 33 bits long. 32 bits are for the data or instruction value plus an additional bit that identifies the value as instruction (1) or data (0). This scan chain must only be used during INTEST. The order of the scan chain from **XTDI** to **XTDO** is:

- **CPDATA** [0:31]
- instruction or data flag.

7.8 ARM7TDM core clocks

ARM7TDM has two clocks:

- the memory clock, **MCLK**, generated by the ARM720T
- an internally **XTCK**-generated clock, **DCLK**.

During normal operation, the core is clocked by **MCLK**, and internal logic holds **DCLK** LOW.

There are two cases in which the clocks switch:

- during debugging
- during testing.

7.8.1 Clock switch during debug

When ARM7TDM is in the debug state, the core is clocked by **DCLK** under the control of the TAP state machine, and **MCLK** can free run. The selected clock is output on the signal **ECLK** for use by the external system.

Note

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

When ARM7TDM enters debug state, it must switch from **MCLK** to **DCLK**. This is handled automatically by logic in the ARM7TDM. On entry to debug state, ARM7TDM asserts **DBGACK** in the HIGH phase of **MCLK**. The switch between the two clocks occurs on the next falling edge of **MCLK**. This is shown in Figure 7-6.

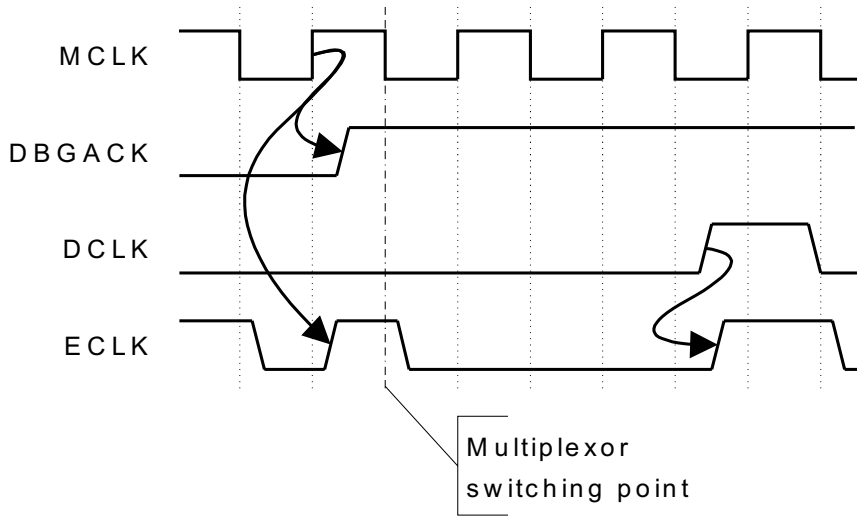


Figure 7-6 Clock switching on entry to debug state

ARM7TDM 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 **MCLK**. This must be done in the following sequence:

1. The final instruction of the debug sequence must be shifted into the data bus scan chain and clocked in by asserting **DCLK**.
2. At this point, BYPASS must be clocked into the TAP instruction register.
3. ARM7TDM now automatically resynchronizes back to **MCLK** and starts fetching instructions from memory at **MCLK** speed.

See *Exit from debug state* on page 7-28 for more information.

7.9 Determining the core and system state

When ARM7TDM is in debug state, you can examine the core and system state. This is done by forcing load and store multiples into the instruction pipeline.

7.9.1 Determining ARM or Thumb state

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. You can achieve this by examining bit 4 of the EmbeddedICE Logic debug status register. If this is HIGH, the core was in Thumb state when it entered debug.

7.9.2 Determining the state of the core

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 state.

While in debug state, only the following instructions can legally be scanned into the instruction pipeline for execution:

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

Moving to ARM state

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

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

As all Thumb instructions are only 16 bits long, the simplest method when shifting them into scan chain 1 is to repeat the instruction twice.

For example, the encoding for `BX R0` is `0x4700`. Therefore, if `0x47004700` is shifted into scan chain 1, the debugger does not have to keep track of which half of the bus the processor expects to read the data from.

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

In ARM state

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

```
STM R0, {R0-R15}
```

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

Note

The use of R0 as the base register for `STM` is for illustration only. Any register can be used.

Accessing banked registers

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

Note

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

For example, assume that the debugger is asked to return the state of the USER and FIQ mode registers, and debug state was entered in Supervisor mode.

The instruction sequence might be as listed below:

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

All these instructions are said to execute at *debug speed*. Debug speed is much slower than system speed because between each core clock, 33 scan clocks occur to shift in an instruction, or shift out data. Executing instructions more slowly than usual is acceptable for accessing the core state because ARM7TDM is fully static. However, this same method cannot be used for determining the state of the rest of the system.

7.9.3 Determining system state

To meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously with it. Therefore, ARM7TDM must be forced to synchronize back to system speed. This is controlled by the 33rd bit of scan chain 1.

You can place any instruction in scan chain 1 with bit 33, the BREAKPT bit, LOW. This instruction is then executed at debug speed. To execute an instruction at system speed, the instruction prior to it must be scanned into scan chain 1 with bit 33 set HIGH.

After the system speed instruction has been scanned into the data bus and clocked into the pipeline, the BYPASS instruction must be loaded into the TAP controller. This makes the ARM7TDM automatically synchronize back to **MCLK,** the system clock, executes the instruction at system speed, and then re-enters debug state and switches itself back to the internally generated **DCLK**. When the instruction has completed, **DBGACK** is HIGH and the core switches back to **DCLK**. At this point, INTEST can be selected in the TAP controller, and debugging can resume.

To determine that a system speed instruction has completed, the debugger must look at both **DBGACK** and **nMREQ**. In order to access memory, ARM7TDM drives **nMREQ** LOW after it has synchronized back to system speed. This transition is used by the memory controller to arbitrate whether ARM7TDM can have the bus in the next cycle. If the bus is not available, ARM7TDM can have its clock stalled indefinitely.

Therefore, the only way to tell that the memory access has completed, is to examine the state of both **nMREQ** and **DBGACK**. When both are HIGH, the access has completed. The debugger normally uses EmbeddedICE Logic to control debugging, and by reading the EmbeddedICE Logic status register, the state of **nMREQ** and **DBGACK** can be determined. Refer to Chapter 8 *EmbeddedICE Logic* for more details.

Using system speed load multiples and debug speed store multiples, the system memory state can be fed back to the debug host.

Restrictions

There are restrictions on which instructions can have the 33rd bit set. The only valid instructions where this bit can be set are:

- loads
- stores

- load multiple
- store multiple.

See also *Exit from debug state*.

When ARM7TDM returns to debug state after a system speed access, bit 33 of scan chain 1 is set HIGH. This gives the debugger information about why the core entered debug state the first time this scan chain is read.

7.9.4 Determining system control coprocessor state

To access the system control coprocessor registers, debug state must be entered by a breakpoint, watchpoint, or debug request. This ensures that the ARM7TDM core stops execution of code that might be dependent on the system control coprocessor.

Scan chain 15 can then be selected using the **SCAN_N** instruction.

Instructions can then be scanned down the scan chain as if being executed from the ARM7TDM core. As the ARM7TDM is idle while scan chain 15 is being accessed, you must provide the register data using the scan chain. The instruction prior to the data must have the instruction or data flag cleared.

The data operation requires an additional clock from the TAP controller. This can be achieved by remaining in the RUN-TEST-IDLE state for an additional **XTCK** cycle.

7.9.5 Exit from debug state

Leaving debug state involves:

1. Restoring ARM7TDM internal state.
2. Branching to the next instruction to be executed.
3. Synchronizing back to **MCLK**.

After restoring internal state, a branch instruction must be loaded into the pipeline. See *The PC during debug* on page 7-30 for details on calculating the branch.

Bit 33 of scan chain 1 is used to force ARM7TDM to resynchronize back to **MCLK**. The penultimate instruction of the debug sequence is scanned in with bit 33 set HIGH. The final instruction of the debug sequence is the branch, and this is scanned in with bit 33 LOW. The core is then clocked to load the branch into the pipeline. Now, the RESTART instruction is selected in the TAP controller.

When the state machine enters the RUN-TEST-IDLE state, the scan chain reverts back to system mode and clock resynchronization to **MCLK** occurs within ARM7TDM. ARM7TDM then resumes normal operation, fetching instructions from memory. This

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

7.10 The PC during debug

The debugger must keep track of what happens to the PC so that ARM7TDM can be forced to branch back to the place at which program flow was interrupted by debug. There are five cases when this occurs:

- *Entry to the debug state from a breakpoint advances the PC by four addresses, or 16 bytes. Each instruction executed in debug state advances the PC by one address, or four bytes. The normal way to exit from debug state after a breakpoint is to remove the breakpoint, and branch back to the previously breakpointed address.*
- *Returning to program execution after entering debug state from a watchpoint is done in the same way as the procedure described above. Debug entry adds four addresses to the PC, and every instruction adds one address. The difference is that because the instruction that caused the watchpoint has executed, the program returns to the next instruction.* on page 7-31
- *Watchpoint with another exception* on page 7-31
- *Debug request* on page 7-32
- *System-speed access* on page 7-32.

A summary of the method used to determine the return address is provided in *Summary of return address calculations* on page 7-33.

7.10.1 Breakpoint

Entry to the debug state from a breakpoint advances the PC by four addresses, or 16 bytes. Each instruction executed in debug state advances the PC by one address, or four bytes. 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 ARM7TDM entered debug state from a breakpoint set on a given address and two debug-speed instructions were executed, a branch of minus seven addresses must occur. Four are for debug entry, plus two for the instructions, plus one for the final branch.

The following sequence shows the data scanned into scan chain 1. This is *Most Significant Bit* (MSB) first, and so the first digit is the value placed in the BREAKPT bit, followed by the instruction data:

```
0 E0802000; ADD R2, R0, R0

1 E1826001; ORR R6, R2, R1

0 EAFFFFF9; B -7 (2s complement)
```


Once in debug state, a minimum of two instructions must be executed before the branch, although these can both be NOPs, for example:

```
MOV R0, R0
```

For small branches, the final branch can be replaced by a subtract with the PC as the destination:

```
SUB PC, PC, #28
```

7.10.2 Watchpoint

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

7.10.3 Watchpoint with another exception

If a watchpointed access simultaneously causes a Data Abort, ARM7TDM enters debug state in abort mode. Entry into debug is held off until the core has changed into abort mode, and fetched the instruction from the abort vector.

A similar sequence is followed when an interrupt, or any other exception, occurs during a watchpointed memory access. ARM7TDM enters debug state in the exception mode, and so 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 does take place, you must give the user the choice of whether to service the exception before debugging.

Exiting from debug state

Exiting debug state if an exception occurred is slightly different from the other cases. Here, entry to debug state causes the PC to be incremented by three addresses rather than four, and this must be taken into account in the return branch calculation. For example, suppose that an abort occurred on a watchpointed access and ten instructions had been executed to determine this. The following sequence can be used to return to program execution:

```
0 E1A00000; MOV R0, R0

1 E1A00000; MOV R0, R0

0 EAFFFFF0; B -16
```

This forces a branch back to the abort vector, causing the instruction at that location to be refetched and executed.

Note

After the abort service routine, the instruction that caused the abort and watchpoint is re-executed. This generates the watchpoint and ARM7TDM enters debug state again.

7.10.4 Debug request

Entry into debug state through a debug request is similar to a breakpoint. However, unlike a breakpoint, the last instruction has 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, suppose that you invoke a debug request, and decide to return to program execution straight away. The following sequence can be used:

```
0 E1A00000; MOV R0, R0

1 E1A00000; MOV R0, R0

0 EAFFFFFA; B -6
```

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

7.10.5 System-speed access

If a system-speed access is performed during debug state, the value of the PC is increased by three addresses. As system-speed instructions access the memory system, aborts can take place. If an abort occurs during a system-speed memory access, ARM7TDM enters abort mode before returning to debug state.

This is similar to an aborted watchpoint except that the problem is much harder to fix, 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 which caused the abort, and therefore the abort address. In this case, the value of the PC is invalid, but the debugger must know what location was being accessed. Therefore, the debugger can be written to help the abort handler fix the memory system.

7.10.6 Summary of return address calculations

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

- For normal breakpoint and watchpoint, the branch is:
  (4 + N + 3S)
- For entry through debug request (**DBGRQ**), or watchpoint with exception, the branch is:
  (3 + N + 3S)

where:

- N is the number of debug speed instructions executed, including the final branch
- S is the number of system speed instructions executed.

7.11 Priorities and exceptions

Because the normal program flow is broken when a breakpoint or a debug request occurs, debug can be considered as being another type of exception. Some of the interaction with other exceptions is been described in *Entering debug state* on page 7-7 and *The PC during debug* on page 7-30. This section summarizes these priorities.

7.11.1 Breakpoint with Prefetch Abort

When a breakpointed instruction fetch causes a Prefetch Abort, the abort is taken and the breakpoint is disregarded. Usually, Prefetch Aborts occur when, for example, an access is made to a virtual address that does not physically exist, and the returned data is therefore invalid.

In this case, the normal action of the operating system is to swap in the page of memory and return to the previously invalid address. Here, when the instruction is fetched, and providing the breakpoint is activated (it might be data-dependent), ARM7TDM enters debug state.

In this case, the Prefetch Abort takes higher priority than the breakpoint.

7.11.2 Interrupt

When ARM7TDM enters debug state, interrupts are automatically disabled. If interrupts are disabled during debug, ARM7TDM is never forced into an interrupt mode. Interrupts only have this effect on watchpointed accesses. They are ignored at all times on breakpoints.

If an interrupt is pending during the instruction prior to entering debug state, ARM7TDM enters debug state in the mode of the interrupt. So, on entry to debug state, the debugger cannot assume that ARM7TDM is in the expected mode of the program. It must check the PC, the CPSR, and the SPSR to fully determine the reason for the exception.

Debug takes higher priority than the interrupt, although ARM7TDM *remembers* that an interrupt has occurred.

7.11.3 Data Aborts

When a Data Abort occurs on a watchpointed access, ARM7TDM enters debug state in abort mode. Therefore, the watchpoint has higher priority than the abort although, as in the case of interrupt, ARM7TDM remembers that the abort happened.

7.12 Scan interface timing

Figure 7-7 and Table 7-2 provide general scan timing information.

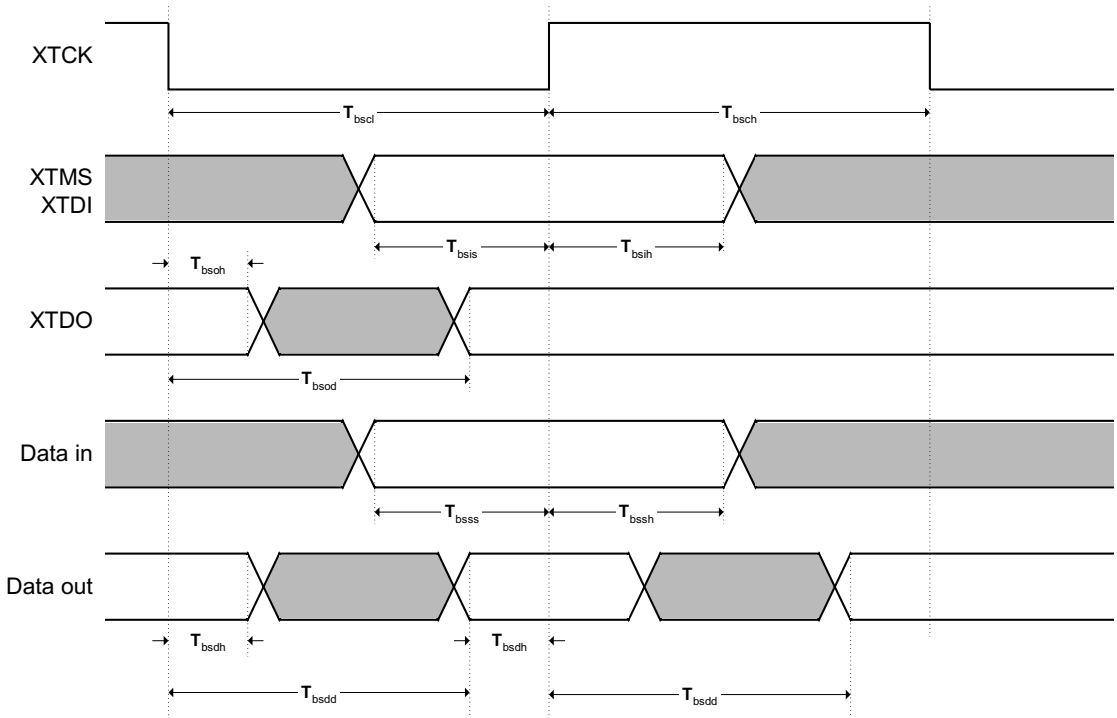


Figure 7-7 Scan general timing

Table 7-2 ARM720T scan interface timing

Symbol	Parameter
T_{bscl}	**XTCK** low period
T_{bsch}	**XTCK** high period
T_{bsis}	**XTDI**, **XTMS** setup to $\mathbf{XTCK_r}$
T_{bsih}	**XTDI**, **XTMS** hold from $\mathbf{XTCK_r}$
T_{bsoh}	**XTDO** hold time from $\mathbf{XTCK_f}$
T_{bsod}	$\mathbf{XTCK_f}$ to **XTDO** valid

Table 7-2 ARM720T scan interface timing (continued)

Symbol	Parameter
T_{bsss}[a]	I/O signal setup to **XTCK$_r$**
T_{bssh}[a]	I/O signal hold from **XTCK$_r$**
T_{bsdh}	Data output hold time from **XTCK**
T_{bsdd}	**XTCK** to data output valid
T_{bsr}	Reset period
T_{bse}	Output enable time
T_{bsz}	Output disable time

a.For correct data latching, the I/O signals (from the core and pads) must be setup and held with respect to the rising edge of **XTCK** in the CAPTURE-DR state of the INTEST and EXTEST instructions

Contact your supplier for AC timing parameter values.

Figure 7-8 shows the T_{bsr} (reset period timing) parameter.

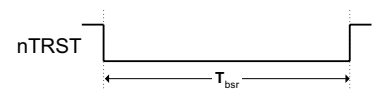


Figure 7-8 Reset period timing

Figure 7-9 shows the T_{bse} parameter (output enable time) and T_{bsz} (output disable time) when the HIGHZ TAP instruction is loaded into the instruction register.

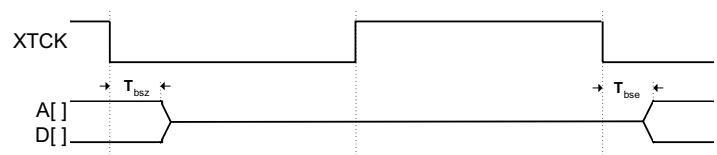


Figure 7-9 Output enable and disable times due to HIGHZ TAP instruction

Figure 7-10 shows the T_{bse} parameter (output enable time) and T_{bsz} (output disable time) when data scanning.

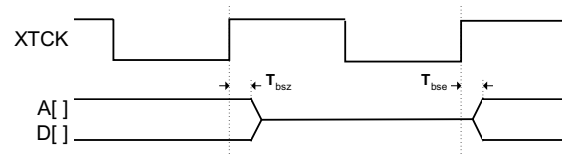


Figure 7-10 Output enable and disable times due to data scanning

Table 7-3 lists the signals and positions for scan chain 0.

Table 7-3 Scan chain 0, signals and positions

Number	Signal	Type
1	**D[0]**	Input/output
2	**D[1]**	Input/output
3	**D[2]**	Input/output
4	**D[3]**	Input/output
5	**D[4]**	Input/output
6	**D[5]**	Input/output
7	**D[6]**	Input/output
8	**D[7]**	Input/output
9	**D[8]**	Input/output
10	**D[9]**	Input/output
11	**D[10]**	Input/output
12	**D[11]**	Input/output
13	**D[12]**	Input/output
14	**D[13]**	Input/output
15	**D[14]**	Input/output
16	**D[15]**	Input/output

Table 7-3 Scan chain 0, signals and positions (continued)

Number	Signal	Type
17	**D[16]**	Input/output
18	**D[17]**	Input/output
19	**D[18]**	Input/output
20	**D[19]**	Input/output
21	**D[20]**	Input/output
22	**D[21]**	Input/output
23	**D[22]**	Input/output
24	**D[23]**	Input/output
25	**D[24]**	Input/output
26	**D[25]**	Input/output
27	**D[26]**	Input/output
28	**D[27]**	Input/output
29	**D[28]**	Input/output
30	**D[29]**	Input/output
31	**D[30]**	Input/output
32	**D[31]**	Input/output
33	**BREAKPT**	Input
34	**NENIN**	Input
35	**NENOUT**	Output
36	**LOCK**	Output
37	**BIGEND**	Input
38	**DBE**	Input
39	**MAS[0]**	Output
40	**MAS[1]**	Output
41	**BL[0]**	Input

Table 7-3 Scan chain 0, signals and positions (continued)

Number	Signal	Type
42	**BL[1]**	Input
43	**BL[2]**	Input
44	**BL[3]**	Input
45	**DCTL**[a]	Output
46	**nRW**	Output
47	**DBGACK**	Output
48	**CGENDBGACK**	Output
49	**nFIQ**	Input
50	**nIRQ**	Input
51	**nRESET**	Input
52	**ISYNC**	Input
53	**DBGRQ**	Input
54	**ABORT**	Input
55	**CPA**	Input
56	**nOPC**	Output
57	**IFEN**	Input
58	**nCPI**	Output
59	**nMREQ**	Output
60	**SEQ**	Output
61	**nTRANS**	Output
62	**CPB**	Input
63	**nM[4]**	Output
64	**nM[3]**	Output
65	**nM[2]**	Output
66	**nM[1]**	Output

Table 7-3 Scan chain 0, signals and positions (continued)

Number	Signal	Type
67	**nM[0]**	Output
68	**nEXEC**	Output
69	**ALE**	Input
70	**ABE**	Input
71	**APE**	Input
72	**TBIT**	Output
73	**nWAIT**	Input
74	**A[31]**	Output
75	**A[30]**	Output
76	**A[29]**	Output
77	**A[28]**	Output
78	**A[27]**	Output
79	**A[26]**	Output
80	**A[25]**	Output
81	**A[24]**	Output
82	**A[23]**	Output
83	**A[22]**	Output
84	**A[21]**	Output
85	**A[20]**	Output
86	**A[19]**	Output
87	**A[18]**	Output
88	**A[17]**	Output
89	**A[16]**	Output
90	**A[15]**	Output
91	**A[14]**	Output

Table 7-3 Scan chain 0, signals and positions (continued)

Number	Signal	Type
92	**A[13]**	Output
93	**A[12]**	Output
94	**A[11]**	Output
95	**A[10]**	Output
96	**A[9]**	Output
97	**A[8]**	Output
98	**A[7]**	Output
99	**A[6]**	Output
100	**A[5]**	Output
101	**A[4]**	Output
102	**A[3]**	Output
103	**A[2]**	Output
104	**A[1]**	Output
105	**A[0]**	Output

a. **DCTL** is not described in this datasheet. **DCTL** is an output from the processor used to control the unidirectional data out latch, **DOUT[31:0]**. This signal is not visible from the periphery of ARM7TDM.

7.13 Scan and debug signals used by the embedded trace logic

The signals listed in Table 7-4 exist on the ARM720T and are used to configure and control the ETM. Refer to the *ETM7 Technical Reference Manual* for more information on scan chain connection between the ARM720T core and ETM7, and DBGRQ connection.

Table 7-4 Scan and debug signals used by the ETM

Signal	Type
DBGRQ	Input
XnTRST	Input
SDOUTBS	Input
XTCK	Input
XTDI	Input
XTMS	Input
RANGEOUT0	Output
RANGEOUT1	Output

Chapter 8
EmbeddedICE Logic

This chapter describes the ARM720T EmbeddedICE Logic. It contains the following sections.

- *About EmbeddedICE Logic* on page 8-2
- *The watchpoint registers* on page 8-4
- *Programming breakpoints* on page 8-9
- *Programming watchpoints* on page 8-11
- *The debug control register* on page 8-13
- *Debug status register* on page 8-15
- *Coupling breakpoints and watchpoints* on page 8-17
- *Debug communications channel* on page 8-19.

8.1 About EmbeddedICE Logic

The ARM7TDM EmbeddedICE Logic, referred to as *EmbeddedICE*, provides integrated on-chip debug support for the ARM7TDM core.

In this chapter ARM7TDM refers to the ARM7TDMI core excluding the EmbeddedICE Logic. EmbeddedICE is programmed in a serial fashion using the ARM7TDM TAP controller. It consists of two real-time watchpoint units, together with a control and status register. You can program one or both watchpoint units to halt the execution of instructions by the ARM7TDM core using the **BREAKPT** signal.

Two independent registers, debug control and debug status, provide overall control of EmbeddedICE operation. Figure 8-1 shows the relationship between the core, EmbeddedICE, and the TAP controller.

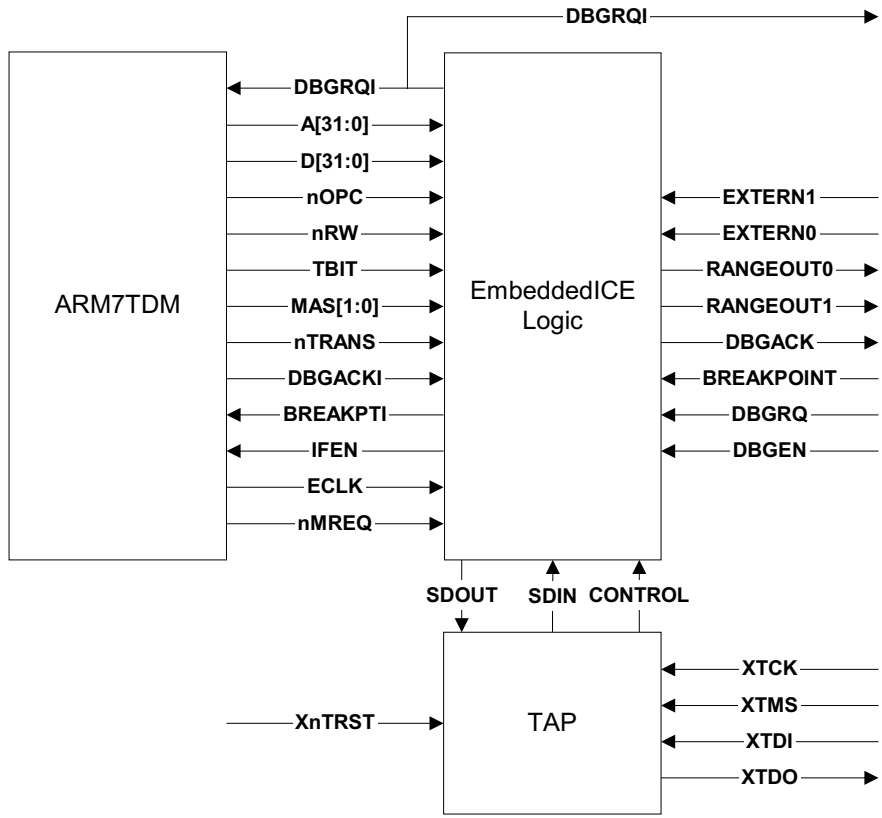


Figure 8-1 ARM7TDMI TAP controller and EmbeddedICE

Execution is halted when a match occurs between the values programmed into EmbeddedICE and the values currently appearing on the address bus, data bus, and various control signals. Any bit can be masked so that its value does not affect the comparison.

Note

- Only those signals that are pertinent to EmbeddedICE are shown.
- In the ARM720T, the EmbeddedICE module is connected directly to the ARM7TDM core and therefore functions on the virtual address of the processor after relocation by the FCSE PID.

Either of the two real-time watchpoint units can be configured to be a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). You can make watchpoints and breakpoints data-dependent.

8.1.1 Disabling EmbeddedICE

You can disable EmbeddedICE by wiring the **DBGEN** input LOW.

When **DBGEN** is LOW, **BREAKPOINT** and **DBGRQ** to the core are forced LOW, **DBGACK** from the ARM7TDM is also forced LOW, and the **IFEN** input to the core is forced HIGH, enabling interrupts to be detected by ARM7TDM.

When **DBGEN** is LOW, EmbeddedICE is also put into a low-power mode.

8.1.2 EmbeddedICE timing

The **EXTERN1** and **EXTERN0** inputs are sampled by EmbeddedICE on the falling edge of **ECLK**. Therefore you must allow sufficient set-up and hold time for these signals.

8.2 The watchpoint registers

The two watchpoint units, known as *watchpoint 0* and *watchpoint 1*. Each contain three pairs of registers:

- address value and address mask
- data value and data mask
- control value and control mask.

Each register is independently programmable and has its own address, as listed in Table 8-1.

Table 8-1 Function and mapping of EmbeddedICE registers

Address	Width	Function
00000	3	Debug control
00001	5	Debug status
00100	6	Debug comms control register
00101	32	Debug comms data register
01000	32	Watchpoint 0 address value
01001	32	Watchpoint 0 address mask
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 1address 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

8.2.1 Programming and reading watchpoint registers

A register is programmed by scanning data into the EmbeddedICE scan chain using scan chain 2. The scan chain consists of a 38-bit shift register comprising:

- a 32-bit data field
- a 5-bit address field
- a read/write bit.

This is shown in Figure 8-2.

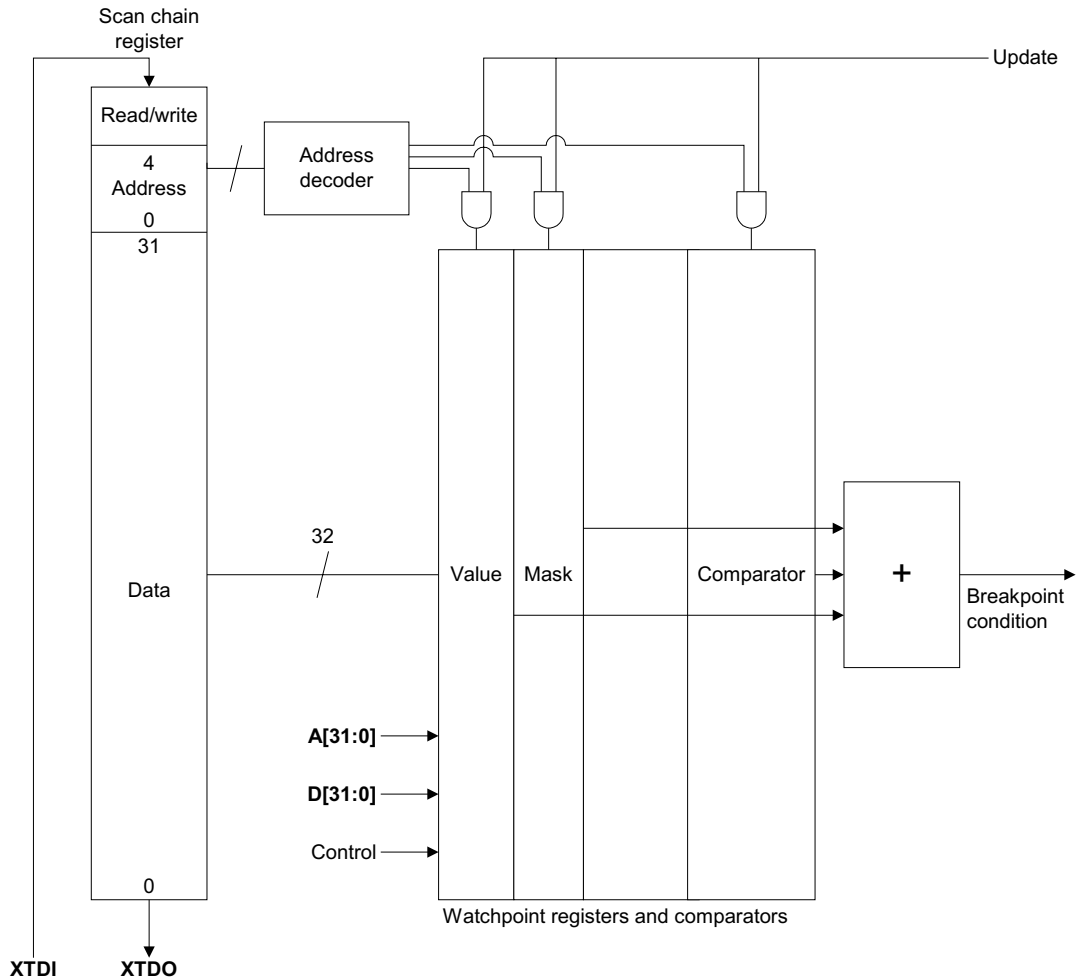


Figure 8-2 EmbeddedICE block diagram

The data to be written is scanned into the 32-bit data field, the address of the register into the 5-bit address field, and a 1 into the read/write bit.

A register is read by scanning its address into the address field and scanning a 0 into the read/write bit. The 32-bit data field is ignored. The register addresses are shown in Table 8-1 on page 8-4.

Note

A read or write takes place when the TAP controller enters the UPDATE-DR state.

8.2.2 Using the mask registers

For each value register in a register pair, there is a mask register of the same format. Setting a bit to 1 in the mask register has the effect of disregarding the corresponding bit in the value register in the comparison. For example, if a watchpoint is required on a particular memory location but the data value is irrelevant, you can program the data mask register to `0xFFFFFFFF`, all bits set to 1, to make the entire data bus field ignored.

Note

The mask is an XNOR mask rather than a conventional AND mask. When a mask bit is set to 1, the comparator for that bit position always matches, irrespective of the value register or the input value.

Setting the mask bit to 0 means that the comparator only matches if the input value matches the value programmed into the value register.

8.2.3 The control registers

Control value and control mask registers are mapped identically in the lower 8 bits. Bit 8 of the control value register is the ENABLE bit, which cannot be masked. The control value and mask format is shown in Figure 8-3.

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	EXTERN	nTRANS	nOPC	MAS[1]	MAS[0]	nRW

Figure 8-3 Watchpoint control value and mask format

The bits have the following functions:

nRW Compares against the not-read/write signal from the core in order to detect the direction of bus activity. **nRW** is 0 for a read cycle and 1 for a write cycle.

MAS[1:0] Compares against the **MAS[1:0]** signal from the core in order to detect the size of bus activity. The encoding is shown in Table 8-2.

Table 8-2 MAS[1:0] signal encoding

Bit 1	Bit 0	Data size
0	0	byte
0	1	halfword
1	0	word
1	1	(reserved)

nOPC Detects if the current cycle is an instruction fetch (nOPC = 0) or a data access (nOPC = 1).

nTRANS Compares against the not-translate signal from the core in order to distinguish between User mode (nTRANS = 0) and non-User mode (nTRANS = 1) accesses.

EXTERN Is an external input to EmbeddedICE that allows the watchpoint to be dependent upon an external condition. The EXTERN input for watchpoint 0 is labeled EXTERN0 and the EXTERN input for watchpoint 1 is labeled EXTERN1. This is known as **nUSER** on ARM720T and has an allocated output.

CHAIN Can be connected to the 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 ARM7TDM EmbeddedICE, 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 **XnTRST** is LOW.

RANGE Can be connected to the range output of another watchpoint register. In the ARM7TDM EmbeddedICE, the RANGEOUT output of watchpoint 1 is connected to the RANGE input of watchpoint 0. This allows the two watchpoints to be coupled for detecting conditions that occur simultaneously, for example, in range-checking.

ENABLE Only exists in the value register and it cannot be masked. If a watchpoint match occurs, the **BREAKPOINT** signal is asserted only when the ENABLE bit is set.

For each of the bits [8:0] in the control value register, there is a corresponding bit in the control mask register. This removes the dependency on particular signals.

8.3 Programming breakpoints

Breakpoints can be classified as hardware breakpoints or software breakpoints:

Hardware These typically monitor the address value and can be set in any code, even in code that is in ROM or code that is self-modifying.

Software These monitor a particular bit pattern being fetched from any address. Therefore you can use one EmbeddedICE watchpoint to support any number of software breakpoints. Software breakpoints can usually only be set in RAM because an instruction has to be replaced by the special bit pattern chosen to cause a software breakpoint.

8.3.1 Hardware breakpoints

To make a watchpoint unit cause hardware breakpoints on instruction fetches:

1. Program its address value register with the address of the instruction to be breakpointed.
2. Program the breakpoint bits for each state as follows:
   a. ARM, set bits [1:0] of the address mask register to one.
   b. Thumb, set bit 0 of the address mask to one.

   In both cases, the remaining bits are set to zero.
3. Program the data value register only if you require a data-dependent breakpoint, that is, only if the actual instruction code fetched must be matched as well as the address. If the data value is not required, program the data mask register to `0xFFFFFFFF`, all bits to one, otherwise program it to `0x00000000`.
4. Program the control value register with **nOPC** = zero.
5. Program the control mask register with **nOPC** = zero, all other bits to one.
6. If you have to make the distinction between User and non-User mode instruction fetches, program the **nTRANS** value and mask bits as above.
7. If required, program the **EXTERN**, **RANGE**, and **CHAIN** bits in the same way.

8.3.2 Software breakpoints

To make a watchpoint unit cause software breakpoints, that is, on instruction fetches of a particular bit pattern:

1. Program its address mask register to `0xFFFFFFFF`, all bits set to one, so that the address is disregarded.
2. Program the data value register with the particular bit pattern that has been chosen to represent a software breakpoint.
3. For a Thumb software breakpoint, the 16-bit pattern must be repeated in both halves of the data value register. For example, if the bit pattern is `0xDFFF`, then `0xDFFFDFFF` must be programmed. When a 16-bit instruction is fetched, EmbeddedICE only compares the valid half of the data bus against the contents of the data value register. In this way, a single watchpoint register can be used to catch software breakpoints on both the upper and lower halves of the data bus.
4. Program the data mask register to `0x00000000`.
5. Program the control value register with **nOPC** = zero.
6. Program the control mask register with **nOPC** = zero, all other bits to one.
7. If you have to make the distinction between User and non-User mode instruction fetches, program the **nTRANS** bit in the control value and control mask registers accordingly.
8. If required, program the **EXTERN**, **RANGE**, and **CHAIN** bits in the same way.

Note

The address value register does not have to be programmed.

Setting the breakpoint

To set the software breakpoint:

1. Read the instruction at the desired address and store it.
2. Write the special bit pattern representing a software breakpoint at the address.

Clearing the breakpoint

To clear the software breakpoint, restore the instruction to the address.

8.4 Programming watchpoints

This section contains examples of how to program the watchpoint register to generate breakpoints and watchpoints. Many other ways of programming the registers are possible. For instance, simple range breakpoints can be provided by setting one or more of the address mask bits.

To make a watchpoint unit cause watchpoints, that is, on data accesses:

1. Program its address value register with the address of the data access to be watchpointed.
2. Program the address mask register to `0x00000000`.
3. Program the data value register only if you require a data-dependent watchpoint, that is, only if the actual data value read or written must be matched as well as the address. If the data value is irrelevant, program the data mask register to `0xFFFFFFFF` (all bits set to one) otherwise program it to `0x00000000`.
4. Program the control value register with:
   a. **nOPC** = one.
   b. **nRW** = zero for a read.
   c. **nRW** = one for a write.
   d. **MAS[1:0]** with the value corresponding to the appropriate data size.
5. Program the control mask register with:
   a. **nOPC** = zero.
   b. **nRW** = zero.
   c. **MAS[1:0]** = zero.
   d. all other bits to zero.

Note

nRW or **MAS[1:0]** can be set to one if both reads and writes or data size accesses are to be watchpointed respectively.

6. If you have to make the distinction between User and non-User mode data accesses, program the **nTRANS** bit in the control value and control mask registers accordingly.
7. If required, program the **EXTERN**, **RANGE**, and **CHAIN** bits in the same way.

8.4.1 Programming restriction

The EmbeddedICE watchpoint units must only be programmed when the clock to the core is stopped. You can achieve this by putting the core into the debug state.

The reason for this restriction is that if the core continues to run at **ECLK** rates when EmbeddedICE is being programmed at **XTCK** rates, it is possible for the **BREAKPOINT** signal to be asserted asynchronously to the core.

This restriction does not apply if **MCLK** and **XTCK** are driven from the same clock, or if it is known that the breakpoint or watchpoint condition can only occur some time after EmbeddedICE has been programmed.

Note

This restriction does not apply to the debug control or status registers.

8.5 The debug control register

The debug control register is 3 bits wide.

- If the register is accessed for a write, with the read/write bit HIGH, the control bits are written.
- If the register is accessed for a read, with the read/write bit LOW, the control bits are read.

The functions of the register bits are shown in Figure 8-4 and described as follows:

- *DBGRQ*
- *DBGACK*
- *INTDIS* on page 8-14.

2	1	0
INTDIS	DBGRQ	DBGACK

Figure 8-4 Debug control register format

Bits 1 and 0 allow you to force the values on **DBGRQ** and **DBGACK**.

8.5.1 DBGRQ

As shown in Figure 8-6 on page 8-16, the value stored in bit 1 of the control register is synchronized and then ORed with the external **DBGRQ** before being applied to the processor. The output of this OR gate is the signal **DBGRQI** which is brought out externally from the macrocell.

The synchronization between control bit 1 and **DBGRQI** assists in multiprocessor environments. The synchronization latch only opens when the TAP controller state machine is in the RUN-TEST-IDLE state. This allows an *enter debug* condition to be set up in all the processors in the system while they are still running. Once the condition is set up in all the processors, you can then applied it to them simultaneously by entering the RUN-TEST-IDLE state.

8.5.2 DBGACK

In the case of **DBGACK**, the value of **DBGACK** from the core is ORed with the value held in bit 0 to generate the external value of **DBGACK** seen at the periphery of ARM7TDM. This allows the debug system to signal to the rest of the system that the core is still being debugged even when system-speed accesses are being performed. The internal **DBGACK** signal from the core is LOW.

8.5.3 INTDIS

If bit 2, **INTDIS**, is asserted, the interrupt enable signal, **IFEN**, of the core is forced LOW. Therefore all interrupts (IRQ and FIQ) are disabled during debugging (**DBGACK** =1) or if the **INTDIS** bit is asserted. The **IFEN** signal is driven as listed in Table 8-3.

Table 8-3 IFEN signal control

DBGACK	INTDIS	IFEN
0	0	1
1	x	0
x	1	0

8.6 Debug status register

The debug status register is 5 bits wide:

- if it is accessed for a write, with the read/write bit set HIGH, the status bits are written
- if it is accessed for a read, with the read/write bit LOW, the status bits are read.

The debug status register is shown in Figure 8-5:.

4	3	2	1	0
TBIT	nMREQ	IFEN	DBGRQ	DBGACK

Figure 8-5 Debug status register format

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. As the capture clock for the scan chain can be asynchronous to the processor clock, the **DBGACK** output from the core is synchronized before being used to generate the IFEN status bit.

Bit 3 Allows the state of the **NMREQ** signal from the core, synchronized to **XTCK** to be read. This allows the debugger to determine that a memory access from the debug state has completed.

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

The structure of the debug status register is shown in Figure 8-6 on page 8-16.

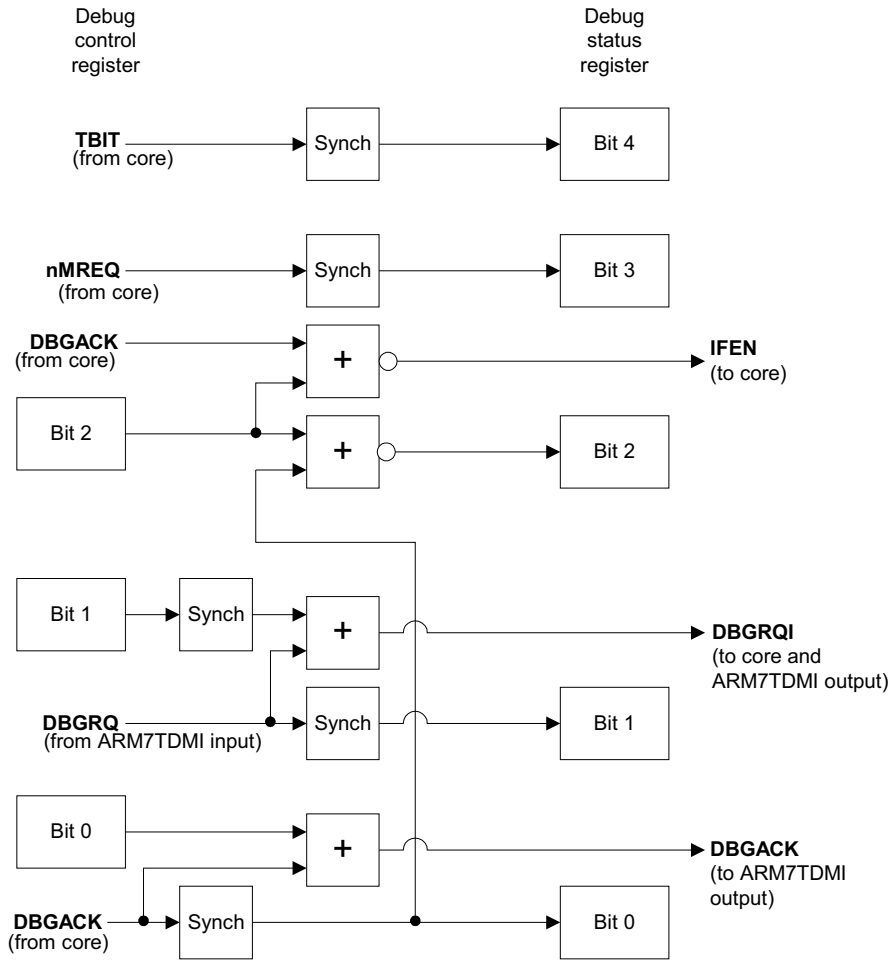


Figure 8-6 Debug control and status register structure

8.7 Coupling breakpoints and watchpoints

You can couple watchpoint units 1 and 0 together using the **CHAIN** and **RANGE** inputs:

- **CHAIN** enables watchpoint 0 to be triggered only if watchpoint 1 has previously matched
- **RANGE** enables simple range checking to be performed by combining the outputs of both watchpoints.

Example 8-1 Coupling breakpoints and watchpoints

Let:

`Av[31:0]`	be the value in the address value register.
`Am[31:0]`	be the value in the address mask register.
`A[31:0]`	be the address bus from the ARM7TDM.
`Dv[31:0]`	be the value in the data value register.
`Dm[31:0]`	be the value in the data mask register.
`D[31:0]`	be the data bus from the ARM7TDM.
`Cv[8:0]`	be the value in the control value register.
`Cm[7:0]`	be the value in the control mask register.
`C[9:0]`	be the combined control bus from the ARM7TDM, other watchpoint registers and the **EXTERN** signal.

8.7.1 CHAINOUT

The **CHAINOUT** signal is then derived as follows:

```
WHEN (({Av[31:0],Cv[4:0]} XNOR {A[31:0],C[4:0]}) OR
{Am[31:0],Cm[4:0]} == 0x1FFFFFFFF)

CHAINOUT = ((({Dv[31:0],Cv[7:5]} XNOR {D[31:0],C[7:5]}) OR
{Dm[31:0],Cm[7:5]}) == 0x7FFFFFFF)
```

The **CHAINOUT** output of watchpoint register 1 provides the **CHAIN** input to watchpoint 0. This allows for quite complicated configurations of breakpoints and watchpoints. For example, consider the request by a debugger to breakpoint on the instruction at location YYY when running process XXX in a multiprocess system.

If the current process ID is stored in memory, you can implement the above function with a watchpoint and breakpoint chained together. The watchpoint address is set to a known memory location containing the current process ID. The watchpoint data is set to the required process ID and the ENABLE bit is set to *off*.

The address comparator output of the watchpoint drives the write enable for the **CHAINOUT** latch, the input to the latch being the output of the data comparator from the same watchpoint. The output of the latch drives the **CHAIN** input of the breakpoint comparator. The address YYY is stored in the breakpoint register and when the **CHAIN** input is asserted, and the breakpoint address matches, the breakpoint triggers correctly.

8.7.2 RANGEOUT

The **RANGEOUT** signal is then derived as follows:

```
RANGEOUT = ((({Av[31:0],Cv[4:0]} XNOR {A[31:0],C[4:0]}) OR
{Am[31:0],Cm[4:0]}) == 0xFFFFFFFFF) AND ((({Dv[31:0],Cv[7:5]}
XNOR {D[31:0],C[7:5]}) OR {Dm[31:0],Cm[7:5]}) == 0x7FFFFFFFF)
```

The **RANGEOUT** output of watchpoint register 1 provides the **RANGE** input to watchpoint register 0. This allows you to couple two breakpoints together to form range breakpoints.

Note

Selectable ranges are restricted to being powers of 2.

If a breakpoint is to occur when the address is in the first 256 bytes of memory, but not in the first 32 bytes, the watchpoint registers must be programmed as follows:

1. Watchpoint 1 is programmed with an address value of `0x00000000` and an address mask of `0x0000001F`. The ENABLE bit is cleared. All other watchpoint 1 registers are programmed as normal for a breakpoint. An address within the first 32 bytes causes the **RANGE** output to go HIGH but the breakpoint is not triggered.
2. Watchpoint 0 is programmed with an address value of `0x00000000` and an address mask of `0x000000FF`. The ENABLE bit is set and the RANGE bit programmed to match a 0. All other watchpoint 0 registers are programmed as normal for a breakpoint.

If watchpoint 0 matches but watchpoint 1 does not (that is, the **RANGE** input to watchpoint 0 is 0), the breakpoint is triggered.

8.8 Debug communications channel

The ARM7TDM EmbeddedICE 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
- a 6-bit wide comms control register for synchronized handshaking between the processor and the asynchronous debugger.

These registers live in fixed locations in the EmbeddedICE memory map (as shown in Table 8-1 on page 8-4) and are accessed from the processor using `MCR` and `MRC` instructions to coprocessor 14.

8.8.1 Debug communications channel registers

The debug comms control register is read-only and allows synchronized handshaking between the processor and the debugger. The register format is shown in Figure 8-7.

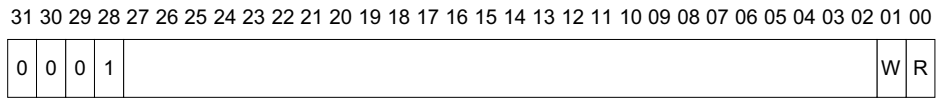


Figure 8-7 Debug comms control register

The function of each register bit is:

Bits [31:28] Contain a fixed pattern that denotes the EmbeddedICE version number, in this case 0001.

Bit [1] Denotes whether the comms data write register is free from the processor point of view.

From the processor point of view:
If the Comms data write register is free (W=0), new data can be written.
If it is not free (W=1), the processor must poll until W=0.

From the debugger point of view, if W=1 then new data has been written which can then be scanned out.

Bit [0] Denotes if there is some new data in the comms data read register.

From the processor point of view:
If R=1, there is some new data which can be read using an `MRC` instruction.

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

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

Instructions

The following instructions must be used:

This instruction returns the debug comms control register into Rd:

```
MRC CP14, 0, Rd, C0, C0
```

This instruction writes the value in Rn to the comms data write register:

```
MCR CP14, 0, Rn, C1, C0
```

This instruction returns the debug data read register into Rd:

```
MRC CP14, 0, Rd, C1, C0
```

Note

As the Thumb instruction set does not contain coprocessor instructions, it is recommended that these are accessed using SWI instructions when in Thumb state.

8.8.2 Communications using the comms channel

Communication between the debugger and the processor occurs as follows:

1. When the processor wishes to send a message to EmbeddedICE, it first checks that the comms data write register is free for use. This is done by reading the debug comms control register to check that the W bit is clear:
   a. If it is clear, the comms data write register is empty and a message is written by a register transfer to the coprocessor. The action of this data transfer automatically sets the W bit.
   b. If it is set, this implies that previously-written data has not been picked up by the debugger and the processor must poll until the W bit is clear.
2. Because the data transfer occurs from the processor to the comms data write register, the W bit is set in the debug comms control register.
3. When the debugger polls this register, it sees a synchronized version of both the R and W bit:
   a. When the debugger sees that the W bit is set, it can read the comms data write register and scan the data out.
   b. The action of reading this data register clears the W bit of the debug comms control register. At this point, the communications process will begin again.

8.8.3 Message transfer

Message transfer from the debugger to the processor is carried out in a similar fashion to *Communications using the comms channel* on page 8-20:

1. The debugger polls the R bit of the debug comms control register:
   a. If the R bit is LOW, the data read register is free and so data can be placed there for the processor to read.
   b. If the R bit is set, previously deposited data has not yet been collected and so the debugger must wait.
2. When the comms data read register is free, data is written there using the scan chain. The action of this write sets the R bit in the debug comms control register.
3. When the processor polls this register, it sees an **MCLK** synchronized version:
   a. If the R bit is set, this denotes that there is data waiting to be collected, and this can be read using a CPRT load. The action of this load clears the R bit in the debug comms control register.
   b. If the R bit is clear, this denotes that the data has been taken and the process can now be repeated.

Chapter 9
Bus Clocking

This chapter describes the bus interface clocking. It contains the following sections:

- *About the ARM720T bus interface* on page 9-2
- *Fastbus extension* on page 9-3
- *Standard mode* on page 9-5.

9.1 About the ARM720T bus interface

The ARM720T bus interface can be operated using either:

- the standard mode of operation
- the new fastbus extension.

As the ARM720T is a fully static design, you can stop the clock indefinitely in either mode of operation.

Note

Take care to ensure that the memory system does not dissipate power in the state in which it is stopped.

9.1.1 Standard mode

For designs using low-cost, low-speed memory, and if operation of the core at a faster speed is required, it is recommended that you use standard mode.

This mode consists of:

- two clocks, **FCLK** and **BCLK**
- synchronous or fully asynchronous operation.

9.1.2 Fastbus extension

For new designs, you can operate the device using the fastbus extension. In fastbus mode, the device is clocked off a single clock, and the bus is operated at the same frequency as the core. This allows the bus interface to be clocked faster than if the device is operated in standard mode. It is recommended that you use this mode of operation in systems with high-speed memory and a single clock.

This mode consists of:

- single device clock
- increased maximum **BCLK** frequency.

9.2 Fastbus extension

Using the fastbus extension, the ARM720T has a single input clock, **BCLK**. This clocks the internals of the device, and qualified by **BWAIT,** controls the memory interface as shown in Figure 9-1.

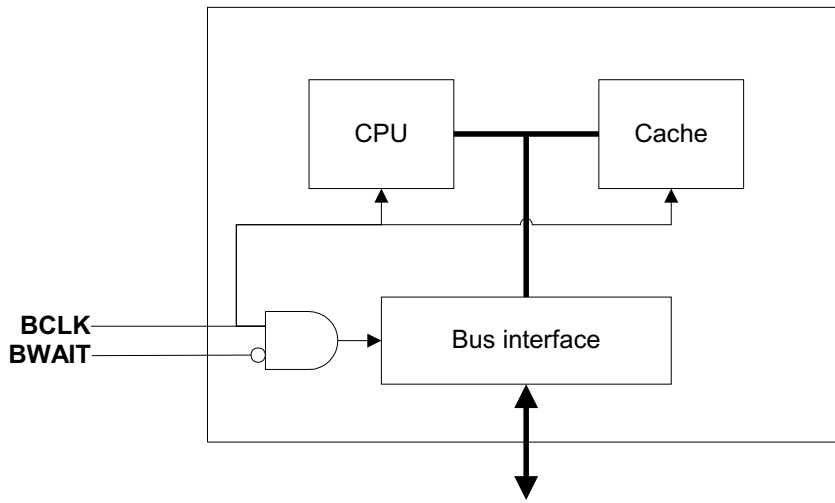


Figure 9-1 Conceptual device clocking using the fastbus extension

When operating the device with **XFASTBUS** HIGH, the inputs **FCLK** and **XSnA** are not used.

Note

To prevent unwanted power dissipation, ensure that they do not float to an undefined level. New designs must tie these signals LOW for compatibility with future products.

9.2.1 Using BWAIT

The **BWAIT** signal inserts entire **BCLK** cycles into the bus cycle timing. **BWAIT** can only change when **BCLK** is LOW, and extends the memory access by inserting **BCLK** cycles into the access while **BWAIT** is asserted.

Figure 10-4 on page 10-11 shows the use of **BWAIT** in more detail.

Memory cycles

It is preferable to use **BWAIT** to extend memory cycles, rather than stretching **BCLK** externally to the device because it is possible for the core to be accessing the cache while bus activity is occurring. This allows the maximum performance, as the core can continue execution in parallel with the memory bus activity. All **BCLK** cycles are available to the CPU and cache, regardless of the state of **BWAIT**.

In some circumstances, it is desirable to stretch **BCLK** phases to match memory timing that is not an integer multiple of **BCLK**. There are certain cases where this results in a higher performance than using **BWAIT** to extend the access by an integer number of cycles.

CPU and cache operation

CPU and cache operation can only continue in parallel with buffered writes to the external bus. For all read accesses, the CPU is stalled until the bus activity has completed. So, if read accesses can be achieved faster by stretching **BCLK** rather than using **BWAIT,** this results in improved performance. An example of where this can be useful is to interface to a ROM which has a cycle time of 2.5 times the **BCLK** period.

9.3 Standard mode

Using the standard mode of operation, without the fastbus extension, and **XFASTBUS** tied LOW, the ARM720T has two input clocks:

- **FCLK**
- **BCLK**.

The bus interface is always controlled by the memory clock, **BCLK**, qualified by **BWAIT**. However, the core and cache are clocked by the fast clock, **FCLK**.

In standard mode, the **FCLK** frequency must be greater than or equal to the **BCLK** frequency at all times. This relationship must be maintained on a cycle-by-cycle basis.

9.3.1 Memory access

When running in this mode, you can stretch memory access cycles by:

- using **BWAIT**
- by stretching phases of **BCLK**.

The resulting performance is determined by the access time, regardless of which method is used. This is shown in Figure 9-2.

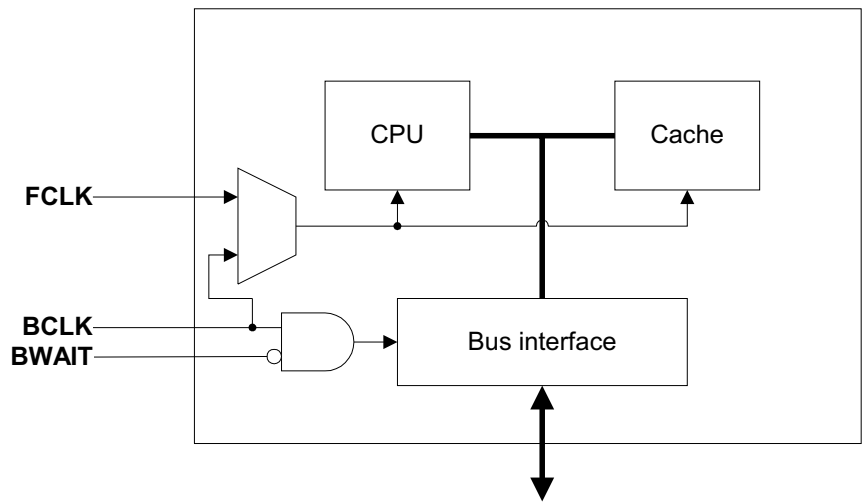


Figure 9-2 Conceptual device clocking in standard mode

9.3.2 Synchronous and asynchronous modes

When not using the fastbus extension, the ARM720T bus interface has two distinct modes of operation:

- synchronous
- asynchronous.

These are selected by tying **XSnA** either HIGH or LOW.

FCLK and BCLK

The two modes differ in the relationship between **FCLK** and **BCLK**:

- In asynchronous mode (**XSnA** LOW), the clocks can be completely asynchronous and of unrelated frequency.
- In synchronous mode (**XSnA** HIGH), **BCLK** can only make transitions before the falling edge of **FCLK**.

In systems where a satisfactory relationship exists between **FCLK** and **BCLK**, synchronization penalties can be avoided by selecting the synchronous mode of operation.

Asynchronous mode

In this mode, **FCLK** and **BCLK** can be completely asynchronous. You must select this mode by tying **XSnA** LOW when the two clocks are of unrelated frequency.

There is a synchronization penalty whenever the internal core clock switches between the two input clocks. This penalty is symmetrical, and varies between zero and a whole period of the clock to which the core is resynchronizing:

- when changing from **FCLK** to **BCLK**, the average resynchronization penalty is half a **BCLK** period
- when changing from **BCLK** to **FCLK**, the average resynchronization penalty is half an **FCLK** period.

Synchronous mode

You select this mode by tying **XSnA** HIGH. In this mode, there is a tightly defined relationship between **FCLK** and **BCLK**, in that **BCLK** can only make transitions on the falling edge of **FCLK**. Some jitter between the two clocks is permitted, but **BCLK** must meet the setup and hold requirements relative to **FCLK**. This is shown in Figure 9-3.

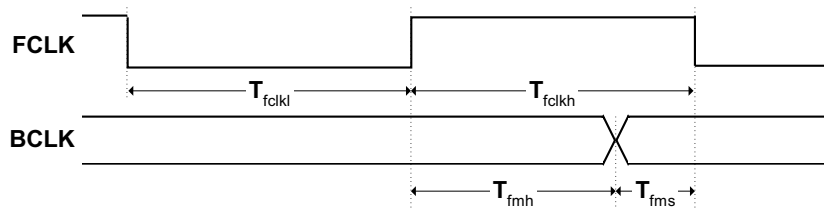


Figure 9-3 Relationship of FCLK and BCLK in synchronous mode

Chapter 10
AMBA Interface

This chapter describes the operation of the AMBA bus interface. It contains the following sections:

- *About the AMBA interface* on page 10-2
- *ASB bus interface signals* on page 10-3
- *Cycle types* on page 10-4
- *Addressing signals* on page 10-7
- *Memory request signals* on page 10-8
- *Data signal timing* on page 10-9
- *Slave response signals* on page 10-10
- *Maximum sequential length* on page 10-12
- *Read-lock-write* on page 10-13
- *Little-endian and big-endian operation* on page 10-14
- *Multi-master operation* on page 10-17
- *Bus master handover* on page 10-19
- *Default bus master* on page 10-21.

10.1 About the AMBA interface

In normal operation, the ARM720T is an *Advanced System Bus* (ASB) bus master. As a bus master it performs a subset of the possible ASB cycle types.

The ASB is further described in the *AMBA Specification.*

10.2 ASB bus interface signals

The signals in the ASB interface can be grouped into four categories:

Addressing	**BA[31:0]** **BWRITE** **BSIZE[1:0]** **BLOK** **BPROT[1:0]**.
Memory request	**BTRAN[1:0]**.
Data sampled	**BD[31:0]**.
Slave response	**BERROR** **BWAIT** **BLAST**.

In addition to the signals provided above, there are also three controls communicating with control logic in the system:

AGNT	Selects the ARM as a bus master.
AREQ	Indicates that the ARM720T requires bus mastership.
DSEL	Selects the ARM as a bus slave.

10.3 Cycle types

In normal operation, the ARM720T bus interface can perform two types of cycle:

- address cycles
- sequential cycles.

These cycles are differentiated by the pipelined signal **BTRAN[1:0]**. Conventionally, cycles are considered to start from the falling edge of **BCLK**, and this is how they are shown in all diagrams.

These cycle types are a subset of the possible ASB cycle types. Other cycle types can be forced by the use of the slave response signals. See the *AMBA Specification* for more details.

The addressing and memory request signals are pipelined ahead of the data addressing by a phase, half a cycle, and **BTRAN[1:0]** by one cycle. This advance information allows the implementation of efficient memory systems.

10.3.1 Single-word memory access

A simple single-word memory access is shown in Figure 10-1.

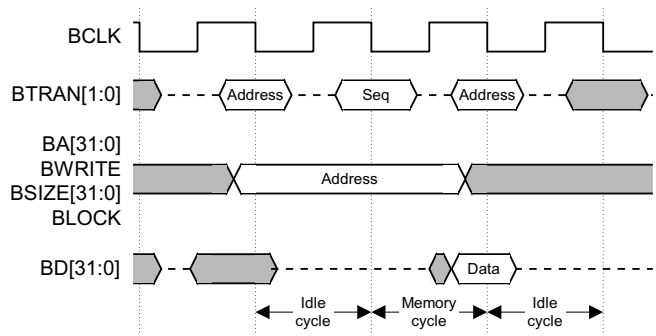


Figure 10-1 Simple single-cycle access

The access starts with the address being broadcast. You can be use this for decoding, but the access is not committed until **BTRAN[1:0]**, bus transaction type, signals a *sequential* cycle in the following HIGH phase of **BCLK**. This indicates that the next cycle is a memory access cycle.

In this example, **BTRAN[1:0]** returns to address after a single cycle, indicating that there is a single memory access cycle, followed by an address cycle. The data is transferred on the falling edge of **BCLK** at the end of the sequential cycle.

Therefore, a memory access consists of:

- an address cycle, with a valid address
- a memory cycle with the same address.

The initial address cycle allows the memory controller more time to decode the address. See Table 10-1 on page 10-8 for the encoding of **BTRAN[1:0]**.

10.3.2 Sequential accesses

ARM720T can perform sequential bursts of accesses. These consist of:

- an address cycle and a sequential cycle, as shown previously
- further sequential cycles to either:
  - — incrementing word addresses, that is, a, a+4, a+8 for example
  - — halfword addresses, that is, a, a+2, a+4 for example.

Figure 10-2 shows that after the initial address cycle, the address is pipelined by half a bus cycle from the data.

Note

BTRAN[1:0] is pipelined by a bus cycle from the data. If **BWAIT** is being used to stretch cycles, **BTRAN[1:0]** no longer refers to the next **BCLK** cycle, but rather to the next bus cycle. See *BWAIT* on page 10-10 for more information.

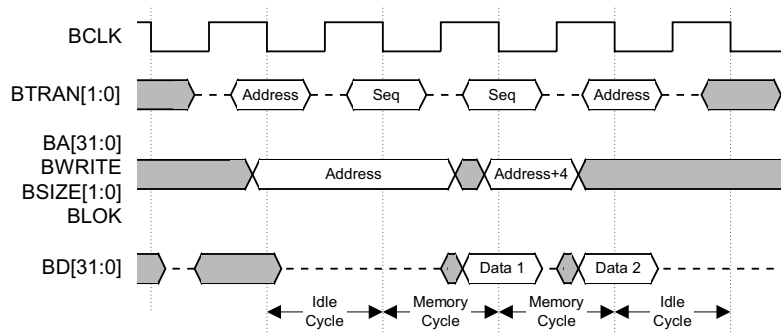


Figure 10-2 Simple sequential access

Sequential bursts can occur on word or halfword accesses, and are always in the same direction, that is, read, **BWRITE** LOW, or write, **BWRITE** HIGH.

A memory controller must always qualify the use of the address with **BTRAN[1:0]**. There are circumstances in which a new address can be broadcast on the address bus, but **BTRAN[1:0]** does not signal a *sequential* access. This only happens when an internal, protection unit generated, abort occurs.

10.3.3 Bus accesses

The minimum interval between bus accesses can occur after a buffered write. In this case, there might only be a single address cycle between two memory cycles to nonsequential addresses. This means that the address for the second access is broadcast on **BA[31:0]** during the HIGH phase of the final memory cycle of the buffered write. This is shown in Figure 10-3.

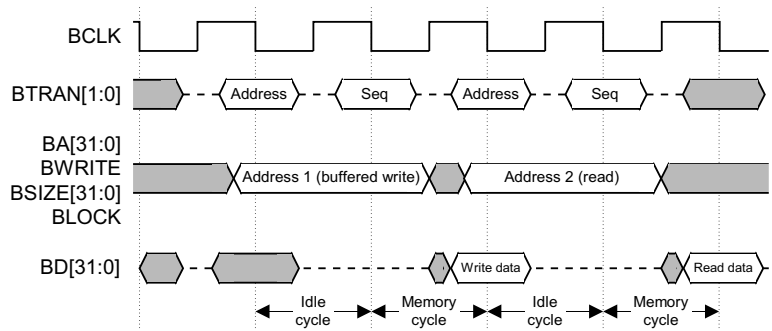


Figure 10-3 Minimum interval between bus accesses

This is the closest case of back-to-back cycles on the bus, and the memory controller must be designed to handle this case. In high-speed systems, one solution is to use **BWAIT** to increase the decode and access time available for the second access.

Note

Memory and peripheral strobes must not be direct decodes of the address bus. This can result in them changing during the last cycle of a write burst.

10.4 Addressing signals

Memory accesses can be read or write, and are differentiated by the signal **BWRITE**.

BWRITE cannot change during a sequential access, so if a read from address A is followed immediately by a write to address (A+4), the write to address (A+4) is performed on the bus as a nonsequential access.

In the same way, any memory access can be a word, a halfword, or a byte. These are differentiated by the signal **BSIZE[1:0]**. Again, **BSIZE[1:0]** can not change during sequential accesses. It is not possible to perform sequential byte accesses.

To reduce system power consumption, the addressing signals are left with their current values at the end of an access, until the next access occurs.

After a buffered write, there might be only a single address cycle between the two memory cycles. In this case, the next nonsequential address is broadcast in the last cycle of the previous access. This is the worst case for address decoding, as shown in Figure 10-3 on page 10-6.

10.5 Memory request signals

The memory request signals, **BTRAN[1:0],** are pipelined by one bus cycle, and refer to the next bus cycle.

Note

You must take care when depipelining these signals if **BWAIT** is being used, as they always refer to the following bus cycle, rather than the following **BCLK** cycle. **BWAIT** stretches the bus cycle by an integer number of **BCLK** cycles. See *BWAIT* on page 10-10. Table 10-1 lists **BTRAN[1:0]** encoding

Table 10-1 BTRAN[1:0] encoding

BTRAN[1:0]	Cycle type	Description	Remarks
00	Address	Address transfer or idle cycle	-
01	-	Reserved	-
10	Nonsequential	Nonsequential data transfer cycle	This cycle can only occur as a result of the slave response signals. In normal operation, ARM720T does not generate this cycle type.
11	Sequential	Sequential data transfer cycle	-

10.6 Data signal timing

During a read access, the data is sampled on the falling edge of **BCLK** at the end of the sequential cycle. During a write access, the data on **BD[31:0]** is timed off the falling edge of **BCLK** at the start of the memory cycle. If **BWAIT** is being used to stretch this cycle, the data is valid from the falling edge of **BCLK** at the end of the previous cycle, when **BWAIT** was HIGH. See *BWAIT* on page 10-10.

Note

In a low-power system, you must ensure that the databus is not allowed to float to an undefined level. This causes power to be dissipated in the inputs of devices connected to the bus. This is particularly important when a system is put into a low-power sleep mode. It is recommended that one set of databus drivers in the system are left enabled during sleep to hold the bus at a defined level.

10.7 Slave response signals

There are two main slave response signals:

- *BERROR*
- *BWAIT*

Other slave response combinations, including bus last and bus retract, are described in the *AMBA Specification*.

10.7.1 BERROR

The **BERROR** signal is sampled on the rising edge of **BCLK** during a sequential cycle, on both read and write accesses. The effect of **BERROR** on the operation of the ARM720T is described in *Exceptions* on page 2-16.

BERROR can be flagged on any sequential cycle. However, it is ignored on buffered writes, which cannot be aborted.

Linefetches

The effect of **BERROR** during linefetches is slightly different to that during other access. During a linefetch the ARM720T fetches four words of data, regardless of which words of data were requested by the ARM core, and the rest of the words are fetched speculatively:

- if **BERROR** is asserted on a word that was requested by the ARM core, the abort functions normally
- if the abort is signaled on a word that was not requested by the ARM core, the access is not aborted, and program flow is not interrupted.

Regardless of which word was aborted, the line of data is not placed in the cache as it is assumed to contain invalid data.

10.7.2 BWAIT

You can use the **BWAIT** pin to extend memory accesses in whole cycle increments.

BWAIT is driven by the selected slave during the LOW phase of **BCLK**. When a slave cannot complete an access in the current cycle, it drives **BWAIT** HIGH to stall the ARM720T.

BWAIT does not prevent changes in **BTRAN[1:0]** and write data on **BD[31:0]** during the cycle in which it was asserted HIGH. Changes in these signals are then prevented until the **BCLK** HIGH phase after **BWAIT** was taken LOW. The addressing signals do not change from the rising **BCLK** edge when **BWAIT** goes HIGH, until the next **BCLK** HIGH phase after **BWAIT** returns LOW.

In Figure 10-4, the heavy bars indicate the cycle for which signals are stable as a result of asserting **BWAIT**.

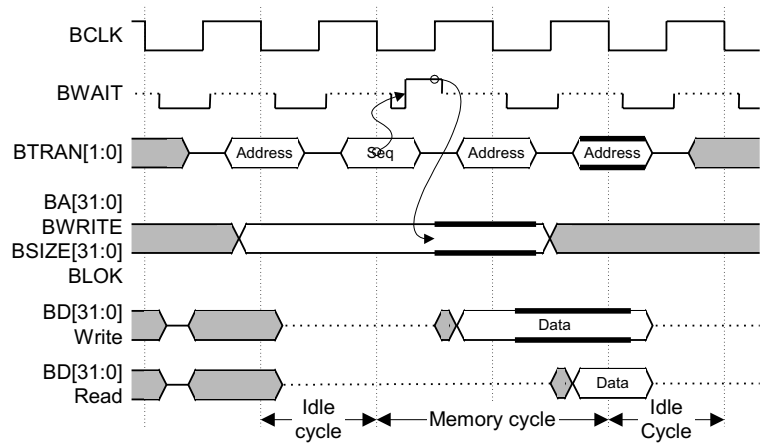


Figure 10-4 Use of the BWAIT pin to stop ARM720T for 1 BCLK cycle

The signal **BTRAN[1:0]** is pipelined by one bus cycle. This pipelining must be taken into account when these signals are being decoded. The value of **BTRAN[1:0]** indicates whether the next bus cycle is a data cycle or an address cycle.

As bus cycles are stretched by **BWAIT**, the boundary between bus cycles is determined by the falling edge of **BCLK** when **BWAIT** was sampled as LOW on the rising edge of **BCLK**. A useful general rule is to sample the value of **BTRAN[1:0]** on the *falling* edge of **BCLK** only when **BWAIT** was LOW on the previous *rising* edge of **BCLK**.

When **BWAIT** is used to stretch a sequential cycle, **BTRAN[1:0]** returns to signaling address during the first phase of the sequential cycle if a single word access is occurring. In this case, it is important that the memory controller does not interpret that an address cycle is signaled when it is a stretched memory cycle.

10.8 Maximum sequential length

The ARM720T can perform sequential memory accesses whenever the cycle is of the same type as the previous cycle (for example, read/write), and the addresses are consecutive. However, sequential accesses are interrupted on a 256-word boundary.

If a sequential access is performed over a 256-word boundary, the access to word 256 is turned into a nonsequential access, and further accesses continue sequentially as before.

This simplifies the design of the memory controller. Provided that peripherals and areas of memory are aligned to 256-word boundaries, sequential bursts are always local to one peripheral or memory device. This means that all accesses to a device always start with a nonsequential access.

A DRAM controller can take advantage of the fact that sequential cycles are always within a DRAM page, provided the page size is greater than 256.

10.9 Read-lock-write

The read-lock-write sequence is generated by a `SWP` instruction.

The **BLOK** signal indicates that the two accesses must be treated as an atomic unit. A memory controller must ensure that no other bus activity is allowed to happen between the accesses when **BLOK** is asserted. When the ARM720T has started a read-lock-write sequence, it cannot be interrupted until it has completed.

On the bus, the sequence consists of:

- a read access
- a write access to the same address.

This sequence is differentiated by the **BLOK** signal. **BLOK**:

- goes HIGH in the HIGH phase of **BCLK** at the start of the read access
- always goes LOW at the end of the write access.

The read cycle is always performed as a single, nonsequential, external read cycle, regardless of the contents of the cache.

The write is forced to be unbuffered, so that it can be aborted if necessary.

The cache is updated on the write.

10.10 Little-endian and big-endian operation

The ARM720T treats words in memory as being stored in big-endian or little-endian format depending on the value of the bigend bit in the control register (see *Memory formats* on page 2-3).

Load and store are the only instructions affected by the endianness. Refer to the *ARM Architecture Reference Manual* for details of the `LDR` and `STR` instructions.

10.10.1 Little-endian format

In little-endian format:

- the lowest-numbered byte in a word is considered to be the least significant
- the highest-numbered byte is the most significant.

Byte zero of the memory system must be connected to data lines seven to zero (**BD[7:0]**) in this format.

Little-endian format is shown in Figure 10-5.

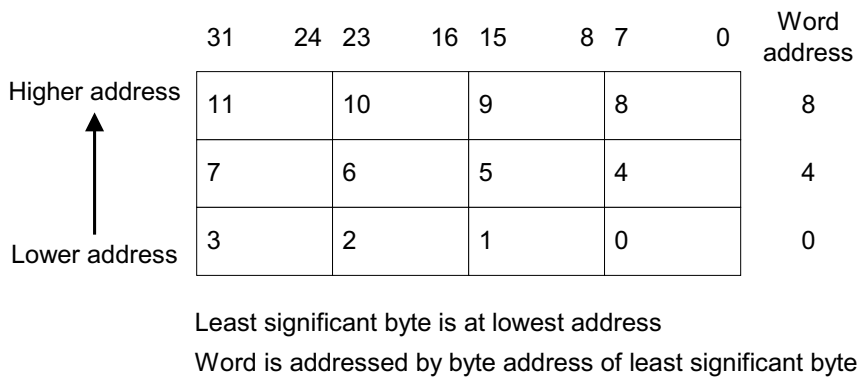


Figure 10-5 Little-endian addresses of bytes within words

10.10.2 Big-endian format

In big-endian format:

- the most significant byte of a word is stored at the lowest-numbered byte
- the least significant byte is stored at the highest-numbered byte.

Byte zero of the memory system must therefore be connected to data lines 31 to 24 (**BD[31:24]**).

Big-endian format is shown in Figure 10-6.

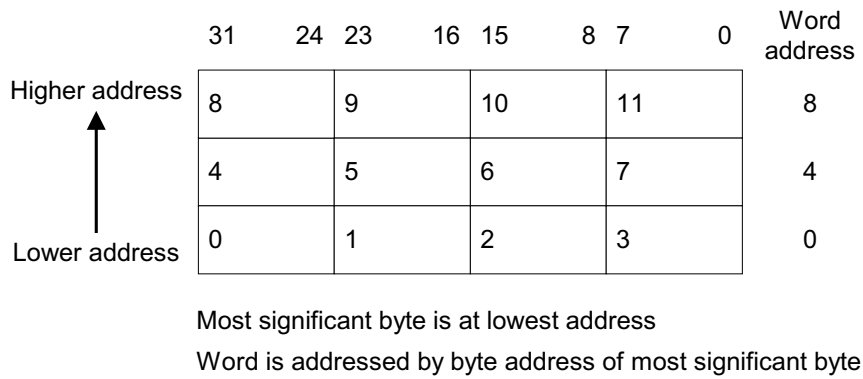


Figure 10-6 Big-endian addresses of bytes within words

10.10.3 Word operations

All word operations expect the data to be presented on data bus inputs 31 to 0. The external memory system ignores the bottom two bits of the address if a word operation is indicated.

10.10.4 Halfword operations

A halfword store, `STRH` repeats the bottom 16 bits of the source register twice across data bus outputs 31 to 0. The external memory system must activate the appropriate byte subsystems to store the data.

Little-endian operation

A halfword load, `LDRH`, expects the data on data bus inputs 15 to 0 if the supplied address is on a word boundary, or on data bus inputs 31 to 16 if it is a word address plus two bytes. The selected halfword is placed in the bottom 16 bits of the destination register. The other two bytes on the databus are ignored (see Figure 10-5 on page 10-14).

Big-endian operation

A halfword load, `LDRH`, expects the data on data bus inputs 31 to 16 if the supplied address is on a word boundary, or on data bus inputs 15 to 0 if it is a word address plus two bytes. The selected halfword is placed in the bottom 16 bits of the destination register. The other two bytes on the databus are ignored, see Figure 10-6 on page 10-15.

10.10.5 Byte operations

A byte store, STRB, repeats the bottom eight bits of the source register four times across data bus outputs 31 to 0. The external memory system activates the appropriate byte subsystem to store the data.

Because ARM720T duplicates the byte to be written across the databus and internally rotates bytes after reading them from the databus, a 32-bit memory system only requires to have control logic to enable the appropriate byte. You do not have to rotate or shift the data externally.

To ensure that all of the databus is driven during a byte read, it is valid to read a word back from the memory.

Little-endian operation

A byte load, LDRB, expects the data on data bus inputs seven to zero if the supplied address is on a word boundary, on data bus inputs 15 to 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom eight bits of the destination register. The other three bytes on the databus are ignored (see Figure 10-5 on page 10-14).

Big-endian operation

A byte load, LDRB, expects the data on data bus inputs 31 to 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom eight bits of the destination register. The other three bytes on the databus are ignored (see Figure 10-6 on page 10-15).

10.11 Multi-master operation

The AMBA bus specification supports multiple bus masters on the high performance ASB. A simple two wire request and grant mechanism is implemented between the arbiter and each bus master. The arbiter ensures that only one bus master is active on the bus and also ensures that when no masters are requesting the bus, a default master is granted.

The specification also supports a shared lock signal. This allows bus masters to indicate that the current transfer is indivisible from the following transfer and prevents other bus masters from gaining access to the bus until the locked transfers have completed.

10.11.1 Arbitration

Efficient arbitration is important to reduce *dead-time* between successive masters being active on the bus. The bus protocol supports pipelined arbitration, so that arbitration for the next transfer is performed during the current transfer.

The arbitration protocol is defined, but the prioritization is flexible and left to the application. Typically, the test interface is given the highest priority to ensure test access under all conditions. Every system must also include a default bus master, which is granted the bus when no bus masters are requesting it.

The request signal, **AREQ**, from each bus master to the arbiter indicates that the bus master requires the bus. The grant signal from the arbiter to the bus master, **AGNT**, indicates that the bus master is currently the highest priority master requesting the bus.

The bus master:

- must drive the **BTRAN** signals during **BCLK** HIGH when **AGNT** is HIGH
- is granted when **AGNT** is HIGH and **BWAIT** is LOW on a rising edge of **BCLK**.

The shared bus lock signal, **BLOK**, indicates to the arbiter that the following transfer is indivisible from the current transfer and no other bus master can be given access to the bus.

A bus master must always drive a valid level on the **BLOK** signal when granted the bus to ensure the arbitration process can continue, even if the bus master is not performing any transfers.

The arbiter functions are:

1. Bus masters assert **AREQ** during the HIGH phase of **BCLK**.
2. The arbiter samples all **AREQ** signals on the falling edge of **BCLK**.

3. During the LOW phase of **BCLK**, the arbiter also samples the **BLOK** signal and then asserts the appropriate **AGNT** signal:
   a. If **BLOK** is LOW, the arbiter grants the highest priority bus master
   b. If **BLOK** is HIGH, the arbiter keeps the same bus master granted.

The arbiter can update the grant signals every bus cycle. However, a new bus master can only become granted and start driving the bus when the current transfer completes, as indicated by **BWAIT** being LOW. Therefore, it is possible for the potential next bus master to change during waited transfers.

The **BLOK** signal is ignored by the arbiter during the single cycle of handover between two different bus masters.

If no bus masters are requesting the bus, the arbiter must grant the default bus master. The arbitration protocol is defined, but the prioritization is flexible and left to the application. A simple fixed-priority scheme can be used. Alternatively, a more complex scheme can be implemented if required by the application.

10.12 Bus master handover

Bus master handover occurs when a bus master, which is not currently granted the bus, becomes the new granted bus master.

A bus master becomes granted when **AGNT** is HIGH and **BWAIT** is LOW. **AGNT** HIGH indicates the bus master is currently the highest priority master requesting the bus and **BWAIT** LOW indicates the previous transfer has completed. Figure 10-7 shows the bus master handover process.

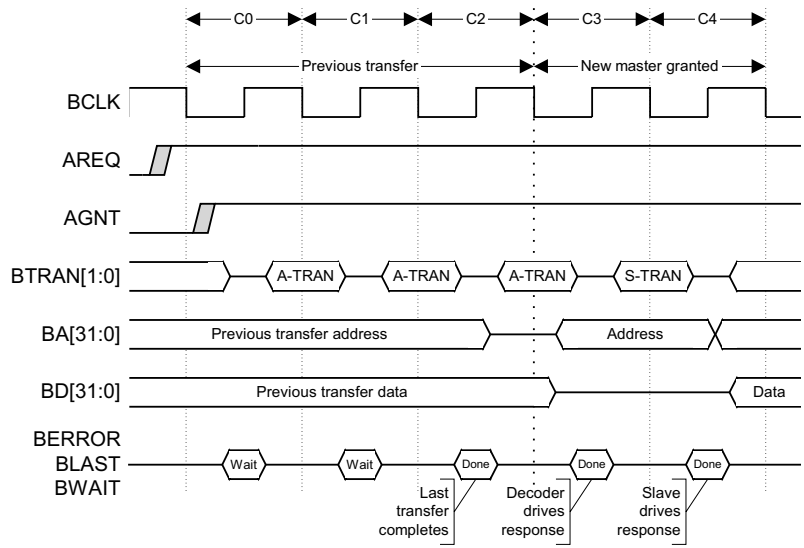


Figure 10-7 Bus master handover

1. When **AGNT** is asserted, a bus master must drive the **BTRAN** signals during **BCLK** HIGH. This can continue for many cycles if the previous transfer is waited.
2. Prior to handover, **BTRAN** must indicate an address-only cycle as the new bus master must commence with an address-only cycle to allow for bus turnaround.
3. When the previous transfer completes, the new bus master is granted.
4. In the last clock HIGH phase of the previous transfer, the address bus stops being driven by the previous bus master.
5. The new bus master starts to drive the address bus and control signals during the clock LOW phase.

6. The first transfer can then commence in the following bus cycle.

During a waited transfer, bus master handover can be delayed and it is possible that the **AGNT** to a particular bus master might be asserted and then negated, if another higher priority bus master then requests the bus before the current transfer has completed.

10.13 Default bus master

Every system must be designed with a single default bus master, which is granted when no other bus master is requesting the bus. The default bus master is responsible for driving the following signals to ensure the bus remains active:

- **BTRAN** must be driven to indicate address-only transfer
- **BLOK** must be driven LOW.

Note

If the ARM720T is to be the default bus master then the **AREQ** signal from the ARM720T must not be used.

Chapter 11
AMBA Test

This chapter describes the AMBA test features of the ARM720T. It contains the following sections:

- *Slave operation, test mode* on page 11-2
- *ARM720T test mode* on page 11-3
- *ARM7TDM core test mode* on page 11-5
- *RAM test mode* on page 11-6
- *TAG test mode* on page 11-8
- *Test register mapping* on page 11-11.

11.1 Slave operation, test mode

When the block is selected as a slave, you can write and read test vectors to the core using the AMBA test methodology.

The ARM720T provides four test modes for this purpose:

- *ARM720T test mode* on page 11-3
- *ARM7TDM core test mode* on page 11-5
- *RAM test mode* on page 11-6
- *TAG test mode* on page 11-8.

To apply test vectors to the ARM720T, the ARM720T block must have been deselected as a master (**AGNT** goes LOW). The *Test Interface Controller* (TIC) becomes the bus master, and the ARM720T is selected as a slave using the signal **DSEL**. This places the ARM720T into test mode, and allows access to the test registers.

The tests are sequenced by the test state machine in the AMBA interface. This generates the appropriate control signals for the test modes.

A sample test sequence is shown in Figure 11-1.

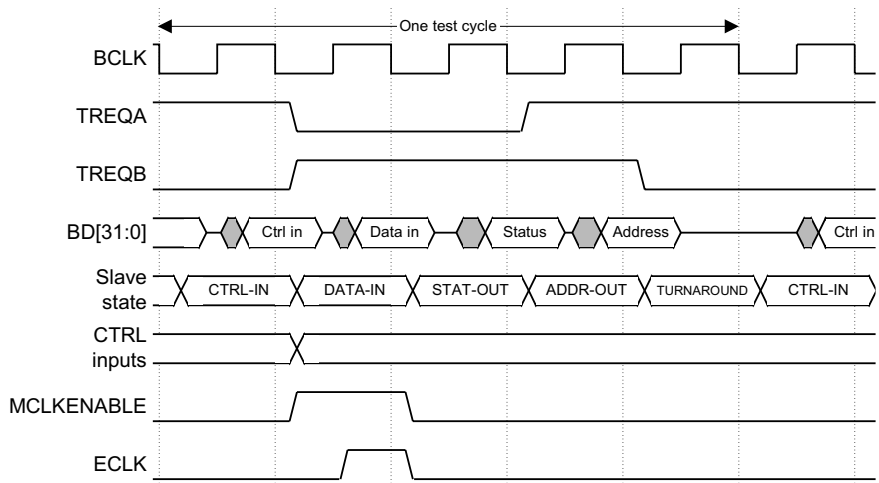


Figure 11-1 Running a test vector on the processor core

11.2 ARM720T test mode

The ARM720T test mode tests the functionality of the:

- cache control logic
- write buffer
- protection unit
- cache.

To perform this test control/stimuli are applied to the control register (see Table 11-4 on page 11-13).

Data packets are read or written as appropriate and the address and status are read back (see Table 11-3 on page 11-11).

The sequencing for this test mode is shown in Figure 11-2. This is the default test mode, and is selected when bits [31:29] of the control register are set LOW (see Table 11-4 on page 11-13).

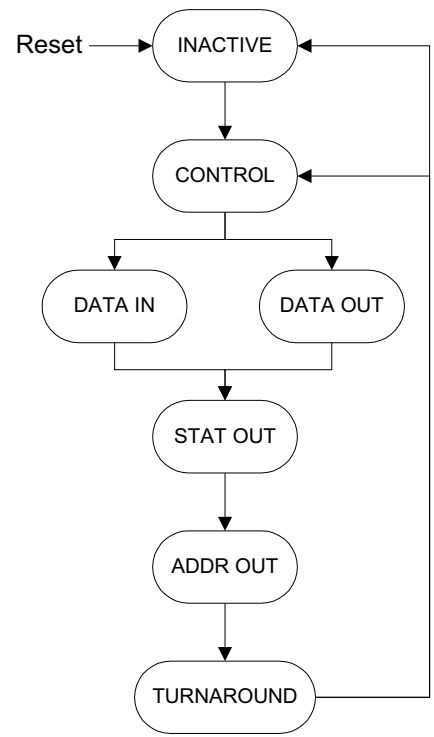


Figure 11-2 State machine for ARM720T and ARM7TDMI test

MCLKENABLE is an internal signal that controls the clocking of the ARM720T and is asserted only in the DataIn and DataOut status.

11.3 ARM7TDM core test mode

The ARM7TDMI test places the ARM720T into a test mode so that the signals of the ARM7TDM are visible to the AMBA interface. In this mode, the rest of ARM720T is held in reset. The ARM720T is placed in the mode by setting bit 31 of the control register (see Table 11-4 on page 11-13).

11.4 RAM test mode

The RAM test mode performs an intensive test of the RAM arrays, to provide full coverage of bit faults. In this test mode, the rest of the ARM720T is held in reset and direct access is provided to the data, address, and control signals of the RAM.

To accommodate this, an alternative test sequence is used as shown in Figure 11-3.

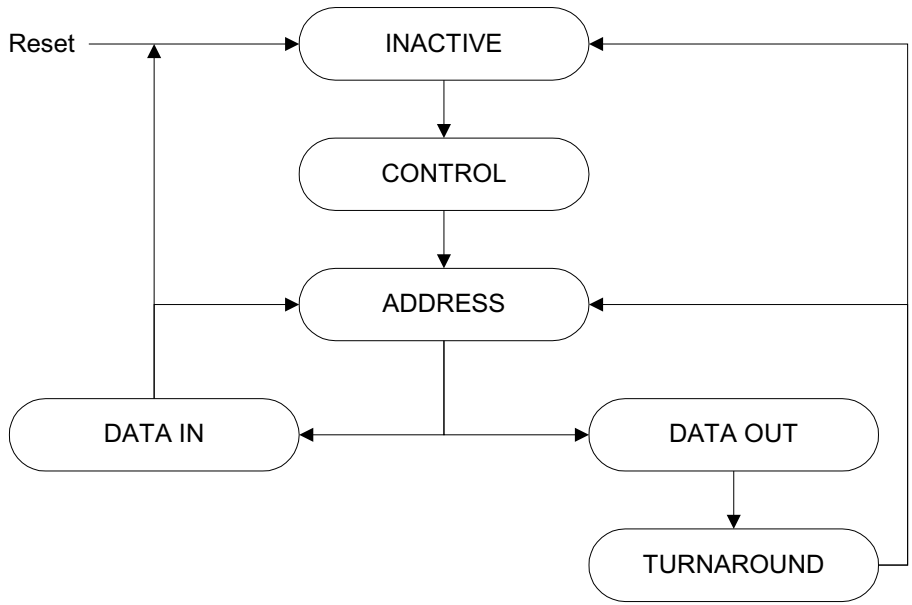


Figure 11-3 .State machine for RAM test mode

In this test mode, the RAM control signals are derived from unused address bits, as shown in Table 11-1.

Table 11-1 RAM test mode address packet bit positions

Address packet bit	RAM signal	Description
[24:23]	**MAS[1:0]**	RAM access size
22	**RSEQ**	RAM sequential signal
21	**IMMED**	Immediate write signal, controls write pipeline, and selects between **RAMSEL[3:0]** and **SETSEL[3:0]**

Table 11-1 RAM test mode address packet bit positions

Address packet bit	RAM signal	Description
20	**WRITE**	RAM write strobe
19	**READ**	RAM read strobe
[18:15]	**RAMSEL[3:0]**	RAM bank select signal, used when **IMMED** is LOW
[14:11]	**SETSEL[3:0]**	RAM bank select signal, used when **IMMED** is HIGH
[10:0]	**ADDR[10:0]**	RAM address

To enter RAM test mode, bits 29 and 28 of the control packet must be set. This places the ARM720T into RAM test mode, and forces the RAM to be clocked from the **FCLK** input.

11.5 TAG test mode

The TAG test mode performs an intensive test of all of the cells of the TAG array, and tests the TAG comparators. In this test mode, the rest of the ARM720T is held in reset and direct access is provided to the data, address, and control signals of the RAM as shown in Figure 11-4. In this test mode the TAG control signals are derived from the TAG CTL packet as listed in Table 11-2 on page 11-9.

To enter TAG test mode, you must set bits 30 and 28 of the control packet. This places the ARM720T into TAG test mode, and forces the TAG to be clocked from the **FCLK** input.

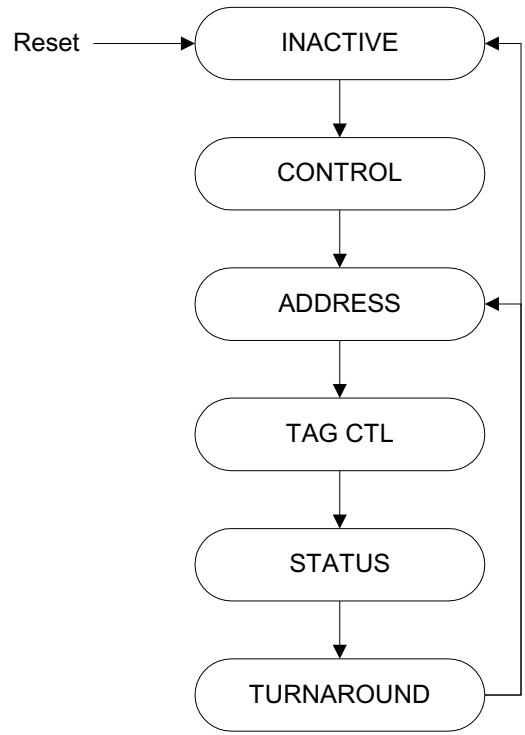


Figure 11-4 State machine for TAG test mode

Table 11-2 TAG test mode TAG CTL packet bit positions

TAG CTL packet bit	TAG signal	Description
[11:8]	**FLUSH[3:0]**	When asserted each bit flushes the appropriate TAG arrays
[7:4]	**TAGSEL[3:0]**	Tag select signal, each bit selects a TAG array
2	**WRITE**	TAG write strobe
1	**READ**	TAG read strobe
0	**VALID**	Valid input, the value on **VALID** is written into the valid cell in the array on a write.

11.6 MMU test mode

The MMU test mode performs an intensive test of all the cells in the TLB array, and tests the protection mechanism. In this test mode the rest of the ARM720T is held in reset and direct access is provided to the data, control, and translated address of the MMU as shown in Figure 11-5.

In this test mode, the MMU control signals are derived from the MMU CM packet.

To enter MMU test mode, you must set bits 28 and 27 of the control packet. This places the ARM720T into MMU test mode and forces the MMU to be clocked from the **FCLK** input.

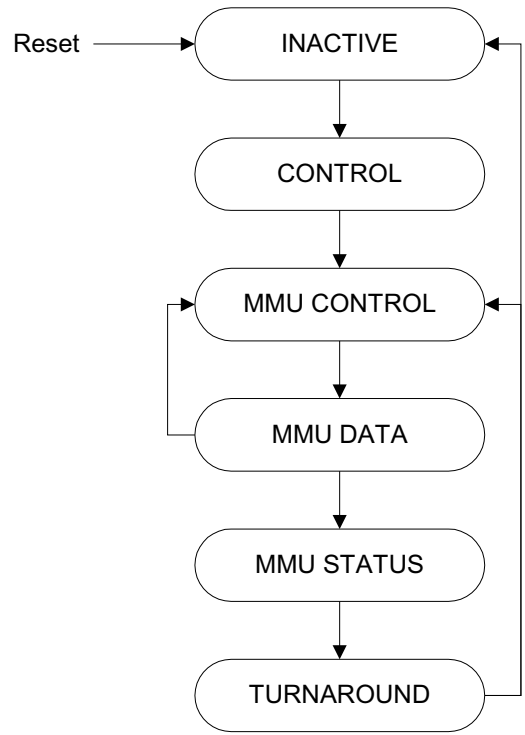


Figure 11-5 State machine for MMU test mode

11.7 Test register mapping

The test registers are defined in the following tables:

- Table 11-3
- Table 11-4 on page 11-13.

Table 11-3 Status packet bit positions bits [31:0]

Bit	ARM7TDMI test	ARM720T test
31	**BUSDIS** Bus disable	-
30	**SCREG[3]** Scan chain register	**SCREG[3]** Scan chain register
29	**SCREG[2]** Scan chain register	**SCREG[2]** Scan chain register
28	**SCREG[1]** Scan chain register	**SCREG[1]** Scan chain register
27	**SCREG[0]** Scan chain register	**SCREG[0]** Scan chain register
26	**HIGHZ** HIGHZ instruction in TAP controller	**HIGHZ** HIGHZ instruction in TAP controller
25	**nTDOEN** not **TDO** enable	**nTDOEN** not **XTDO** enable
24	**DBGRQI** Internal debug request	**DBGRQI** Internal debug request
23	**RANGEOUT0** ICEbreaker rangeout0	**RANGEOUT0** ICEbreaker rangeout0
22	**RANGEOUT1** ICEbreaker rangeout1	**RANGEOUT1** ICEbreaker rangeout1
21	**COMMRX** Communications channel receive	**COMMRX** Communications channel receive
20	**COMMTX** Communications channel transmit	**COMMTX** Communications channel transmit
19	**DBGACK** Debug acknowledge	**DBGACK** Debug acknowledge

Table 11-3 Status packet bit positions bits [31:0] (continued)

Bit	ARM7TDMI test	ARM720T test
18	**TDO** Test data out	**XTDO** Test data out
17	**nENOUT**[a] Not enable output	**nENOUT** Not enable output
16	**nENOUTI**[b] Not enable output	**PROTWATCH[3]** Protection unit test output
15	**TBIT** Thumb state	**PROTWATCH[2]** Protection unit test output
14	**nCPI** Not coprocessor instruction	-
13	**nM[4]** Not processor mode	**CAMWATCH[2]** Replacement test output
12	**nM[3]** Not processor mode	**CAMWATCH[1]** Replacement test output
11	**nM[2]** Not processor mode	**CAMWATCH[0]** Replacement test output
10	**nM[1]** Not processor mode	**IDCWATCH[3]** Cache test output
9	**nM[0]** Not processor mode	**IDCWATCH[2]** Cache test output
8	**nTRANS** Not memory translate	**IDCWATCH[1]** Cache test output
7	**nEXEC** Not executed	**IDCWATCH[0]** Cache test output
6	**LOCK** Locked operation	**LOCK** Locked operation
5	**MAS[1]** Memory access size	**MAS[1]** Memory access size
4	**MAS[0]** Memory access size	**MAS[0]** Memory access size
3	**nOPC** Not op-code fetch	**nENDOUT** Not enable output

Table 11-3 Status packet bit positions bits [31:0] (continued)

Bit	ARM7TDMI test	ARM720T test
2	**nRW** Not read/write	**nRW** Not read/write
1	**nMREQ** Not memory request	**nMREQ** Not memory request
0	**SEQ** Sequential address	**SEQ** Sequential address

a.**nENOUT** is only valid during the data access cycle, so **MCLKENABLE** is used to clock a transparent latch that captures the correct state.

b.**nENOUTI** as **nENOUT.**

Table 11-4 Control packet bit positions bits [31:0]

Bit	ARM7TDMI input	ARM720T input
31	**TESTCPU** ARM7TDMI test enable	**TESTCPU** ARM7TDMI test enable
30	-	**TAGTEST** TAG test mode enable
29	-	**RAMTEST** RAM test mode enable
28	**nENIN** NOT enable input	**FORCEFCLK** Clock select override
27	**SDOUTBS** Boundary scan serial output data	**MMUTEST** MMU test mode enable
26	**TBE** Test bus enable	-
25	**APE** Address pipeline enable	-
24	**BL[3]**[a] Byte latch control	-
23	**BL[2]**[a] Byte latch control	-

Table 11-4 Control packet bit positions bits [31:0] (continued)

Bit	ARM7TDMI input	ARM720T input
22	**BL[1]**[a] Byte latch control	-
21	**BL[0]**[a] Byte latch control	-
20	**TMS** Test mode select	**XTMS** Test mode select
19	**TDI** Test data in	**XTDI** Test data in
18	**TCK**[b] Test clock	**XTCK** Test clock
17	**nTRST** Not test reset.	**XnTRST** Not test reset
16	**EXTERN1** External input 1	**EXTERN1** External input 1
15	**EXTERN0** External input 0	**EXTERN0** External input 0
14	**DBGRQ** Debug request	**DBGRQ** Debug request
13	**BREAKPT** Breakpoint	**BREAKPOINT** Breakpoint
12	**DBGEN** Debug enable	**DBGEN** Debug enable
11	**ISYNC** Synchronous interrupts	-
10	**BIGEND** Big Endian configuration	**WINCE EN** WinCe enhancements enable
9	**CPA** Coprocessor absent	**CPA** Coprocessor absent
8	**CPB** Coprocessor busy	**CPB** Coprocessor busy
7	**ABE**[c] Address bus enable	**XSnA** Clock configuration

Table 11-4 Control packet bit positions bits [31:0] (continued)

Bit	ARM7TDMI input	ARM720T input
6	**ALE** Address latch enable	**ALE** Address latch enable
5	**DBE**[d] Data bus enable	**XFASTBUS** Clock configuration
4	**nFIQ** Not fast interrupt request.	**nFIQ** Not fast interrupt request
3	**nIRQ** Not interrupt request	**nIRQ** Not interrupt request
2	**ABORT** Memory abort	**ABORT** Memory abort
1	**nWAIT**[e] Not wait	**nWAIT** Not wait
0	**nRESET** Not reset	**nRESET** Not reset

a.ANDed with **MCLKENABLE**, so is only valid during data access cycle.

b.ANDed with **MCLKENABLE** and **BCLK.**

c.This must normally be set HIGH, because if the bus is tristated, with **ABE** LOW, then it is not possible to read address values.

d.**DBE** to the ARM7DMT is ANDed with the state machine generated **DBE** and **BCLK** to prevent bus conflict

e.ANDed with **MCLKENABLE**, so that the core state can only change during the data access cycle.

Chapter 12
Trace Interface Port

This chapter describes the ETM support for the ARM720T. It contains the following sections.

- *About the ETM* on page 12-2
- *ETM interface* on page 12-3.

12.1 About the ETM

The ETM provides instruction and data trace for the ARM family of processors.

The ETM comprises two parts:

A trace port A trace protocol has been developed to provide a real time trace capability for ARM processor cores that are embedded in large *Application-Specific Integrated Circuits* (ASICs). Because the ASIC usually contains on-chip memory and other circuitry, it is not possible for you to determine processor core operation by observing the ASIC pins. The trace port is necessary for you to understand processor operation.

Triggering facilities
A specification has been developed that allows you to specify the exact set of trigger resources necessary for a particular application. Resources include address and data comparators, counters, and sequencers.

A software debugger provides you with the interface to the ETM. The debugger allows all of the ETM facilities to be configured through a JTAG interface. If a trace port has been implemented then the debugger displays the captured trace information in an easily understandable format.

You can use the JTAG interface for other debugging functions, such as downloading code or single-stepping through a program.

The ETM compresses the trace information and exports it through the trace port. An external *Trace Port Analyzer* (TPA) is used to capture the trace information.

When you have captured the trace then the debugger extracts compressed information from the TPA and decompresses it to provide a full disassembly of the executed code. The debugger can also link this to the original high level code to provide you with information on how the code was executed on the target system.

12.2 ETM interface

The ARM720T trace interface port enables connection of an *ARM7 ETM* (ETM7) Rev 1 to an ARM720T Rev 3. This interface does not exist on ARM720T Rev 0 to Rev 2. The ETM7 provides instruction and data trace for the ARM7 family of processors.

The interface is made up as follows:

ETMCLK Is a clock signal output from the ARM720T to use in the ETM7 to provide synchronization with the clock in the ARM720T core. The internal clock signal used is **CPCLK** which is inverted to form the **ETMCLK** output.

ETMCLKEN **ETMCLK** is gated when it enters the ETM7 by exporting another signal (**ETMCLKEN**) from the ARM720T. This signal is based on the **CPnWAIT** signal.

ETM<signal> Outputs to the ETM7.

The ETM7 is reset by **XnTRST**, no extra signal is used to achieve this.

The **ETMCLK** output is used by the ETM7 to register the **ETM<signal>** outputs on the rising edge of **ETMCLK**.

The ETM interface (**ETM<signal>**) timings are shown in Figure 12-1. These signals all change in the low phase of **ETMCLK**.

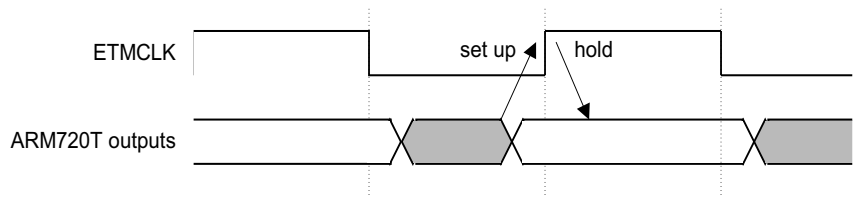


Figure 12-1 ETM interface signal timing

The ARM720T **ETM<signal>** descriptions are provided in *Embedded trace macrocell interface signals* on page A-10.

The *ETM7 Technical Reference Manual* describes how to integrate an ETM7 with the ARM7 family of processors.

12.2.1 ETMCLK gating for power saving

For lowest power operation, it is essential that the clock provided to the ETM7 is gated off when the ETM7 is powered down. You must insert a clock gate between ARM720T and ETM7 for this purpose. This is shown in Figure 12-2.

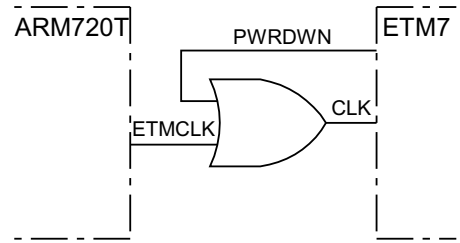


Figure 12-2 ETMCLK power saving

Note

You must take care during implementation to minimize the delay caused by insertion of this gate.

Appendix A
Signal Descriptions

This chapter describes the interface signals of the ARM720T. It contains the following sections:

- *AMBA interface signals* on page A-2
- *Coprocessor interface signals* on page A-5
- *JTAG signals* on page A-7
- *Debugger signals* on page A-9
- *Embedded trace macrocell interface signals* on page A-10
- *Miscellaneous signals* on page A-12.

A.1 AMBA interface signals

The AMBA interface signals are listed in Table A-1.

Table A-1 AMBA signal descriptions

Name	Type	Source or destination	Description
AGNT	In	Arbiter	Access grant. This signal from the bus arbiter indicates that the ARM720T is currently the highest priority master requesting the bus. If **AGNT** is asserted at the end of a transfer (**BWAIT** LOW), the master is granted the bus. **AGNT** changes during the LOW phase of **BCLK** and remains valid through the high phase.
AREQ	Out	Arbiter	Access request. This signal indicates that the master requires the bus. It changes during the HIGH phase of **BCLK**. This signal is intended for use where the ARM720T is not the lowest priority or default bus master.
BA[31:0]	Out	Current bus master	Bus address. This is the system address bus.
BCLK	In		System (bus) clock. This clock times all bus transfers.
BD[31:0]	In/out	Bus master	Bidirectional system data bus. This data bus is driven by the current bus master during write cycles, and by the appropriate bus slave during read cycles.
BERROR	In/out	System decoder and current bus master	Bus error. This signal indicates a transfer error 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.

Table A-1 AMBA signal descriptions (continued)

Name	Type	Source or destination	Description
BLAST	In/out	System decoder and current bus master	Bus class. This signal is driven by the selected slave to indicate if the current transfer must be the last of a burst sequence. When **BLAST** is HIGH, the next bus transfer must allow sufficient time for address decoding. When **BLAST** is LOW, the next transfer can continue as a burst sequence. This signal is also used in combination with the **BERROR** signal to indicate a bus retract operation.
BLOK	Out	Arbiter	Bus lock. When HIGH, this signal indicates that the following bus transfer is to be indivisible and no other bus master must be given access to the bus.
BnRES	In	Reset state machine	Bus reset. This signal indicates the reset status of the bus.
BPROT[1:0]	Out	System decoder	Bus protections. These signals provide additional information about the transfer being performed. All write cycles are indicated as being Supervisor accesses. These signals have the same timing as the **BA** signals.
BSIZE[1:0]	Out	Current bus master	Bus size. These signals indicate the size of the transfer, which can be byte, halfword, or word. These signals have the same timing as the address bus.
BTRAN[1:0]	Out	Bus master	Bus transaction type. These signals indicate the type of the next transaction which can be address-only, nonsequential, or sequential. These signals are driven when **AGNT** is asserted, and are valid during the HIGH phase of **BCLK** before the transfer to which they refer.

Table A-1 AMBA signal descriptions (continued)

Name	Type	Source or destination	Description
BWAIT	In/out	System decoder and current bus master	Bus wait. This signal is driven by the selected slave to indicate if the current transfer can complete. If **BWAIT** is HIGH, a further bus cycle is required. If **BWAIT** is LOW, the current transfer can complete in the current bus cycle.
BWRITE	In/out	Current bus master	Bus write. When HIGH, this signal indicates a bus write cycle and when LOW, a read cycle. This signal has the same timing as the address bus.
DSEL	In	System decoder	Slave select. This signal puts the ARM core into a test mode so that vectors can be written in and out of the core.

A.2 Coprocessor interface signals

The coprocessor interface signals are listed in Table A-2.

Table A-2 Coprocessor interface signal descriptions

Name	Type	Description
CPCLK	Out	Coprocessor clock. This clock controls the operation of the coprocessor interface.
CPDATA[31:0]	In/out	Coprocessor data bus. Data is transferred to and from the coprocessor using this bus. Data is valid on the falling edge of **CPCLK**.
CPDBE	In	Coprocessor data bus enable. This signal when HIGH, indicates that the coprocessor intends to drive the coprocessor data bus, **CPDATA.** If the coprocessor interface is not to be used then this signal must be tied LOW.
CPnWAIT	Out	Coprocessor not wait. The coprocessor clock **CPCLK** is qualified by **CPnWAIT** to allow the ARM720T to control the transfer of data on the coprocessor interface.
CPTESTREAD	In	Coprocessor test read. This signal can be used for test of a coprocessor, if attached, and must only be used with the ARM720T held in reset. When HIGH, it enables *Data Bus* (DB) to be driven on to **CPDATA**, and must be held LOW. It must never be asserted at the same time as **CPTESTWRITE**.
CPTESTWRITE	In	Coprocessor test write. This signal can be used for test of a coprocessor, if attached, and must only be used with the ARM720T held in reset. When HIGH, it enables DB to be driven on to **CPDATA**, and must be held LOW. It must never be asserted at the same time as **CPTESTREAD**.
EXTCPA	In	External coprocessor absent. A coprocessor that is capable of performing the operation that ARM720T is requesting, by asserting **nCPI** takes **EXTCPA** LOW immediately. If **EXTCPA** is HIGH at the end of the low phase of the cycle in which **nCPI** went LOW, ARM720T aborts the coprocessor instruction and takes the undefined instruction trap. If **EXTCPA** is LOW and remains LOW, ARM720T busy-waits until **EXTCPB** is LOW, and then completes the coprocessor instruction.
EXTCPB	In	External coprocessor busy. A coprocessor that is capable of performing the operation that ARM720T is requesting, by asserting **nCPI**, but cannot commit to starting it immediately, indicates this by driving **EXTCPB** HIGH. When the coprocessor is ready to start it takes **EXTCPB** LOW. ARM720T samples **EXTCPB** at the low phases of each cycle in which **nCPI** is LOW.

Table A-2 Coprocessor interface signal descriptions (continued)

Name	Type	Description
nOPC	Out	Not opcode fetch. When LOW, this signal indicates that the processor is fetching an instruction from memory. When HIGH, data, if present, is being transferred. This signal is used by the coprocessor to track the ARM pipeline.
nCPI	Out	Not coprocessor instruction. When LOW, this signal indicates that the ARM720T is executing a coprocessor instruction.
nUSER	Out	Not User mode. When LOW, this signal indicates that the processor is in User mode. It is used by a coprocessor to qualify instructions.
TBIT	Out	Thumb state. This signal, when HIGH, indicates that the processor is executing the THUMB instruction set. When LOW, the processor is executing the ARM instruction set.

A.3 JTAG signals

JTAG signal descriptions are listed in Table A-3.

Table A-3 JTAG signal descriptions

Name	Type	Description
HIGHZ	Out	This signal denotes that the **HIGHZ** instruction has been loaded into the TAP controller.
IR[3:0]	Out	TAP instruction register. These signals reflect the current instruction loaded into the TAP controller instruction register. The signals change on the falling edge of **XTCK** when the TAP state machine is in the UPDATE-DR state. You can use these signals to allow more scan chains to be added using the ARM720T TAP controller.
RSTCLKBS	Out	Reset boundary scan clock. This signal denotes that either the TAP controller state machine is in the RESET state or that **XnTRST** has been asserted. You can use this to reset boundary scan cells outside the ARM720T.
SCREG[3:0]	Out	Scan chain register. These signals reflect the ID number of the scan chain currently selected by the TAP controller. These signals change on the falling edge of **XTCK** when the TAP state machine is in the UPDATE-DR state.
SDINBS	Out	Boundary scan serial data in. This signal is the serial data to be applied to an external scan chain.
SDOUTBS	In	Boundary scan serial data out. This signal is the serial data from an external scan chain. It allows a single **XTDO** port to be used. If an external scan chain is not connected, this input must be tied LOW.
TAPSM[3:0]	Out	Tap controller status. These signals represent the current state of the TAP controller machine. These signals change on the rising edge of **XTCK** and can be used to allow more scan chains to be added using the ARM720T TAP controller.
TCK1	Out	Test clock one. This clock represents the HIGH phase of **XTCK**. **TCK1** is HIGH when **XTCK** is HIGH. This signal can be used to allow more scan chains to be added using the ARM720T TAP controller.
TCK2	Out	Test clock two. This clock represents the LOW phase of **XTCK**. **TCK2** is HIGH when **XTCK** is LOW. You can use this signal to allow more scan chains to be added using the ARM720T TAP controller. **TCK2** is the non-overlapping complement of **TCK1**.
XnTDOEN	Out	Not test data out output enable. When LOW, this signal denotes that serial data is being driven out on the **XTDO** output.

Table A-3 JTAG signal descriptions (continued)

Name	Type	Description
XnTRST	In	Not test reset. When LOW, this signal resets the JTAG interface.
XTCK	In	Test clock. This signal is the JTAG test clock.
XTDI	In	Test data in. JTAG test data in signal.
XTDO	Out	Test data out. JTAG test data out signal.
XTMS	In	Test mode select. JTAG test mode select signal.

A.4 Debugger signals

The debugger signal descriptions are listed in Table A-4.

Table A-4 Debugger signal descriptions

Name	Type	Description
BREAKPOINT	In	Breakpoint. This signal allows external hardware to halt execution of the processor for debug purposes. When HIGH, this causes the current memory access to be breakpointed. If memory access is an instruction Fetch, the core enters debug state if the instruction reaches the Execute stage of the core pipeline. If the memory access is for data, the core enters the debug state after the current instruction completes execution. This allows extension of the internal breakpoints provided by the EmbeddedICE module.
COMMRX	Out	Communication receive full. When HIGH, this signal denotes that the comms channel receive buffer contains data for the core to read.
COMMTX	Out	Communication transmit empty. When HIGH, this signal denotes that the comms channel transmit buffer is empty.
DBGACK	Out	Debug acknowledge. When HIGH, this signal denotes that the ARM is in debug state.
DBGEN	In	Debug enable. This signal allows the debug features of ARM720T to be disabled. When **DBGEN** is LOW, it inhibits **BREAKPOINT** and **DBGRQ** to the core, **DBGACK** from the ARM720T is always LOW.
DBGRQ	In	Debug request. This signal causes the core to enter debug state after executing the current instruction. This allows external hardware to force the core into debug state, in addition to the debugging features provided by the EmbeddedICE Logic.
EXTERN [1:0]	In	External condition. These signals allow breakpoints and watchpoints to depend on an external condition.
RANGEOUT[1:0]	Out	Range out. These signals indicate that the relevant EmbeddedICE watchpoint register has matched the conditions currently present on the address, data, and control buses. These signals are independent of the state of the watchpoint enable control bits.

A.5 Embedded trace macrocell interface signals

The ETM interface signals are listed in Table A-5.

Table A-5 ETM interface signal descriptions

Output name	Type	Description
ETMnMREQ	Out	Not memory request. When LOW, indicates that the processor requires memory access during the following cycle.
ETMSEQ	Out	Sequential address. When HIGH, indicates that the address of the next memory cycle is related to that of the last memory cycle. The new address is one of the following: • the same as the previous one • four greater in ARM state • two greater in Thumb state. This signal can be used, with the low order address lines, to indicate that the next cycle can use a fast memory mode and bypass the address translation system.
ETMnEXEC	Out	Not executed. When HIGH, indicates that the instruction in the execution unit is not being executed. For example it might have failed the condition check code.
ETMnCPI	Out	Not coprocessor instruction. When the ARM720T executes a coprocessor instruction, it takes the **ETMnCPI** LOW and waits for a response from the coprocessor. The actions taken depend on this response, which the coprocessor signals on the **CPA** and **CPB** inputs.
ETMA[31:0]	Out	Addresses. This is the retimed internal address bus.
ETMnOPC	Out	Not opcode fetch. When LOW, indicates that the processor is fetching an instruction from memory. When HIGH, indicates that data, if present, is being transferred.
ETMnRW	Out	Not read/write. When HIGH, indicates a processor write cycle. When LOW, indicates a processor read cycle.
ETMCLK	Out	ETM clock. Exported clock signal for use in ETM. Internal signal is inverted version of **CPCLK**. See Table A-2 on page A-5 for a description of **CPCLK**.

Table A-5 ETM interface signal descriptions (continued)

Output name	Type	Description
ETMCLKEN	Out	ETM clock enable. Exported signal used to gate **ETMCLK**. Internal signal is based on the **CPnWAIT** signal that is first phase two latched by **CPCLK**. This ensures that it changes at the start of phase two, the HIGH phase of **CPCLK**. It is held throughout the next phase, that is phase one, the LOW phase of **CPCLK.** See Table A-2 on page A-5 for a description of **CPnWAIT**.
ETMMAS[1:0]	Out	Memory access size. Indicates the width of the bus transaction to the current address, this signal can take the following values: 00 = 8-bit 01 = 16-bit 10 = 32-bit 11 is reserved. The above values are valid for both read and write cycles.
ETMDBGACK	Out	Debug acknowledge. When HIGH, indicates that the processor is in debug state. When LOW, indicates that the processor is in normal system state.
ETMD[31:0]	Out	Coprocessor data bus. This is the retimed internal data bus.
ETMABORT	Out	Memory abort or bus error. Indicates that a requested access has been disallowed.
ETMCPA	Out	Coprocessor absent handshake. The coprocessor absent signal. It is a buffered version of the coprocessor absent signal.
ETMCPB	Out	Coprocessor busy handshake. The coprocessor busy signal. It is a buffered version of the coprocessor absent signal.
PROCID[31:0]	Out	Trace PROCID bus.
PROCIDWR	Out	Trace PROCID write. Indicates to ETM7 that the Trace PROCID, CP15 register 13, has been written.

Note

- The signal **TBIT** is also used as an ETM interface signal. For a description of **TBIT**, see Table A-2 on page A-5.
- The signal **BIGEND** is also used as an ETM interface signal. For a description of **BIGEND**, see Table A-6 on page A-12.

A.6 Miscellaneous signals

— Miscellaneous signals used by the ARM720T are listed in Table A-6.

Table A-6 Miscellaneous signal descriptions

Name	Type	Source or destination	Description
BIGEND	Out	Configuration output	Big-endian format. When this signal is HIGH, the processor treats bytes in memory as being in big-endian format. When it is LOW, memory is treated as little-endian.
CACHEDIS[a]	In	Configuration input	Disable cache. This signal is used to disable the IDC for use in certain applications. See *IDC disable for secure applications* on page 4-6 for a description of this signal.
FCLK	In	External clock source	Fast clock input. This clock is used to clock the ARM core when **XFASTBUS** is LOW. During testing, the signal allows efficient testing of the RAM, TAG, and MMU blocks.
XFASTBUS	In	Configuration input	Bus clocking mode configuration signal. When HIGH, the ARM720T operates from a single clock, **BCLK**. When LOW, selects standard mode operating from two clocks, **BCLK** and **FCLK**.
XnFIQ	In	Interrupt controller	ARM fast interrupt request signal.
XnIRQ	In	Interrupt controller	ARM interrupt request signal. The interrupt controller mixes several interrupt sources, and produces **XnIRQ**.
XSnA	In	Configuration input	Synchronous and not asynchronous configuration pin. In standard ARM bus mode this signal determines the bus interface mode and must be wired HIGH or LOW depending on the desired relationship between **FCLK** and **BCLK**. See *Standard mode* on page 9-5. This pin is ignored when operating **XFASTBUS** is high.

a.ARM does not support the use of this feature.

A.7 Additional signal outputs

Three additional signal outputs are provided to aid the interface of AMBA signals to Input and Output pads when building an ARM test chip. These signals are:

- **BABE**
- **WDEN**
- **BDEN**.

Note

ARM advises that these signals are not used.

Index

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

A

Abort mode 2-7
Aborts
  CPU 6-18
  Data 2-19, 7-34
    indexed addressing 2-24
  external
    buffered writes 6-25
    cachable reads 6-25
  prefetch 2-18
  types 2-18
Access faults
  checking 6-22
Accessing banked registers 7-26
Addressing signals 10-7
AMBA interface
  about 10-2
  signals A-2
ARM instruction set 1-6
  addressing mode
    five 1-14
    four 1-13
    three 1-12
    two 1-11
    two, privileged 1-12
  condition fields 1-15
  fields 1-14
  operand two 1-14
ARM state
  register organization 2-9
ARM720T
  block diagram 1-3
  description 1-2
  scan chain arrangement 7-5
ASB bus interface 10-3

B

Banked registers
  accessing 7-26
Big endian
  format 10-14
Big endian operation 10-14
Big endian. *See* memory format
Breakpoints
  clearing 8-10
  entering debug state from 7-30
  programming 8-9
    hardware 8-9
    software 8-9
  setting 8-10
  with prefetch abort 7-34
Bus interface 9-2
  asynchronous mode 9-6
  fastbus extension 9-2
  standard mode 9-2
  synchronous mode 9-7

Bus master
default 10-21
Bus master handover 10-19
BYPASS
public instruction 7-13
Bypass register 7-16
Byte (data type) 2-6
Byte operations 10-16

C

CLAMP
public instruction 7-14
CLAMPZ
public instruction 7-15
Clock switching
debug state 7-23
Communications channel
message transfer 8-21
using 8-20
Condition code flags 2-13
Configuration
compatibility 3-2
description 3-2
notation 3-2
Control coprocessor state
determining 7-28
Control registers
Registers
control 8-6
Coprocessor 1-4
Coprocessor interface
signals A-5
Core clocks 7-23
Core state
accessing banked registers 7-25
determining 7-25
moving to ARM state 7-25
CPSR (Current Processor Status Register) 2-13
format of 2-13
CPU aborts 6-18
Cycle types
bus interface 10-4

D

Data signal timing 10-9
Data types 2-6
alignment 2-6
byte 2-6
halfword 2-6
word 2-6
Debug
host 7-4
program counter 7-30
protocol converter 7-4
reset 7-11
systems 7-4
Debug extensions 7-2
debug state 7-2
internal state 7-2
Debug interface
definition 7-2
Debug request
entering debug state via 7-32
Debug state
entering 7-7
entering on breakpoint 7-7
entering on debug-request 7-8
entering on watchpoint 7-7
exiting from 7-28
switching clock state 7-23
Debugger
signals A-9
device identification code register 7-16
Domain access control 6-21
Domain access control register
format 6-21
interpreting access bits 6-21

E

Early termination
definition 2-24
EmbeddedICE
about 8-2
breakpoints 8-9
coupling 8-17
BREAKPT signal 8-2
communications channel 8-19
control registers 8-6
debug control register 8-13
debug status register 8-15
definition 8-2
disabling 8-3
TAP controller 8-2, 8-6
timing 8-3
ETM
about 12-2
interface 12-3
ETM interface
signals A-10
Exception
entering 2-16
entry and exit summary 2-17
leaving 2-17
priorities 2-21
restrictions 2-21
returning to THUMB state from 2-17
vectors 2-20, 2-21
addresses 2-20
External aborts
Aborts
external 6-25
buffered writes 6-25
cachable reads 6-25
EXTEST
public instruction 7-12

F

Fast Context Switch Extension 2-22
Fastbus extension 9-3
Fault address register 6-19
Fault checking 6-22
Fault status register 6-19
Faults
alignment 6-23
domain 6-23
permission 6-24
section 6-24
subpage 6-24
translation 6-23
FCSE
relocation of low virtual addresses 2-22
FIQ mode 2-7
definition 2-18

H

Halfword operations 10-15
High register
accessing from THUMB state 2-11
description 2-11
HIGHZ
public instruction 7-14

I

IDC
cacheable bit 4-2
disable 4-5
disable for secure applications 4-6
enable 4-5
interaction with MMU and write buffer 6-26
operation 4-2
read-lock-write 4-3
reset 4-5
validity 4-4
double-mapped space 4-4
software IDC flush 4-4
IDCODE
public instruction 7-13

Instruction register 7-17
Instruction set 1-5
  ARM 1-6
  Thumb 1-15
Instruction types 1-5
Internal coprocessor instructions 3-3
Interrupts 7-34
INTEST
  public instruction 7-12
IRQ mode 2-7
  definition 2-18

J

JTAG signals A-7
JTAG state machine 7-10

L

Large page references
  translating 6-16
Level 1
  descriptor 6-7
  fetch 6-6
Level 2
  descriptor 6-12
Little endian
  format 10-14
  operation 10-14
Little endian. *See* memory format
Low registers 2-12

M

Memory access
  use of the BWAIT pin 10-10
Memory format
  big endian
    description 2-3
  little endian
    description 2-4
Memory request signals 10-7, 10-8
Miscellaneous signals A-12
MMU
  description 6-2
  disabling 6-27
  domains 6-2
  effect of reset 6-3
  enabling 3-6, 6-26
  faults 6-18
  interaction with IDC and write buffer 6-26
  memory accesses 6-2
  program accessible registers 6-4
  TLB 6-2
Multi master operation 10-17

N

nWAIT pin
  use of 10-10

O

Operating modes
  Abort
    mode 2-7
  changing 2-7
  FIQ 2-7
  IRQ mode 2-7
  Supervisor mode 2-7
  System mode 2-7
  Undefined mode 2-7
  User mode 2-7
Operating state
  ARM 2-2
  reading 2-14
  switching 2-2
    to ARM 2-2
    to THUMB 2-2
  THUMB 2-2

P

Page table descriptor
  bits 6-8
Program status registers
  control bits 2-13
  mode bit values 2-14
  reserved bits 2-14
Programming watchpoints 8-11
Public instructions 7-12
  BYPASS 7-13
  CLAMP 7-14
  CLAMPZ 7-15
  EXTEST 7-12
  HIGHZ 7-14
  IDCODE 7-13
  INTEST 7-12
  RESTART 7-15
  SAMPLE/PRELOAD 7-15
  SCAN_N 7-12

R

Read-lock-write 10-13
Registers 3-4
  ARM 2-8
    interrupt modes 2-9
  BYPASS 7-16
  debug communications channel 8-19
  debug control
    DBGACK 8-13
    DBGRQ 8-13
    INTDIS 8-14
  debug status 8-15
  device ID 7-16
  fault address 6-19
  fault status 6-19
  instruction 7-17
  MMU 6-4
  register 0, ID register 3-4
  register 1, control register 3-5
  register 13, process identifier register 3-10
    changing FCSE PID 3-11
    FCSE PID 3-11
  register 2, translation table base register 3-7
  register 3, domain access control register 3-7
  register 4, reserved 3-8
  register 5, fault status register 3-8
  register 6, fault address register 3-9
  register 7, cache operations register 3-9
  register 8, translation lookaside buffer register 3-9
  register 9-12, reserved 3-10
  relationship between ARM and Thumb 2-11
  scan chain select 7-17
  test data types 7-16
  Thumb 2-10
  watchpoint 8-4
    programming and reading 8-5
Reset
  action of processor on 2-23
RESTART public instruction 7-15
Return address calculations 7-33

S

SAMPLE/PRELOAD
  public instruction 7-15
Scan and debug
  signals used by ETM 7-42
Scan chain 0 7-20
Scan chain 1 7-21
Scan chain 15 7-22
Scan chain 2 7-22
Scan chain select register 7-17
Scan Chains 7-18
Scan interface timing 7-35
Scan limitations 7-9
SCAN_N
  public instruction 7-12
Section descriptor 6-9

Sequential memory accesses
  Memory accesses
    sequential 10-12
Signals
  AMBA interface A-2
  coprocessor interface A-5
  debugger A-9
  ETM interface A-10
  JTAG A-7
  miscellaneous A-12
Slave operation, test mode 11-2
Slave response signals 10-10
Small page references
  translating 6-14
Software Interrupt 2-19
Software interrupt 2-19
SPSR (Saved Processor Status Register) 2-13
  format of 2-13
Standard mode 9-5
Supervisor mode 2-7
SWI 2-19
System mode 2-7
System speed access
  during debug state 7-32
System state
  determining 7-27

T

T bit (in CPSR) 2-14
Test data register types 7-16
Thumb instruction set 1-15
Thumb state 2-2
  register organization 2-10
Translating references 6-5
Translating section references 6-11
Translation table base 6-5

U

Undefined instruction trap 2-20
Undefined mode 2-7
User mode 2-7

W

Watchpoint
  registers 8-4
    programming and reading 8-5
Watchpoints
  entering debug state from 7-31
  programming 8-11
  programming restriction 8-12
  with another exception 7-31
Word operations 10-15
Write buffer
  bufferable bit 5-2
  definition 5-2
  interaction with MMU and IDC 6-26
  operation 5-3
    bufferable write 5-3
    read-lock-write 5-4
    unbufferable write 5-3

