ARM9TDMI™

(Rev 2)

Technical Reference Manual

ARM®

ARM DDI 0168A

ARM9TDMI
Technical Reference Manual

Release Information

Change History

Date	Issue	Change
July 1999	A	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM9TDMI Technical Reference Manual

List of Tables
ARM9TDMI Technical Reference Manual

List of Figures
ARM9TDMI Technical Reference Manual

Preface

This preface introduces the ARM9TDMI (Revision 2), which is a member of the ARM family of general-purpose microprocessors. It contains the following sections:

About this manual

This document is a reference manual for the ARM9TDMI microprocessor. The ARM9TDMI includes the following features:

- The option, selectable using the **UNIEN** signal, of using two unidirectional buses **DD[31:0]** and **DDIN[31:0]**, instead of a single bidirectional data bus. This is described in *Unidirectional/bidirectional mode interface* on page 3-10.
- The value returned by the JTAG TAP controller IDCODE instruction is the value present on the new **TAPID[31:0]** input bus. This allows the ID code to be easily changed for each chip design.

Intended audience

This manual is written for experienced hardware and software engineers who might or might not have experience of ARM products.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams* on page xiii
- *Signals* on page xiii
- *Numbering* on page xiv.

Typographical

The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

monospace italic Denotes arguments to monospace text where the argument is to be replaced by a specific value.

monospace bold Denotes language keywords when used outside example code.

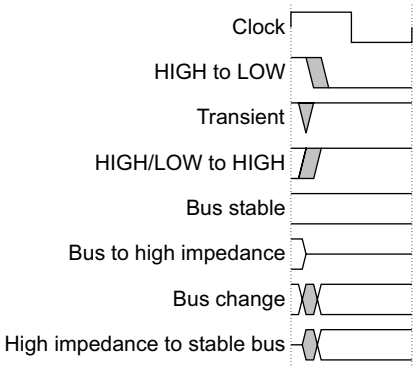
< and > Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example:

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

Timing diagrams

The figure named *Key to timing diagram conventions* explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

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.



Key to timing diagram conventions

Signals

The signal conventions are:

Signal level The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.

Lower-case n Denotes an active-LOW signal.

Prefix A	Denotes global *Advanced eXtensible Interface* (AXI) signals.
Prefix AR	Denotes AXI read address channel signals.
Prefix AW	Denotes AXI write address channel signals.
Prefix B	Denotes AXI write response channel signals.
Prefix C	Denotes AXI low-power interface signals.
Prefix H	Denotes *Advanced High-performance Bus* (AHB) signals.
Prefix P	Denotes *Advanced Peripheral Bus* (APB) signals.
Prefix R	Denotes AXI read data channel signals.
Prefix W	Denotes AXI write data channel signals.

Numbering

The numbering convention is:

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

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

ARM Limited periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the Frequently Asked Questions list.

ARM publications

This manual contains information that is specific to the ARM9TDMI microprocessor. See the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM7TDMI™ Data Sheet* (ARM DDI 0029).

Other publications

This section lists relevant documents published by third parties:

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

Feedback

ARM Limited welcomes feedback on the ARM9TDMI microprocessor and its documentation.

Feedback on this product

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

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

Feedback on this manual

If you have any comments on this manual, send email to `errata@arm.com` giving:

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

ARM Limited also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the *ARM9TDMI (Revision 2)* and shows its processor block diagram under the headings:

1.1 About the ARM9TDMI

The ARM9TDMI is a member of the ARM family of general-purpose microprocessors. The ARM9TDMI is targeted at embedded control applications where high performance, low die size and low power are all important. The ARM9TDMI supports both the 32-bit ARM and 16-bit Thumb instruction sets, allowing the user to trade off between high performance and high code density. The ARM9TDMI supports the ARM debug architecture and includes logic to assist in both hardware and software debug. The ARM9TDMI supports both bidirectional and unidirectional connection to external memory systems. The ARM9TDMI also includes support for coprocessors.

The ARM9TDMI processor core is implemented using a five-stage pipeline consisting of fetch, decode, execute, memory and write stages. The device has a Harvard architecture, and the simple bus interface eases connection to either a cached or SRAM-based memory system. A simple handshake protocol is provided for coprocessor support.

1.2 Processor block diagram

Figure 1-1 shows the ARM9TDMI processor block diagram.

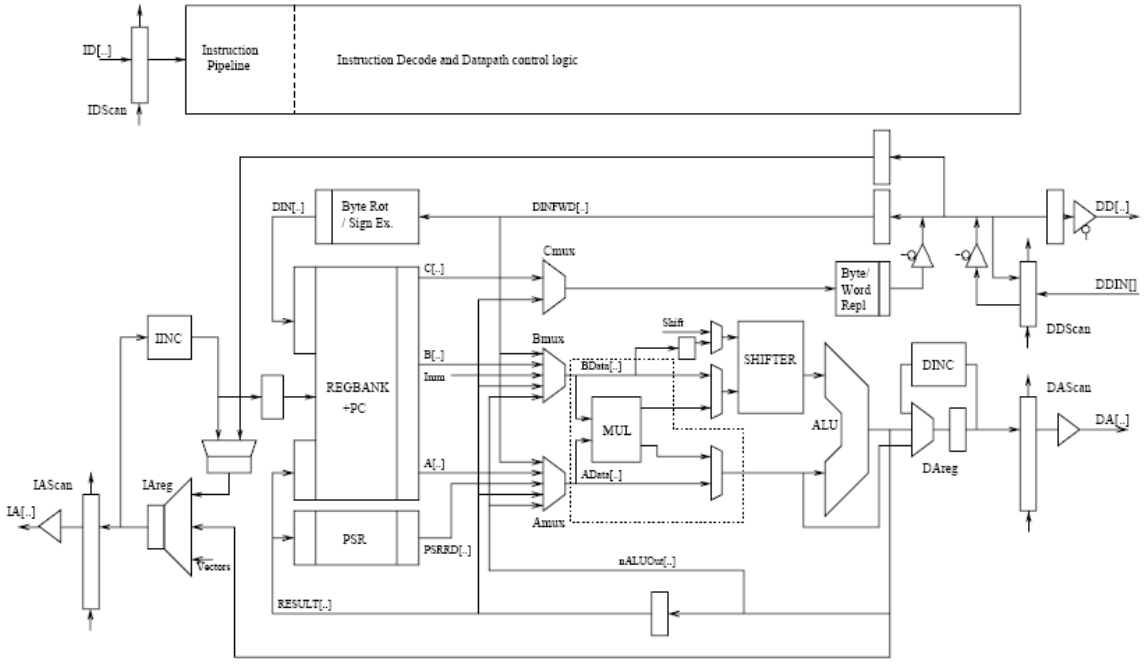


Figure 1-1 ARM9TDMI processor block diagram

Chapter 2
Programmer's Model

This chapter describes the programmer's model for the ARM9TDMI under the headings:

2.1 About the programmer's model

The ARM9TDMI processor core implements ARM Architecture v4T, and so executes the ARM 32-bit instruction set and the compressed Thumb 16-bit instruction set. The programmer's model is fully described in the *ARM Architecture Reference Manual*.

The ARM v4T architecture specifies a small number of implementation options. The options selected in the ARM9TDMI implementation are listed in the table below. For comparison, the options selected for the ARM7TDMI implementation are also shown:

Table 2-1 ARM9TDMI implementation option

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

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

- The ARM9TDMI implements the Base Restored Data Abort model, which significantly simplifies the software data abort handler.
- The ARM9TDMI fully implements the instruction set extension spaces added to the ARM (32-bit) instruction set in Architecture v4 and v4T.

These differences are explained in more detail below.

2.1.1 Data abort model

The ARM9TDMI implements the Base Restored Data Abort Model, which differs from the Base updated data abort model implemented by ARM7TDMI.

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

The Base Restored Data Abort Model significantly simplifies the software data abort handler.

2.1.2 Instruction set extension spaces

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

ARM Architecture v4 and v4T also introduced a number of instruction set extension spaces to the ARM instruction set. These are:

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

Instructions in these spaces are UNDEFINED (they cause an Undefined Instruction Exception). The ARM9TDMI fully implements all the instruction set extension spaces defined in ARM Architecture v4T as UNDEFINED instructions, allowing emulation of future instruction set additions.

2.2 Pipeline implementation and interlocks

The ARM9TDMI implementation uses a five-stage pipeline design. These five stages are:

- instruction fetch (F)
- instruction decode (D)
- execute (E)
- data memory access (M)
- register write (W).

ARM implementations are fully interlocked, so that software will function identically across different implementations without concern for pipeline effects. Interlocks do affect instruction execution times. For example, the following sequence suffers a single cycle penalty due to a load-use interlock on register R0:

```
LDR R0, [R7]
ADD R5, R0, R1
```

For more details, see Chapter 7 *Instruction Cycle Summary and Interlocks*. Figure 2-1 shows the timing of the pipeline, and the principal activity in each stage.

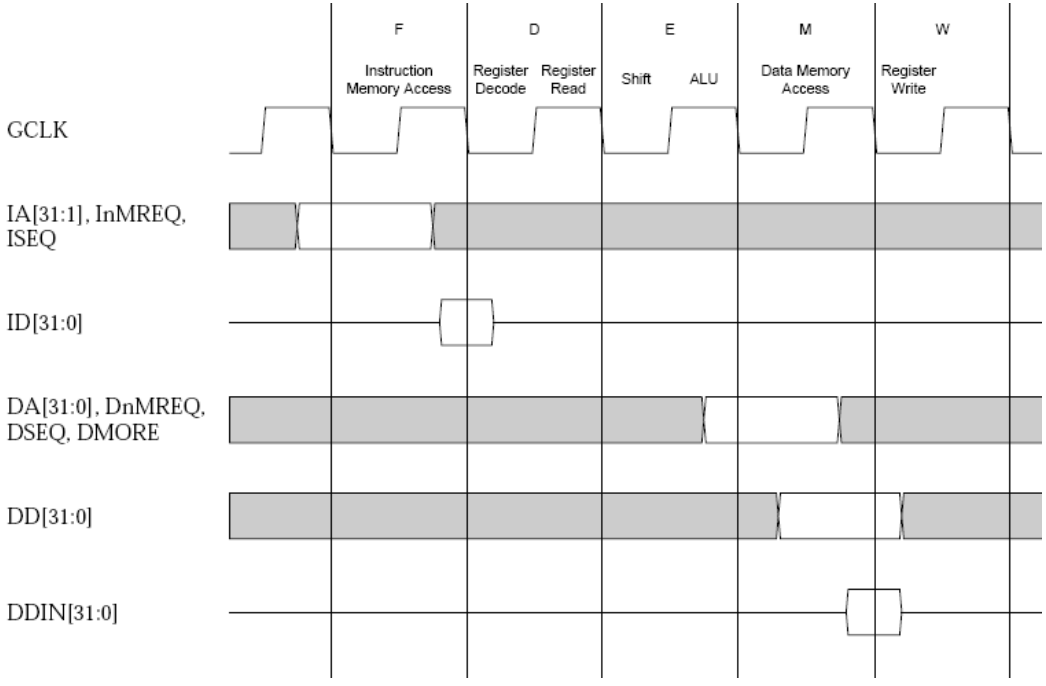


Figure 2-1 ARM9TDMI processor core instruction pipeline

Chapter 3
ARM9TDMI Processor Core Memory Interface

This chapter describes the memory interface of the ARM9TDMI processor core. The processor core has a Harvard memory architecture, and so the memory interface is separated into the instruction interface and the data interface. The information in this chapter is broken down as follows:

3.1 About the memory interface

The ARM9TDMI has a Harvard bus architecture with separate instruction and data interfaces. This allows concurrent instruction and data accesses, and greatly reduces the CPI of the processor. For optimal performance, single cycle memory accesses for both interfaces are required, although the core can be wait-stated for non-sequential accesses, or slower memory systems.

For each interface there are different types of memory access:

For both instruction and data interfaces, the ARM9TDMI process core uses pipelined addressing. The address and control signals are generated the cycle before the data transfer takes place, giving any decode logic as much advance notice as possible. All memory accesses are generated from **GCLK**.

- non-sequential
- sequential
- internal
- coprocessor transfer (for the data interface).

These accesses are determined by **InMREQ** and **ISEQ** for the instruction interface, and by **DnMREQ** and **DSEQ** for the data interface.

The ARM9TDMI can operate in both big-endian and little-endian memory configurations, and this is selected by the **BIGEND** input. The endian configuration affects both interfaces, so care must be taken in designing the memory interface logic to allow correct operation of the processor core.

For system purposes, it is normally necessary to provide some mechanism whereby the data interface can access instruction memory. There are two main reasons for this:

- The use of in-line data for literal pools is very common. This data will be fetched via the data interface but will normally be contained in the instruction memory space.
- To enable debug via the JTAG interface it must be possible to download code into the instruction memory. This code has to be written to memory via the data data bus as the instruction data bus is unidirectional. This means in this instance it is essential for the data interface to have access to the instruction memory.

A typical implementation of an ARM9TDMI-based cached processor has Harvard caches and a unified memory structure beyond the caches, thereby giving the data interface access to the instruction memory space. The ARM940T is an example of such a system. However, for an SRAM-based system this technique cannot be used, and an alternative method must be employed.

It is not as critical for the instruction interface to have access to the data memory area unless the processor needs to execute code from data memory.

3.1.1 Wait states

For memory accesses which require more than one cycle, the processor can be halted by using **nWAIT**. This signal halts the processor, including both the instruction and data interfaces. The **nWAIT** signal should be driven LOW by the end of phase 2 to stall the processor (it is inverted and ORed with **GCLK** to stretch the internal processor clock). The **nWAIT** signal must only change during phase 2 of **GCLK**. For debug purposes the internal core clock is exported on the **ECLK** signal. This timing is shown below in Figure 3-1.

Alternatively, wait states may be inserted by stretching either phase of **GCLK** before it is applied to the processor. ARM9TDMI does not contain any dynamic logic which relies on regular clocking to maintain its state. Therefore there is no limit on the maximum period for which **GCLK** may be stretched, in either phase, or the time for which **nWAIT** may be held LOW.

The system designer must take care when adding wait states because the interface is pipelined. When a wait state is asserted, the current data and instruction transfers are suspended. However, the address buses and control signals will have already changed to indicate the next transfer. It is therefore necessary to latch the address and control signals of each interface when using wait states.

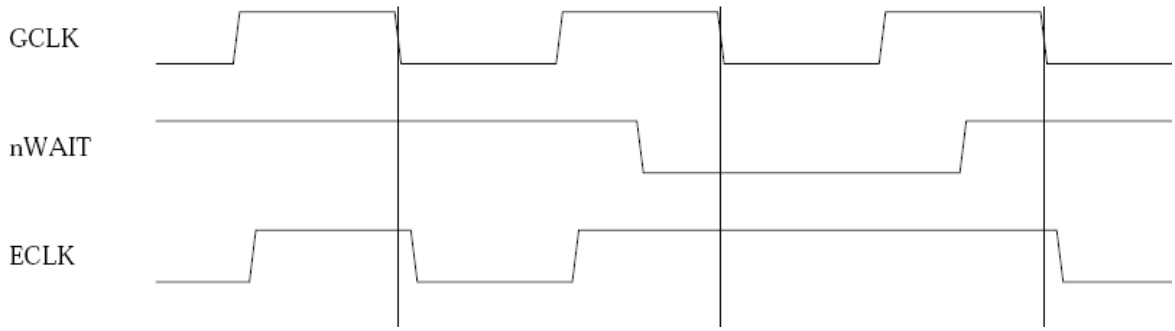


Figure 3-1 ARM9TDMI clock stalling using nWAIT

3.2 Instruction interface

Whenever an instruction enters the execute stage of the pipeline, a new opcode is fetched from the instruction bus. The ARM9TDMI processor core may be connected to a variety of cache/SRAM systems, and it is optimized for single cycle access systems.

However, in order to ease the system design, it is possible to connect the ARM9TDMI to memory which takes two (or more) cycles for a non-sequential (N) access, and one cycle for a sequential (S) access. Although this increases the effective CPI, it considerably eases the memory design.

The ARM9TDMI indicates that an instruction fetch will take place by driving **InMREQ** LOW. The instruction address bus, **IA[31:1]** will contain the address for the fetch, and the **ISEQ** signal will indicate whether the fetch is sequential or non-sequential to the previous access. All these signals become valid towards the end of phase 2 of the cycle that precedes the instruction fetch.

If **ITBIT** is LOW, and thus ARM9TDMI is performing word reads, then **IA[1]** should be ignored.

The timing is shown in Figure 3-2 on page 3-5. The full encoding of **InMREQ** and **ISEQ** is as follows:

Table 3-1 InMREQ and ISEQ encoding

InMREQ	ISEQ	Cycle type
0	0	Non-sequential
0	1	Sequential
1	0	Internal
1	1	Reserved for future use

Note

The 1,1 case does not occur in this implementation but may be used in the future.

Instruction fetches may be marked as aborted. The **IABORT** signal is an input to the processor with the same timing as the instruction data. If, and when, the instruction reaches the execute stage of the pipeline, the prefetch abort vector is taken. The timing for this is shown in Figure 3-2 on page 3-5. If the memory control logic does not make use of the **IABORT** signal, it must be tied LOW.

Internal cycles occur when the processor is stalled, either waiting for an interlock to resolve, or completing a multi-cycle instruction.

Note

A sequential cycle can occur immediately after an internal cycle.

Figure 3-2 shows the cycle timing for an N followed by an S cycle, where there is a prefetch abort on the S cycle:

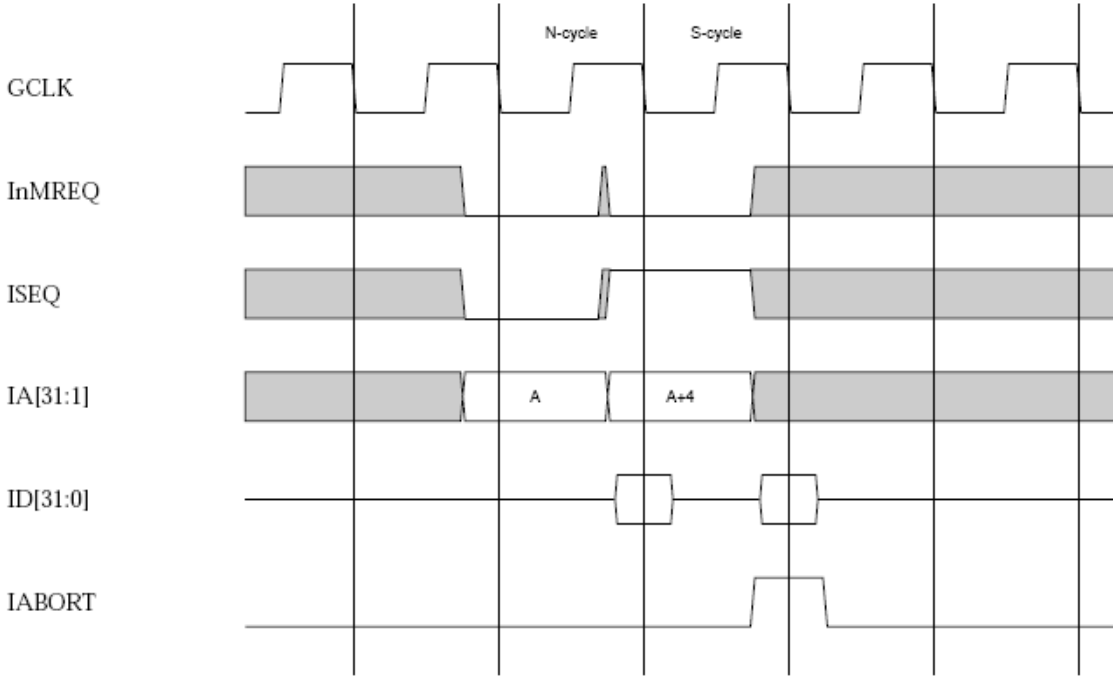


Figure 3-2 Instruction fetch timing

3.3 Endian effects for instruction fetches

The ARM9TDMI will perform 32-bit or 16-bit instruction fetches depending on whether the processor is in ARM or Thumb state. The processor state may be determined externally by the value of the **ITBIT** signal. When this signal is LOW, the processor is in ARM state, and 32-bit instructions are fetched. When it is HIGH, the processor is in Thumb state and 16-bit instructions are fetched.

When the processor is in ARM state, its endian configuration does not affect the instruction fetches, as all 32 bits of **ID[31:0]** are read. However, in Thumb state the processor will read either from the upper half of the instruction data bus, **ID[31:16]**, or from the lower half, **ID[15:0]**. This is determined by the endian configuration of the memory system, which is indicated by the **BIGEND** signal, and the state of **IA[1]**.

Table 3-2 shows which half of the data bus is sampled in the different configurations:

Table 3-2 Endian effect on instruction position

	Little BIGEND = 0	Big BIGEND = 1
IA[1] = 0	ID[15:0]	ID[31:16]
IA[1] = 1	ID[31:16]	ID[15:0]

When a 16-bit instruction is fetched, the ARM9TDMI ignores the unused half of the data bus.

3.4 Data interface

Data transfers take place in the memory stage of the pipeline. The operation of the data interface is very similar to the instruction interface.

The interface is pipelined with the address and control signals, becoming valid in phase 2 of the cycle before the transfer. There are four types of data cycle, and these are indicated by **DnMREQ** and **DSEQ**. The timing for these signals is shown in Figure 3-3 on page 3-9. The full encoding for these signals is given in Table 3-3:

Table 3-3 DnMREQ and DSEQ encoding

DnMREQ	DSEQ	Cycle Type
0	0	Non-sequential
0	1	Sequential
1	0	Internal
1	1	Coprocessor Transfer

For internal cycles, data memory accesses are not required in this instance, the data interface outputs will retain the state of the previous transfer.

DnRW indicates the direction of the transfer, LOW for reads and HIGH for writes. The signal becomes valid at approximately the same time as the data address bus.

- For reads, **DDIN[31:0]** must be driven with valid data for the falling edge of **GCLK** at the end of phase 2.
- For writes by the processor, data will become valid in phase 1, and remain valid throughout phase 2.

Both reads and writes are illustrated in Figure 3-3 on page 3-9.

See *About the coprocessor interface* on page 4-2 for further information on using **DDIN[31:0]** and **DD[31:0]** in unidirectional mode or connecting together to form a bidirectional bus.

Data transfers may be marked as aborted. The **DABORT** signal is an input to the processor with the same timing as the data. Upon completion of the current instruction in the memory stage of the pipeline, the data abort vector is taken. If the memory control logic does not make use of the **DABORT** signal, it must be tied LOW, but with the exception that data can be transferred to and from the ARM9TDMI core.

The size of the transfer is indicated by **DMAS[1:0]**. These signals become valid at approximately the same time as the data address bus. The encoding is given below in Table 3-4:

Table 3-4 DMAS[1:0] encoding

DMAS[1:0]	Transfer size
00	Byte
01	Half word
10	Word
11	Reserved

For coprocessor transfers, access to memory is not required, but there will be a transfer of data between the ARM9TDMI and coprocessor using the data buses, **DD[31:0]** and **DDIN[31:0]**. **DnRW** indicates the direction of the transfer and **DMAS[1:0]** indicates word transfers, as all coprocessor transfers are word sized.

The **DMORE** signal is active during load and store multiple instructions and only ever goes HIGH when **DnMREQ** is LOW. This signal effectively gives the same information as **DSEQ**, but a cycle ahead. This information is provided to allow external logic more time to decode sequential cycles.

Figure 3-3 on page 3-9 shows a load multiple of four words followed by an MCR, followed by an aborted store. Note the following:

- The **DMORE** signal is active in the first three cycles of the load multiple to indicate that a sequential word will be loaded in the following cycle.
- From the behavior of **InMREQ** during the LDM, it can be seen that an instruction fetch takes place when the instruction enters the execute stage of the pipeline, but that thereafter the instruction pipeline is stalled until the LDM completes.

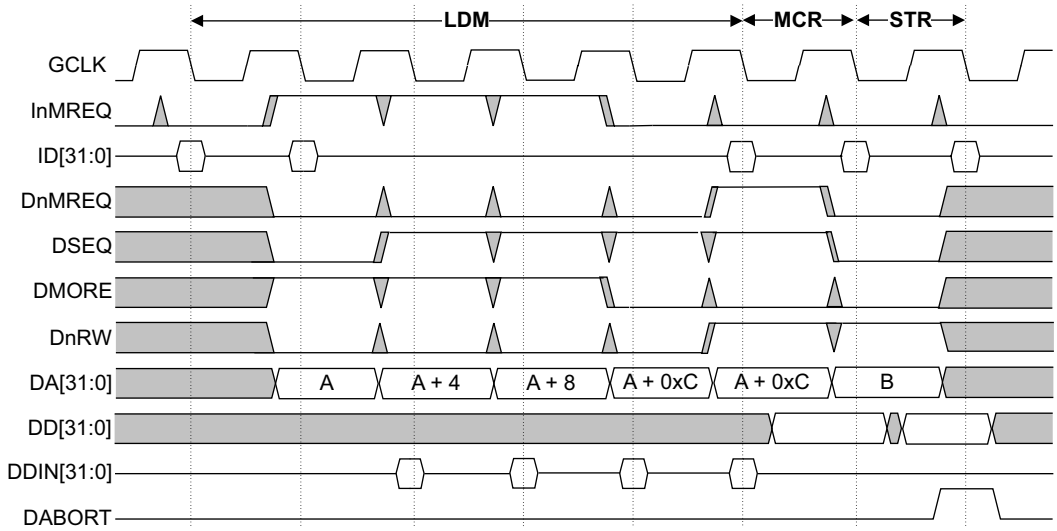


Figure 3-3 Data access timings

3.5 Unidirectional/bidirectional mode interface

The ARM9TDMI supports connection to external memory systems using either a bidirectional data data bus or two unidirectional buses. This is controlled by the **UNIEN** input.

If **UNIEN** is LOW, **DD[31:0]** is a tristate output bus used to transfer write data. It is only driven when the ARM9TDMI is performing a write to memory. By wiring **DD[31:0]** to the input **DDIN[31:0]** bus (externally to the ARM9TDMI), a bidirectional data data bus can be formed.

If **UNIEN** is HIGH, then **DD[31:0]**, and all other ARM9TDMI outputs, are permanently driven. **DD[31:0]** then forms a unidirectional write data data bus. In this mode, the tristate enable pins **IABE**, **DABE**, **DDBE**, **TBE**, and the TAP instruction **nHIGHZ**, have no effect. Therefore all outputs are always driven.

All timing diagrams in this manual, except where tristate timing is shown explicitly, assume **UNIEN** is HIGH.

3.6 Endian effects for data transfers

The ARM9TDMI supports 32-bit, 16-bit and 8-bit data memory access sizes. The endian configuration of the processor, set by **BIGEND**, affects only non-word transfers (16-bit and 8-bit transfers).

For data writes by the processor, the write data is duplicated on the data bus. So for a 16-bit data store, one copy of the data appears on the upper half of the data bus, **DD[31:16]**, and the same data appears on the lower half, **DD[15:0]**. For 8-bit writes four copies are output, one on each byte lane, **DD[31:24]**, **DD[23:16]**, **DD[15:8]** and **DD[7:0]**. This considerably eases the memory control logic design and helps overcome any endian effects.

For data reads, the processor will read a specific part of the data bus. This is determined by the endian configuration, the size of the transfer, and bits 1 and 0 of the data address bus. Table 3-5 shows which bits of the data bus are read for 16-bit reads, and Table 3-6 shows which bits are read for 8-bit reads.

For simplicity of design, 32 bits of data can be read from memory and the processor will ignore any unwanted bits.

Table 3-5 Endian effects for 16-bit data fetches

DA[1:0]	Little (BIGEND = 0)	Big (BIGEND = 1)
00	DDIN[15:0]	DDIN[31:16]
10	DDIN[31:16]	DDIN[15:0]

Table 3-6 Endian effects for 8-bit data fetches

DA[1:0]	Little (BIGEND = 0)	Big (BIGEND = 1)
00	DDIN[7:0]	DDIN[31:24]
01	DDIN[15:8]	DDIN[23:16]
10	DDIN[23:16]	DDIN[15:8]
11	DDIN[31:24]	DDIN[7:0]

3.7 ARM9TDMI reset behavior

When **nRESET** is driven LOW, the currently executing instruction terminates abnormally. If **GCLK** is HIGH, **InMREQ**, **ISEQ**, **DnMREQ**, **DSEQ** and **DMORE** will asynchronously change to indicate an internal cycle. If **GCLK** is LOW, they will not change until after the **GCLK** goes HIGH.

When **nRESET** is driven HIGH, the ARM9TDMI starts requesting memory again once the signal has been synchronized, and the first memory access will start two cycles later. The **nRESET** signal is sampled on the falling edge of **GCLK**. The behavior of the memory interfaces coming out of reset is shown in Figure 3-4 on page 3-13.

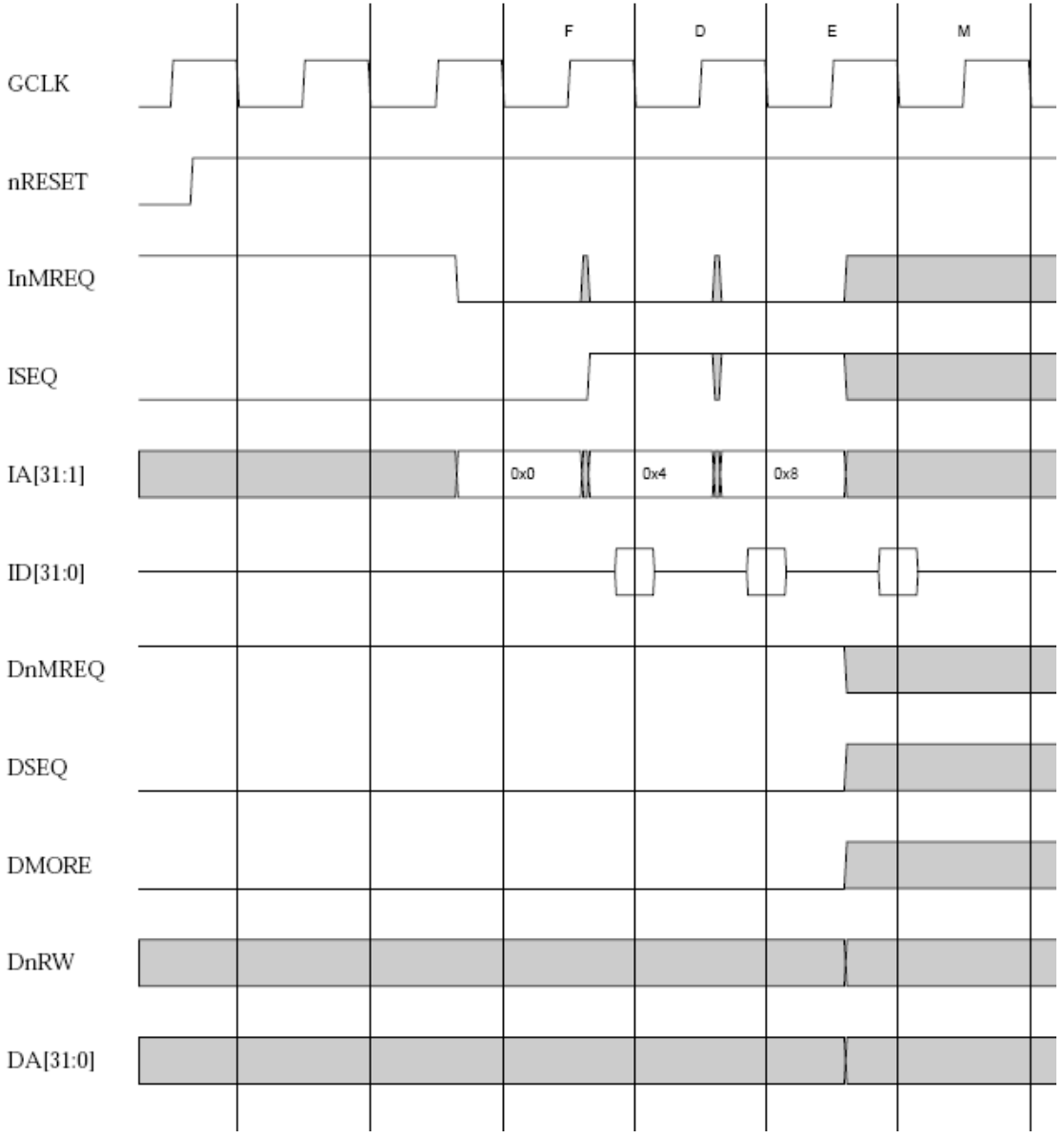


Figure 3-4 ARM9TDMI reset behavior

Chapter 4
ARM9TDMI Coprocessor Interface

This chapter describes the ARM9TDMI coprocessor interface, and details the following operations:

4.1 About the coprocessor interface

The ARM9TDMI supports the connection of coprocessors. All types of ARM coprocessor instructions are supported. Coprocessors determine the instructions they need to execute using a *pipeline follower* in the coprocessor. As each instruction arrives from memory, it enters both the ARM pipeline and the coprocessor pipeline. Typically, a coprocessor operates one clock phase behind the ARM9TDMI pipeline. The coprocessor determines when an instruction is being fetched by the ARM9TDMI, so that the instruction can be loaded into the coprocessor, and the pipeline follower advanced.

Note

A cached ARM9TDMI core typically has an external coprocessor interface block, the main purpose of which is to latch the instruction data bus, **ID**, one of the data buses, **DD[31:0] or DDIN[31:0]**, and relevant ARM9TDMI control signals before exporting them to the coprocessors. For a description of all the interface signals referred to in this chapter, see *Coprocessor interface signals* on page A-5.

There are three classes of coprocessor instructions:

- LDC/STC
- MCR/MRC
- CDP.

The following sections give examples of how a coprocessor should execute these instruction classes.

4.2 LDC/STC

The number of words transferred is determined by how the coprocessor drives the **CHSD[1:0]** and **CHSE[1:0]** buses. In the example, four words of data are transferred. Figure 4-1 on page 4-4 shows the ARM9TDMI LDC/STC cycle timing.

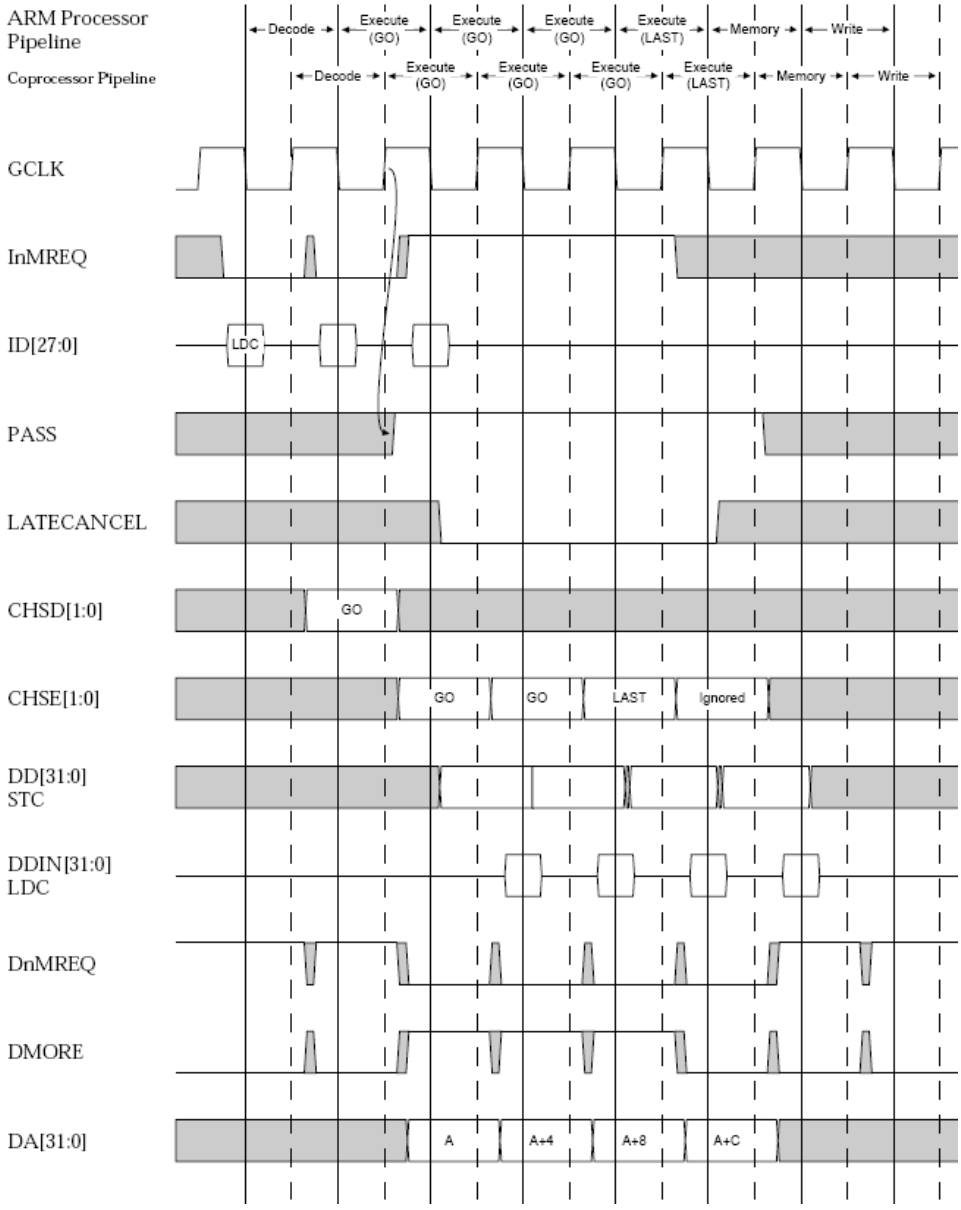


Figure 4-1 ARM9TDMI LDC / STC cycle timing

As with all other instructions, the ARM9TDMI processor core performs the main decode off the rising edge of the clock during the decode stage. From this, the core commits to executing the instruction, and so performs an instruction fetch. The coprocessor instruction pipeline keeps in step with the ARM9TDMI by monitoring **InMREQ**.

At the falling edge of **GCLK**, if **nWAIT** is HIGH, and **InMREQ** is LOW, an instruction fetch is taking place, and **ID[31:0]** will contain the fetched instruction on the next falling edge of the clock, when **nWAIT** is HIGH. This means that:

- the last instruction fetched should enter the decode stage of the coprocessor pipeline
- the instruction in the decode stage of the coprocessor pipeline should enter its execute stage
- the fetched instruction should be latched.

In all other cases, the ARM9TDMI pipeline is stalled, and the coprocessor pipeline should not advance.

Figure 4-2 shows the timing for these signals, and indicates when the coprocessor pipeline should advance its state. In this timing diagram, Coproc Clock shows a processed version of **GCLK** with **InMREQ** and **nWAIT**. This is one method of generating a clock to reflect the advance of the ARM9TDMI pipeline.

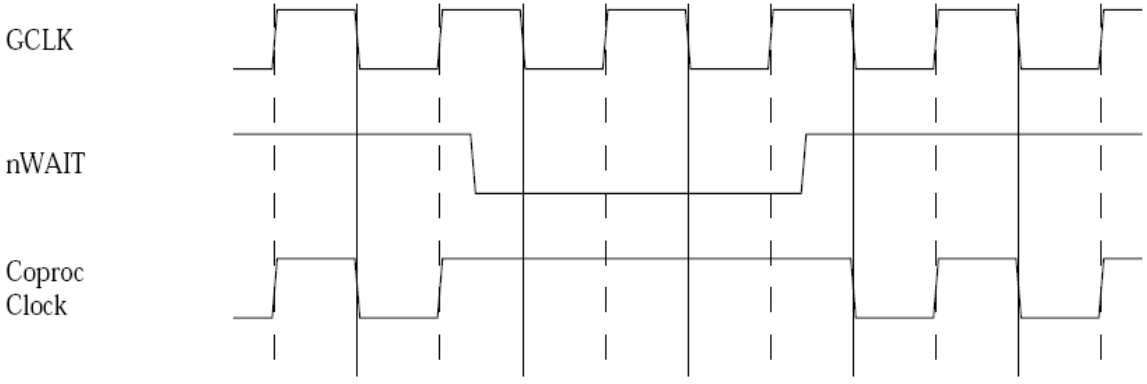


Figure 4-2 ARM9TDMI coprocessor clocking

During the execute stage, the condition codes are combined with the flags to determine whether the instruction really executes or not. The output **PASS** is asserted (HIGH) if the instruction in the execute stage of the coprocessor pipeline:

- is a coprocessor instruction

- has passed its condition codes.

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

A further output, **LATECANCEL**, is used to cancel a coprocessor instruction when the instruction preceding it caused a data abort. This is valid on the rising edge of **GCLK** on the cycle that follows the first execute cycle of the coprocessor instructions. This is the only cycle in which **LATECANCEL** can be asserted.

On the falling edge of the clock, the ARM9TDMI processor core examines the coprocessor handshake signals **CHSD[1:0]** or **CHSE[1:0]**:

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

The handshake signals encode one of four states:

ABSENT If there is no coprocessor attached that can execute the coprocessor instruction, the handshake signals indicate the ABSENT state. In this case, the ARM9TDMI processor core takes the undefined instruction trap.

WAIT If there is a coprocessor attached that can handle the instruction, but not immediately, the coprocessor handshake signals are driven to indicate that the ARM9TDMI processor core should stall until the coprocessor can catch up. This is known as the *busy-wait* condition. In this case, the ARM9TDMI processor core loops in an idle state waiting for **CHSE[1:0]** to be driven to another state, or for an interrupt to occur. If **CHSE[1:0]** changes to **ABSENT**, the undefined instruction trap will be taken. If **CHSE[1:0]** changes to GO or LAST, the instruction will proceed as described below. If an interrupt occurs, the ARM9TDMI processor core is forced out of the busy-wait state. This is indicated to the coprocessor by the **PASS** signal going LOW. The instruction will be restarted at a later date and so the coprocessor must not commit to the instruction (it must not change any of the coprocessor's state) until it has seen **PASS** HIGH, when the handshake signals indicate the GO or LAST condition.

GO The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires another cycle of execution. Both the ARM9TDMI processor core and the coprocessor must also consider the state of the **PASS** signal before actually committing to the instruction.

For an LDC or STC instruction, the coprocessor instruction drives the handshake signals with GO when two or more words still need to be transferred. When only one further word is to be transferred, the coprocessor drives the handshake signals with LAST. In phase 2 of the execute stage, the ARM9TDMI processor core outputs the address for the LDC/STC. Also in this phase, **DnMREQ** is driven LOW, indicating to the memory system that a memory access is required at the data end of the device. The timing for the data on **DD[31:0]** for an LDC and **DD[31:0]** for an STC is shown in Figure 4-1 on page 4-4.

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

For both MRC and STC instructions, the **DDIN[31:0]** bus is owned by the coprocessor, and can hence be driven by the coprocessor from the cycle after the relevant instruction enters the execute stage of the coprocessor pipeline, until the next instruction enters the execute stage of the coprocessor pipeline. This is the case even if the instruction is subject to a **LATECANCEL** or the **PASS** signal is not asserted.

For efficient coprocessor design, an unmodified version of **GCLK** should be applied to the execution stage of the coprocessor. This will allow the coprocessor to continue executing an instruction even when the ARM9TDMI pipeline is stalled.

4.2.1 Coprocessor handshake encoding

Table 4-1 shows how the handshake signals **CHSD[1:0]** and **CHSE[1:0]** are encoded.

Table 4-1 Handshake signals

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

If a coprocessor is not attached to the ARM9TDMI, the handshake signals must be driven with “10” ABSENT, otherwise the ARM9TDMI processor will hang if a coprocessor enters the pipeline.

If multiple coprocessors are to be attached to the interface, the handshaking signals can be combined by ANDing bit 1, and ORing bit 0. In the case of two coprocessors which have handshaking signals **CHSD1**, **CHSE1** and **CHSD2**, **CHSE2** respectively:

CHSD[1] <= **CHSD1[1]** AND **CHSD2[1]**

CHSD[0] <= **CHSD1[0]** OR **CHSD2[0]**

CHSE[1] <= **CHSE1[1]** AND **CHSE2[1]**

CHSE[0] <= **CHSE1[0]** OR **CHSE2[0]**

4.3 MCR/MRC

These cycles look very similar to STC/LDC. An example, with a busy-wait state, is shown in Figure 4-3:

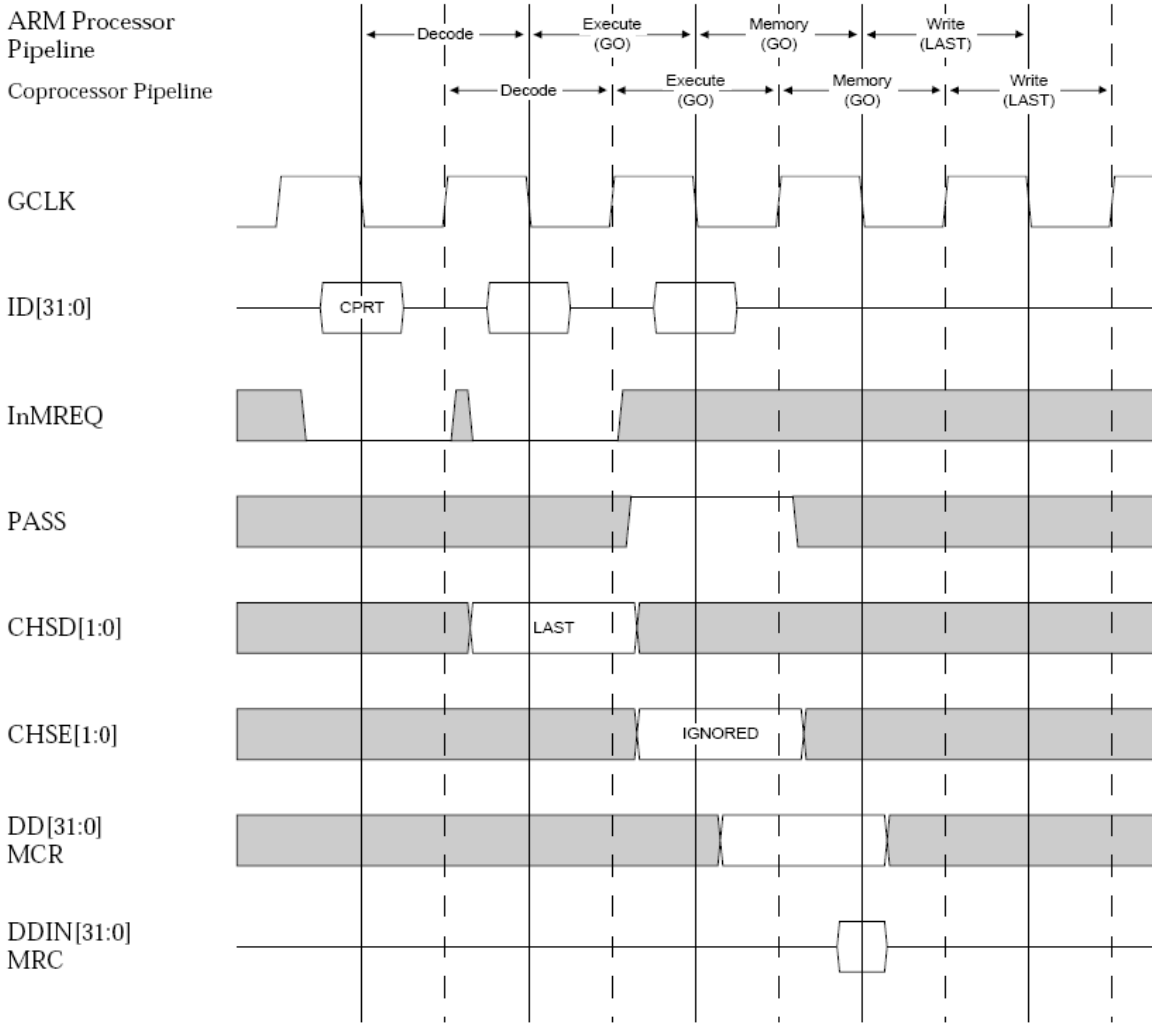


Figure 4-3 ARM9TDMI MCR / MRC transfer timing

First **InMREQ** is driven LOW to denote that the instruction on **ID** is entering the decode stage of the pipeline. This causes the coprocessor to decode the new instruction and drive **CHSD[1:0]** as required. In the next cycle **InMREQ** is driven LOW to denote that the instruction has now been issued to the execute stage. If the condition codes pass,

and hence the instruction is to be executed, the **PASS** signal is driven HIGH and the **CHSD[1:0]** handshake bus is examined (it is ignored in all other cases). For any successive execute cycles the **CHSE[1:0]** handshake bus is examined. When the LAST condition is observed, the instruction is committed. In the case of an MCR, the **DD[31:0]** bus is driven with the register data. In the case of an MRC, **DDIN[31:0]** is sampled at the end of the ARM9TDMI memory stage and written to the destination register during the next cycle.

4.4 Interlocked MCR

If the data for an MCR operation is not available inside the ARM9TDMI pipeline during its first decode cycle, the ARM9TDMI pipeline will interlock for one or more cycles until the data is available. An example of this is where the register being transferred is the destination from a preceding LDR instruction. In this situation the MCR instruction will enter the decode stage of the coprocessor pipeline, and remain there for a number of cycles before entering the execute stage. Figure 4-4 on page 4-12 gives an example of an interlocked MCR.

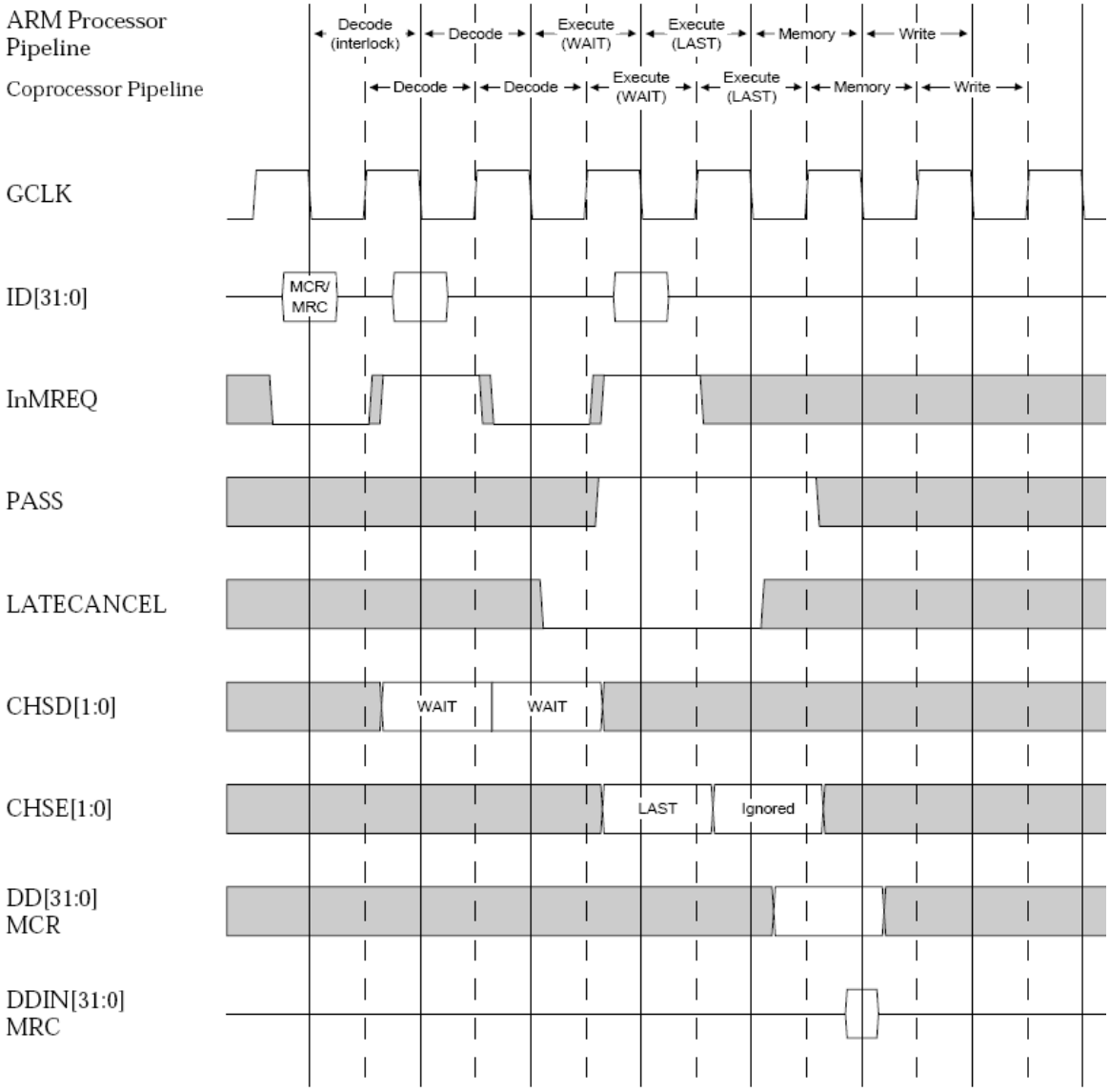


Figure 4-4 ARM9TDMI interlocked MCR

4.5 CDP

CDP signals normally execute in a single cycle. Like all the previous cycles, **InMREQ** is driven LOW to signal when an instruction is entering the decode and then the execute stage of the pipeline:

- if the instruction really is to be executed, the **PASS** signal is be driven HIGH during phase 2 of execute
- if the coprocessor can execute the instruction immediately it drives **CHSD[1:0]** with LAST
- if the instruction requires a busy-wait cycle, the coprocessor drives **CHSD[1:0]** with WAIT and then **CHSE[1:0]** with LAST.

Figure 4-5 on page 4-14 shows a CDP which is cancelled due to the previous instruction causing a data abort. The CDP instruction enters the execute stage of the pipeline and is signalled to execute by **PASS**. In the following phase **LATECANCEL** is asserted. This causes the coprocessor to terminate execution of the CDP instruction and for it to cause no state changes to the coprocessor.

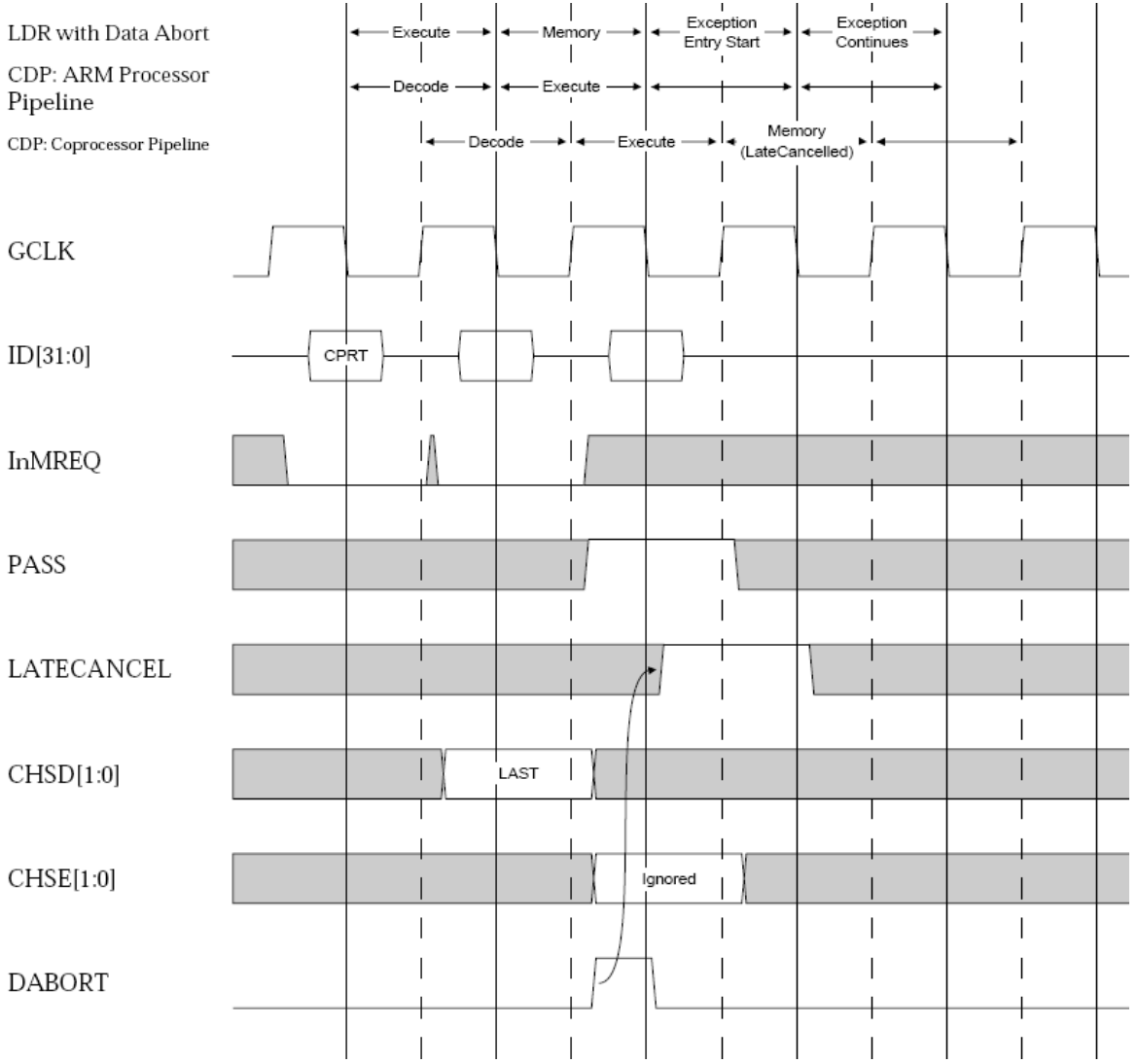


Figure 4-5 ARM9TDMI late cancelled CDP

4.6 Privileged instructions

The coprocessor may restrict certain instructions for use in privileged modes only. To do this, the coprocessor will have to track the **InTRANS** output. Figure 4-6 shows how **InTRANS** changes after a mode change.

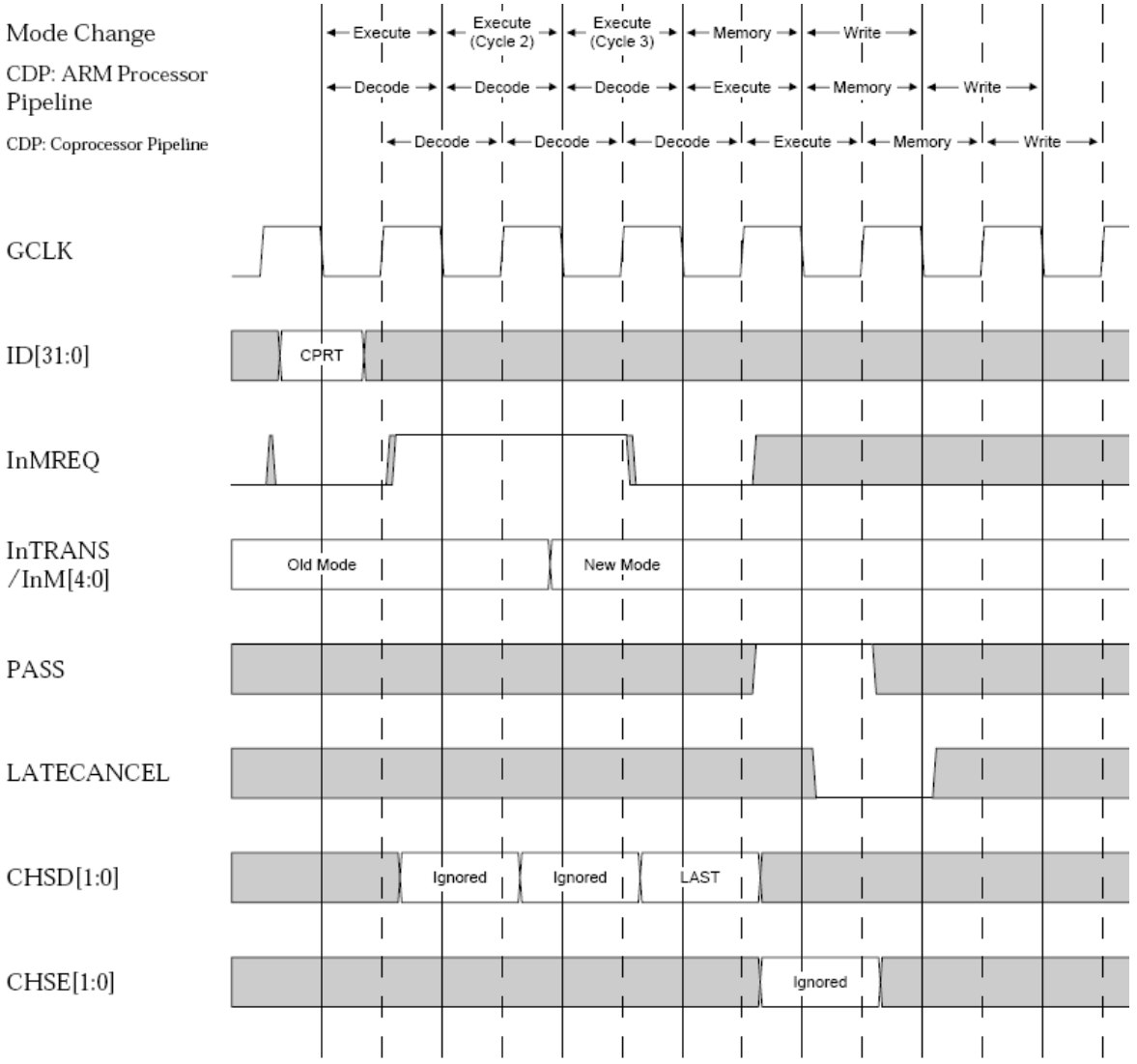


Figure 4-6 ARM9TDMI privileged instructions

The first two **CHSD** responses are ignored by the ARM9TDMI because it is only the final **CHSD** response, as the instruction moves from decode into execute, that counts. This allows the coprocessor to change its response as **InTRANS/InM[4:0]** changes.

4.7 Busy-waiting and interrupts

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

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

If it is HIGH, the instruction should still be executed. If it is LOW, the instruction should be abandoned. Figure 4-7 on page 4-18 shows a busy-waited coprocessor instruction being abandoned due to an interrupt:

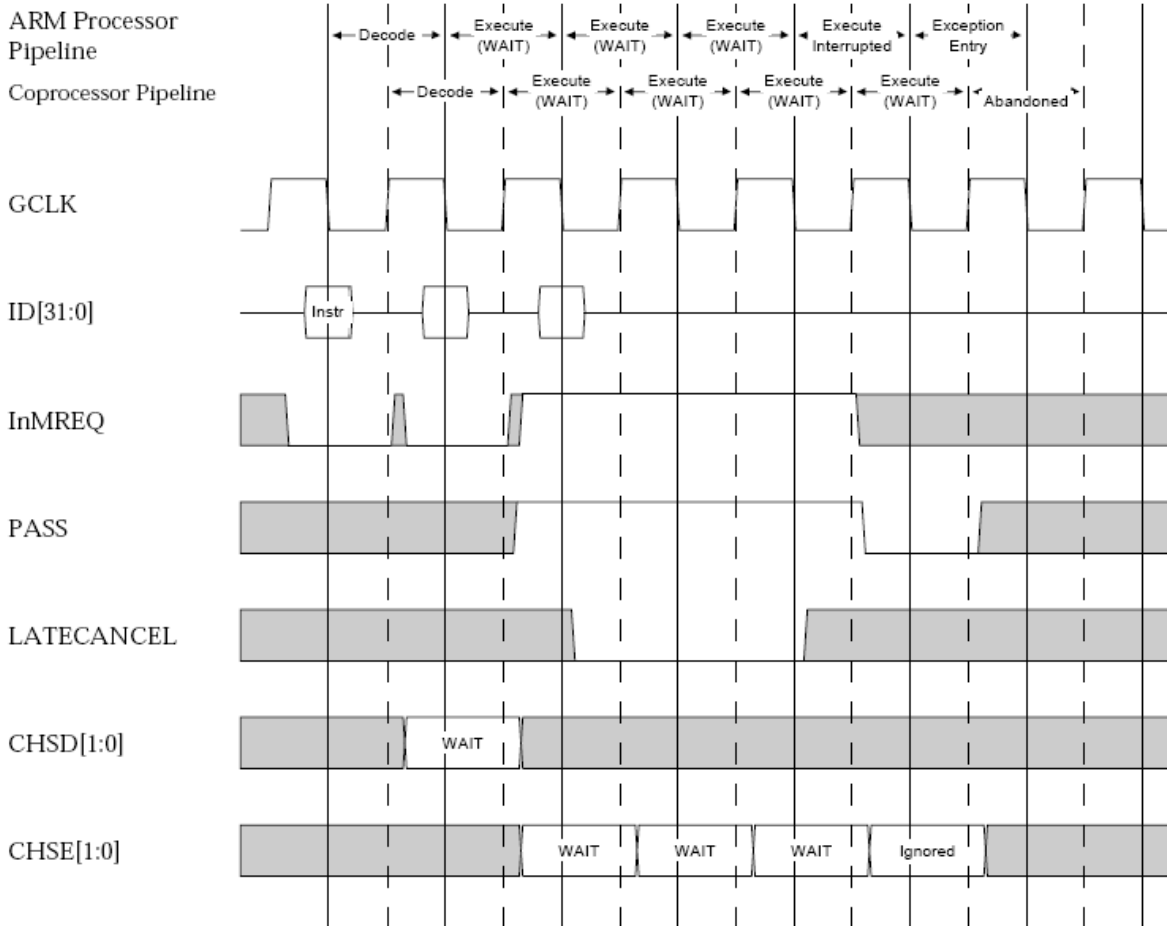


Figure 4-7 ARM9TDMI busy waiting and interrupts

4.8 Coprocessor 15 MCRs

Coprocessor 15 is typically reserved for use as a system control coprocessor. For an MCR to coprocessor 15, it is possible to transfer the coprocessor data to the coprocessor on the **IA** and **DA** buses. To do this the coprocessor should drive GO on the coprocessor handshake signals for a number of cycles. For each cycle that the coprocessor responded with GO on the handshake signals the coprocessor data will be driven onto **IA** and **DA** as shown in Figure 4-8.

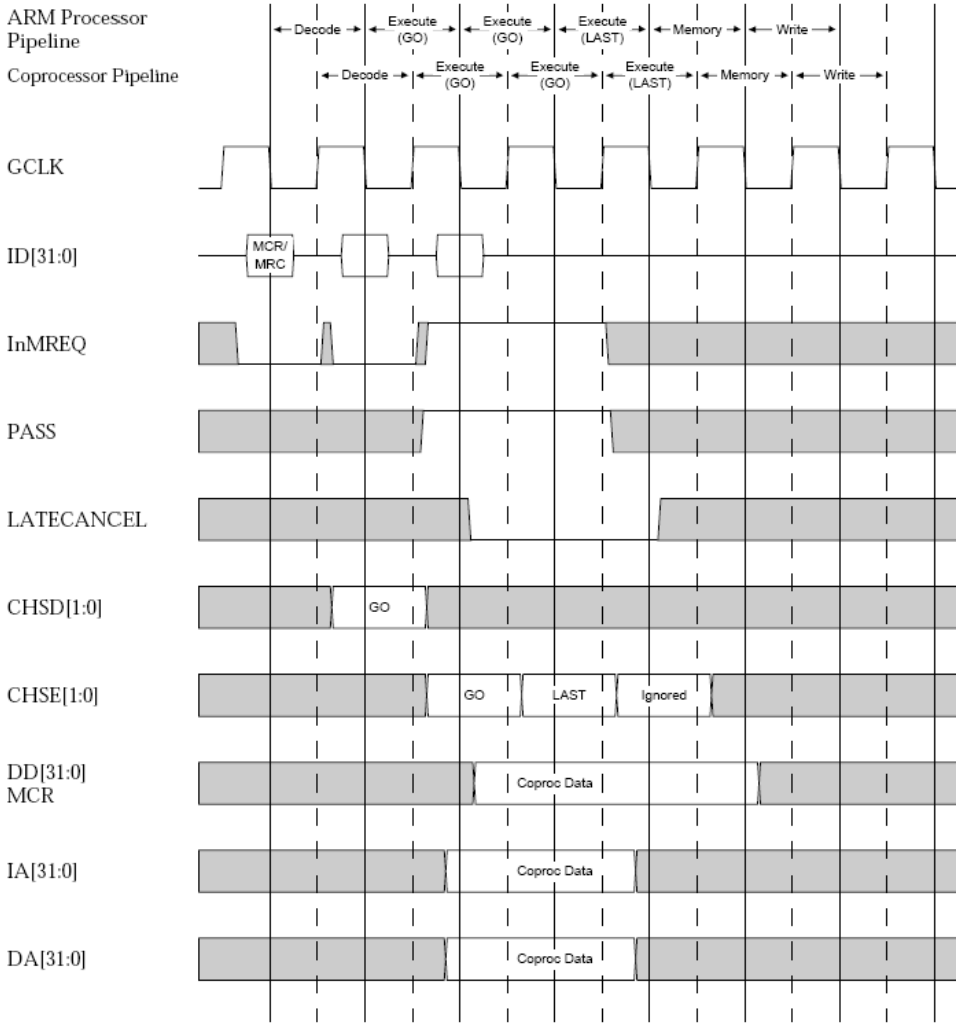


Figure 4-8 ARM9TDMI coprocessor 15 MCRs

Chapter 5
Debug Support

This chapter describes the debug support for the ARM9TDMI, including the EmbeddedICE™ macrocell:

5.1 About debug

The ARM9TDMI debug interface is based on IEEE Std. 1149.1- 1990, Standard Test Access Port and Boundary-Scan Architecture. Refer to this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

The ARM9TDMI contains hardware extensions for advanced debugging features. These are intended to ease the user's development of application software, operating systems, and the hardware itself.

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

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

When this happens, the ARM9TDMI is said to be in debug state. At this point, the internal state of the core and the external state of the system may be examined. Once examination is complete, the core and system state may be restored and program execution resumed.

The ARM9TDMI is forced into debug state either by a request on one of the external debug interface signals, or by an internal functional unit known as the EmbeddedICE macrocell. Once in debug state, the core isolates itself from the memory system. The core can then be examined while all other system activity continues as normal.

The internal state of the ARM9TDMI is examined via a JTAG-style serial interface, which allows instructions to be serially inserted into the pipeline of the core without using the external data bus. Thus, when in debug state, a store-multiple (STM) could be inserted into the instruction pipeline, and this would export the contents of the ARM9TDMI registers. This data can be serially shifted out without affecting the rest of the system.

5.2 Debug systems

The ARM9TDMI forms one component of a debug system that interfaces from the high-level debugging performed by the user to the low-level interface supported by the ARM9TDMI. A typical system is shown in Figure 5-1.

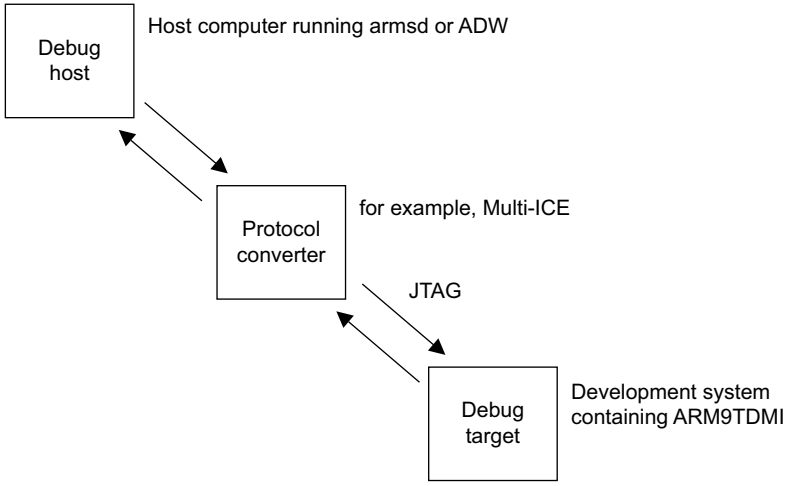


Figure 5-1 Typical debug system

Such a system typically has three parts:

- *The debug host*
- *The protocol converter*
- *The ARM9TDMI* on page 5-4.

These are described in the following paragraphs.

5.2.1 The debug host

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

5.2.2 The protocol converter

The debug host is connected to the ARM9TDMI development system via an interface (an RS232, for example). The messages broadcast over this connection must be converted to the interface signals of the ARM9TDMI. This function is performed by the protocol converter, for example, Multi-ICE™.

5.2.3 The ARM9TDMI

The ARM9TDMI, with hardware extensions to ease debugging, is the lowest level of the system. The debug extensions allow the user to stall the core from program execution, examine its internal state and the state of the memory system, and then resume program execution.

The debug host and the protocol converter are system dependent. The rest of this chapter describes the ARM9TDMI hardware debug extensions.

5.3 Debug interface signals

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

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

5.3.1 Entry into debug state on breakpoint

Any instruction being fetched for memory is latched at the end of phase 2. To apply a breakpoint to that instruction, the breakpoint signal must be asserted by the end of the following phase1. This minimizes the setup time, giving the EmbeddedICE macrocell an entire phase in which to perform the comparison. This is shown in Figure 5-2 on page 5-6.

External logic, such as additional breakpoint comparators, may be built to extend the functionality of the EmbeddedICE macrocell. Their output should be applied to the **IEBKPT** input. This signal is simply ORed with the internally generated **Breakpoint** signal before being applied to the ARM9TDMI core control logic.

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

The decode cycle of the debug entry sequence occurs during the execute cycle of the breakpointed instruction. The latched **Breakpoint** signal forces the processor to start the debug sequence.

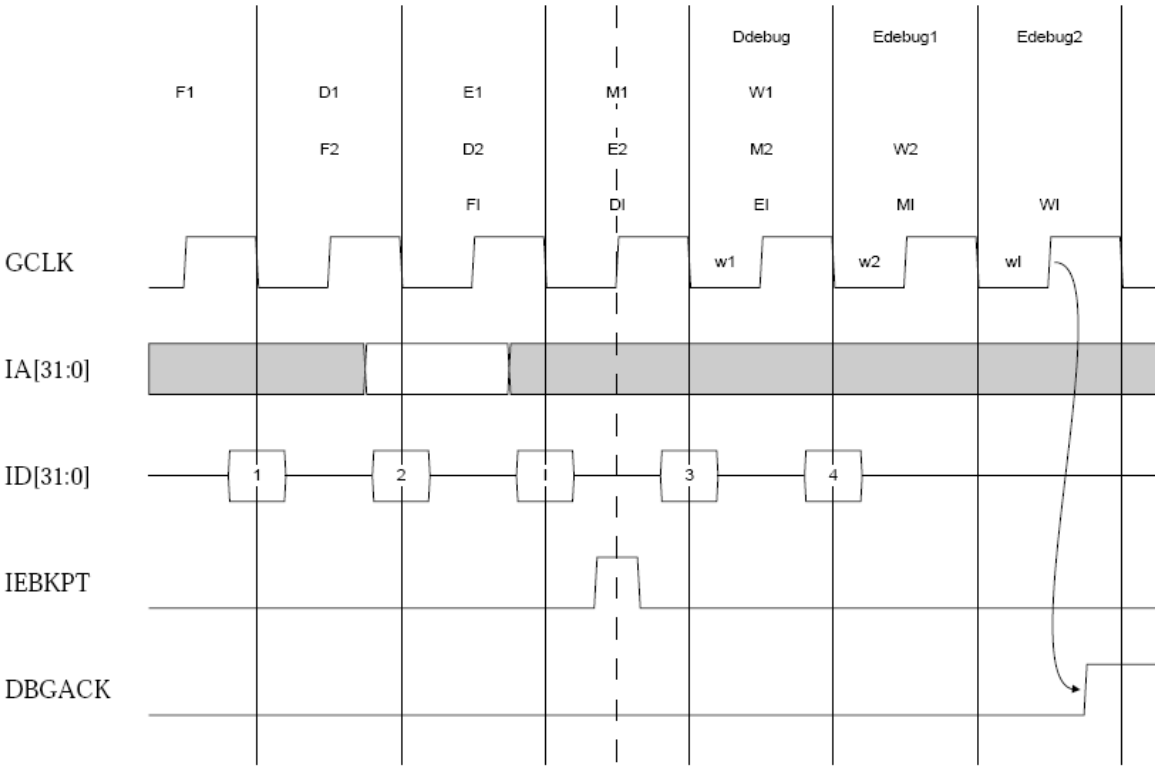


Figure 5-2 Breakpoint timing

5.3.2 Breakpoints and exceptions

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

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

On an instruction boundary, if there is a breakpointed instruction and an interrupt (**IRQ** or **FIQ**), the interrupt is taken and the breakpointed instruction is discarded. Once the interrupt has been serviced, the execution flow is returned to the original program.

This means that the instruction which was previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters debug state once it reaches the execute stage of the pipeline.

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

5.3.3 Watchpoints

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

After a watchpointed access, the next instruction in the processor pipeline is always allowed to complete execution. Where this instruction is a single-cycle data-processing instruction, entry into debug state is delayed for one cycle while the instruction completes. The timing of debug entry following a watchpointed load in this case is shown in Figure 5-3 on page 5-8.

Note

Although instruction 5 enters the execute state, it is not executed, and there is no state update as a result of this instruction. Once the debugging session is complete, normal continuation would involve a return to instruction 5, the next instruction in the code sequence which has not yet been executed.

The instruction following the instruction which generated the watchpoint could have modified the *Program Counter (PC)*. If this has happened, it will not be possible to determine the instruction which caused the watchpoint. A timing diagram showing debug entry after a watchpoint where the next instruction is a branch is shown in Figure 5-4 on page 5-9. However, it is always possible to restart the processor.

Once the processor has entered debug state, the ARM9TDMI core may be interrogated to determine its state. In the case of a watchpoint, the PC contains a value that is five instructions on from the address of the next instruction to be executed. Therefore, if on entry to debug state, in ARM state, the instruction SUB PC, PC, #20 is scanned in and the processor restarted, execution flow would return to the next instruction in the code sequence.

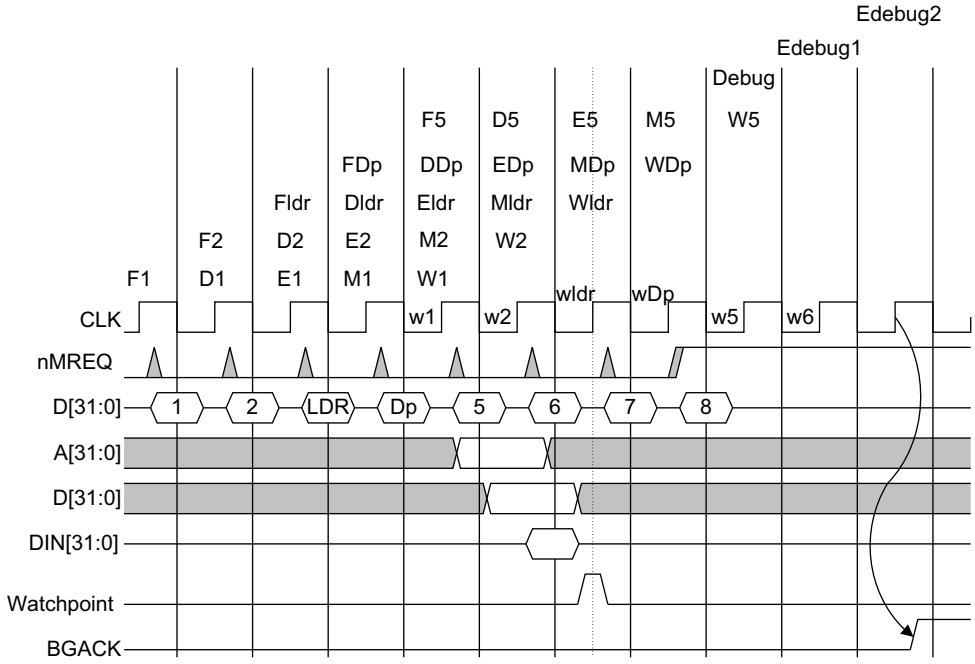


Figure 5-3 Watchpoint entry with data processing instruction

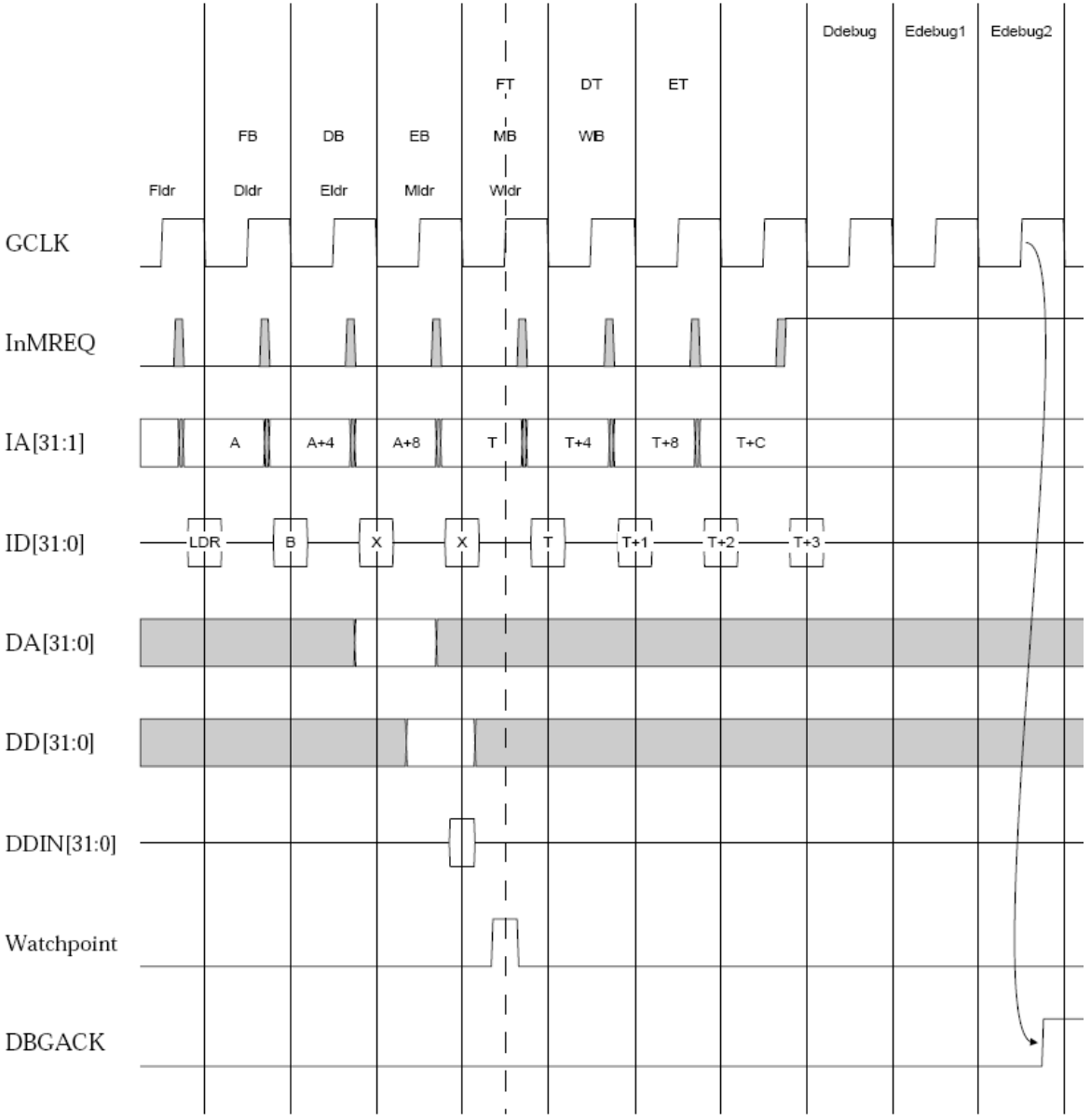


Figure 5-4 Watchpoint entry with branch

5.3.4 Watchpoints and exceptions

If there is an abort with the data access as well as a watchpoint, the watchpoint condition is latched, the exception entry sequence performed, and then the processor enters debug state. If there is an interrupt pending, again the ARM9TDMI allows the exception entry sequence to occur and then enters debug state.

5.3.5 Debug request

A debug request can take place through the EmbeddedICE macrocell or by asserting the **EDBGRQ** signal. The request is synchronized and passed to the processor. Debug request takes priority over any pending interrupt. Following synchronization, the core will enter debug state when the instruction at the execution stage of the pipeline has completely finished executing (once memory and write stages of the pipeline have completed). While waiting for the instruction to finish executing, no more instructions will be issued to the execute stage of the pipeline.

5.3.6 Actions of the ARM9TDMI in debug state

Once the ARM9TDMI is in debug state, both memory interfaces will indicate internal cycles. This allows the rest of the memory system to ignore the ARM9TDMI and function as normal. Since the rest of the system continues operation, the ARM9TDMI will ignore aborts and interrupts.

The **BIGEND** signal should not be changed by the system while in debug state. If it changes, not only will there be a synchronization problem, but the programmer's view of the ARM9TDMI will change without the knowledge of the debugger. The **nRESET** signal must also be held stable during debug. If the system applies reset to the ARM9TDMI (**nRESET** is driven LOW), the state of the ARM9TDMI will change without the knowledge of the debugger.

When instructions are executed in debug state, the ARM9TDMI will change asynchronously to the memory system outputs (except for **InMREQ**, **ISEQ**, **DnMREQ**, and **DSEQ** which change synchronously from GCLK). For example, every time a new instruction is scanned into the pipeline, the instruction address bus will change. If the instruction is a load or store operation, the data address bus will change as the instruction executes. Although this is asynchronous, it should not affect the system, because both interfaces will be indicating internal cycles. Care must be taken with the design of the memory controller to ensure that this does not become a problem.

5.4 Scan chains and JTAG interface

There are three scan chains inside the ARM9TDMI. These allow testing, debugging and programming of the EmbeddedICE macrocell watchpoint units. The scan chains are controlled by a JTAG-style *Test Access Port* (*TAP*) controller. In addition, support is provided for an optional fourth scan chain. This is intended to be used for an external boundary scan chain around the pads of a packaged device. The signals provided for this scan chain are described on *Scan chain 3* on page 5-22.

The three scan chains of the ARM9TDMI are referred to as scan chain 0, 1 and 2.

Note

The ARM9TDMI TAP controller supports 32 scan chains. Scan chains 0 to 15 have been reserved for use by ARM. Any extension scan chains should be implemented in the remaining space. The **SCREG[4:0]** signals indicate which scan chain is being accessed.

5.5 The JTAG state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure 5-5 shows the state transitions that occur in the TAP controller.

The state numbers are also shown on the diagram. These are output from the ARM9TDMI on the **TAPSM[3:0]** bits.

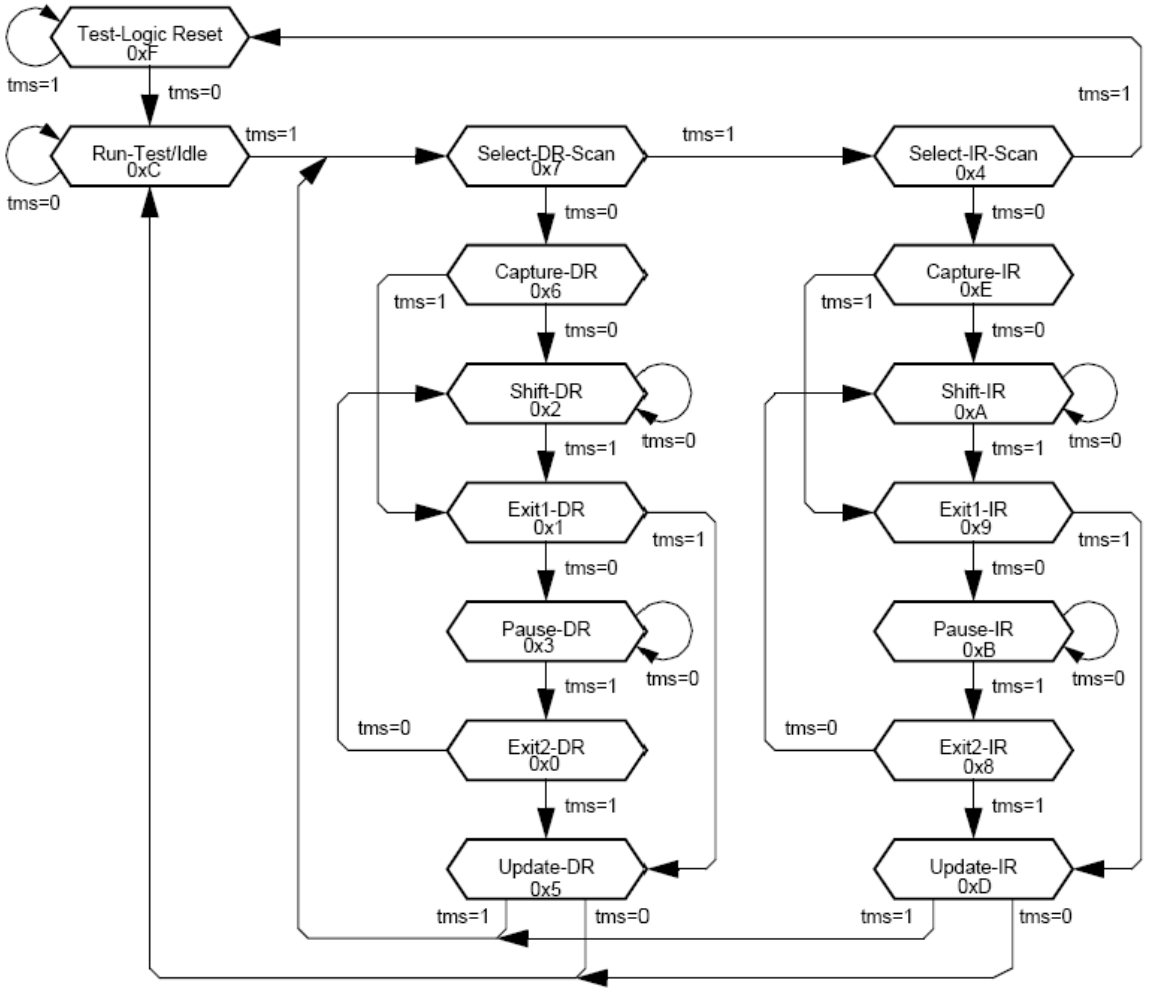


Figure 5-5 Test access port (TAP) controller state transitions

5.5.1 Reset

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

Note

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

The action of reset is as follows:

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

5.5.2 Pullup resistors

The IEEE 1149.1 standard effectively requires **TDI** and **TMS** to have internal pullup resistors. In order to minimize static current draw, these resistors are not fitted to the ARM9TDMI. Accordingly, the four inputs to the test interface (the **TDO**, **TDI** and **TMS** signals plus **TCK**) must all be driven to valid logic levels to achieve normal circuit operation.

5.5.3 Instruction register

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

5.5.4 Public instructions

The following public instructions are supported:

Table 5-1 Public instructions

Instruction	Binary code
EXTEST	0000
SCAN_N	0010
INTEST	1100

Table 5-1 Public instructions (continued)

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

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

EXTEST (0000)

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

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

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

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

SCAN_N (0010)

This instruction connects the scan path select register between **TDI** and **TDO**.

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

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

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

INTEST (1100)

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

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

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

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

IDCODE (1110)

The IDCODE instruction connects the device identification register (or ID register) between **TDI** and **TDO**. The ID register is a 32-bit register that allows the manufacturer, part number, and version of a component to be determined through the TAP. The ID register is loaded from the **TAPID[31:0]** input bus, which should be tied to a constant value being the unique JTAG IDCODE for the device.

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

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

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

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

BYPASS (1111)

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

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

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

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

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

Note

All unused instruction codes default to the BYPASS instruction.

CLAMP (0101)

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

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

Note

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

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

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

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

HIGHZ (0111)

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

When the HIGHZ instruction is loaded into the instruction register, all ARM9TDMI outputs are 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.

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

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

CLAMPZ (1001)

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

When the CLAMPZ instruction is loaded into the instruction register and scan chain 0 is selected, all the 3-state outputs (as described above) are placed in their inactive state, but the data supplied to the outputs is derived from the scan cells. The purpose of this instruction is to ensure that, during production test, each output can be disabled when its data value is either a logic 0 or logic 1.

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

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

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

SAMPLE/PRELOAD (0011)

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

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

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

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

RESTART (0100)

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

5.6 Test data registers

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

- *Bypass register*
- *ARM9TDMI device identification (ID) code register*
- *Instruction register* on page 5-19
- *Scan chain select register* on page 5-19
- *Scan chains 0, 1, 2, and 3* on page 5-20.

These are described below.

5.6.1 Bypass register

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

Length 1 bit.

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

5.6.2 ARM9TDMI device identification (ID) code register

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

Length 32 bits.

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

The IEEE format of the ID register is as shown in Figure 5-2 on page 5-6:

Table 5-2 ID code register

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

5.6.3 Instruction register

Purpose Changes the current TAP instruction.

Length 4 bits.

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

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

5.6.4 Scan chain select register

Purpose Changes the current active scan chain.

Length 5 bits.

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

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

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

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

The number of the currently selected scan chain is reflected on the **SCREG[4:0]** output bus. The TAP controller may be used to drive external scan chains in addition to those within the ARM9TDMI macrocell. The external scan chain must be assigned a number and control signals for it, and can be derived from **SCREG[4:0]**, **IR[3:0]**, **TAPSM[3:0]**, **TCK1** and **TCK2**.

The list of scan chain numbers allocated by ARM are shown in Table 5-3. An external scan chain may take any other number. The serial data stream applied to the external scan chain is made present on **SDIN**. The serial data back from the scan chain must be presented to the TAP controller on the **SDOUTBS** input.

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

Table 5-3 Scan chain number allocation

Scan chain number	Function
0	Macrocell scan test
1	Debug
2	EmbeddedICE macrocell programming
3	External boundary scan
4–15	Reserved
16–31	Unassigned

5.6.5 Scan chains 0, 1, 2, and 3

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

Scan chain 0

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

Length 184 bits.

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

- During CAPTURE-DR, the current outputs from the core's logic are captured in the output cells.
- During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known stimuli to the inputs.
- During RUN-TEST/IDLE, the core is clocked. Normally, the TAP controller should only spend one cycle in RUN-TEST/IDLE. The whole operation may then be repeated.

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

- During CAPTURE-DR, the current inputs to the core's logic from the system are captured in the input cells.
- During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known values on the core's outputs.
- During RUN-TEST/IDLE, the core is not clocked.

The operation may then be repeated.

Scan chain 1

Purpose Primarily for debugging, although it can be used for EXTEST on the data data bus **DD[31:0]** and the instruction data bus **ID[31:0]**. Scan chain 1 is selected via the SCAN_N TAP controller instruction.

Length 67 bits.

This scan chain is 67 bits long, 32 bits for data values, 32 bits for instruction data, and three additional bits, SYSSPEED, **DDEN** and an used bit. The three bits serve four different purposes:

- Under normal INTEST test conditions, the **DDEN** signal can be captured and examined.
- During EXTEST test conditions, a known value can be scanned into **DDEN** to be driven into the rest of the system. If a logic 1 is scanned into **DDEN**, the data data bus **DD[31:0]** will drive out the values stored in its scan cells. If a logic 0 is scanned into **DDEN**, **DD[31:0]** will capture the current input values.

- While debugging, the value placed in the SYSSPEED control bit determines whether the ARM9TDMI synchronizes back to system speed before executing the instruction.
- After the ARM9TDMI has entered debug state, the first time SYSSPEED is captured and scanned out, its value tells the debugger whether the core has entered debug state due to a breakpoint (SYSSPEED LOW), or a watchpoint (SYSSPEED HIGH).

Scan chain 2

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

Length 38 bits.

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

No action is taken during CAPTURE-DR.

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

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

Scan chain 3

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

Length User-defined.

Scan chain 3 is provided so that an optional external boundary scan chain may be controlled via the ARM9TDMI. Typically this would be used for a scan chain around the pad ring of a packaged device. The following control signals are provided and are generated only when scan chain 3 has been selected. These outputs are inactive at all other times.

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

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

ICAPCLKBS, ECAPCLKBS

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

SHCLK1BS, SHCLK2BS

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

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

5.7 ARM9TDMI core clocks

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

Note

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

The two cases in which the clocks switch are during debugging and during testing.

5.8 Clock switching during debug

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

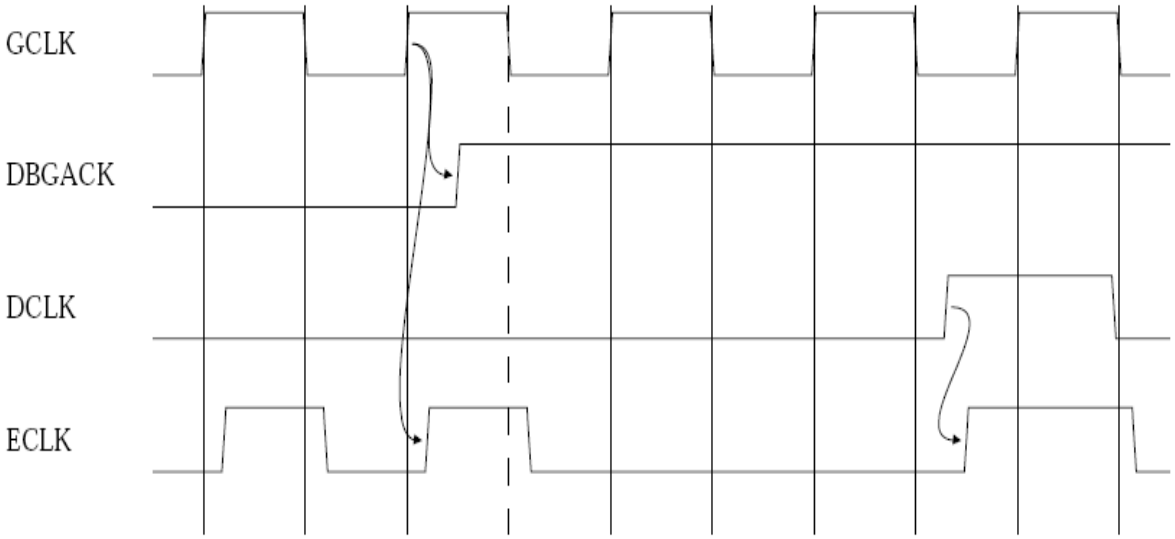


Figure 5-6 Clock switching on entry to debug state

The ARM9TDMI is forced to use **DCLK** as the primary clock until debugging is complete. On exit from debug, the core must be allowed to synchronize back to **GCLK**. This must be done in the following sequence. The final instruction of the debug sequence must be shifted into the instruction data bus scan chain, and clocked in by asserting **DCLK**. At this point, RESTART must be clocked into the TAP controller register.

The ARM9TDMI will now automatically resynchronize back to **GCLK** when the TAP controller enters the RUN-TEST/IDLE mode and start fetching instructions from memory at **GCLK** speed. For more information, refer to *Exit from debug state* on page 5-30.

5.9 Clock switching during test

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

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

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

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

5.10 Determining the core state and system state

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

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

5.10.1 Determining the core state

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

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

```
STR R0, [R1]; Save R0 before use
MOV R0, PC;   Copy PC into R0
STR R0, [R1]; Save the PC in R0
BX PC; Jump into ARM state
MOV R8, R8; NOP (no operation)
MOV R8, R8; NOP
```

The above use of R1 as the base register for the stores is for illustration only—any register could be used.

Since all Thumb instructions are only 16 bits long, the simplest course of action when shifting them into scan chain 1 is to repeat the instruction twice on the instruction data bus bits. For example, the encoding for `BX R0` is 0x4700. Thus, if 0x47004700 is shifted into the 32 bits of the instruction data bus of scan chain 1, the debugger does not have to track from which half of the bus the processor expects to read instructions.

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

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

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

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

After determining the values in the current bank of registers, it may be desirable to access banked registers. This can only be done by changing mode. Normally, a mode change may only occur if the core is already in a privileged mode. However, while in debug state, a mode change from any mode into any other mode may occur. Note that the debugger must restore the original mode before exiting debug state.

For example, assume that the debugger has been asked to return the state of the user mode and FIQ mode registers, and debug state was entered in supervisor mode.

The instruction sequence could be:

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

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

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

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

5.10.2 Determining system state

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

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

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

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

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

5.10.3 Instructions which may have the SYSSPEED bit set

The only valid instructions on which to set this bit are:

- loads
- stores
- load multiple
- store multiple.

When the ARM9TDMI returns to debug state after a system speed access, the SYSSPEED bit is set HIGH.

5.11 Exit from debug state

Leaving debug state involves restoring the internal state of the ARM9TDMI, causing a branch to the next instruction to be executed, and synchronizing back to **GCLK**. After restoring the internal state, a branch instruction must be loaded into the pipeline. For details on calculating the branch, see *The behavior of the program counter during debug* on page 5-33.

Bit 33 of scan chain 1 is used to force the ARM9TDMI to resynchronize back to **GCLK**. The penultimate instruction in the debug sequence is a branch to the instruction at which execution is to resume. This is scanned in with bit 33 set LOW. The core is then clocked to load the branch into the pipeline. The final instruction to be scanned in is a NOP (such as `MOV R0, R0`), with bit 33 set HIGH. The core is then clocked to load this instruction 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 will revert back to system mode and clock resynchronization to **GCLK** will occur within the ARM9TDMI. Normal operation will then resume, with instructions being fetched from memory.

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

The function of **DBGACK** is to tell the rest of the system when the ARM9TDMI is in debug state. This can be used to inhibit peripherals such as watchdog timers that have real time characteristics. Also, with a small amount of external logic, **DBGACK** can be used to mask out all memory accesses caused by the debugging process, so that the same number of memory accesses are seen independent of debug entry. This, however, is only possible if debugging is performed through breakpoints. It is not possible to precisely mask memory accesses due to debugging if watchpoints are used.

For example, when the ARM9TDMI enters debug state after a breakpoint, the instruction pipeline contains the breakpointed instruction plus two other instructions which have been prefetched. On entry to debug state the pipeline is flushed. So, on exit from debug state, the pipeline must be refilled to its previous state. Therefore, because of the debugging process, more memory accesses occur than would normally be expected. Through the use of **DBGACK**, together with a small amount of external logic it is possible for a peripheral that simply counts the number of instruction fetches to return the same answer after a program has run both with and without debugging.

It can be seen in Figure 5-8 on page 5-32 that two instructions are fetched after that which breakpoints. Figure 5-7 on page 5-31, shows **DBGACK** normally masks the first three instruction fetches out of the debug state, corresponding to the breakpoint instruction, and the two instructions prefetched after it. Since under some circumstances

DBGACK may remain HIGH for more than three instruction fetches, if precise instruction access counting is required, some external logic must generate a modified **DBGACK** that always falls after three instruction fetches.

Note

When a system speed access occurs, **DBGACK** remains HIGH throughout the system speed memory accesses. It then falls after the system speed memory accesses are completed, and finally rises again as the processor re-enters debug state. Therefore **DBGACK** masks all system speed memory accesses.

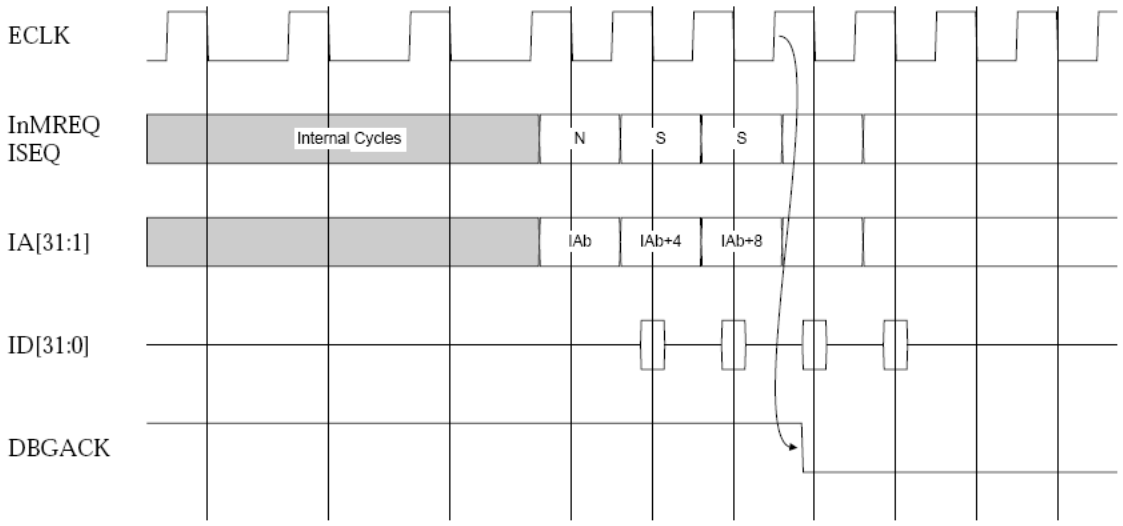


Figure 5-7 Debug exit sequence

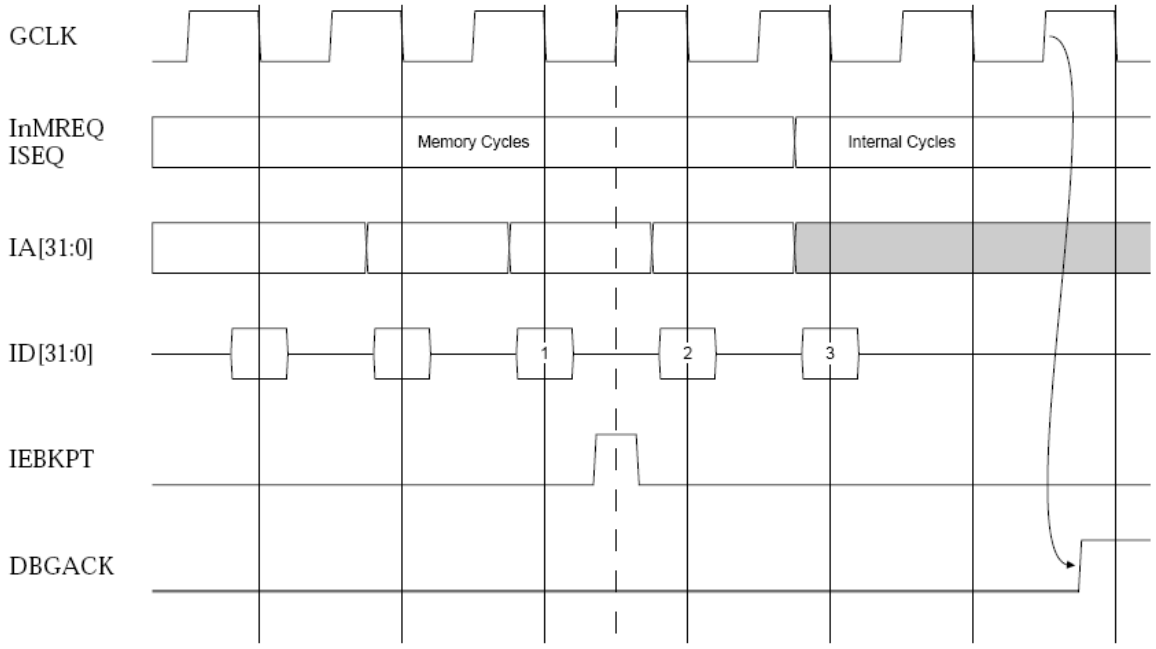


Figure 5-8 Debug state entry

5.12 The behavior of the program counter during debug

To force the ARM9TDMI to branch back to the place at which program flow was interrupted by debug, the debugger must keep track of what happens to the PC. There are six cases:

- *Breakpoint*
- *Watchpoint*
- *Watchpoint with another exception* on page 5-34
- *Watchpoint and breakpoint* on page 5-34
- *Debug request* on page 5-34
- *System speed accesses* on page 5-35.

In each case the same equation is used to determine where to resume execution. These are explained below.

5.12.1 Breakpoint

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

For example, if the ARM9TDMI entered debug state from a breakpoint set on a given address and two debug speed instructions were executed, a branch of 7 addresses must occur (four for debug entry, plus two for the instructions, plus one for the final branch). The following sequence shows ARM instructions scanned into scan chain 1. This is the *Most Significant Bit* (MSB) first, so the first digit represents the value to be scanned into the SYSSPEED bit, followed by the instruction.

```
0 EAFFFFF9 ; B -7 addresses (two's complement)
1 E1A00000 ; NOP (MOV R0, R0), SYSSPEED bit is set
```

For small branches, the final branch could be replaced with a subtract with the PC as the destination. For example, `SUB PC, PC, #28` for ARM code.

5.12.2 Watchpoint

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

5.12.3 Watchpoint with another exception

If a watchpoint access simultaneously causes a data abort, the ARM9TDMI will enter debug state in abort mode. Entry into debug is held off until the core has changed into abort mode, and fetched the instruction from the abort vector.

A similar sequence is followed when an interrupt, or any other exception, occurs during a watchpointed memory access. The ARM9TDMI will enter debug state in the mode of the exception, 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 did take place, the user should be given the choice of whether to service the exception before debugging.

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

```
0 EAFFFFF1; B -15 addresses (two's complement)
1 E1A00000; NOP (MOV R0, R0), SYSSPEED bit is set
```

This will force a branch back to the abort vector, causing the instructions at that location to be refetched and executed. Note that after the abort service routine, the instruction that caused the abort and watchpoint will be re-executed. This will cause the watchpoint to be generated and hence the ARM9TDMI will enter debug state again.

5.12.4 Watchpoint and breakpoint

It is possible to have a watchpoint and breakpoint condition occurring simultaneously. This can happen when an instruction causes a watchpoint, and the following instruction has been breakpointed. The same calculation should be performed as for *Breakpoint* on page 5-33 to determine where to resume. In this case, it will be at the breakpoint instruction, since this has not been executed.

5.12.5 Debug request

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

For example, the following sequence handles a situation in which the user has invoked a debug request, and decides to return to program execution immediately:

```
0 EAFFFFFB; B -5 addresses (2's complement)
1 E1A00000; NOP (MOV R0, R0), SYSSPEED bit is set
```


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

5.12.6 System speed accesses

If a system speed access is performed during debug state, the value of the PC is increased by five addresses. Since system speed instructions access the memory system, it is possible for aborts to take place. If an abort occurs during a system speed memory access, the ARM9TDMI enters abort mode before returning to debug state.

This is similar to an aborted watchpoint. However, 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 that caused the abort, and hence the abort address. In this case, the value of the PC is invalid, but the debugger will know the address of the location that was being accessed. Thus the debugger can be written to help the abort handler fix the memory system.

5.12.7 Summary of return address calculations

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

`-(4 + N +5S)`

where `N` is the number of debug speed instructions executed (including the final branch), and `S` is the number of system speed instructions executed.

5.13 EmbeddedICE macrocell

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

Because the ARM9TDMI processor core has a Harvard Architecture, the user must specify whether the watchpoint registers examine the instruction or the data interface. This is specified by bit 3:

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

There can be no don't care case for this bit because the comparators cannot compare the values on both buses simultaneously. Therefore, bit 3 of the control mask register is always clear and cannot be programmed HIGH. Bit 3 also determines whether the internal **Breakpoint** or **Watchpoint** signal should be driven by the result of the comparison. Figure 5-9 on page 5-38 gives an overview of the operation of the EmbeddedICE macrocell.

The ARM9TDMI EmbeddedICE macrocell has logic that allows single stepping through code. This reduces the work required by an external debugger, and removes the need to flush the instruction cache. There is also hardware to allow efficient trapping of accesses to the exception vectors. These blocks of logic free the two general-purpose hardware breakpoint/watchpoint units for use by the programmer at all times.

The general arrangement of the EmbeddedICE macrocell is shown in Figure 5-9 on page 5-38.

5.13.1 Register map

The EmbeddedICE macrocell register map is shown below:

Table 5-4 ARM9TDMI EmbeddedICE macrocell register map

Address	Width	Function
00000	4	Debug control
00001	5	Debug status
00010	8	Vector catch control
00100	6	Debug comms control
00101	32	Debug comms data

Table 5-4 ARM9TDMI EmbeddedICE macrocell register map (continued)

Address	Width	Function
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 1 address value
10001	32	Watchpoint 1 address mask
10010	32	Watchpoint 1 data value
10011	32	Watchpoint 1 data mask
10100	9	Watchpoint 1 control value
10101	8	Watchpoint 1 control mask

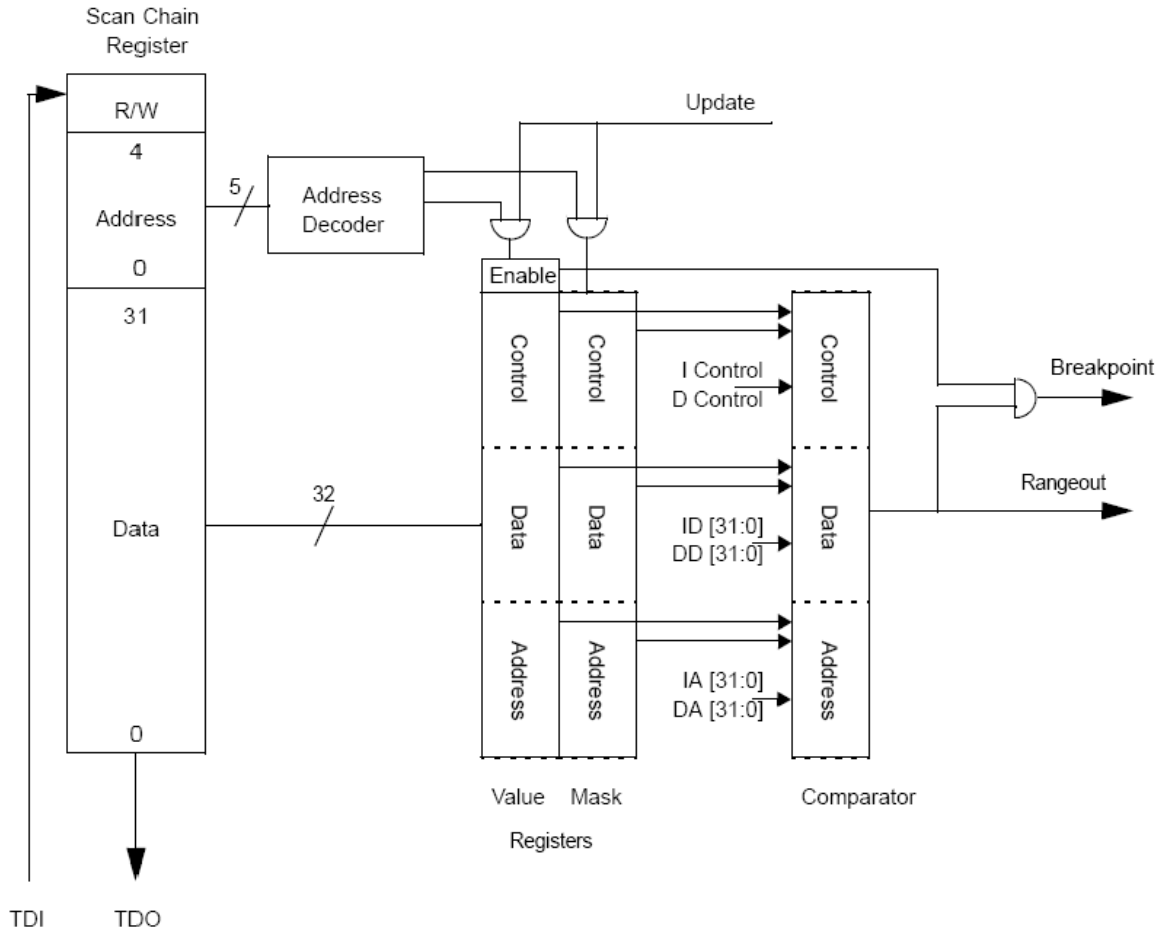


Figure 5-9 ARM9TDMI EmbeddedICE macrocell overview

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

5.13.2 Using the mask registers

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

5.13.3 Control registers

The format of the control registers depends on how bit 3 is programmed. If bit 3 is programmed to be 1, the breakpoint comparators examine the data address, data and control signals.

In this case, the format of the register is as shown in Figure 5-10.

Note

Bit 8 and bit 3 cannot be masked.

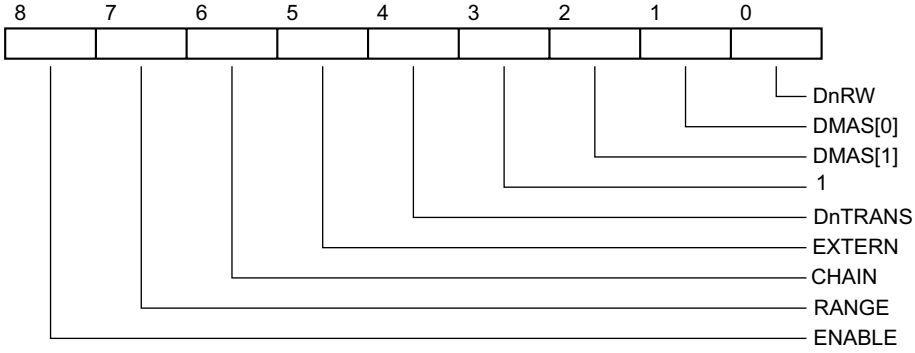


Figure 5-10 Watchpoint control register for data comparison

The bits have the following functions:

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

Bit	Function
DnRW	Compares against the data not read/write signal from the core in order to detect the direction of the data data bus activity. **nRW** is 0 for a read, and 1 for a write.
DMAS[1:0]	Compares against the **DMAS[1:0]** signal from the core in order to detect the size of the data data bus activity.
DnTRANS	Compares against the data not translate signal from the core in order to determine between a user mode (**DnTRANS** = 0) data transfer, and a privileged mode (**DnTRANS** = 1) transfer.

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

Bit	Function
EXTERN	Is an external input into the EmbeddedICE macrocell that allows the watchpoint to be dependent upon some external condition. The **EXTERN** input for watchpoint 0 is labelled **EXTERN0**, and the **EXTERN** input for watchpoint 1 is labelled **EXTERN1**.
CHAIN	Can be connected to chain output of another watchpoint in order to implement, for example, debugger requests of the form "breakpoint on address YYY only when in process XXX". In the ARM9TDMI EmbeddedICE macrocell, the **CHAINOUT** output of watchpoint 1 is connected to the **CHAIN** input of watchpoint 0. The **CHAINOUT** output is derived from a latch. The address/control field comparator drives the write enable for the latch and the input to the latch is the value of the data field comparator. The **CHAINOUT** latch is cleared when the control value register is written or when **nTRST** is LOW.
RANGE	Can be connected to the range output of another watchpoint register. In the ARM9TDMI EmbeddedICE macrocell, the Address comparator output from watchpoint 1 is connected to the **RANGE** input of watchpoint 0. This allows two watchpoints to be coupled for detecting conditions that occur simultaneously, for example, for range-checking.
ENABLE	If a watchpoint match occurs, the internal Watchpoint signal will only be asserted when the ENABLE bit is set. This bit only exists in the value register, it cannot be masked.

If bit 3 of the control register is programmed to 0, the comparators will examine the instruction address, instruction data and instruction control buses. In this case bits [1:0] of the mask register must be set to "don't care" (programmed to 11). The format of the register in this case is as shown in Figure 5-11 on page 5-41.

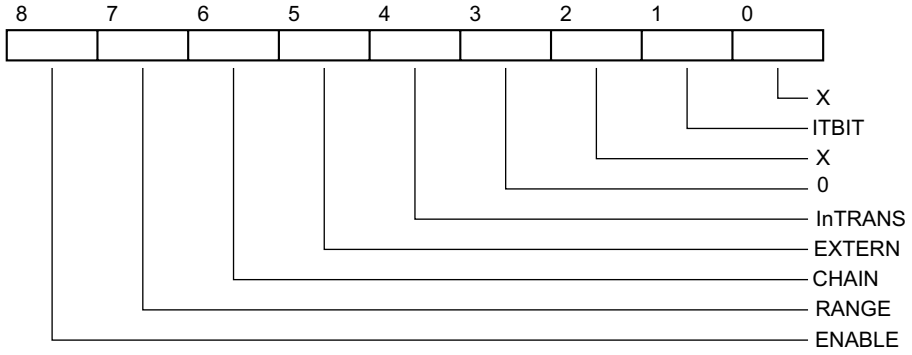


Figure 5-11 Watchpoint control register for instruction comparison

Table 5-6 Watchpoint control register for instruction comparison bit functions

Bit	Function
ITBIT	Compares against the Thumb state signal from the core to determine between a Thumb (**ITBIT** = 1) instruction fetch or an ARM (**ITBIT** = 0) fetch.
InTRANS	Compares against the not translate signal from the core in order to determine between a user mode (**InTRANS** = 0) instruction fetch, and a privileged mode (**InTRANS** = 1) fetch.
EXTERN	Is an external input into the EmbeddedICE macrocell that allows the watchpoint to be dependent upon some external condition. The **EXTERN** input for watchpoint 0 is labelled **EXTERN0**, and the **EXTERN** input for watchpoint 1 is labelled **EXTERN1**.

Table 5-6 Watchpoint control register for instruction comparison bit functions (continued)

Bit	Function
CHAIN	Can be connected to chain output of another watchpoint in order to implement, for example, debugger requests of the form “breakpoint on address YYY only when in process XXX”. In the ARM9TDMI EmbeddedICE macrocell, the **CHAINOUT** output of watchpoint 1 is connected to the **CHAIN** input of watchpoint 0. The **CHAINOUT** output is derived from a latch. The address/control field comparator drives the write enable for the latch, and the input to the latch is the value of the data field comparator. The **CHAINOUT** latch is cleared when the control value register is written, or when **nTRST** is LOW.
RANGE	Can be connected to the range output of another watchpoint register. In the ARM9TDMI EmbeddedICE macrocell, the RANGEOUT output of watchpoint 1 is connected to the RANGE input of watchpoint 0. This allows two watchpoints to be coupled for detecting conditions that occur simultaneously, for example, for range-checking.
ENABLE	If a watchpoint match occurs, the internal **Breakpoint** signal will only be asserted when the **ENABLE** bit is set. This bit only exists in the value register, it cannot be masked.

5.13.4 Debug control register

The ARM9TDMI debug control register is four bits wide and is shown in Figure 5-12.

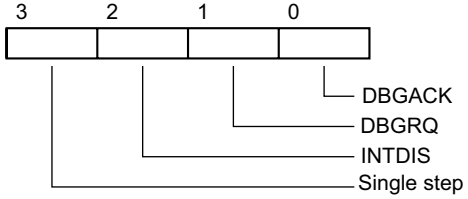


Figure 5-12 Debug control register

Bit 3 controls the single-step hardware, and this is explained in more detail in *Single stepping* on page 5-46.

5.13.5 Debug status register

The debug status register is five bits wide. It is a read only register and any writes will be ignored. If it is accessed for a read (with the read/write bit LOW), the status bits are read.

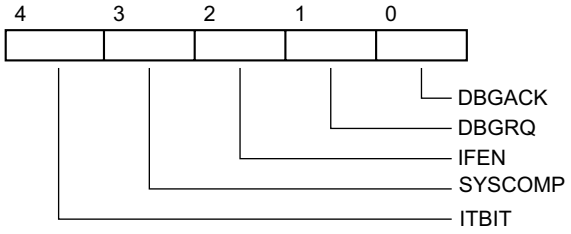


Figure 5-13 Debug status register

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

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

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

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

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

5.13.6 Vector catch register

The ARM9TDMI EmbeddedICE macrocell controls logic to enable accesses to the exception vectors to be trapped in an efficient manner. This is controlled by the vector catch register, as shown in Figure 5-14 on page 5-44. The functionality is described in *Vector catching* on page 5-45.

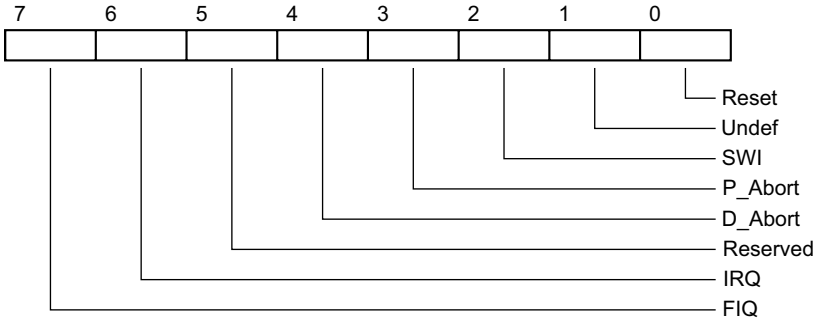


Figure 5-14 Vector catch register

5.14 Vector catching

The ARM9TDMI EmbeddedICE macrocell contains logic that allows efficient trapping of fetches from the vectors during exceptions. This is controlled by the Vector catch register. If one of the bits in this register is set HIGH and the corresponding exception occurs, the processor enters debug state as if a breakpoint has been set on an instruction fetch from the relevant exception vector.

For example, if the processor executes a SWI instruction while bit 2 of the Vector catch register is set, the ARM9TDMI fetches an instruction from location 0x8. The vector catch hardware detects this access and forces the internal **Breakpoint** signal HIGH into the ARM9TDMI control logic. This, in turn, forces the ARM9TDMI to enter debug state.

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

5.15 Single stepping

The ARM9TDMI EmbeddedICE macrocell contains logic that allows efficient single stepping through code. This leaves the macrocell watchpoint comparators free for general use.

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

5.16 Debug communications channel

The ARM9TDMI EmbeddedICE macrocell contains a communication channel for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The communications channel consists of a 32-bit wide comms data read register, a 32-bit wide comms data write register and a 6-bit wide comms control register for synchronized handshaking between the processor and the asynchronous debugger. These registers are located in fixed locations in the EmbeddedICE register map (as shown in Figure 5-9 on page 5-38) and are accessed from the processor via MCR and MRC instructions to coprocessor 14.

5.16.1 Debug comms channel registers

The debug comms control register is read only, and allows synchronized handshaking between the processor and the debugger.

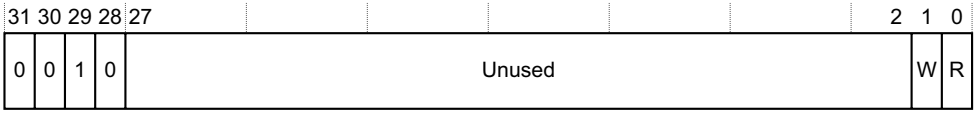


Figure 5-15 Debug comms control register

The function of each register bit is described below:

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

Bits 27:2 Unused.

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

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

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

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

Returns the debug comms control register into Rd.

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

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

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

Returns the debug data read register into Rd.

Note

The Thumb instruction set does not support coprocessors so the ARM9TDMI must be operated in ARM state in order to access the debug comms channel.

5.16.2 Communications via the comms channel

There are two methods of communicating via the comms channel, transmitting and receiving. The following descriptions detail their usage.

Sending a message to the debugger

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

It reads the debug comms control register to check status of the W bit.

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

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

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

As an alternative to polling, the debug comms channel can be interrupt driven by connecting the ARM9TDMI **COMMRX** and **COMMTX** signals to the systems interrupt controller.

Receiving a message from the debugger

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

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

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

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

Note

It is not possible to read EmbeddedICE registers through serialized vectors applied through scan chain 0.

Chapter 6
Test Issues

This chapter examines the test issues for the ARM9TDMI and lists the scan chain 0 bit order under the headings:

6.1 About testing

The ARM9TDMI processor core supports parallel and serial test methodologies. The parallel test patterns are derived from assembler ARM code programs written to achieve a high fault coverage.

The ARM9TDMI processor core has a fully JTAG-compatible scan chain which intersects all the inputs and outputs. This allows the test patterns to be serialized and injected to the processor via the JTAG interface. Both the parallel and serial test patterns are supplied to ARM9TDMI processor core licensees. The scan chain also supports EXTEST, allowing the connections between the ARM9TDMI processor core and other JTAG-compatible peripherals to be tested.

6.2 Scan chain 0 bit order

Table 6-1 Scan chain 0 bit order

Number	Signal	Direction
1	ID[0]	Input
2	ID[1]	Input
3:31	ID[2:30]	Input
32	ID[31]	Input
33	SYSSPEED	Internal
34	Unused	Internal
35	DDEN	Output
36	DD[31]	Bidirectional
37	DD[30]	Bidirectional
38:66	DD[29:1]	Bidirectional
67	DD[0]	Bidirectional
68	DA[31]	Output
69	DA[30]	Output
70:98	DA[29:1]	Output
99	DA[0]	Output
100	IA[31]	Output
101	IA[30]	Output
102:129	IA[29:2]	Output
130	IA[1]	Output
131	IEBKPT	Input
132	DEWPT	Input
133	EDBGRQ	Input
134	EXTERN0	Input
135	EXTERN1	Input

Table 6-1 Scan chain 0 bit order (continued)

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

Table 6-1 Scan chain 0 bit order (continued)

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

Chapter 7
Instruction Cycle Summary and Interlocks

This chapter gives the instruction cycle times and shows the timing diagrams for interlock timing:

7.1 Instruction cycle times

Table 7-1 describes the symbols used in tables.

Table 7-1 Symbols used in tables

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

Table 7-2 summarizes the ARM9TDMI instruction cycle counts and bus activity when executing the ARM instruction set.

Table 7-2 Instruction cycle bus times

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

Table 7-2 Instruction cycle bus times (continued)

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

Table 7-3 shows the instruction cycle times from the perspective of the data bus:

Table 7-3 Data bus instruction times

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

7.1.1 Multiplier cycle counts

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

- For ARM MUL, MLA, SMULL, SMLAL, and Thumb MUL, m is:
  1 if bits [31:8] of the multiplier operand are all zero or one
  2 if bits [31:16] of the multiplier operand are all zero or one
  3 if bits [31:24] of the multiplier operand are all zero or all one
  4 otherwise.
- For ARM UMULL, UMLAL, m is:
  1 if bits [31:8] of the multiplier operand are all zero
  2 if bits [31:16] of the multiplier operand are all zero
  3 if bits [31:24] of the multiplier operand are all zero
  4 otherwise.

7.2 Interlocks

Pipeline interlocks occur when the data required for an instruction is not available due to the incomplete execution of an earlier instruction. When an interlock occurs, instruction fetches stop on the instruction memory interface of the ARM9TDMI. Four examples of this are given below.

7.2.1 Example 1

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

```
LDR R0, [R1]
ADD R2, R0, R1
```

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

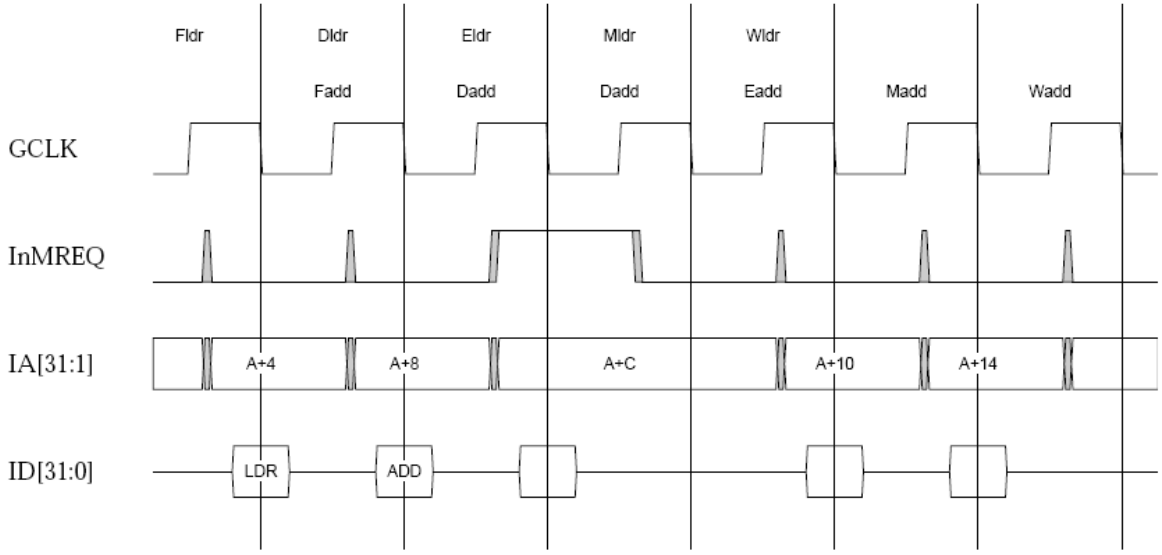


Figure 7-1 Single load interlock timing

7.2.2 Example 2

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

```
LDRB R0, [R1,#1]
ADD R2, R0, R1
```

Now, because a rotation must occur on the loaded data, there is a second interlock cycle. The behavior on the instruction memory interface is shown in Figure 7-2.

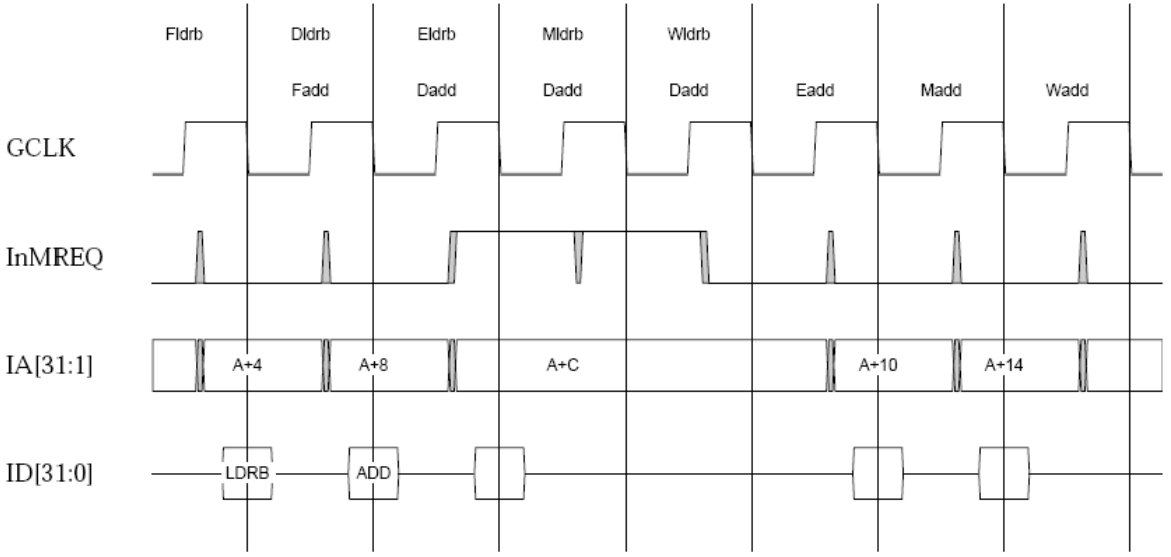


Figure 7-2 Two cycle load interlock

7.2.3 Example 3

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

```
LDM R12,{R1-R3}
ADD R2, R2, R1
```

The LDM takes three cycles to execute in the memory stage of the pipeline. The ADD is therefore delayed until the LDM begins its final memory fetch. The behavior of both the instruction and data memory interface are shown in Figure 7-3 on page 7-7.

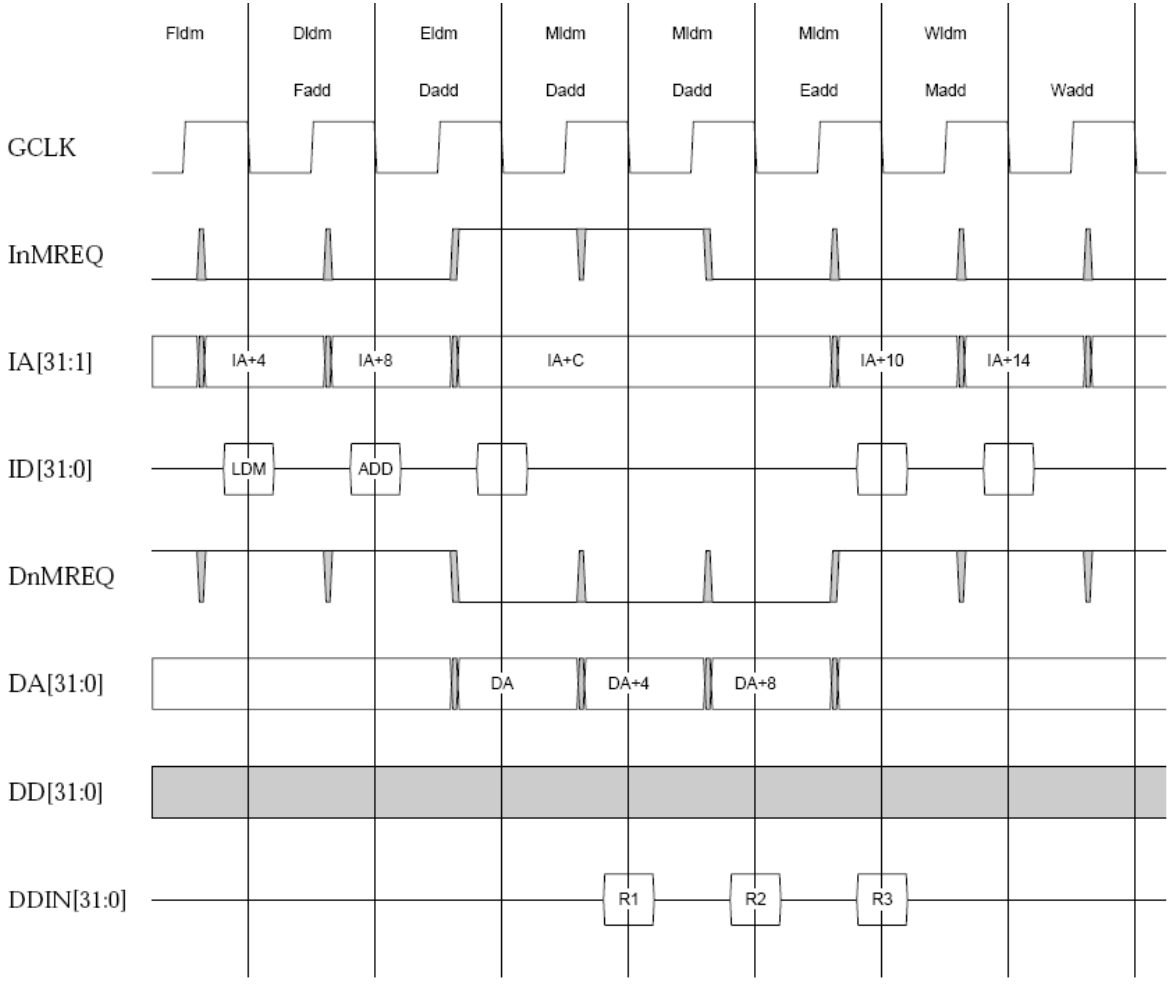


Figure 7-3 LDM interlock

7.2.4 Example 4

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

```
LDM R12,{R1-R3}
ADD R4, R3, R1
```

The code is the same code as in example 3, but in this instance the ADD instruction uses R3. Due to the nature of load multiples, the lowest register specified is transferred first, and the highest specified register last. Because the ADD is dependent on R3, there must be a further cycle of interlock while R3 is loaded. The behavior on the instruction and data memory interface is shown in Figure 7-4.

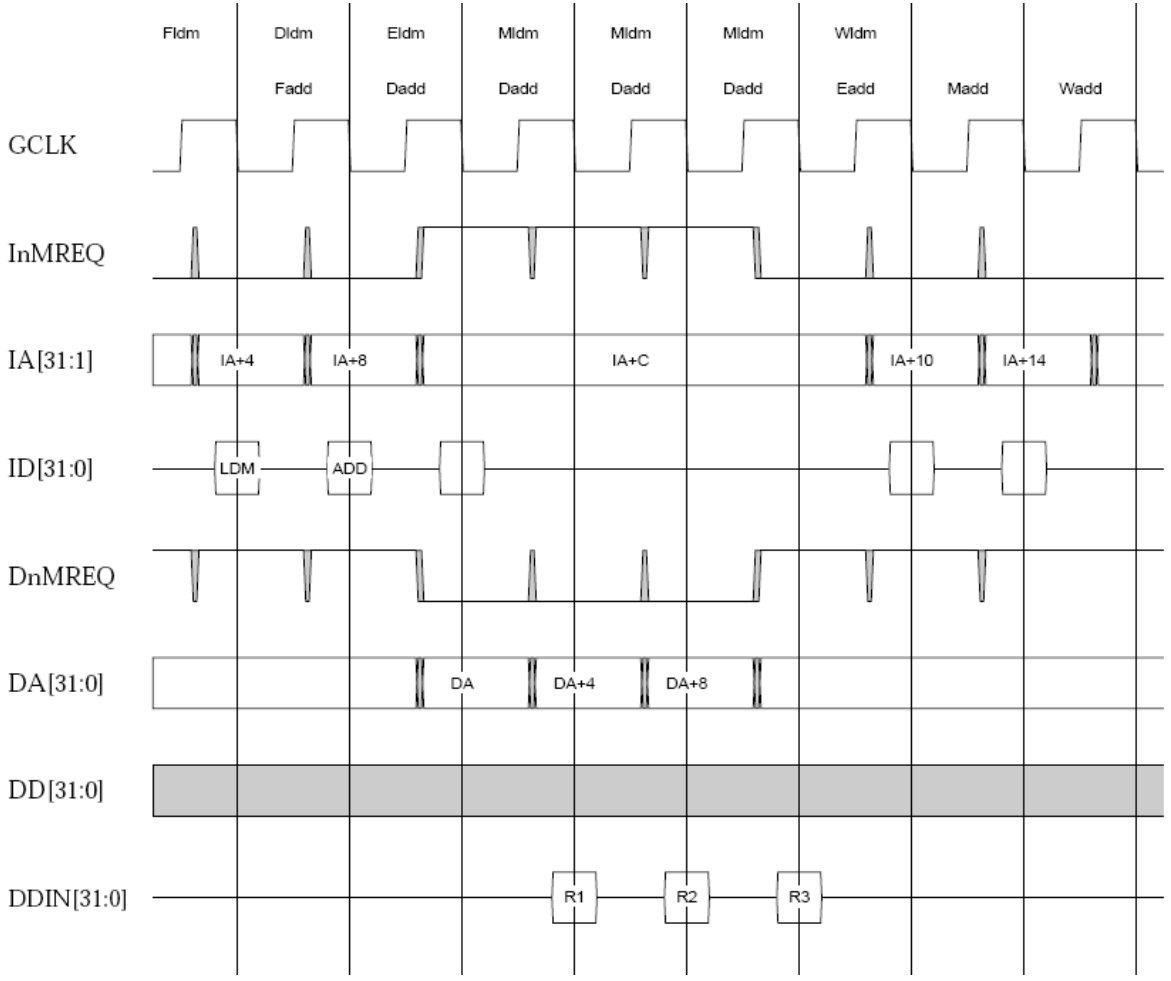


Figure 7-4 LDM dependent interlock

Chapter 8
ARM9TDMI AC Characteristics

This chapter gives the timing diagrams and timing parameters for the ARM9TDMI:

8.1 ARM9TDMI timing diagrams

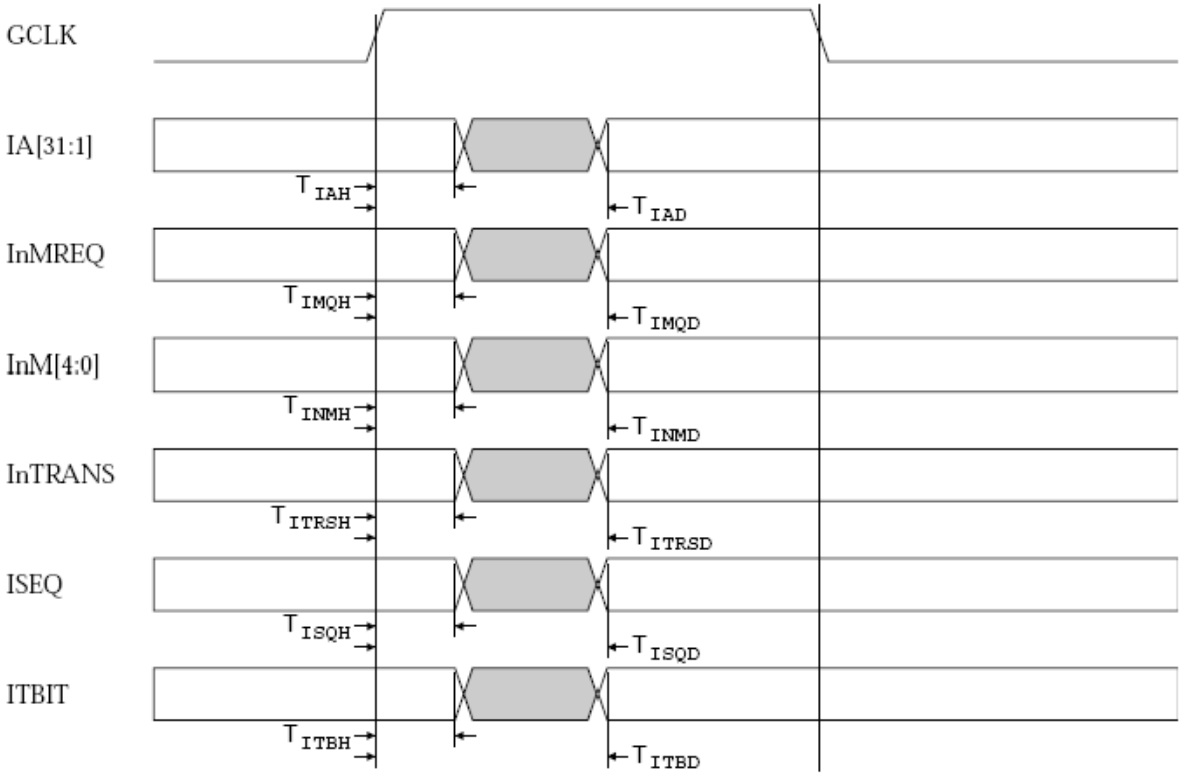


Figure 8-1 ARM9TDMI instruction memory interface output timing

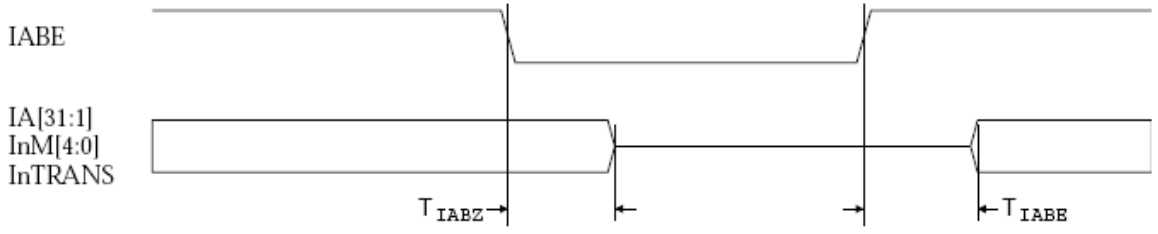


Figure 8-2 ARM9TDMI instruction address bus enable

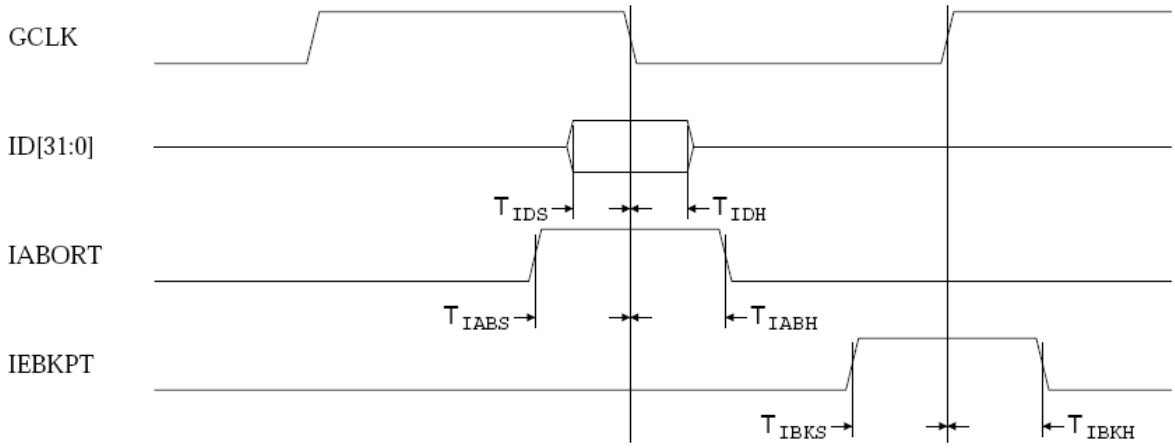


Figure 8-3 ARM9TDMI instruction memory interface input timing

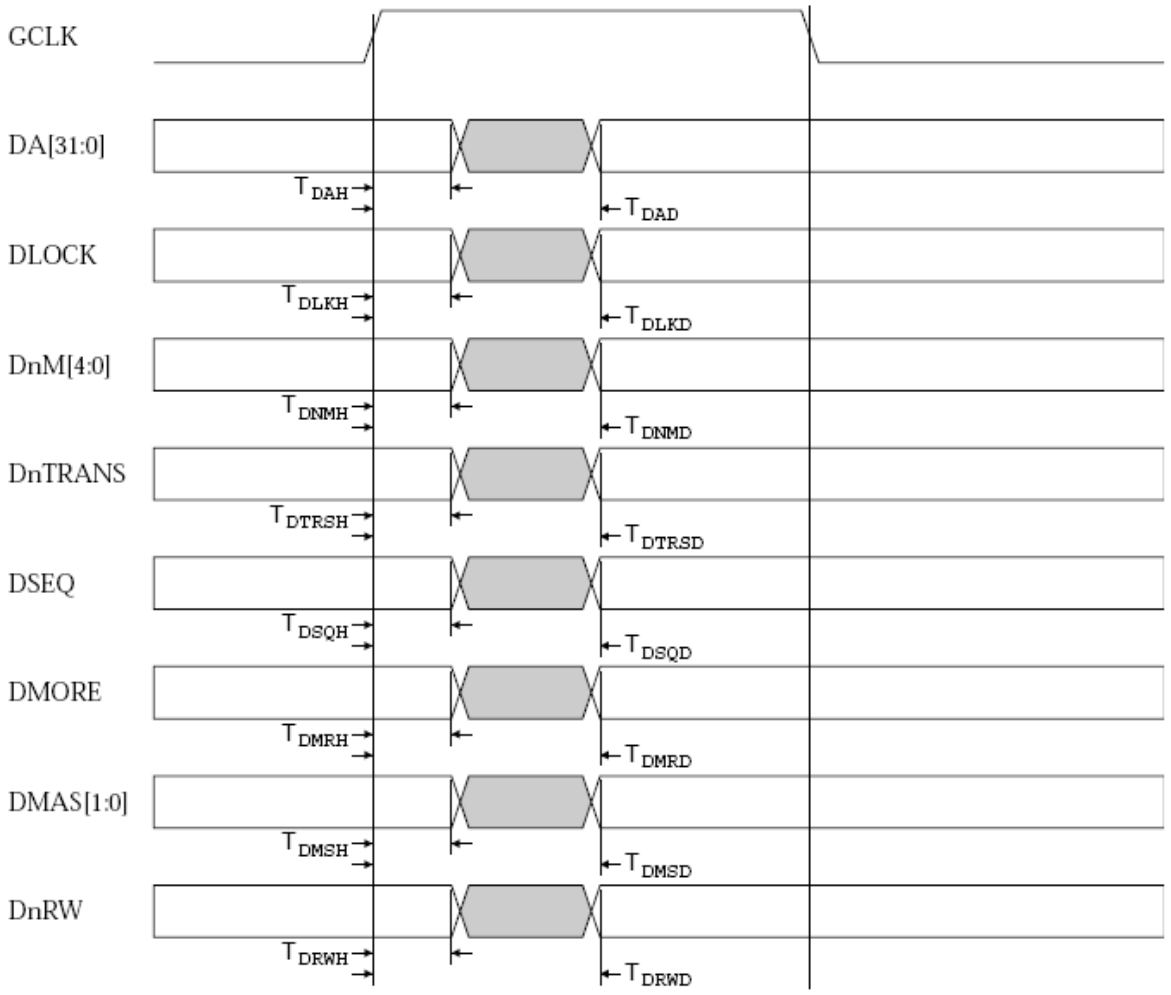


Figure 8-4 ARM9TDMI data memory interface output timing

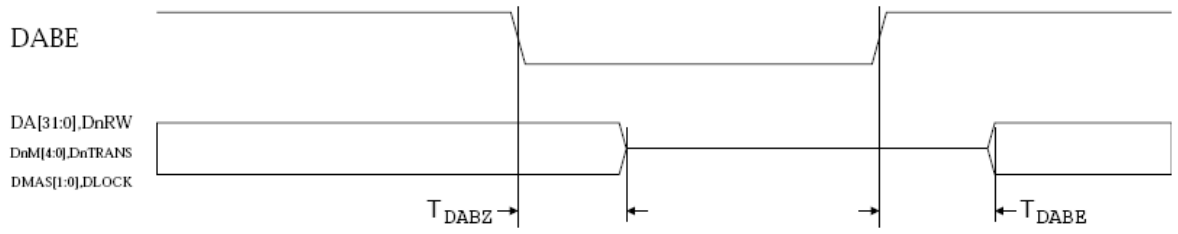


Figure 8-5 ARM9TDMI data address bus timing

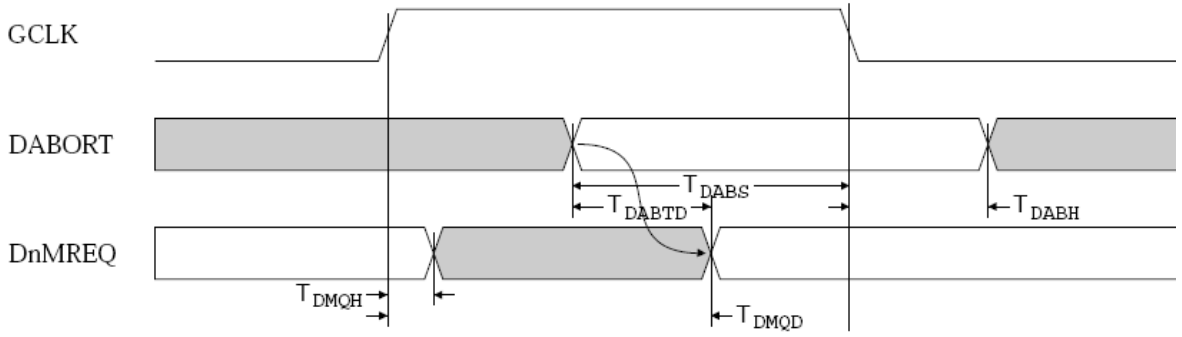


Figure 8-6 ARM9TDMI data ABORT and DnMREQ timing

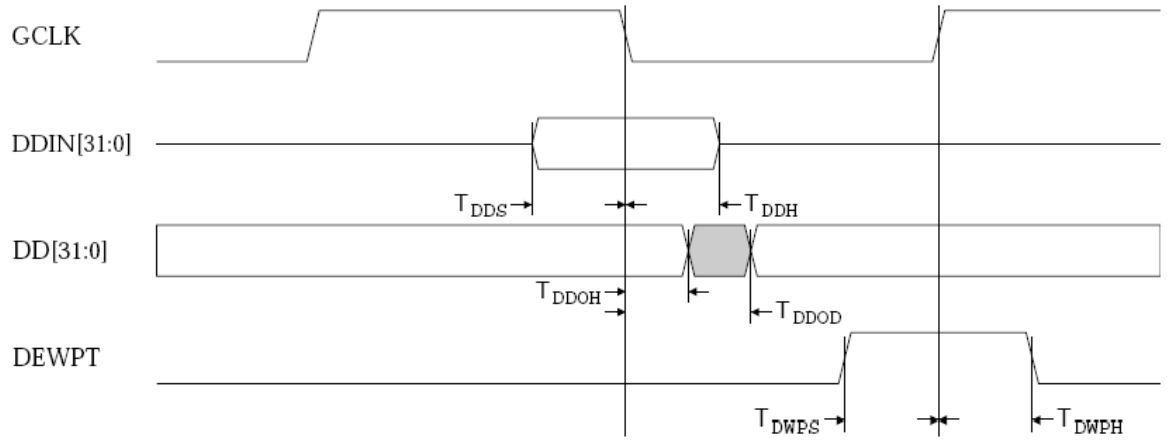


Figure 8-7 ARM9TDMI data data bus timing

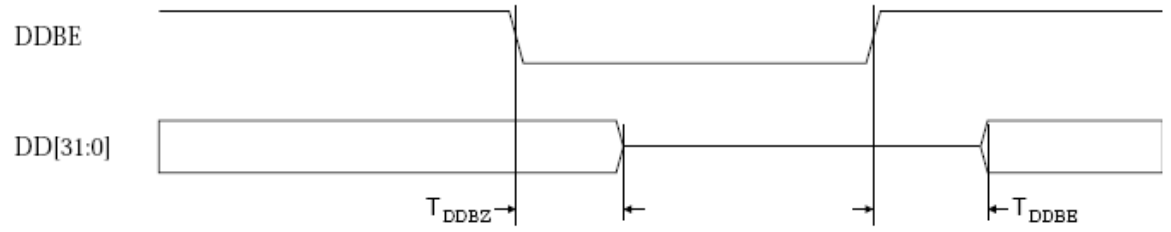


Figure 8-8 ARM9TDMI data bus enable

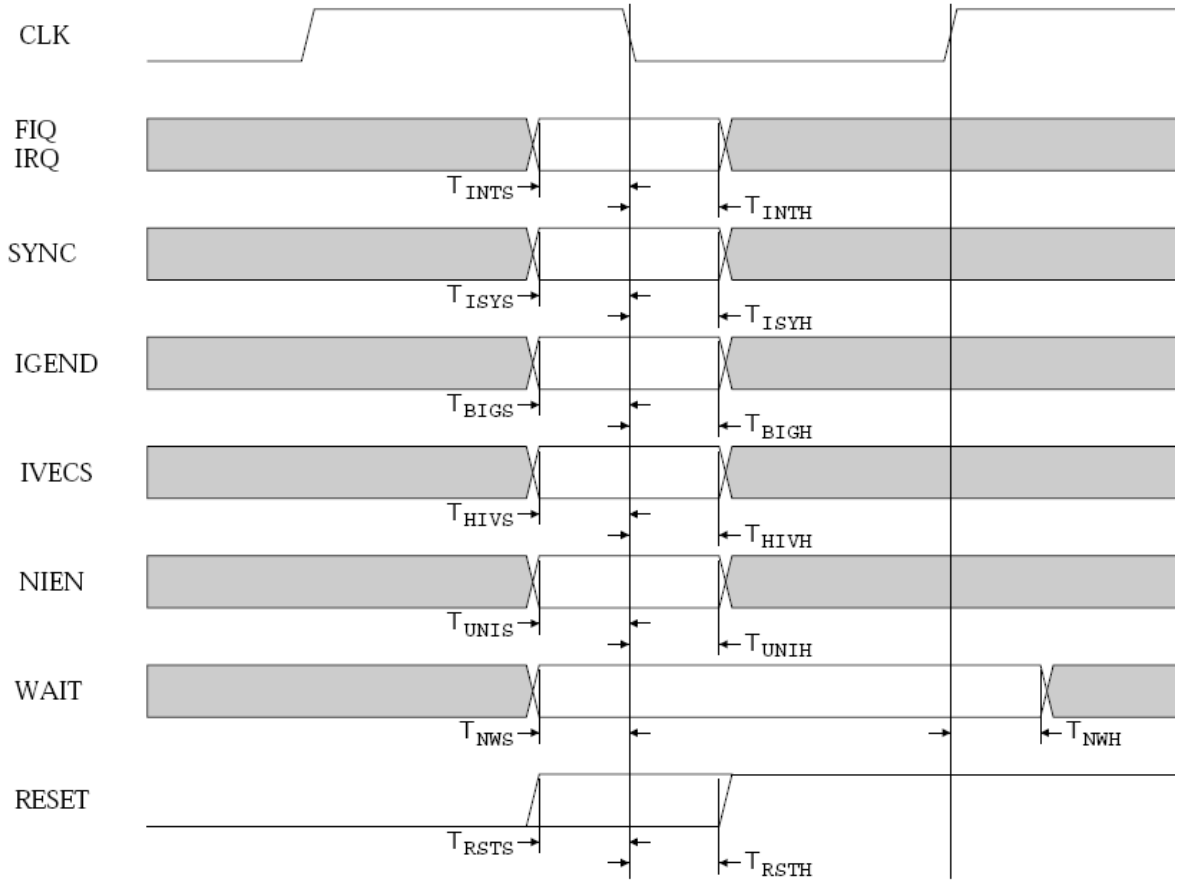


Figure 8-9 ARM9TDMI miscellaneous signal timing

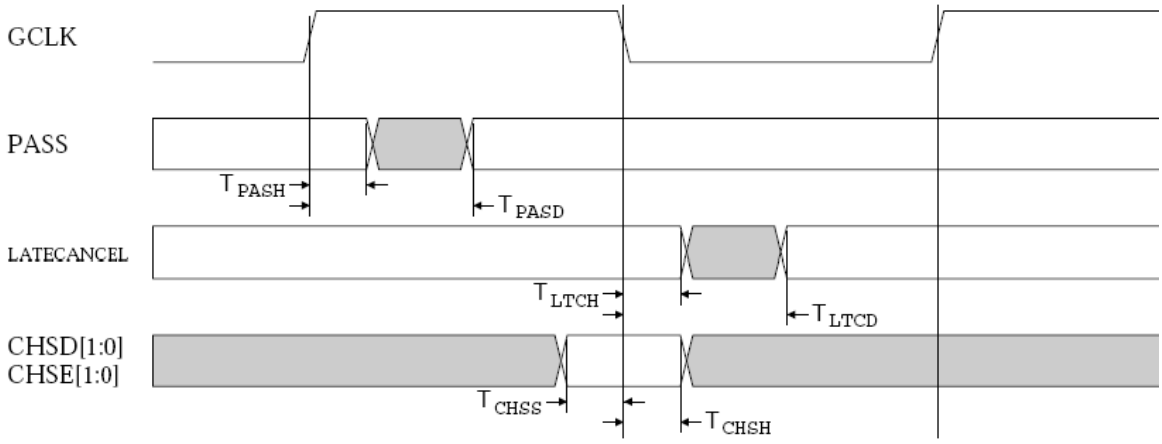


Figure 8-10 ARM9TDMI coprocessor interface signal timing

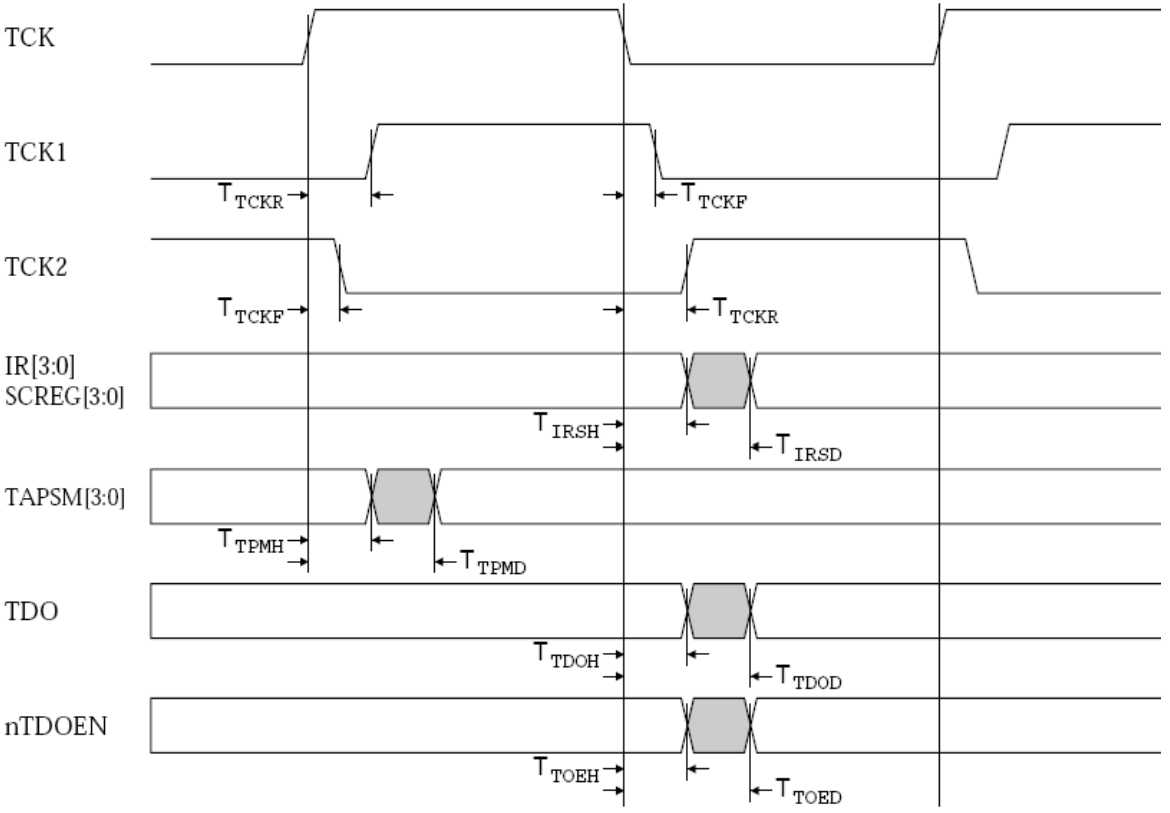


Figure 8-11 ARM9TDMI JTAG output signals

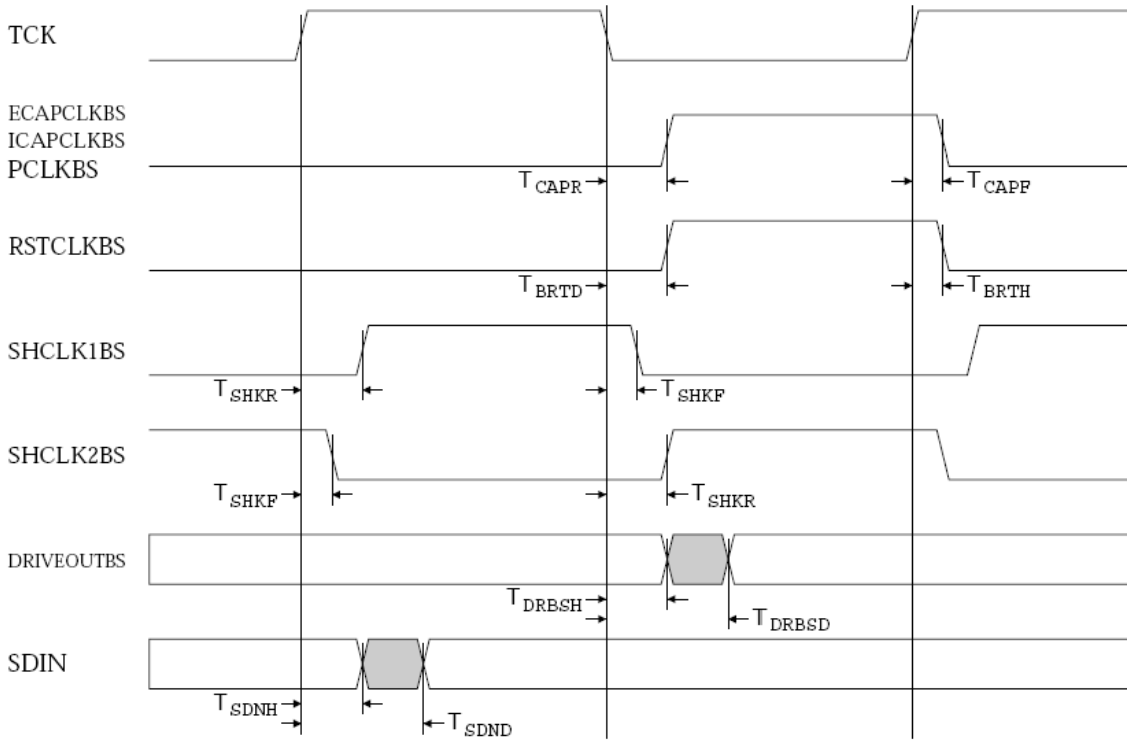


Figure 8-12 ARM9TDMI external boundary scan chain output signals

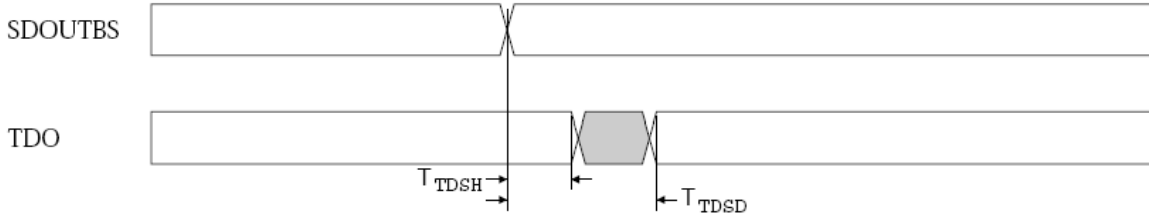


Figure 8-13 ARM9TDMI SDOUTBS to TDO relationship

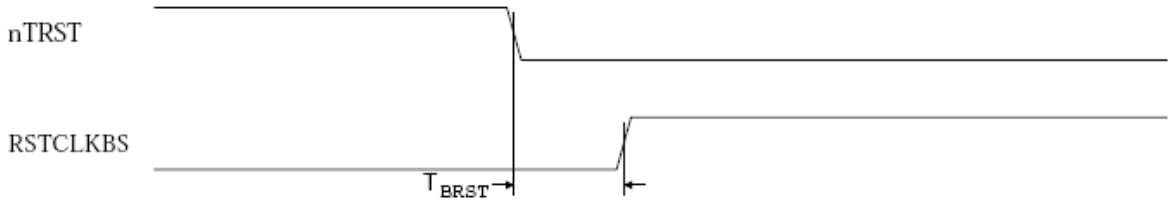


Figure 8-14 ARM9TDMI nTRST to RSTCLKBS relationship

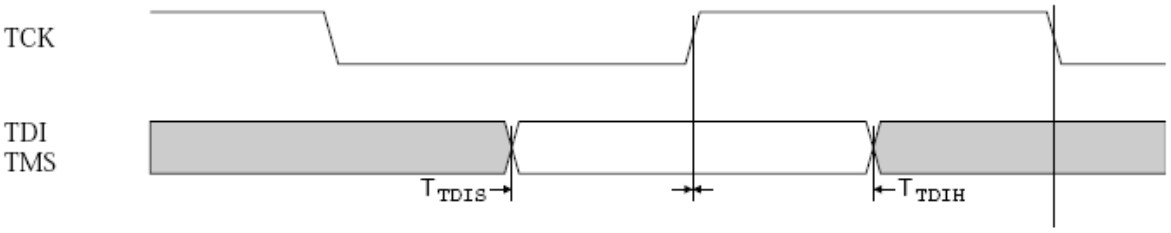


Figure 8-15 ARM9TDMI JTAG input signal timing

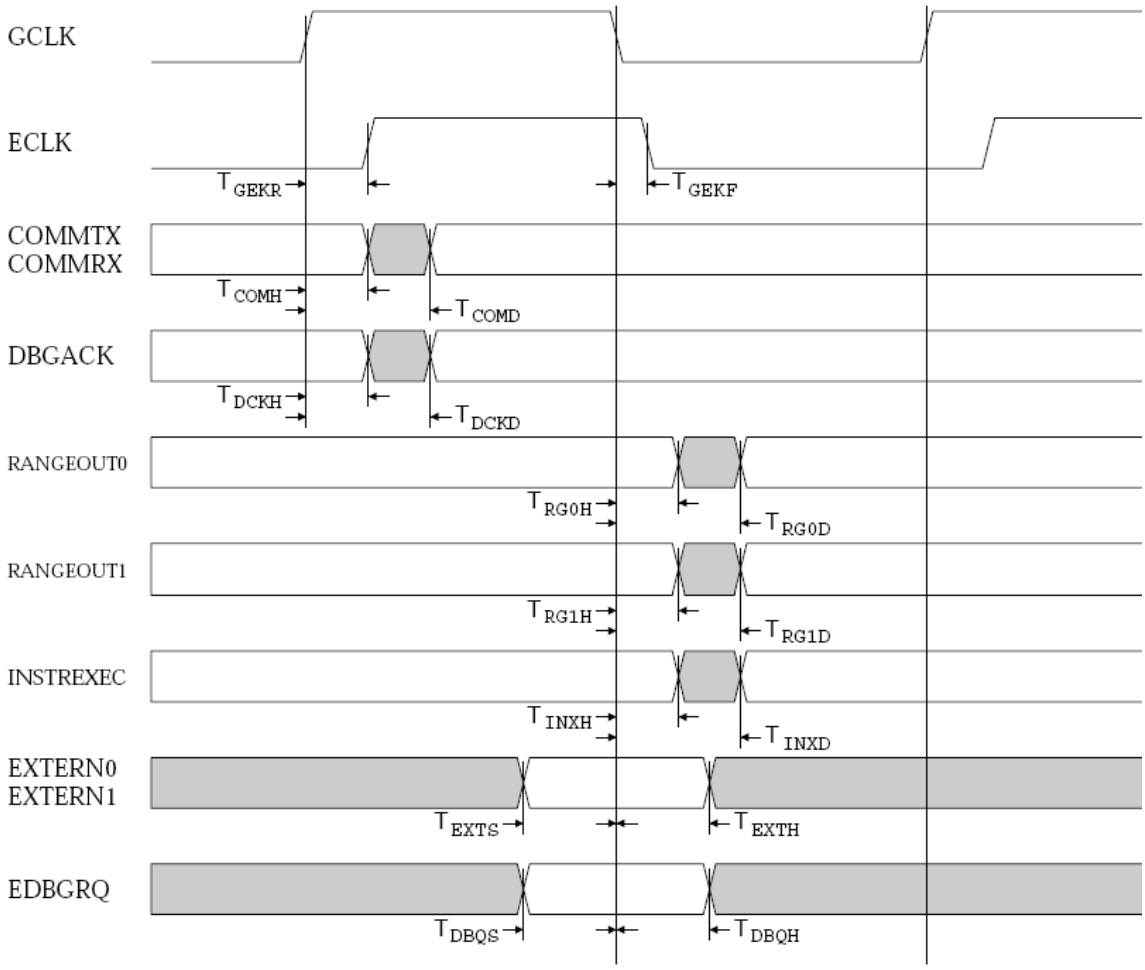


Figure 8-16 ARM9TDMI GCLK related debug output timings

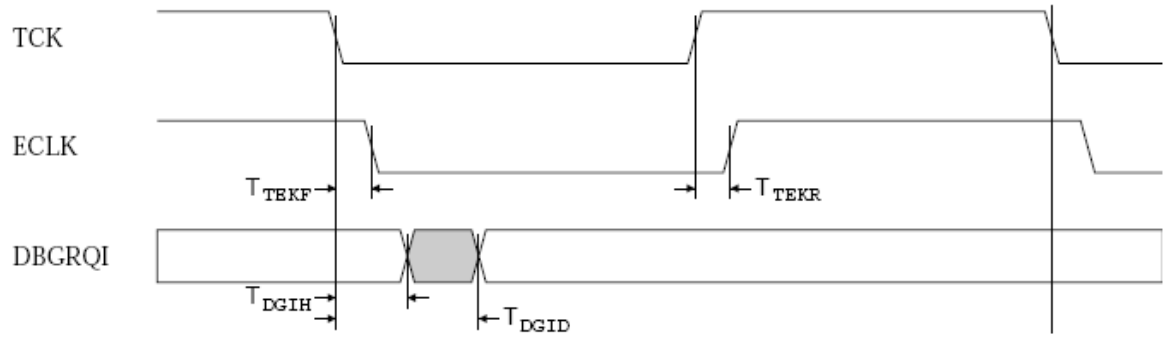


Figure 8-17 ARM9TDMI TCK related debug output timings

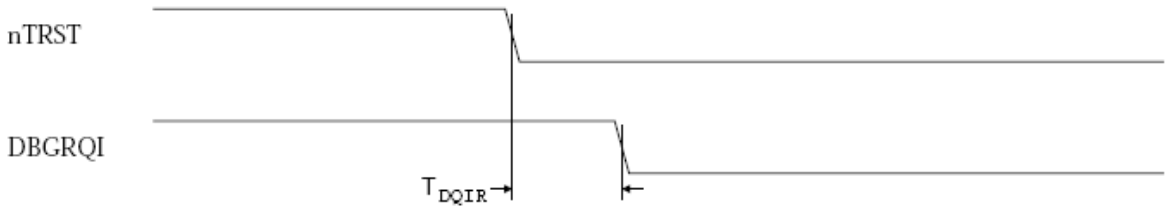


Figure 8-18 ARM9TDMI nTRST to DBGRQI relationship

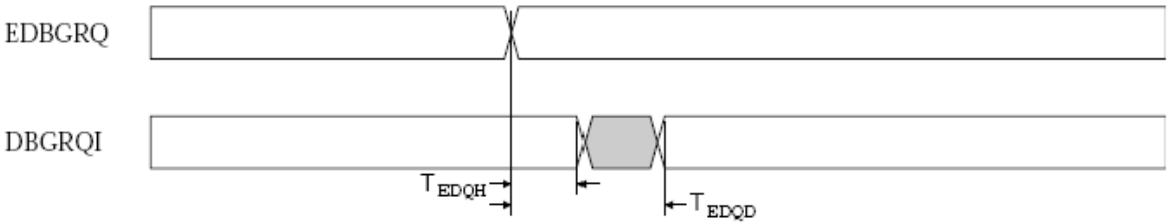


Figure 8-19 ARM9TDMI EDBGRQ to DBGRQI relationship

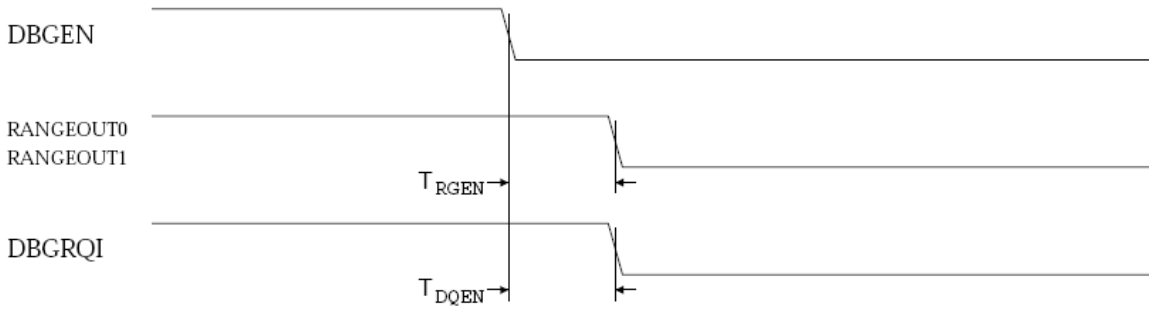


Figure 8-20 ARM9TDMI DBGEN to output effects

8.2 ARM9TDMI timing parameters

Table 8-1 ARM9TDMI timing parameters

Timing parameter	Description
T_{bigh}	**BIGEND** hold time from **GCLK** falling
T_{bigs}	**BIGEND** setup time to **GCLK** falling
T_{brst}	Delay from **nTRST** falling to **RSTCLKBS** rising
T_{brtd}	**RSTCLKBS** rising from **TCK** falling
T_{brth}	**RSTCLKBS** falling from **TCK** rising
T_{capf}	**ECAPCLKBS/ICAPCLKBS/PCLKBS** falling from **TCK** rising
T_{caph}	Input hold time to **TCK** falling (EXTEST capture)
T_{capr}	**ECAPCLKBS/ICAPCLKBS/PCLKBS** rising from **TCK** falling
T_{caps}	Input setup time to **TCK** falling (EXTEST capture)
T_{chsh}	**CHSD[1:0]**/**CHSE[1:0]** hold time from **GCLK** falling
T_{chss}	**CHSD[1:0]/CHSE[1:0]** setup time to **GCLK** falling
T_{comd}	**COMMTX/COMMRX** output delay
T_{comh}	**COMMTX/COMMRX** output hold time
T_{dabe}	Delay from **DABE** rising to **DA[31:0]/DnTRANS/DnM[4:0]/DMAS[1:0]/DnRW/DLOCK** driven valid
T_{dabh}	**DABORT** hold time from **GCLK** falling
T_{dabs}	**DABORT** setup time to **GCLK** falling
T_{dabtd}	**DnMREQ** delay from **DABORT**
T_{dabz}	Delay from **DABE** falling to **DA[31:0]/DnTRANS/DnM[4:0]/DMAS[1:0]/DnRW/DLOCK** high impedance
T_{dad}	**DA[31:0]** delay from **GCLK** rising
T_{dah}	**DA[31:0]** hold time from **GCLK** rising
T_{dbqh}	**EDBGRQ** input hold time from **GCLK** falling
T_{dbqs}	**EDBGRQ** input setup time to **GCLK** falling

Table 8-1 ARM9TDMI timing parameters (continued)

Timing parameter	Description
T_{dckd}	**DBGACK** output delay
T_{dckh}	**DBGACK** output hold time
T_{ddbe}	Delay from **DDBE** rising to **DD[31:0]** (output) driven valid
T_{ddbz}	Delay from **DDBE** falling to **DD[31:0]** (output) high impedance
T_{ddend}	**DDEN** delay from **GCLK** falling
T_{ddenh}	**DDEN** hold time from **GCLK** falling
T_{ddh}	**DD[31:0]** (input) hold time from **GCLK** falling
T_{ddod}	**DD[31:0]** (output) delay from **GCLK** falling
T_{ddoh}	**DD[31:0]** (output) hold time from **GCLK** falling
T_{dds}	**DD[31:0]** (input) setup time to **GCLK** falling
T_{dgid}	**DBGRQI** output delay from **TCK** falling
T_{dgih}	**DBGRQI** output hold time from **TCK** falling
T_{dih}	**TDI** and **TMS** hold time from **TCK** rising
T_{dis}	**TDI** and **TMS** setup time to **TCK** rising
T_{dlkd}	**DLOCK** delay from **GCLK** rising
T_{dlkh}	**DLOCK** hold time from **GCLK** rising
T_{dmqd}	**DnMREQ** delay from **GCLK** rising
T_{dmqh}	**DnMREQ** hold time from **GCLK** rising
T_{dmrd}	**DMORE** delay from **GCLK** rising
T_{dmrh}	**DMORE** hold time from **GCLK** rising
T_{dmsd}	**DMAS[1:0]** delay from **GCLK** rising
T_{dmsh}	**DMAS[1:0]** hold time from **GCLK** rising
T_{dnmd}	**DnM[4:0]** delay from **GCLK** rising
T_{dnmh}	**DnM[4:0]** hold time from **GCLK** rising

Table 8-1 ARM9TDMI timing parameters (continued)

Timing parameter	Description
T_{dqen}	**DBGRQI** falling delay from **DBGEN** falling
T_{dqir}	**nTRST** falling to **DBGRQI** falling delay
T_{drbsd}	**DRIVEOUTBS** delay from TCK falling
T_{drbsh}	**DRIVEOUTBS** hold time from TCK falling
T_{drwd}	**DnRW** delay from **GCLK** rising
T_{drwh}	**DnRW** hold time from **GCLK** rising
T_{dsqd}	**DSEQ** delay from **GCLK** rising
T_{dsqh}	**DSEQ** hold time from **GCLK** rising
T_{dtrsd}	**DnTRANS** delay from **GCLK** rising
T_{dtrsh}	**DnTRANS** hold time from **GCLK** rising
T_{dwph}	**DEWPT** hold time from **GCLK** rising
T_{dwps}	**DEWPT** setup time to **GCLK** rising
T_{edqd}	**DBGRQI** output delay from **EDBGRQ** changing
T_{edqh}	**DBGRQI** output hold time from **EDBGRQ** changing
T_{exth}	**EXTERN0/EXTERN1** input hold time from **GCLK** falling
T_{exts}	**EXTERN0/EXTERN1** input setup time to **GCLK** falling
T_{gclkh}	Minimum **GCLK** HIGH period
T_{gclkl}	Minimum **GCLK** LOW period
T_{gekf}	**GCLK** falling to **ECLK** falling delay
T_{gekr}	**GCLK** rising to **ECLK** rising delay
T_{hivh}	**HIVECS** hold time from **GCLK** rising
T_{hivs}	**HIVECS** setup time to **GCLK** rising
T_{iabe}	Delay from **IABE** rising to **IA[31:1]/InM[4:0]/InTRANS** driven valid
T_{iabh}	**IABORT** hold time from **GCLK** falling

Table 8-1 ARM9TDMI timing parameters (continued)

Timing parameter	Description
T_{iabs}	**IABORT** setup time to **GCLK** falling
T_{iabz}	Delay from **IABE** falling to **IA[31:1]/InM[4:0]/InTRANS** high impedance
T_{iad}	**IA[31:1]** delay from **GCLK** rising
T_{iah}	**IA[31:1]** hold time from **GCLK** rising
T_{ibkh}	**IEBKPT** hold time from **GCLK** rising
T_{ibks}	**IEBKPT** setup time to **GCLK** rising
T_{idh}	**ID[31:0]** hold time from **GCLK** falling
T_{ids}	**ID[31:0]** setup time to **GCLK** falling
T_{imqd}	**InMREQ** delay from **GCLK** rising
T_{imqh}	**InMREQ** hold time from **GCLK** rising
T_{inmd}	**InM[4:0]** delay from **GCLK** rising
T_{inmh}	**InM[4:0]** hold time from **GCLK** rising
T_{inth}	Interrupt (**nFIQ/nIRQ**) hold time from **GCLK** falling
T_{ints}	Interrupt (**nFIQ/nIRQ**) setup time to **GCLK** falling
T_{inxd}	**INSTREXEC** output delay
T_{inxh}	**INSTREXEC** output hold time
T_{irsd}	**IREG[3:0]/SCREG[4:0]** output delay from **TCK** falling
T_{irsh}	**IREG[3:0]/SCREG[4:0]** hold time from **TCK** falling
T_{isqd}	**ISEQ** delay from **GCLK** rising
T_{isqh}	**ISEQ** hold time from **GCLK** rising
T_{isyh}	**ISYNC** hold time from **GCLK** falling
T_{isys}	**ISYNC** setup time to **GCLK** falling
T_{itbd}	**ITBIT** delay from **GCLK** rising
T_{itbh}	**ITBIT** hold time from **GCLK** rising

Table 8-1 ARM9TDMI timing parameters (continued)

Timing parameter	Description
T_{itrsd}	**InTRANS** delay from **GCLK** rising
T_{itrsh}	**InTRANS** hold time from **GCLK** rising
T_{ltcd}	**LATECANCEL** delay from **GCLK** falling
T_{ltch}	**LATECANCEL** hold time from **GCLK** falling
T_{nwh}	**nWAIT** hold time from **GCLK** rising
T_{nws}	**nWAIT** setup time to **GCLK** falling
T_{pasd}	**PASS** output delay from **GCLK** rising
T_{pash}	**PASS** hold time from **GCLK** rising
T_{rg0d}	**RANGEOUT0** output delay
T_{rg0h}	**RANGEOUT0** output hold time
T_{rg1d}	**RANGEOUT1** output delay
T_{rg1h}	**RANGEOUT1** output hold time
T_{rgen}	**RANGEOUT0/RANGEOUT1** falling delay from **DBGEN** falling
T_{rsth}	**nRESET** hold time from **GCLK** rising
T_{rsts}	**nRESET** setup time to **GCLK** rising
T_{sdnd}	**SDIN** output delay from **TCK** falling
T_{sdnh}	**SDIN** hold time from **TCK** falling
T_{shkf}	**SHCLK1BS/SHCLK2BS** falling from **TCK** changing
T_{shkr}	**SHCLK1BS/SHCLK2BS** rising from TCK changing
T_{tapidh}	**TAPID[31:0]** hold time to **TCK** falling
T_{tapids}	**TAPID[31:0]** setup time to **TCK** falling
T_{tbe}	Delay from **TBE** rising, to outputs driven valid
T_{tbz}	Delay from **TBE** falling, to outputs high impedance
T_{tckf}	**TCK1/TCK2** falling from **TCK** changing

Table 8-1 ARM9TDMI timing parameters (continued)

Timing parameter	Description
T_{tckh}	Minimum **TCK** HIGH period
T_{tckl}	Minimum **TCK** LOW period
T_{tckr}	**TCK1/TCK2** rising from **TCK** changing
T_{tdod}	**TDO** output delay from **TCK** falling
T_{tdoh}	**TDO** hold time from **TCK** falling
T_{tdsd}	**TDO** delay from **SDOUTBS** changing
T_{tdsh}	**TDO** hold time from **SDOUTBS** changing
T_{tekf}	**TCK** falling to **ECLK** falling delay
T_{tekr}	**TCK** rising to **ECLK** rising delay
T_{toed}	**nTDOEN** output delay from **TCK** falling
T_{toeh}	**nTDOEN** hold time from **TCK** falling
T_{tpmd}	**TAPSM[3:0]** output delay from **TCK** falling
T_{tpmh}	**TAPSM[3:0]** hold time from **TCK** falling
T_{unis}	**UNIEN** input setup time to **GCLK** falling
T_{unih}	**UNIEN** input hold time to **GCLK** falling

Appendix A
ARM9TDMI Signal Descriptions

This chapter lists and describes the ARM9TDMI signals:

A.1 Instruction memory interface signals

Table A-1 Instruction memory interface signals

Name	Direction	Description
IA[31:1]	Output	Instruction Address Bus. This is the processor instruction address bus. It changes when **GCLK** is HIGH.
IABE	Input	Instruction Address Bus Enable. This is an input which, when LOW, it puts the instruction address bus, **IA[31:1]**, drivers into a high impedance state. This signal has the same effect on **InTRANS** and **InM[4:0]**. If **UNIEN** is HIGH this signal is ignored.
IABORT	Input	Instruction Abort. This is an input which allows the memory system to tell the processor that the requested instruction memory access is not allowed.
ID[31:0]	Input	Instruction Data Bus. This input bus should be driven with the requested instruction data before the end of phase 2 of **GCLK**.
InM[4:0]	Output	Instruction Mode. These signals indicate the current mode of the processor and are in the same form as the mode bits in the CPSR.
InMREQ	Output	Not Instruction Memory Request. If LOW at the end of **GCLK** phase 2, the processor requires an instruction memory access during the following cycle.
InTRANS	Output	Not Memory Translate. When LOW, the processor is in user mode. When HIGH, the processor is in a privileged mode.
ISEQ	Output	Instruction Sequential Address. If HIGH at the end of **GCLK** phase 2, any instruction memory access during the following cycle is sequential from the last instruction memory access.
ITBIT	Output	Instruction Thumb Bit. When HIGH, the processor is in Thumb state. When LOW, the processor is in ARM state.

A.2 Data memory interface signals

Table A-2 Data memory interface signals

Name	Direction	Description
DA[31:0]	Output	Data Address Bus. This is the processor data address bus. It changes when **GCLK** is HIGH.
DABE	Input	Data Address Bus Enable. When LOW, this input puts the data address bus, **DA[31:0]**, drivers into a high impedance state. This signal has the same effect on **DnTRANS**, **DLOCK**, **DMAS[1:0]**, **DnRW**, and **DnM[4:0]**. If **UNIEN** is HIGH this signal is ignored.
DABORT	Input	Data Abort. This input allows the memory system to tell the processor that the requested data memory access is not allowed.
DD[31:0]	Output	Data Output Bus. This output bus is used to transfer write data between the processor and external memory. The output data will become valid during phase 1 and remain valid through **GCLK** phase 2. If **UNIEN** is LOW, this is a tristate output bus and is only driven during write cycles. If **UNIEN** is HIGH, this bus is always driven.
DDBE	Input	Data Data Bus Enable. This is an input which, when LOW, puts the Data Data Bus **DD[31:0]** into a high impedance state. If **UNIEN** is HIGH this signal is ignored.
DDEN	Output	Data Data Bus Output Enabled. This signal indicates when the processor is performing a write transfer on the Data Data Bus, **DD[31:0]**.
DDIN[31:0]	Input	Data Input Bus. This input is used to transfer load data between external memory and the processor. It should be driven with the requested data by the end of **GCLK** phase 2.
DLOCK	Output	Data Lock. If HIGH at the end of **GCLK** phase 2, any data memory access in the following cycle is locked, and the memory controller must wait until **DLOCK** goes LOW before allowing another device to access memory.
DMAS[1:0]	Output	Data Memory Access Size. These outputs encode the size of a data memory access in the following cycle. A word access is encoded as 10 (binary), a halfword access as 01, and a byte access as 00. The encoding 11 is reserved.
DMORE	Output	Data More. If HIGH at the end of **GCLK** phase 2, the data memory access in the following cycle will be directly followed by a sequential data memory access.
DnM[4:0]	Output	Data Mode. The processor mode within which the data memory access should be performed. Note that the data memory access mode may differ from the current processor mode.

Table A-2 Data memory interface signals (continued)

Name	Direction	Description
DnMREQ	Output	Not Data Memory Request. If LOW at the end of **GCLK** phase 2, the processor requires a data memory access in the following cycle.
DnRW	Output	Data not Read, Write. If LOW at the end of phase 2, any data memory access in the following cycle is a read. If HIGH, it is a write.
DnTRANS	Output	Data Not Memory Translate. If LOW, the next data memory access is to be performed as a user mode access, if HIGH the data memory access is to performed as a privileged mode access. Note that the data memory access mode may differ from the current processor mode.
DSEQ	Output	Data Sequential Address. If HIGH at the end of phase 2, any data memory access in the next cycle is sequential from the current data memory access.

A.3 Coprocessor interface signals

Table A-3 Coprocessor interface signals

Name	Direction	Description
CHSD[1:0]	Input	Coprocessor Handshake Decode. The handshake signals from the decode stage of the coprocessors pipeline follower. Note, if no coprocessor is present in the system, **CHSD[1]** should be tied HIGH, and **CHSD[0]** should be tied LOW.
CHSE[1:0]	Input	Coprocessor Handshake Execute. The handshake signals from the execute stage of the coprocessors pipeline follower. Note, if no coprocessor is present in the system, **CHSE[1]** should be tied HIGH, and **CHSE[0]** should be tied LOW.
LATECANCEL	Output	Coprocessor Late Cancel. If HIGH during the first memory cycle of a coprocessor instruction's execution, the coprocessor should cancel the instruction without having updated its state.
PASS	Output	Coprocessor **PASS**. This signal indicates that there is a coprocessor instruction in the execute stage of the pipeline, and it should be executed.

For further information on the coprocessor interface refer to Chapter 4 *ARM9TDMI Coprocessor Interface*.

A.4 JTAG and TAP controller signals

Table A-4 JTAG and TAP controller signals

Name	Direction	Description
DRIVEOUTBS	Output	Boundary Scan Cell Enable. This signal is used to control the multiplexers in the scan cells of an external boundary scan chain. This signal changes in the UPDATE-IR state when scan chain 3 is selected and either the INTEST, EXTEST, CLAMP or CLAMPZ instruction is loaded. When an external boundary scan chain is not connected, this output should be left unconnected.
ECAPCLKBS	Output	Extest Capture Clock for Boundary Scan. This is a **TCK2** wide pulse generated when the TAP controller state machine is in the CAPTURE-DR state, the current instruction is EXTEST and scan chain 3 is selected. This signal is used to capture the chip level inputs during EXTEST. When an external boundary scan chain is not connected, this output should be left unconnected.
ICAPCLKBS	Output	Intest Capture Clock. This is a **TCK2** wide pulse generated when the TAP controller state machine is in the CAPTURE-DR state, the current instruction is INTEST and scan chain 3 is selected. This signal is used to capture the chip level outputs during INTEST. When an external boundary scan chain is not connected, this output should be left unconnected.
IR[3:0]	Output	Tap Controller Instruction Register. These four bits reflect the current instruction loaded into the TAP controller instruction register. The bits change on the falling edge of **TCK** when the state machine is in the UPDATE-IR state.
PCLKBS	Output	Boundary Scan Update Clock. This is a **TCK2** wide pulse generated when the TAP controller state machine is in the UPDATE-DR state and scan chain 3 is selected. This signal is used by an external boundary scan chain as the update clock. When an external boundary scan chain is not connected, this output should be left unconnected.
RSTCLKBS	Output	Boundary Scan Reset Clock. This signal denotes that either the TAP controller state machine is in the RESET state, or that **nTRST** has been asserted. This may be used to reset external boundary scan cells.
SCREG[4:0]	Output	Scan Chain Register. These four bits reflect the ID number of the scan chain currently selected by the TAP controller. These bits change on the falling edge of **TCK** when the TAP state machine is in the UPDATE-DR state.
SDIN	Output	Boundary Scan Serial Input Data. This signal contains the serial data to be applied to an external scan chain, and is valid around the falling edge of **TCK**.
SDOUTBS	Input	Boundary Scan Serial Output Data. This is the serial data out of the boundary scan chain (or other external scan chain). It should be set up to the rising edge of **TCK**. When an external boundary scan chain is not connected, this input should be tied LOW.

Table A-4 JTAG and TAP controller signals (continued)

Name	Direction	Description
SHCLK1BS	Output	Boundary Scan Shift Clock Phase 1. This control signal is provided to ease the connection of an external boundary scan chain. **SHCLK1BS** is used to clock the master half of the external scan cells. When the state machine is in SHIFT-DR state, scan chain 3 is selected, **SHCLK1BS** follows **TCK1**. When not in the SHIFT-DR state, or when scan chain 3 is not selected, this clock is LOW. When an external boundary scan chain is not connected, this output must be left unconnected.
SHCLK2BS	Output	Boundary Scan Shift Clock Phase 2. This control signal is provided to ease the connection of an external boundary scan chain. **SHCLK2BS** is used to clock the slave half of the external scan cells. When the state machine is in SHIFT-DR state, scan chain 3 is selected, **SHCLK2BS** follows **TCK2**. When not in the SHIFT-DR state, or when scan chain 3 is not selected, this clock is LOW. When an external boundary scan chain is not connected, this output must be left unconnected.
TAPID[31:0]	Input	TAP Identification. The value on this bus will be captured when using the IDCODE instruction on the TAP controller state machine.
TAPSM[3:0]	Output	TAP Controller State Machine. This bus reflects the current state of the TAP controller state machine. These bits change off the rising edge of **TCK**.
TCK	Input	The JTAG clock (the test clock).
TCK1	Output	**TCK**, Phase 1. **TCK1** is HIGH when **TCK** is HIGH, although there is a slight phase lag due to the internal clock non-overlap.
TCK2	Output	**TCK**, Phase 2. **CK2** is HIGH when **TCK** is LOW, although there is a slight phase lag due to the internal clock non-overlap.
TDI	Input	Test Data Input, the JTAG serial input.
TDO	Output	Test Data Output, the JTAG serial output.
nTDOEN	Output	Not **TDO** Enable. When LOW, this signal denotes that serial data is being driven out on the **TDO** output. The **nTDOEN** signal would normally be used as an output enable for a **TDO** pin in a packaged part.
TMS	Input	Test Mode Select. **TMS** selects to which state the TAP controller state machine should change.
nTRST	Input	Not Test Reset. Active-low reset signal for the boundary scan logic. This pin must be pulsed or driven LOW after power up to achieve normal device operation, in addition to the normal device reset (**nRESET**).

A.5 Debug signals

Table A-5 Debug signals

Name	Direction	Description
COMMRX	Output	Communications Channel Receive. When HIGH, this signal denotes that the comms channel receive buffer contains data waiting to be read by the ARM9TDMI.
COMMTX	Output	Communications Channel Transmit. When HIGH, this signal denotes that the comms channel transmit buffer is empty and the ARM9TDMI can write new data to the comms channel.
DBGACK	Output	Debug Acknowledge. When HIGH, this signal indicates the ARM9TDMI is in debug state.
DBGEN	Input	Debug Enable. This input signal allows the debug features of the ARM9TDMI to be disabled. This signal should be LOW only when debugging will not be required.
DBGRQI	Output	Internal Debug Request. This signal represents the debug request signal which is presented to the processor core. This is a combination of **EDBGRQ**, as presented to the ARM9TDMI, and bit 1 of the debug control register.
DEWPT	Input	Data Watchpoint. This is an input which allows external hardware to halt execution of the processor for debug purposes. If HIGH at the end of phase 1 following a data memory request cycle, it will cause the ARM9TDMI to enter debug state.
EDBGRQ	Input	External Debug Request. When driven HIGH, this causes the processor to enter debug state after execution of the current instruction completes.
EXTERN0	Input	External Input 0. This is an input to watchpoint unit 0 of the EmbeddedICE macrocell in the processor which allows breakpoints/watchpoints to be dependent on an external condition.
EXTERN1	Input	External Input 1. This is an input to watchpoint unit 1 of the EmbeddedICE macrocell in the processor which allows breakpoints/watchpoints to be dependent on an external condition.
IEBKPT	Input	Instruction Breakpoint. This is an input which allows a external hardware to halt the execution of the processor for debug purposes. If HIGH at the end of phase 1 following an instruction memory request cycle, it causes the ARM9TDMI to enter debug state if the relevant instruction reaches the execute stage of the processor pipeline.
INSTREXEC	Output	Instruction Executed. Indicates that in the previous cycle the instruction in the execute stage of the pipeline passed its condition codes, and was executed.

Table A-5 Debug signals (continued)

Name	Direction	Description
RANGEOUT0	Output	EmbeddedICE Rangeout 0. This signal indicates that the EmbeddedICE macrocell watchpoint unit 0 has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint's enable control bit.
RANGEOUT1	Output	EmbeddedICE Rangeout 1. This signal indicates that the EmbeddedICE macrocell watchpoint unit 1 has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint's enable control bit.
TBE	Input	Test Bus Enable. When driven LOW, **TBE** forces the following signals to HIGH impedance: **DD[31:0]** **DA[31:0]** **DLOCK** **DMAS[1:0]** **DnM[4:0]** **DnRW** **DnTRANS** **DMORE** **DnMREQ** **DSEQ** **IA[31:0]** **InM[4:0]** **InTRANS** **InMREQ** **ISEQ** **ITBIT** **LATECANCEL** **PASS.** Under normal operating conditions, **TBE** should be held HIGH at all times. If **UNIEN** is HIGH, this signal is ignored.

A.6 Miscellaneous signals

Table A-6 Miscellaneous signals

Name	Direction	Description
BIGEND	Input	Big-Endian Configuration. When this input is HIGH, the ARM9TDMI processor treats bytes in memory as being in big-endian format. When it is LOW, memory is treated as little-endian.
ECLK	Output	External Clock. The clock by which the ARM9TDMI is currently being clocked. This clock will reflect any wait states applied by **nWAIT**, and once debug state has been entered by the debug clock.
nFIQ	Input	Not Fast Interrupt request. This input causes the core to be interrupted if taken LOW, and if the appropriate enable in the processor is active. The signal is level-sensitive and must be held LOW until a suitable response is received from the processor. The **nFIQ** signal may be synchronous or asynchronous, depending on the state of ISYNC.
GCLK	Input	Clock. This clock times all ARM9TDMI memory accesses (both data and instruction), and internal operations. The clock has two distinct phases—phase 1 in which **GCLK** is LOW and phase 2 inwhich **GCLK** is HIGH. The clock may be stretched indefinitely in either phase to allow access to slow peripherals or memory. Alternatively, **nWAIT** may be used with a free running **GCLK** to stretch phase 2.
HIVECS	Input	High Vectors Configuration. When LOW, the ARM9TDMI exception vectors start at address `0x00000000` (hexadecimal). When HIGH, the ARM9TDMI exception vectors start at address `0xFFFF0000`.
nIRQ	Input	Not Interrupt Request. As **nFIQ**, but with lower priority. May be taken LOW to interrupt the processor when theappropriate enable is active. The **nIRQ** signal may be synchronous or asynchronous,depending on the state of ISYNC.
ISYNC	Input	Synchronous Interrupts. When LOW, this input indicates that the **nIRQ** and **nFIQ** inputs are to be synchronized by the processor. When HIGH it disables this synchronization for inputs that are already synchronous.
nRESET	Input	Not Reset. This is a level-sensitive input signal which is used to start the processor from a known address. The ARM9TDMI processor asynchronously enters reset when **nRESET** goes LOW.

Table A-6 Miscellaneous signals (continued)

Name	Direction	Description
nWAIT	Input	Not Wait. When a memory request cannot be processed in a single cycle, the ARM9TDMI can be made to wait for a number of **GCLK** cycles by driving **nWAIT** LOW. Internally, the inverse of **nWAIT** is ORed with **GCLK**, and must only change when **GCLK** is HIGH. If **nWAIT** is not used, it must be tied HIGH.
UNIEN	Input	Unidirectional Enable. When HIGH, all ARM9TDMI outputs are permanently driven, (the state of IABE, DABE, DDBE and TBE is ignored). The **DDIN[31:0]** and **DD[31:0]** buses form a unidirectional data bus. When LOW, outputs can go tristate and the **DD[31:0]** bus is only driven during write cycles. If **DD[31:0]** and **DDIN[31:0]** are wired together, they form a bidirectional data bus.

Index

E

F

H

I

J

L

M

N

P

R

S

T

U

V

W