Atmel Corporation ARM7TDMI™ (Thumb®) Datasheet

January 1999

Document Details
Title: ARM7TDMI (Thumb) Data Sheet
Literature Number: 0673B
Revision: B
Date: January 1999

Printed and distributed by Atmel ES2 in accordance with the license agreement existing between ARM for the ARM7TDMI microprocessor.

Revision History
Revision A: July 1996

Revision B: Reformatting of Revision A (numbering removed) and electrical characteristics removed. From now on, please see one of the following datasheets for electrical characteristics:

- ARM7TDMI Embedded Core ATC50 Electrical Characteristics (0.5 micron three-layer-metal CMOS process intended for use with a supply voltage of 3.3V ± 0.3V)
- ARM7TDMI Embedded Core ATC50/E^2 Electrical Characteristics (0.5 micron three-layer-metal CMOS/NVM process intended for use with a supply voltage of 3.3V ± 0.3V)
- ARM7TDMI Embedded Core ATC35 Electrical Characteristics (0.35 micron three-layer-metal CMOS process intended for use with a supply voltage of 3.3V ± 0.3V)

Atmel ES2
Zone Industrielle
13106 Rousset Cedex
France

Tel: (+33) (0)4 42 53 60 00
Fax: (+33) (0)4 42 53 60 01

For other Atmel addresses see back page.

Table of Contents

Architectural Overview

This chapter introduces the ARM7TDMI architecture and shows block, core, and functional diagrams for the ARM7TDMI.

Introduction

The ARM7TDMI is a member of the Advanced RISC Machines (ARM) family of general purpose 32-bit microprocessors, which offer high performance for very low power consumption and price.

The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler than those of microprogrammed Complex Instruction Set Computers. This simplicity results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective chip.

Pipelining is employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The ARM memory interface has been designed to allow the performance potential to be realised without incurring high costs in the memory system. Speed-critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic, and these control signals facilitate the exploitation of the fast local access modes offered by industry standard dynamic RAMs.

Rev. 0673B–12/98

ARM7TDMI Architecture

The ARM7TDMI is a 3-stage pipeline, 32-bit RISC processor. The processor architecture is Von Neumann load/store architecture, which is characterized by a single data and address bus for instructions and data. The CPU has two instruction sets, the ARM and the Thumb instruction set. The ARM instruction set has 32-bit wide instructions and provides maximum performance. Thumb instructions are 16-bits wide and give maximum code-density. Instructions operate on 8-, 16-, and 32-bit data types.

The CPU has seven operating modes (see Operating Modes on page 17). Each operating mode has dedicated banked registers for fast exception handling. The processor has a total of 37 32-bit registers, including 6 status registers (see Registers).

The THUMB Concept

The ARM7TDMI processor employs a unique architectural strategy known as *THUMB*, which makes it ideally suited to high-volume applications with memory restrictions, or applications where code density is an issue.

The key idea behind THUMB is that of a super-reduced instruction set. Essentially, the ARM7TDMI processor has two instruction sets:

- the standard 32-bit ARM set
- a 16-bit THUMB set

The THUMB set's 16-bit instruction length allows it to approach twice the density of standard ARM code while retaining most of the ARM's performance advantage over a traditional 16-bit processor using 16-bit registers. This is possible because THUMB code operates on the same 32-bit register set as ARM code.

THUMB code is able to provide up to 65% of the code size of ARM, and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system.

THUMB's Advantages

THUMB instructions operate with the standard ARM register configuration, allowing excellent interoperability between ARM and THUMB states. Each 16-bit THUMB instruction has a corresponding 32-bit ARM instruction with the same effect on the processor model.

The major advantage of a 32-bit (ARM) architecture over a 16-bit architecture is its ability to manipulate 32-bit integers with single instructions, and to address a large address space efficiently. When processing 32-bit data, a 16-bit architecture will take at least two instructions to perform the same task as a single ARM instruction.

However, not all the code in a program will process 32-bit data (for example, code that performs character string handling), and some instructions, like Branches, do not process any data at all.

If a 16-bit architecture only has 16-bit instructions, and a 32-bit architecture only has 32-bit instructions, then overall the 16-bit architecture will have better code density, and better than one half the performance of the 32-bit architecture. Clearly 32-bit performance comes at the cost of code density.

THUMB breaks this constraint by implementing a 16-bit instruction length on a 32-bit architecture, making the processing of 32-bit data efficient with a compact instruction coding. This provides far better performance than a 16-bit architecture, with better code density than a 32-bit architecture.

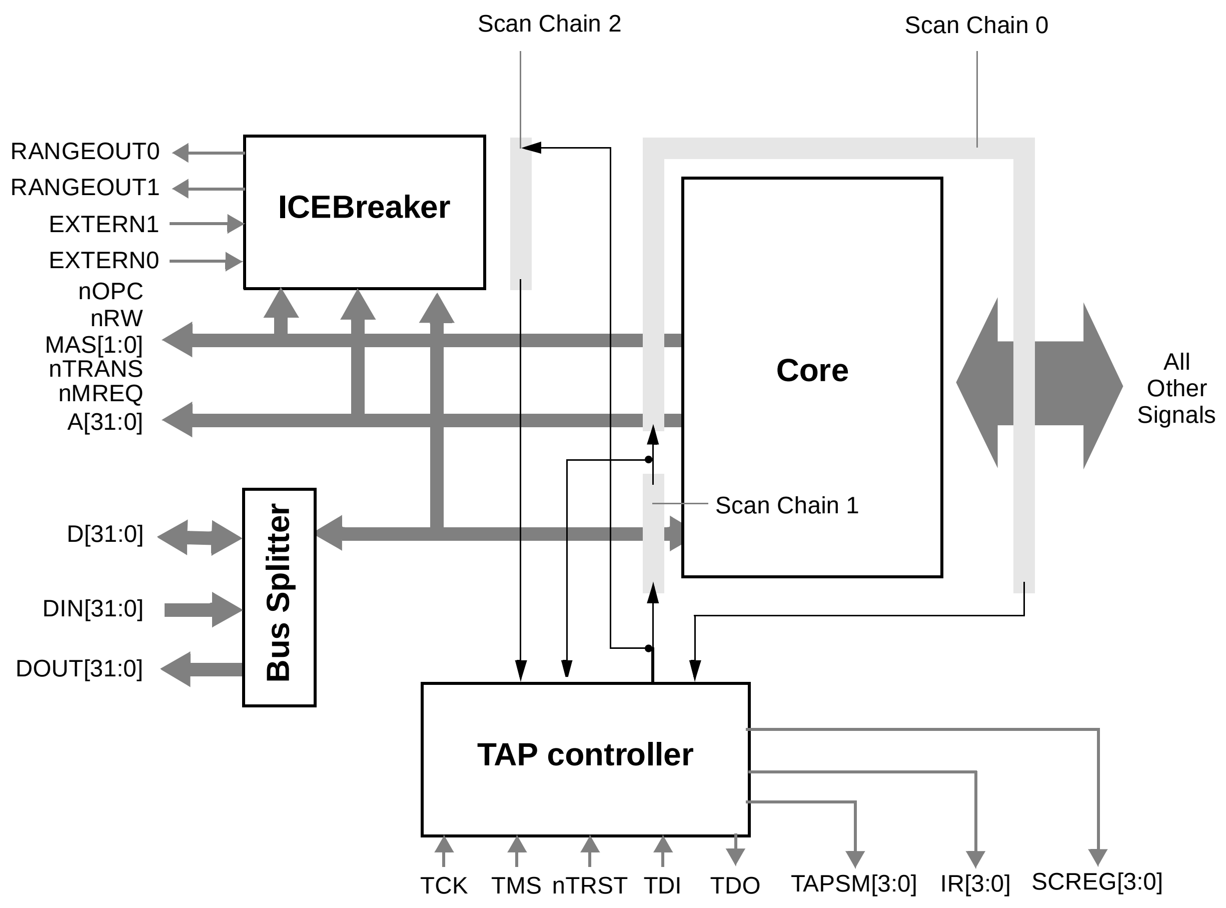
THUMB also has a major advantage over other 32-bit architectures with 16-bit instructions. This is the ability to switch back to full ARM code and execute at full speed. Thus critical loops for applications such as

- fast interrupts
- DSP algorithms

can be coded using the full ARM instruction set, and linked with THUMB code. The overhead of switching from THUMB code to ARM code is folded into sub-routine entry time. Various portions of a system can be optimised for speed or for code density by switching between THUMB and ARM execution as appropriate.

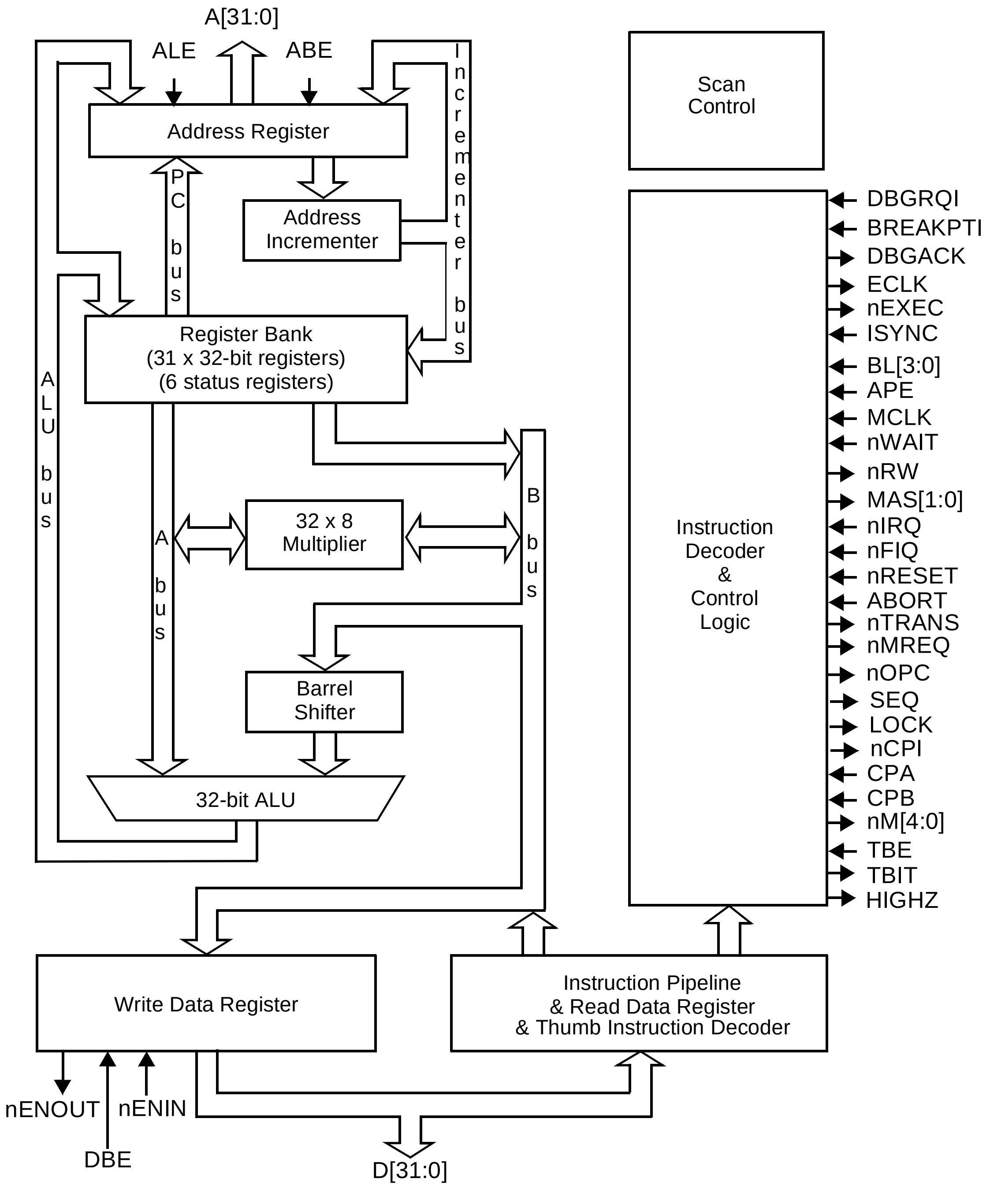
ARM7TDMI Block Diagram

Figure 1. ARM7TDMI Block Diagram



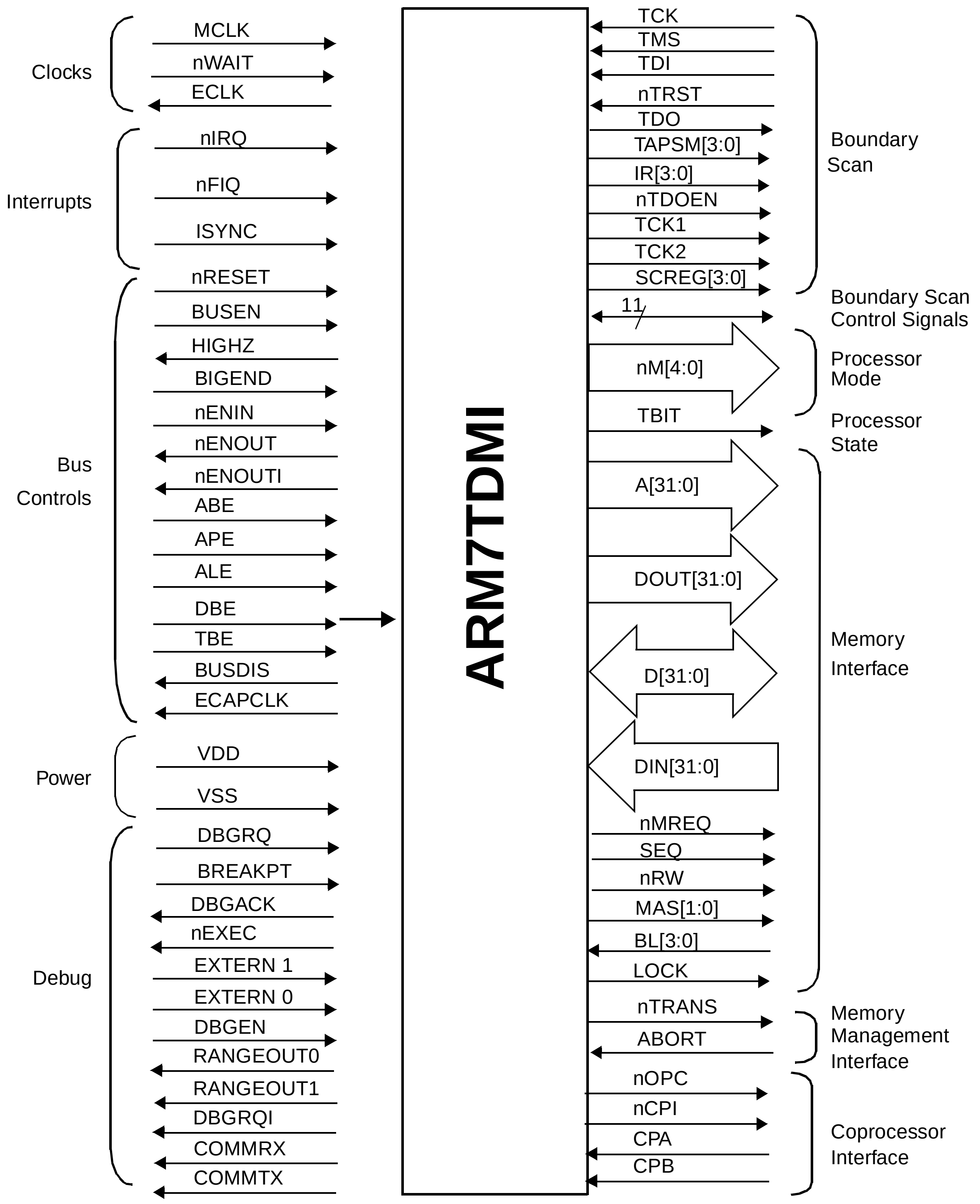
ARM7TDMI Core Diagram

Figure 2. ARM7TDMI Core



ARM7TDMI Functional Diagram

Figure 3. ARM7TDMI Functional Diagram



Signal Description

This chapter lists and describes the input/output signals for the ARM7TDMI.

The following table (Table 1) lists and describes all of the signals for the ARM7TDMI.

Key to signal types

IC	Input with CMOS thresholds
P	Power
O4	Output with INV4 driver
O8	Output with INV8 driver

Table 1. Signal Description

Name	Type	Description
A[31:0] Addresses	08	This is the processor address bus. If **ALE** (address latch enable) is HIGH and **APE** (Address Pipeline Enable) is LOW, the addresses become valid during phase 2 of the cycle before the one to which they refer and remain so during phase 1 of the referenced cycle. Their stable period may be controlled by **ALE** or **APE** as described below.
ABE Address bus enable	IC	This is an input signal which, when LOW, puts the address bus drivers into a high impedance state. This signal has a similar effect on the following control signals: **MAS[1:0]**, **nRW**, **LOCK**, **nOPC** and **nTRANS**. **ABE** must be tied HIGH when there is no system requirement to turn off the address drivers.
ABORT Memory Abort	IC	This is an input which allows the memory system to tell the processor that a requested access is not allowed.
ALE Address latch enable.	IC	This input is used to control transparent latches on the address outputs. Normally the addresses change during phase 2 to the value required during the next cycle, but for direct interfacing to ROMs they are required to be stable to the end of phase 2. Taking **ALE** LOW until the end of phase 2 will ensure that this happens. This signal has a similar effect on the following control signals: **MAS[1:0]**, **nRW**, **LOCK**, **nOPC** and **nTRANS**. If the system does not require address lines to be held in this way, **ALE** must be tied HIGH. The address latch is static, so **ALE** may be held LOW for long periods to freeze addresses.
APE Address pipeline enable.	IC	When HIGH, this signal enables the address timing pipeline. In this state, the address bus plus **MAS[1:0]**, **nRW**, **nTRANS**, **LOCK** and **nOPC** change in the phase 2 prior to the memory cycle to which they refer. When **APE** is LOW, these signals change in the phase 1 of the actual cycle. Please refer to Memory Interface on page 117 for details of this timing.
BIGEND Big Endian configuration.	IC	When this signal is HIGH the processor treats bytes in memory as being in Big Endian format. When it is LOW, memory is treated as Little Endian.
BL[3:0] Byte Latch Control.	IC	These signals control when data and instructions are latched from the external data bus. When **BL[3]** is HIGH, the data on **D[31:24]** is latched on the falling edge of **MCLK**. When **BL[2**] is HIGH, the data on **D[23:16]** is latched and so on. Please refer to for details on the use of these signals.
BREAKPT Breakpoint.	IC	This signal allows external hardware to halt the execution of the processor for debug purposes. When HIGH causes the current memory access to be break-pointed. If the memory access is an instruction fetch, ARM7TDMI will enter debug state if the instruction reaches the execute stage of the ARM7TDMI pipeline. If the memory access is for data, ARM7TDMI will enter debug state after the current instruction completes execution.This allows extension of the internal breakpoints provided by the ICEBreaker module. See ICEBreaker Module on page 163.
BUSDIS Bus Disable	O	This signal is HIGH when INTEST is selected on scan chain 0 or 4 and may be used to disable external logic driving onto the bidirectional data bus during scan testing. This signal changes on the falling edge of **TCK**.
BUSEN Data bus configuration	IC	This is a static configuration signal which determines whether the bidirectional data bus, **D[31:0]**, or the unidirectional data busses, **DIN[31:0]** and **DOUT[31:0]**, are to be used for transfer of data between the processor and memory. Refer also to Memory Interface on page 117. When **BUSEN** is LOW, the bidirectional data bus, **D[31:0]** is used. In this case, **DOUT[31:0]** is driven to value 0x00000000, and any data presented on **DIN[31:0]** is ignored. When **BUSEN** is HIGH, the bidirectional data bus, **D[31:0]** is ignored and must be left unconnected. Input data and instructions are presented on the input data bus, **DIN[31:0]**, output data appears on **DOUT[31:0]**.
COMMRX Communications Channel Receive	O	When HIGH, this signal denotes that the comms channel receive buffer is empty. This signal changes on the rising edge of **MCLK**. See Debug Communications Channel for more information on the debug comms channel.

Table 1. Signal Description (Continued)

Name	Type	Description
COMMTX Communications Channel Transmit	O	When HIGH, this signal denotes that the comms channel transmit buffer is empty. This signal changes on the rising edge of **MCLK**. See Debug Communications Channel for more information on the debug comms channel.
CPA Coprocessor absent.	IC	A coprocessor which is capable of performing the operation that ARM7TDMI is requesting (by asserting **nCPI**) should take **CPA** LOW immediately. If **CPA** is HIGH at the end of phase 1 of the cycle in which **nCPI** went LOW, ARM7TDMI will abort the coprocessor handshake and take the undefined instruction trap. If **CPA** is LOW and remains LOW, ARM7TDMI will busy-wait until **CPB** is LOW and then complete the coprocessor instruction.
CPB Coprocessor busy.	IC	A coprocessor which is capable of performing the operation which ARM7TDMI is requesting (by asserting **nCPI**), but cannot commit to starting it immediately, should indicate this by driving **CPB** HIGH. When the coprocessor is ready to start it should take **CPB** LOW. ARM7TDMI samples **CPB** at the end of phase 1 of each cycle in which **nCPI** is LOW.
D[31:0] Data Bus.	IC 08	These are bidirectional signal paths which are used for data transfers between the processor and external memory. During read cycles (when **nRW** is LOW), the input data must be valid before the end of phase 2 of the transfer cycle. During write cycles (when **nRW** is HIGH), the output data will become valid during phase 1 and remain valid throughout phase 2 of the transfer cycle. Note that this bus is driven at all times, irrespective of whether **BUSEN** is HIGH or LOW. When **D[31:0]** is not being used to connect to the memory system it must be left unconnected. See Memory Interface on page 117.
DBE Data Bus Enable.	IC	This is an input signal which, when driven LOW, puts the data bus **D[31:0]** into the high impedance state. This is included for test purposes, and should be tied HIGH at all times.
DBGACK Debug acknowledge.	04	When HIGH indicates ARM is in debug state.
DBGEN Debug Enable.	IC	This input signal allows the debug features of ARM7TDMI to be disabled. This signal should be driven LOW when debugging is not required.
DBGRQ Debug request.	IC	This is a level-sensitive input, which when HIGH causes ARM7TDMI to enter debug state after executing the current instruction. This allows external hardware to force ARM7TDMI into the debug state, in addition to the debugging features provided by the ICEBreaker block. See ICEBreaker Module on page 163 for details.
DBGRQI Internal debug reques**t**	04	This signal represents the debug request signal which is presented to the processor. This is the combination of external **DBGRQ**, as presented to the ARM7TDMI macrocell, and bit 1 of the debug control register. Thus there are two conditions where this signal can change. Firstly, when **DBGRQ** changes, **DBGRQI** will change after a propagation delay. When bit 1 of the debug control register has been written, this signal will change on the falling edge of **TCK** when the TAP controller state machine is in the RUN-TEST/IDLE state. See ICEBreaker Module on page 163 for details.
DIN[31:0] Data input bus	IC	This is the input data bus which may be used to transfer instructions and data between the processor and memory.This data input bus is only used when **BUSEN** is HIGH. The data on this bus is sampled by the processor at the end of phase 2 during read cycles (i.e. when **nRW** is LOW).
DOUT[31:0] Data output bus	08	This is the data out bus, used to transfer data from the processor to the memory system. Output data only appears on this bus when **BUSEN** is HIGH. At all other times, this bus is driven to value 0x00000000. When in use, data on this bus changes during phase 1 of store cycles (i.e. when **nRW** is HIGH) and remains valid throughout phase 2.

Table 1. Signal Description (Continued)

Name	Type	Description
DRIVEBS Boundary scan cell enable	04	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.
ECAPCLK Extest capture clock	O	This signal removes the need for the external logic in the test chip which was required to enable the internal tristate bus during scan testing. This need not be brought out as an external pin on the test chip.
ECAPCLKBS Extest capture clock for Boundary Scan	04	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 is used to capture the macrocell outputs during EXTEST. When an external boundary scan chain is not connected, this output should be left unconnected.
ECLK External clock output.	04	In normal operation, this is simply **MCLK** (optionally stretched with **nWAIT**) exported from the core. When the core is being debugged, this is **DCLK**. This allows external hardware to track when the ARM7TDMI core is clocked.
EXTERN0 External input 0.	IC	This is an input to the ICEBreaker logic in the ARM7TDMI which allows breakpoints and/or watchpoints to be dependent on an external condition.
EXTERN1 External input 1.	IC	This is an input to the ICEBreaker logic in the ARM7TDMI which allows breakpoints and/or watchpoints to be dependent on an external condition.
HIGHZ	04	This signal denotes that the HIGHZ instruction has been loaded into the TAP controller. See Debug Interface on page 139 for details.
ICAPCLKBS Intest capture clock	04	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 is used to capture the macrocell outputs during INTEST. When an external boundary scan chain is not connected, this output should be left unconnected.
IR[3:0] TAP controller Instruction register	04	These 4 bits reflect the current instruction loaded into the TAP controller instruction register. The instruction encoding is as described in Public Instructions. These bits change on the falling edge of **TCK** when the state machine is in the UPDATE-IR state.
ISYNC Synchronous interrupts.	IC	When LOW indicates that the **nIRQ** and **nFIQ** inputs are to be synchronised by the ARM core. When HIGH disables this synchronisation for inputs that are already synchronous.
LOCK Locked operation.	08	When **LOCK** is HIGH, the processor is performing a “locked” memory access, and the memory controller must wait until **LOCK** goes LOW before allowing another device to access the memory. **LOCK** changes while **MCLK** is HIGH, and remains HIGH for the duration of the locked memory accesses. It is active only during the data swap (SWP) instruction. The timing of this signal may be modified by the use of **ALE** and **APE** in a similar way to the address, please refer to the **ALE** and **APE** descriptions. This signal may also be driven to a high impedance state by driving **ABE** LOW.
MAS[1:0] Memory Access Size.	08	These are output signals used by the processor to indicate to the external memory system when a word transfer or a half-word or byte length is required. The signals take the value 10 (binary) for words, 01 for half-words and 00 for bytes. 11 is reserved. These values are valid for both read and write cycles. The signals will normally become valid during phase 2 of the cycle before the one in which the transfer will take place. They will remain stable throughout phase 1 of the transfer cycle. The timing of the signals may be modified by the use of **ALE** and **APE** in a similar way to the address, please refer to the **ALE** and **APE** descriptions. The signals may also be driven to high impedance state by driving **ABE** LOW.

Table 1. Signal Description (Continued)

Name	Type	Description
MCLK Memory clock input.	IC	This clock times all ARM7TDMI memory accesses and internal operations. The clock has two distinct phases - *phase 1* in which **MCLK** is LOW and *phase 2* in which **MCLK** (and **nWAIT**) is HIGH. The clock may be stretched indefinitely in either phase to allow access to slow peripherals or memory. Alternatively, the **nWAIT** input may be used with a free running **MCLK** to achieve the same effect.
nCPI Not Coprocessor instruction.	04	When ARM7TDMI executes a coprocessor instruction, it will take this output LOW and wait for a response from the coprocessor. The action taken will depend on this response, which the coprocessor signals on the **CPA** and **CPB** inputs.
nENIN NOT enable input.	IC	This signal may be used in conjunction with **nENOUT** to control the data bus during write cycles. See Memory Interface on page 117.
nENOUT Not enable output.	04	During a data write cycle, this signal is driven LOW during phase 1, and remains LOW for the entire cycle. This may be used to aid arbitration in shared bus applications. See Memory Interface on page 117.
nENOUTI Not enable output.	O	During a coprocessor register transfer C-cycle from the ICEbreaker comms channel coprocessor to the ARM core, this signal goes LOW during phase 1 and stays LOW for the entire cycle. This may be used to aid arbitration in shared bus systems.
nEXEC Not executed.	04	When HIGH indicates that the instruction in the execution unit is not being executed, because for example it has failed its condition code check.
nFIQ Not fast interrupt request.	IC	This is an interrupt request to the processor which causes it to be interrupted if taken LOW when 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. **nFIQ** may be synchronous or asynchronous, depending on the state of **ISYNC**.
nHIGHZ Not **HIGHZ**	04	This signal is generated by the TAP controller when the current instruction is HIGHZ. This is used to place the scan cells of that scan chain in the high impedance state. When a external boundary scan chain is not connected, this output should be left unconnected.
nIRQ Not interrupt request.	IC	As **nFIQ**, but with lower priority. May be taken LOW to interrupt the processor when the appropriate enable is active. **nIRQ** may be synchronous or asynchronous, depending on the state of **ISYNC**.
nM[4:0] Not processor mode.	04	These are output signals which are the inverses of the internal status bits indicating the processor operation mode.
nMREQ Not memory request.	04	This signal, when LOW, indicates that the processor requires memory access during the following cycle. The signal becomes valid during phase 1, remaining valid through phase 2 of the cycle preceding that to which it refers.
nOPC Not op-code fetch.	08	When LOW this signal indicates that the processor is fetching an instruction from memory; when HIGH, data (if present) is being transferred. The signal becomes valid during phase 2 of the previous cycle, remaining valid through phase 1 of the referenced cycle. The timing of this signal may be modified by the use of **ALE** and **APE** in a similar way to the address, please refer to the **ALE** and **APE** descriptions. This signal may also be driven to a high impedance state by driving **ABE** LOW.
nRESET Not reset.	IC	This is a level sensitive input signal which is used to start the processor from a known address. A LOW level will cause the instruction being executed to terminate abnormally. When **nRESET** becomes HIGH for at least one clock cycle, the processor will re-start from address 0. **nRESET** must remain LOW (and **nWAIT** must remain HIGH) for at least two clock cycles. During the LOW period the processor will perform dummy instruction fetches with the address incrementing from the point where reset was activated. The address will overflow to zero if **nRESET** is held beyond the maximum address limit.

Table 1. Signal Description (Continued)

Name	Type	Description
nRW Not read/write.	08	When HIGH this signal indicates a processor write cycle; when LOW, a read cycle. It becomes valid during phase 2 of the cycle before that to which it refers, and remains valid to the end of phase 1 of the referenced cycle. The timing of this signal may be modified by the use of **ALE** and **APE** in a similar way to the address, please refer to the **ALE** and **APE** descriptions. This signal may also be driven to a high impedance state by driving **ABE** LOW.
nTDOEN Not **TDO** Enable.	04	When LOW, this signal denotes that serial data is being driven out on the **TDO** output. **nTDOEN** would normally be used as an output enable for a **TDO** pin in a packaged part.
nTRANS Not memory translate.	08	When this signal is LOW it indicates that the processor is in user mode. It may be used to tell memory management hardware when translation of the addresses should be turned on, or as an indicator of non-user mode activity. The timing of this signal may be modified by the use of **ALE** and **APE** in a similar way to the address, please refer to the **ALE** and **APE** description. This signal may also be driven to a high impedance state by driving **ABE** LOW.
nTRST Not Test Reset.	IC	Active-low reset signal for the boundary scan logic. This pin must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset **(nRESET)**. For more information, see Debug Interface on page 139.
nWAIT Not wait.	IC	When accessing slow peripherals, ARM7TDMI can be made to wait for an integer number of **MCLK** cycles by driving **nWAIT** LOW. Internally, **nWAIT** is ANDed with **MCLK** and must only change when **MCLK** is LOW. If **nWAIT** is not used it must be tied HIGH.
PCLKBS Boundary scan update clock	04	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 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.
RANGEOUT0 ICEbreaker Rangeout0	04	This signal indicates that ICEbreaker watchpoint register 0 has matched the conditions currently present on the address, data and control busses. This signal is independent of the state of the watchpoint's enable control bit. **RANGEOUT0** changes when **ECLK** is LOW.
RANGEOUT1 ICEbreaker Rangeout1	04	As **RANGEOUT0** but corresponds to ICEbreaker's watchpoint register 1.
RSTCLKBS Boundary Scan Reset Clock	O	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[3:0] Scan Chain Register	O	These 4 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.
SDINBS Boundary Scan Serial Input Data	O	This signal contains the serial data to be applied to an external scan chain and is valid around the falling edge of **TCK**.
SDOUTBS Boundary scan serial output data	IC	This control signal is provided to ease the connection of an external boundary scan chain. This is the serial data out of the boundary 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.
SEQ Sequential address.	O4	This output signal will become HIGH when the address of the next memory cycle will be related to that of the last memory access. The new address will either be the same as the previous one or 4 greater in ARM state, or 2 greater in THUMB state. The signal becomes valid during phase 1 and remains so through phase 2 of the cycle before the cycle whose address it anticipates. It may be used, in combination with the low-order address lines, to indicate that the next cycle can use a fast memory mode (for example DRAM page mode) and/or to bypass the address translation system.

Table 1. Signal Description (Continued)

Name	Type	Description
SHCLKBS Boundary scan shift clock, phase 1	04	This control signal is provided to ease the connection of an external boundary scan chain. **SHCLKBS** is used to clock the master half of the external scan cells. When in the SHIFT-DR state of the state machine and scan chain 3 is selected, **SHCLKBS** 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 should be left unconnected.
SHCLK2BS Boundary scan shift clock, phase 2	04	This control signal is provided to ease the connection of an external boundary scan chain. **SHCLK2BS** is used to clock the master half of the external scan cells. When in the SHIFT-DR state of the state machine and scan chain 3 is selected, **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 should be left unconnected.
TAPSM[3:0] TAP controller state machine	04	This bus reflects the current state of the TAP controller state machine, as shown in The JTAG state machine. These bits change off the rising edge of **TCK**.
TBE Test Bus Enable.	IC	When driven LOW, **TBE** forces the data bus **D[31:0]**, the Address bus **A[31:0]**, plus **LOCK**, **MAS[1:0]**, **nRW**, **nTRANS** and **nOPC** to high impedance. This is as if both **ABE** and **DBE** had both been driven LOW. However, **TBE** does not have an associated scan cell and so allows external signals to be driven high impedance during scan testing. Under normal operating conditions, **TBE** should be held HIGH at all times.
TBIT	O4	When HIGH, this signal denotes that the processor is executing the THUMB instruction set. When LOW, the processor is executing the ARM instruction set. This signal changes in phase 2 in the first execute cycle of a BX instruction.
TCK	IC	Test Clock.
TCK1 TCK, phase 1	04	This clock represents phase 1 of **TCK**. **TCK1** is HIGH when **TCK** is HIGH, although there is a slight phase lag due to the internal clock non-overlap.
TCK2 TCK, phase 2	04	This clock represents phase 2 of **TCK**. **TCK2** is HIGH when **TCK** is LOW, although there is a slight phase lag due to the internal clock non-overlap.**TCK2** is the non-overlapping compliment of **TCK1**.
TDI	IC	Test Data Input.
TDO Test Data Output.	O4	Output from the boundary scan logic.
TMS	IC	Test Mode Select.
VDD Power supply.	P	These connections provide power to the device.
VSS Ground.	P	These connections are the ground reference for all signals.

Programmer's Model

This chapter describes the two operating states of the ARM7TDMI.

Processor Operating States

From the programmer's point of view, the ARM7TDMI can be in one of two states:

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

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

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

Switching State

Entering THUMB state

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

Transition to THUMB state will also occur automatically on return from an exception (IRQ, FIQ, UNDEF, ABORT, SWI etc.), if the exception was entered with the processor in THUMB state.

Entering ARM state

Entry into ARM state happens:

1. On execution of the BX instruction with the state bit clear in the operand register.
2. On the processor taking an exception (IRQ, FIQ, RESET, UNDEF, ABORT, SWI etc.).

In this case, the PC is placed in the exception mode's link register, and execution commences at the exception's vector address.

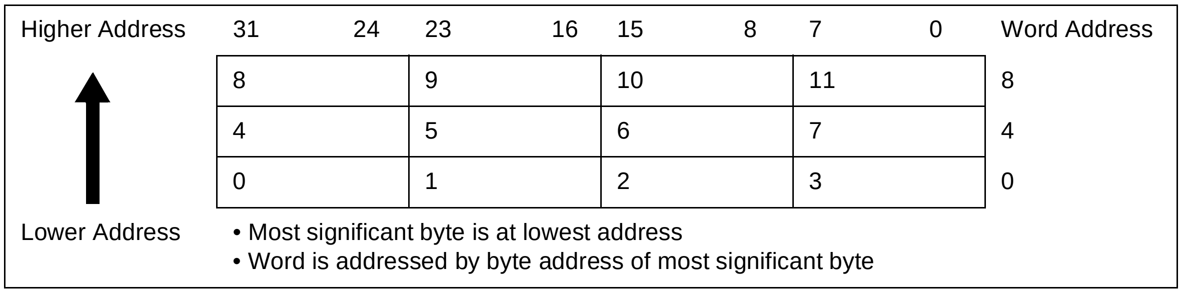
Memory Formats

ARM7TDMI views memory as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second and so on. ARM7TDMI can treat words in memory as being stored either in *Big Endian* or *Little Endian* format.

Big endian format

In Big Endian format, the most significant byte of a word is stored at the lowest numbered byte and the least significant byte at the highest numbered byte. Byte 0 of the memory system is therefore connected to data lines 31 through 24.

Figure 4. Big Endian Addresses of Bytes within Words



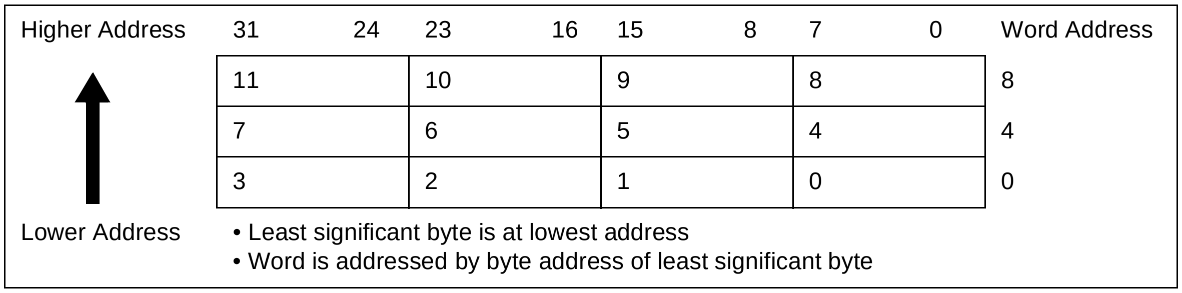
Higher Address	31 24	23 16	15 8	7 0	Word Address
	8	9	10	11	8
	4	5	6	7	4
	0	1	2	3	0
Lower Address					

• Most significant byte is at lowest address
• Word is addressed by byte address of most significant byte

Little endian format

In Little Endian format, the lowest numbered byte in a word is considered the word's least significant byte, and the highest numbered byte the most significant. Byte 0 of the memory system is therefore connected to data lines 7 through 0

Figure 5. Little Endian Addresses of Bytes within Words



Higher Address	31 24	23 16	15 8	7 0	Word Address
	11	10	9	8	8
	7	6	5	4	4
	3	2	1	0	0
Lower Address					

• Least significant byte is at lowest address
• Word is addressed by byte address of least significant byte

Instruction Length

Instructions are either 32 bits long (in ARM state) or 16 bits long (in THUMB state).

Data Types

ARM7TDMI supports byte (8-bit), halfword (16-bit) and word (32-bit) data types. Words must be aligned to four-byte boundaries and half words to two-byte boundaries.

Operating Modes

ARM7TDMI supports seven modes of operation:

User (usr):	The normal ARM program execution state
FIQ (fiq):	Designed to support a data transfer or channel process
IRQ (irq):	Used for general-purpose interrupt handling
Supervisor (svc):	Protected mode for the operating system
Abort mode (abt):	Entered after a data or instruction prefetch abort
System (sys):	A privileged user mode for the operating system
Undefined (und):	Entered when an undefined instruction is executed

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

Registers

ARM7TDMI has a total of 37 registers - 31 general-purpose 32-bit registers and six status registers - but these cannot all be seen at once. The processor state and operating mode dictate which registers are available to the programmer.

The ARM state register set

In ARM state, 16 general registers and one or two status registers are visible at any one time. In privileged (non-User) modes, mode-specific banked registers are switched in. Figure 7 shows which registers are available in each mode: the banked registers are marked with a shaded triangle.

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

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

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

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

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

Figure 6. Register Organization in ARM State

ARM State General Registers and Program Counter

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	◺ R8_fiq	R8	R8	R8	R8
R9	◺ R9_fiq	R9	R9	R9	R9
R10	◺ R10_fiq	R10	R10	R10	R10
R11	◺ R11_fiq	R11	R11	R11	R11
R12	◺ R12_fiq	R12	R12	R12	R12
R13	◺ R13_fiq	◺ R13_svc	◺ R13_abt	◺ R13_irq	◺ R13_und
R14	◺ R14_fiq	◺ R14_svc	◺ R14_abt	◺ R14_irq	◺ R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM State Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	◺ SPSR_fiq	◺ SPSR_svc	◺ SPSR_abt	◺ SPSR_irq	◺ SPSR_und

◺ *= banked register*

The THUMB state register set

The THUMB state register set is a subset of the ARM state set. The programmer has direct access to eight general registers, R0-R7, as well as the Program Counter (PC), a stack pointer register (SP), a link register (LR), and the CPSR. There are banked Stack Pointers, Link Registers and Saved Process Status Registers (SPSRs) for each privileged mode. This is shown in Figure 7.

Figure 7. Register Organization in Thumb State

THUMB State General Registers and Program Counter

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
SP	◺ SP_fiq	◺ SP_svc	◺ SP_abt	◺ SP_irq	◺ SP_und
LR	◺ LR_fiq	◺ LR_svc	◺ LR_abt	◺ LR_irq	◺ LR_und
PC	PC	PC	PC	PC	PC

THUMB State Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	◺ SPSR_fiq	◺ SPSR_svc	◺ SPSR_abt	◺ SPSR_irq	◺ SPSR_und

◺ *= banked register*

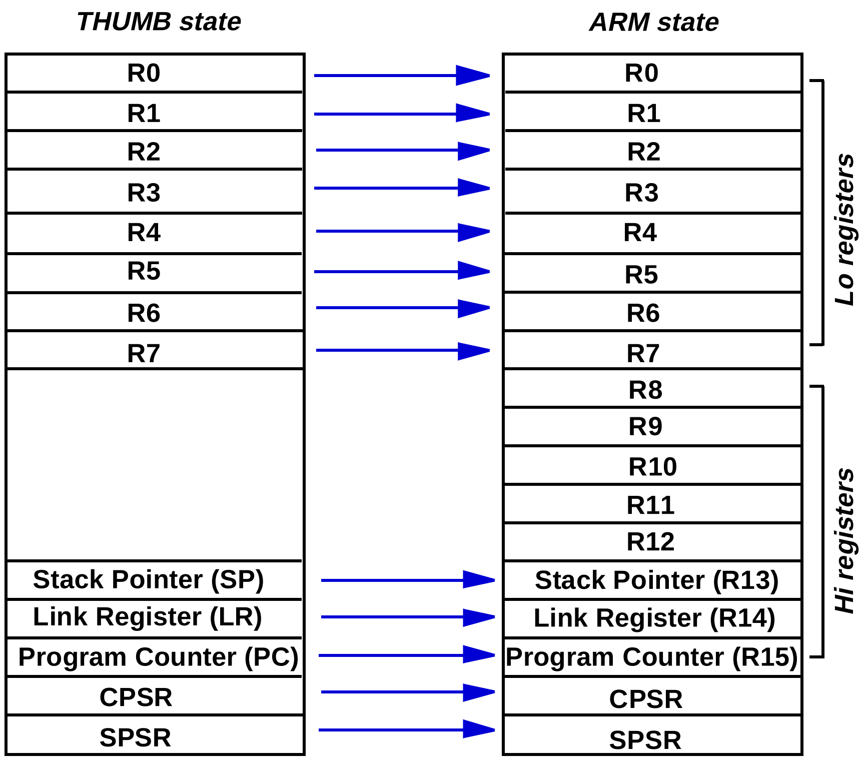
The relationship between ARM and THUMB state registers

The THUMB state registers relate to the ARM state registers in the following way:

- THUMB state R0-R7 and ARM state R0-R7 are identical
- THUMB state CPSR and SPSRs and ARM state CPSR and SPSRs are identical
- THUMB state SP maps onto ARM state R13
- THUMB state LR maps onto ARM state R14
- The THUMB state Program Counter maps onto the ARM state Program Counter (R15)

This relationship is shown in Figure 8.

Figure 8. Mapping of THUMB State Registers onto ARM State Registers



Accessing Hi registers in THUMB state

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

A value may be transferred from a register in the range R0-R7 (a *Lo register*) to a Hi register, and from a Hi register to a Lo register, using special variants of the `MOV` instruction. Hi register values can also be compared against or added to Lo register values with the `CMP` and `ADD` instructions. See Format 5: Hi register operations/branch exchange on page 86.

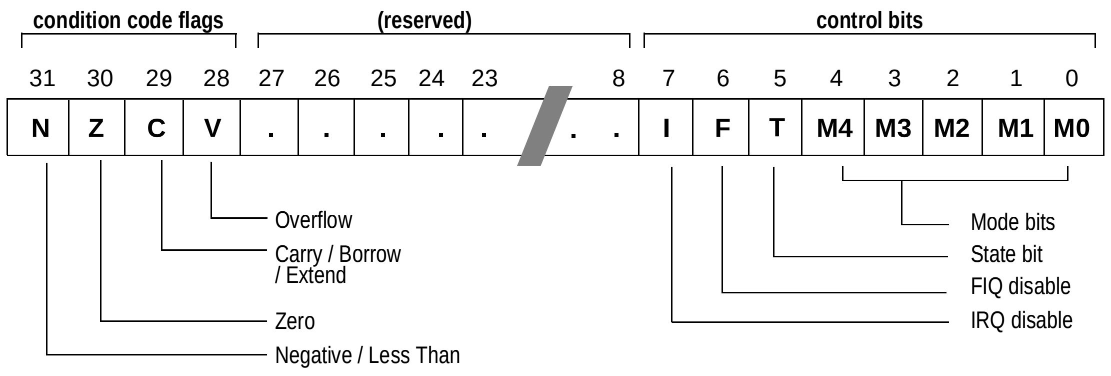
The Program Status Registers

The ARM7TDMI contains a Current Program Status Register (CPSR), plus five Saved Program Status Registers (SPSRs) for use by exception handlers. These registers

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

The arrangement of bits is shown in Figure 9.

Figure 9. Program Status Register Format



The condition code flags

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

In ARM state, all instructions may be executed conditionally: see The Condition Field on page 30 for details.

In THUMB state, only the Branch instruction is capable of conditional execution: see Format 17: software interrupt on page 107.

The control bits

The bottom 8 bits of a PSR (incorporating I, F, T and M[4:0]) are known collectively as the control bits. These will change when an exception arises. If the processor is operating in a privileged mode, they can also be manipulated by software.

The T bit This reflects the operating state. When this bit is set, the processor is executing in THUMB state, otherwise it is executing in ARM state. This is reflected on the **TBIT** external signal.

Note that the software must never change the state of the **TBIT** in the CPSR. If this happens, the processor will enter an unpredictable state.

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

The mode bits The M4, M3, M2, M1 and M0 bits (M[4:0]) are the mode bits. These determine the processor's operating mode, as shown in Table 2. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described shall be used. The user should be aware that if any illegal value is programmed into the mode bits, M[4:0], then the processor will enter an unrecoverable state. If this occurs, reset should be applied.

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

Table 2. PSR Mode Bit Values

M[4:0]	Mode	Visible THUMB state registers	Visible ARM state registers
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq..R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc..R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt..R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und..R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

Exceptions

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

It is possible for several exceptions to arise at the same time. If this happens, they are dealt with in a fixed order - see Exception priorities on page 25.

Action on entering an exception

When handling an exception, the ARM7TDMI:

1. Preserves the address of the next instruction in the appropriate Link Register. If the exception has been entered from ARM state, then the address of the next instruction is copied into the Link Register (that is, current PC + 4 or PC + 8 depending on the exception). (See Table 3 for details). If the exception has been entered from THUMB state, then the value written into the Link Register is the current PC offset by a value such that the program resumes from the correct place on return from the exception. This means that the exception handler need not determine which state the exception was entered from. For example, in the case of SWI, `MOVS PC, R14_svc` will always return to the next instruction regardless of whether the SWI was executed in ARM or THUMB state.
2. Copies the CPSR into the appropriate SPSR
3. Forces the CPSR mode bits to a value which depends on the exception
4. Forces the PC to fetch the next instruction from the relevant exception vector

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

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

Action on leaving an exception

On completion, the exception handler:

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

Note: An explicit switch back to THUMB state is never needed, since restoring the CPSR from the SPSR automatically sets the T bit to the value it held immediately prior to the exception.

Exception entry/exit summary

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

Table 3. Exception Entry/Exit

	Return Instruction	Previous State		Notes
		ARM R14_x	THUMB R14_x	
BL	MOV PC, R14	PC + 4	PC + 2	1
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	1
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	1
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4	2
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4	2
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	1
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8	3
RESET	NA	-	-	4

Notes

1. Where PC is the address of the BL/SWI/Undefined Instruction fetch which had the prefetch abort.
2. Where PC is the address of the instruction which did not get executed since the FIQ or IRQ took priority.
3. Where PC is the address of the Load or Store instruction which generated the data abort.
4. The value saved in R14_svc upon reset is unpredictable.

FIQ

The FIQ (Fast Interrupt Request) exception is designed to support a data transfer or channel process, and in ARM state has sufficient private registers to remove the need for register saving (thus minimising the overhead of context switching).

FIQ is externally generated by taking the **nFIQ** input LOW. This input can except either synchronous or asynchronous transitions, depending on the state of the **ISYNC** input signal. When **ISYNC** is LOW, **nFIQ** and **nIRQ** are considered asynchronous, and a cycle delay for synchronization is incurred before the interrupt can affect the processor flow.

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

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

FIQ may be disabled by setting the CPSR's F flag (but note that this is not possible from User mode). If the F flag is clear, ARM7TDMI checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

IRQ

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

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

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

Abort

An abort indicates that the current memory access cannot be completed. It can be signalled by the external **ABORT** input. ARM7TDMI checks for the abort exception during memory access cycles.

There are two types of abort:

Prefetch abort occurs during an instruction prefetch.

Data abort occurs during a data access.

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

If a data abort occurs, the action taken depends on the instruction type:

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

The abort mechanism allows the implementation of a demand paged virtual memory system. In such a system the processor is allowed to generate arbitrary addresses. When the data at an address is unavailable, the Memory Management Unit (MMU) signals an abort. The abort handler must then work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

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

`SUBS PC,R14_abt,#4` for a prefetch abort, or

`SUBS PC,R14_abt,#8` for a data abort

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

Software interrupt

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

```
MOV PC, R14_svc
```

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

Undefined instruction

When ARM7TDMI comes across an instruction which it cannot handle, it takes the undefined instruction trap. This mechanism may be used to extend either the THUMB or ARM instruction set by software emulation.

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

```
MOVS PC,R14_und
```

This restores the CPSR and returns to the instruction following the undefined instruction.

Exception vectors

The following table shows the exception vector addresses.

Table 4. Exception Vectors

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

Exception priorities

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

Highest priority:

1. Reset
2. Data abort
3. FIQ
4. IRQ
5. Prefetch abort

Lowest priority:

6. Undefined Instruction, Software interrupt.

Not all exceptions can occur at once:
Undefined Instruction and Software Interrupt are mutually exclusive, since they each correspond to particular (non-overlapping) decodings of the current instruction.

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (ie the CPSR's F flag is clear), ARM7TDMI enters the data abort handler and then immediately proceeds to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection. The time for this exception entry should be added to worst-case FIQ latency calculations.

Interrupt Latencies

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchroniser (*Tsyncmax* if asynchronous), plus the time for the longest instruction to complete (*Tldm*, the longest instruction is an LDM which loads all the registers including the PC), plus the time for the data abort entry (*Texc*), plus the time for FIQ entry (*Tfiq*). At the end of this time ARM7TDMI will be executing the instruction at 0x1C.

Tsyncmax is 3 processor cycles, *Tldm* is 20 cycles, *Texc* is 3 cycles, and *Tfiq* is 2 cycles. The total time is therefore 28 processor cycles. This is just over 1.4 microseconds in a system which uses a continuous 20 MHz processor clock. The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time. The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchroniser (*Tsyncmin*) plus *Tfiq*. This is 4 processor cycles.

Reset

When the **nRESET** signal goes LOW, ARM7TDMI abandons the executing instruction and then continues to fetch instructions from incrementing word addresses.

When **nRESET** goes HIGH again, ARM7TDMI:

1. Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and SPSR is not defined.
2. Forces M[4:0] to 10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR's T bit.
3. Forces the PC to fetch the next instruction from address 0x00.
4. Execution resumes in ARM state.

ARM Instruction Set

This chapter describes the ARM instruction set.

Instruction Set Summary

Format summary

The ARM instruction set formats are shown below.

Figure 10. ARM Instruction Set Formats

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2												*Data Processing / PSR Transfer*
Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				*Multiply*
Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				*Multiply Long*
Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				*Single Data Swap*
Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				*Branch and Exchange*
Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				*Halfword Data Transfer: register offset*
Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				*Halfword Data Transfer: immediate offset*
Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset												*Single Data Transfer*
Cond				0	1	1																					1					*Undefined*
Cond				1	0	0	P	U	S	W	L	Rn				Register List																*Block Data Transfer*
Cond				1	0	1	L	Offset																								*Branch*
Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset								*Coprocessor Data Transfer*
Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm				*Coprocessor Data Operation*
Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm				*Coprocessor Register Transfer*
Cond				1	1	1	1	Ignored by processor																								*Software Interrupt*
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Note: Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 6 changed to a 1. These instructions should not be used, as their action may change in future ARM implementations.

Instruction summary

Table 5. The ARM Instruction Set

Mnemonic	Instruction	Action	See Page
ADC	Add with carry	Rd := Rn + Op2 + Carry	34
ADD	Add	Rd := Rn + Op2	34
AND	AND	Rd := Rn AND Op2	34
B	Branch	R15 := address	32
BIC	Bit Clear	Rd := Rn AND NOT Op2	34
BL	Branch with Link	R14 := R15, R15 := address	32
BX	Branch and Exchange	R15 := Rn, T bit := Rn[0]	31
CDP	Coprocesor Data Processing	(Coprocessor-specific)	66
CMN	Compare Negative	CPSR flags := Rn + Op2	34
CMP	Compare	CPSR flags := Rn - Op2	34
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)	34
LDC	Load coprocessor from memory	Coprocessor load	68
LDM	Load multiple registers	Stack manipulation (Pop)	56
LDR	Load register from memory	Rd := (address)	48, 52
MCR	Move CPU register to coprocessor register	cRn := rRn {<op>cRm}	70
MLA	Multiply Accumulate	Rd := (Rm * Rs) + Rn	44, 46
MOV	Move register or constant	Rd : = Op2	34
MRC	Move from coprocessor register to CPU register	Rn := cRn {<op>cRm}	70
MRS	Move PSR status/flags to register	Rn := PSR	40
MSR	Move register to PSR status/flags	PSR := Rm	40
MUL	Multiply	Rd := Rm * Rs	44, 46
MVN	Move negative register	Rd := 0xFFFFFFFF EOR Op2	34
ORR	OR	Rd := Rn OR Op2	34
RSB	Reverse Subtract	Rd := Op2 - Rn	34
RSC	Reverse Subtract with Carry	Rd := Op2 - Rn - 1 + Carry	34
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry	34
STC	Store coprocessor register to memory	address := CRn	68
STM	Store Multiple	Stack manipulation (Push)	56
STR	Store register to memory	<address> := Rd	48, 52
SUB	Subtract	Rd := Rn - Op2	34
SWI	Software Interrupt	OS call	64
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm	62
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2	34
TST	Test bits	CPSR flags := Rn AND Op2	34

The Condition Field

In ARM state, all instructions are conditionally executed according to the state of the CPSR condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfils the conditions encoded by the field, the instruction is executed, otherwise it is ignored.

There are sixteen possible conditions, each represented by a two-character suffix that can be appended to the instruction's mnemonic. For example, a Branch (B in assembly language) becomes BEQ for "Branch if Equal", which means the Branch will only be taken if the Z flag is set.

In practice, fifteen different conditions may be used: these are listed in Table 6. The sixteenth (1111) is reserved, and must not be used.

In the absence of a suffix, the condition field of most instructions is set to "Always" (sufix AL). This means the instruction will always be executed regardless of the CPSR condition codes.

Table 6. Condition Code Summary

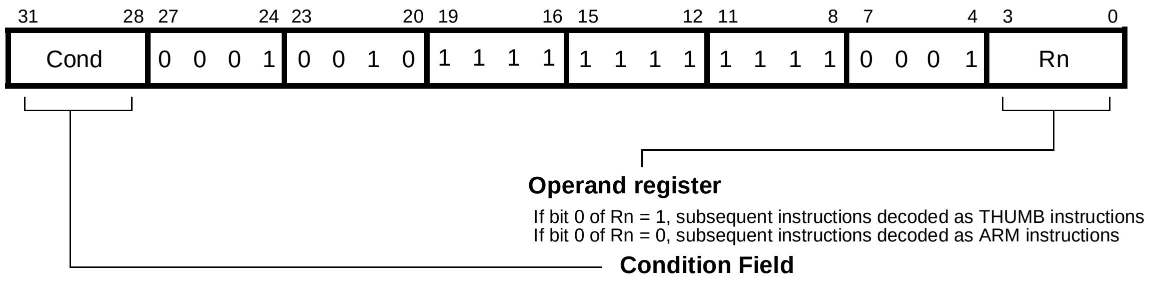
Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Branch and Exchange (BX)

This instruction is only executed if the condition is true. The various conditions are defined in Table 6.

This instruction performs a branch by copying the contents of a general register, Rn, into the program counter, PC. The branch causes a pipeline flush and refill from the address specified by Rn. This instruction also permits the instruction set to be exchanged. When the instruction is executed, the value of Rn[0] determines whether the instruction stream will be decoded as ARM or THUMB instructions.

Figure 11. Branch and Exchange Instructions


31
28
27
24
23
20
19
16
15
12
11
8
7
4
3
0
Cond
0 0 0 1
0 0 1 0
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
Rn
Operand register
If bit 0 of Rn = 1, subsequent instructions decoded as THUMB instructions
If bit 0 of Rn = 0, subsequent instructions decoded as ARM instructions
Condition Field

Instruction cycle times

The BX instruction takes 2S + 1N cycles to execute, where S and N are as defined in Cycle Types.

Assembler syntax

BX - branch and exchange.

```
BX{cond} Rn
```

{cond}: Two character condition mnemonic. See Table 6.

Rn: is an expression evaluating to a valid register number.

Using R15 as an operand

If R15 is used as an operand, the behaviour is undefined.

Examples

```
      ADR R0, Into_THUMB + 1; Generate branch target address
                     ; and set bit 0 high - hence
                     ; arrive in THUMB state.
      BX R0          ; Branch and change to THUMB
                     ; state.
      CODE16         ; Assemble subsequent code as
Into_THUMB           ; THUMB instructions

.
.
      ADR R5, Back_to_ARM: Generate branch target to word
                     : aligned ; address - hence bit 0
                     ; is low and so change back to ARM
                     ; state.
      BX R5          ; Branch and change back to ARM
                     ; state.
      .
      .
      ALIGN             ; Word align
      CODE32            ; Assemble subsequent code as ARM
Back_to_ARM          ; instructions

      .
      .
```

Branch and Branch with Link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined Table 6. The instruction encoding is shown in Figure 12 below.

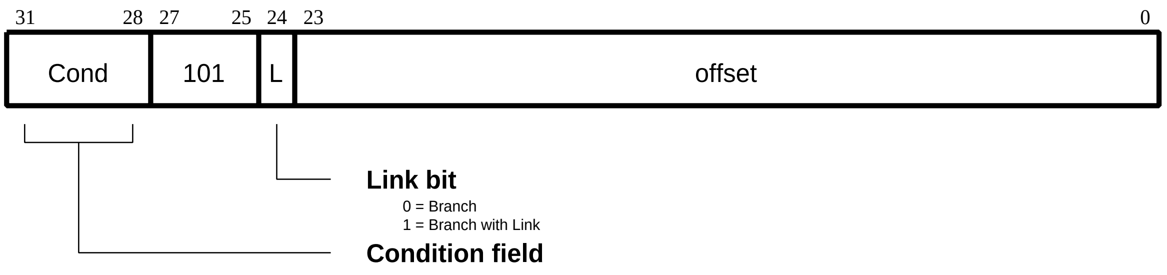


Figure 12. Branch Instructions

Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.

Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC and R14[1:0] are always cleared.

To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

Instruction cycle times

Branch and Branch with Link instructions take 2S + 1N incremental cycles, where S and N are as defined in Cycle Types.

Assembler syntax

Items in {} are optional. Items in <> must be present.

```
B{L}{cond} <expression>
```

{L} is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.

{cond} is a two-character mnemonic as shown in Table 6. If absent then AL (ALways) will be used.

<expression> is the destination. The assembler calculates the offset.

Examples

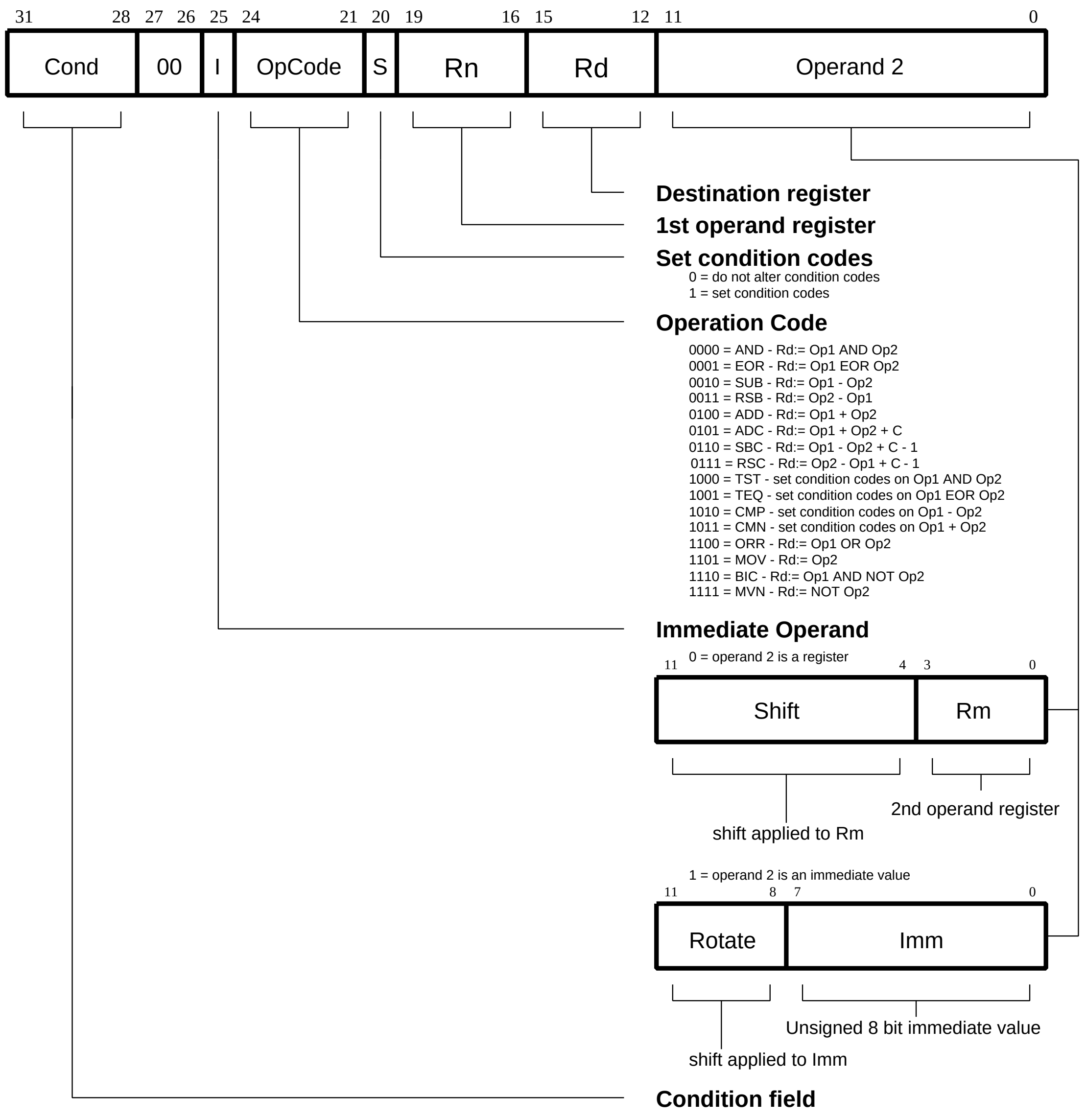
```
here  BAL      here  ; assembles to 0xEAFFFFFE (note effect of
                     ; PC offset).
      B        there ; Always condition used as default.
      CMP      R1,#0 ; Compare R1 with zero and branch to fred
                     ; if R1 was zero, otherwise continue
      BEQ      fred  ; continue to next instruction.

      BL       sub+ROM; Call subroutine at computed address.
      ADDS     R1,#1 ; Add 1 to register 1, setting CPSR flags
                     ; on the result then call subroutine if
      BLCC     sub   ; the C flag is clear, which will be the
                     ; case unless R1 held 0xFFFFFFFF.
```

Data Processing

The data processing instruction is only executed if the condition is true. The conditions are defined in Table 6.

The instruction encoding is shown in Figure 13 below.

Figure 13. Data Processing Instructions



31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11 4 3 0
Shift
Rm
2nd operand register
shift applied to Rm
1 = operand 2 is an immediate value
11 8 7 0
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn).

The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction.

Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set. The instructions and their effects are listed in Table 7**.**

CPSR flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of bit 31 of the result.

Table 7. ARM Data Processing Instructions

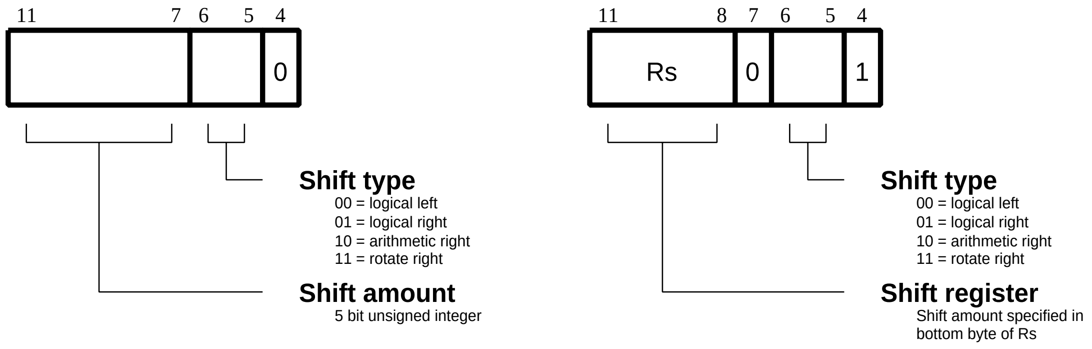
Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in Figure 14.

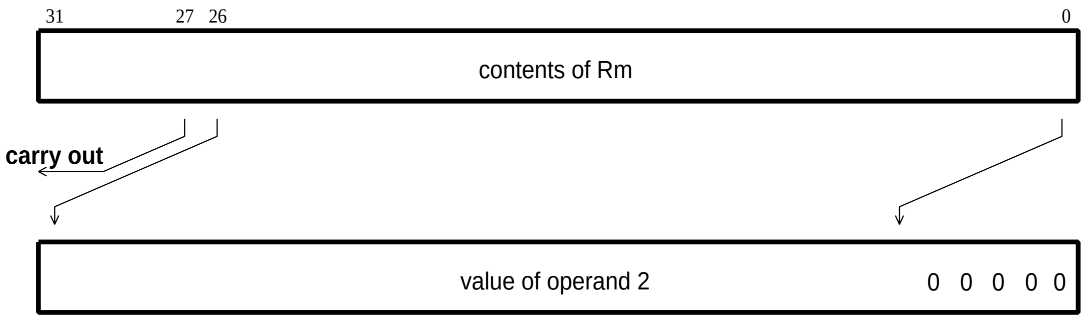
Figure 14. ARM Shift Operations



Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in Figure 15.

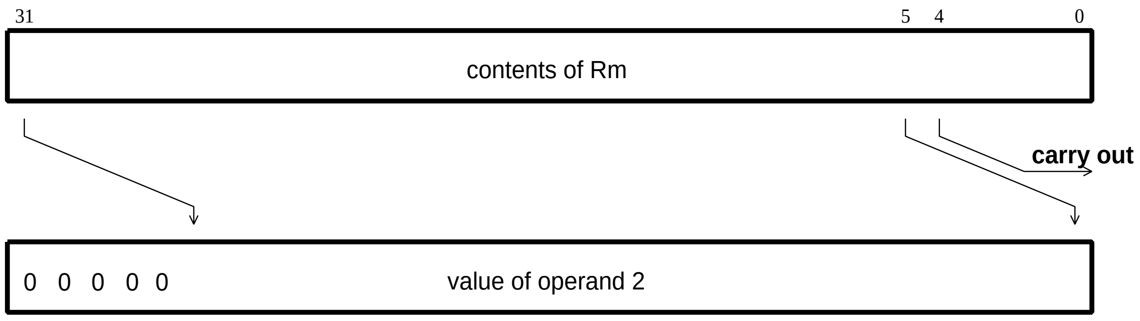
Figure 15. Logical Shift Left



Note: LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in Figure 16.

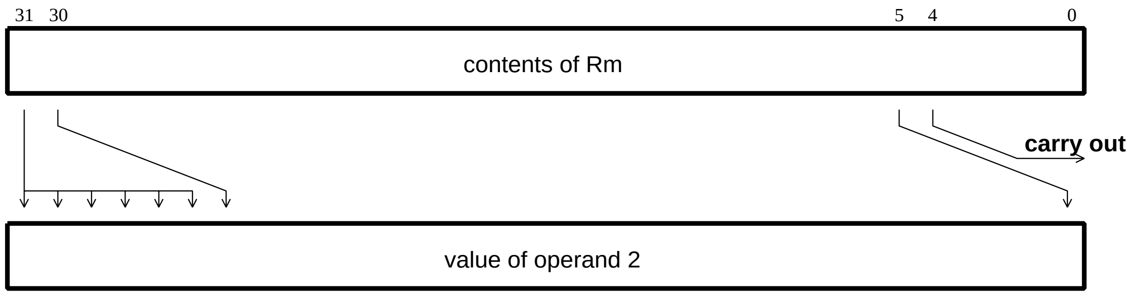
Figure 16. Logical Shift Right



The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in Figure 17.

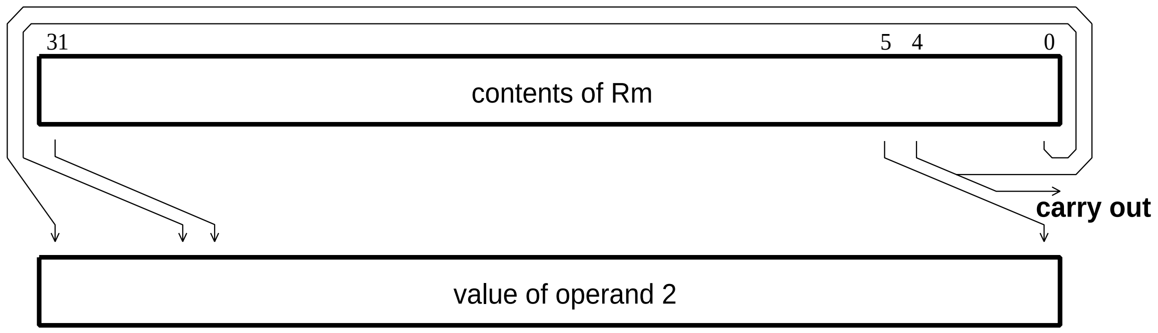
Figure 17. Arithmetic Shift Right



The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeros, according to the value of bit 31 of Rm.

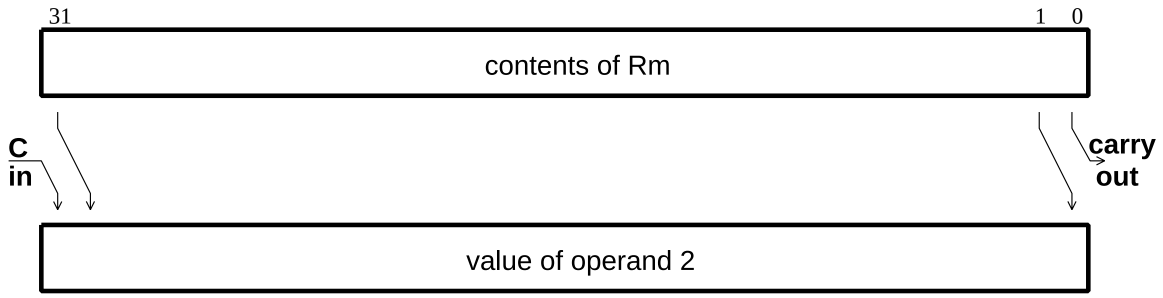
Rotate right (ROR) operations reuse the bits which “over-shoot” in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in Figure 18.

Figure 18. Rotate Right



The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in Figure 19.

Figure 19. Rotate Right Extended



Register specified shift amount

Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output.

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shift described above:

1. LSL by 32 has result zero, carry out equal to bit 0 of Rm.
2. LSL by more than 32 has result zero, carry out zero.
3. LSR by 32 has result zero, carry out equal to bit 31 of Rm.
4. LSR by more than 32 has result zero, carry out zero.
5. ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.
6. ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.
7. ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note: The zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or undefined instruction.

Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

Writing to R15

When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags as described above.

When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of instruction should not be used in User mode.

Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction the register is used directly.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount the PC will be 12 bytes ahead.

TEQ, TST, CMP and CMN opcodes

Note: TEQ, TST, CMP and CMN do not write the result of their operation but do set flags in the CPSR. An assembler should always set the S flag for these instructions even if this is not specified in the mnemonic.

The TEQP form of the TEQ instruction used in earlier ARM processors must not be used: the PSR transfer operations should be used instead.

The action of TEQP in the ARM7TDMI is to move SPSR_<mode> to the CPSR if the processor is in a privileged mode and to do nothing if in User mode.

Instruction cycle times

Data Processing instructions vary in the number of incremental cycles taken as follows:

Table 8. Incremental Cycle Times

Processing Type	Cycles
Normal Data Processing	1S
Data Processing with register specified shift	1S + 1I
Data Processing with PC written	2S + 1N
Data Processing with register specified shift and PC written	2S + 1N + 1I

S, N and I are as defined in Cycle Types.

Assembler syntax

1. MOV,MVN (single operand instructions.)

   `<opcode>{cond}{S} Rd,<Op2>`

2. CMP,CMN,TEQ,TST (instructions which do not produce a result.)

   `<opcode>{cond} Rn,<Op2>`

3. AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC

   `<opcode>{cond}{S} Rd,Rn,<Op2>`

where:

<Op2> is Rm{,<shift>} or,<#expression>

{cond} is a two-character condition mnemonic. See Table 6.

{S} set condition codes if S present (implied for CMP, CMN, TEQ, TST).

Rd, Rn and Rm are expressions evaluating to a register number.

<#expression> if this is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

<shift> is <shiftname> <register> or <shiftname> #expression, or RRX (rotate right one bit with extend).

<shiftname>s are: ASL, LSL, LSR, ASR, ROR. (ASL is a synonym for LSL, they assemble to the same code.)

Examples

```
ADDEQ R2,R4,R5        ; If the Z flag is set make R2:=R4+R5
TEQS  R4,#3           ; test R4 for equality with 3.
                      ; (The S is in fact redundant as the
                      ; assembler inserts it automatically.)
SUB   R4,R5,R7,LSR R2; Logical right shift R7 by the number in
                      ; the bottom byte of R2, subtract result
                      ; from R5, and put the answer into R4.
MOV   PC,R14          ; Return from subroutine.
MOVS  PC,R14          ; Return from exception and restore CPSR
                      ; from SPSR_mode.
```

PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6.

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in Figure 20.

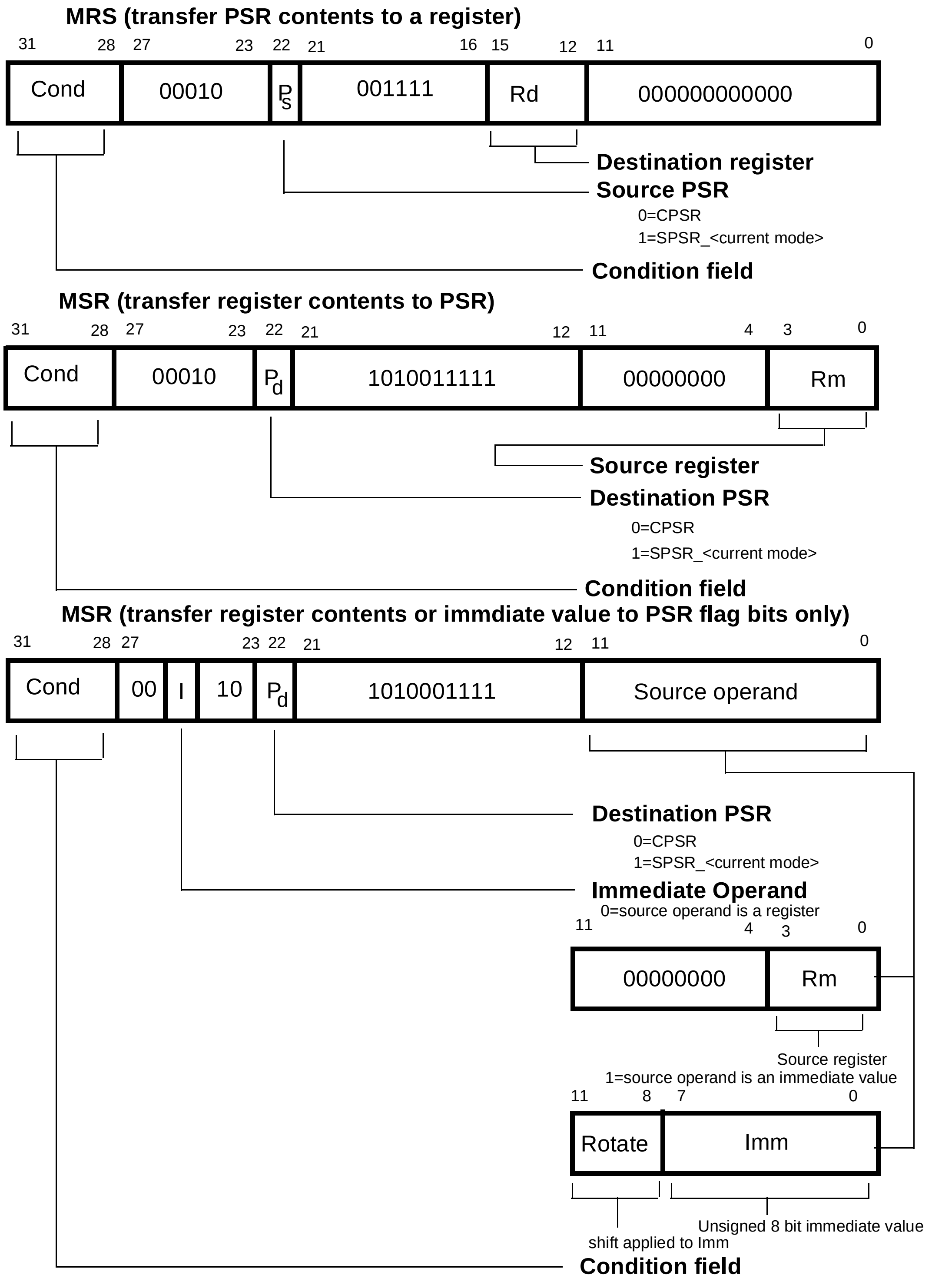
These instructions allow access to the CPSR and SPSR registers. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register. The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register.

The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32 bit immediate value are written to the top four bits of the relevant PSR.

Operand restrictions

- In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.
- Note that the software must never change the state of the T bit in the CPSR. If this happens, the processor will enter an unpredictable state.
- The SPSR register which is accessed depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.
- You must not specify R15 as the source or destination register.
- Also, do not attempt to access an SPSR in User mode, since no such register exists.

Figure 20. PSR Transfer



Reserved bits

Only twelve bits of the PSR are defined in ARM7TDMI (N,Z,C,V,I,F, T & M[4:0]); the remaining bits are reserved for use in future versions of the processor. Refer to Figure 9 for a full description of the PSR bits.

To ensure the maximum compatibility between ARM7TDMI programs and future processors, the following rules should be observed:

- The reserved bits should be preserved when changing the value in a PSR.
- Programs should not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

A read-modify-write strategy should therefore be used when altering the control bits of any PSR register; this involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits and then transferring the modified value back to the PSR register using the MSR instruction.

Example

The following sequence performs a mode change:

```
MRS   R0,CPSR                ; Take a copy of the CPSR.
BIC   R0,R0,#0x1F            ; Clear the mode bits.
ORR   R0,R0,#new_mode        ; Select new mode
MSR   CPSR,R0                ; Write back the modified
                             ; CPSR.
```

When the aim is simply to change the condition code flags in a PSR, a value can be written directly to the flag bits without disturbing the control bits. The following instruction sets the N,Z,C and V flags:

```
MSR   CPSR_flg,#0xF0000000   ; Set all the flags
                             ; regardless of their
                             ; previous state (does not
                             ; affect any control bits).
```

No attempt should be made to write an 8 bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.

Instruction cycle times

PSR Transfers take 1S incremental cycles, where S is as defined in Cycle Types.

Assembler syntax

1. MRS - transfer PSR contents to a register

   `MRS{cond} Rd,<psr>`

2. MSR - transfer register contents to PSR

   `MSR{cond} <psr>,Rm`

3. MSR - transfer register contents to PSR flag bits only

   `MSR{cond} <psrf>,Rm`

The most significant four bits of the register contents are written to the N,Z,C & V flags respectively.

4. MSR - transfer immediate value to PSR flag bits only

   `MSR{cond} <psrf>,<#expression>`

The expression should symbolise a 32 bit value of which the most significant four bits are written to the N,Z,C and V flags respectively.

Key:

{cond} two-character condition mnemonic. See Table 6.

Rd and Rm are expressions evaluating to a register number other than R15

<psr> is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)

<psrf> is CPSR_flg or SPSR_flg

<#expression>where this is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

Examples

In User mode the instructions behave as follows:

```
MSR   CPSR_all,Rm    ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg,Rm    ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg,#0xA0000000; CPSR[31:28] <- 0xA
                     ;(set N,C; clear Z,V)
MRS   Rd,CPSR        ; Rd[31:0] <- CPSR[31:0]
```

In privileged modes the instructions behave as follows:

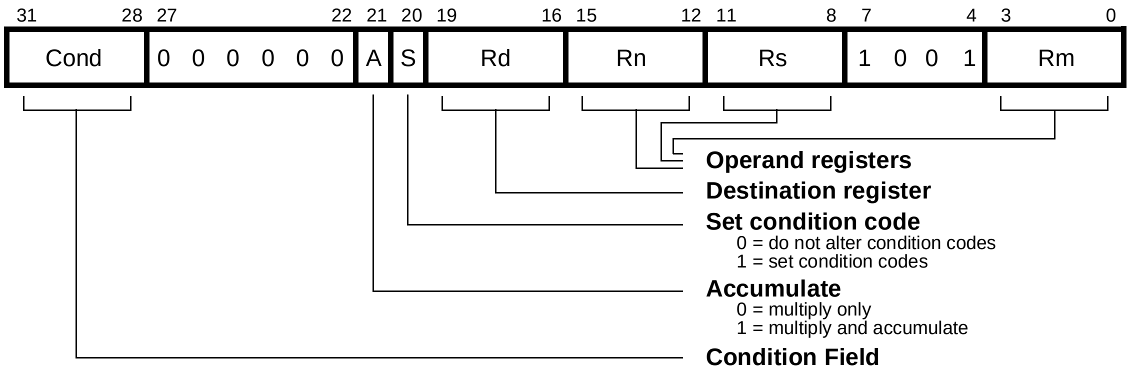
```
MSR   CPSR_all,Rm    ; CPSR[31:0]  <- Rm[31:0]
MSR   CPSR_flg,Rm    ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg,#0x50000000; CPSR[31:28] <- 0x5
                     ;(set Z,V; clear N,C)
MRS   Rd,CPSR        ; Rd[31:0] <- CPSR[31:0]
MSR   SPSR_all,Rm    ;SPSR_<mode>[31:0]<- Rm[31:0]
MSR   SPSR_flg,Rm    ; SPSR_<mode>[31:28] <- Rm[31:28]
MSR   SPSR_flg,#0xC0000000; SPSR_<mode>[31:28] <- 0xC
                     ;(set N,Z; clear C,V)
MRS   Rd,SPSR        ; Rd[31:0] <- SPSR_<mode>[31:0]
```

Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 21.

The multiply and multiply-accumulate instructions use an 8 bit Booth's algorithm to perform integer multiplication.

Figure 21. Multiply Instructions



The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

Both forms of the instruction work on operands which may be considered as signed (2’s complement) or unsigned integers.

The results of a signed multiply and of an unsigned multiply of 32 bit operands differ only in the upper 32 bits - the low 32 bits of the signed and unsigned results are identical. As these instructions only produce the low 32 bits of a multiply, they can be used for both signed and unsigned multiplies.

For example consider the multiplication of the operands:

Operand A	Operand B	Result
0xFFFFFFF6	0x00000014	0xFFFFFF38

If the operands are interpreted as signed

Operand A has the value -10, operand B has the value 20, and the result is -200 which is correctly represented as 0xFFFFFF38

If the operands are interpreted as unsigned

Operand A has the value 4294967286, operand B has the value 20 and the result is 85899345720, which is represented as 0x13FFFFFF38, so the least significant 32 bits are 0xFFFFFF38.

Operand restrictions

The destination register Rd must not be the same as the operand register Rm. R15 must not be used as an operand or as the destination register.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N (Negative) and Z (Zero) flags are set correctly on the result (N is made equal to bit 31 of the result, and Z is set if and only if the result is zero). The C (Carry) flag is set to a meaningless value and the V (oVerflow) flag is unaffected.

Instruction cycle times

MUL takes 1S + mI and MLA 1S + (m+1)I cycles to execute, where S and I are as defined in Cycle Types.

m is the number of 8 bit multiplier array cycles required to complete the multiply, which is controlled by the value of the multiplier operand specified by Rs. Its possible values are as follows

1 if bits [32:8] of the multiplier operand are all zero or all one.

2 if bits [32:16] of the multiplier operand are all zero or all one.

3 if bits [32:24] of the multiplier operand are all zero or all one.

4 in all other cases.

Assembler syntax

```
MUL{cond}{S} Rd,Rm,Rs
MLA{cond}{S} Rd,Rm,Rs,Rn
```

{cond} two-character condition mnemonic. See Table 6.

{S} set condition codes if S present

Rd, Rm, Rs and Rnare expressions evaluating to a register number other than R15.

Examples

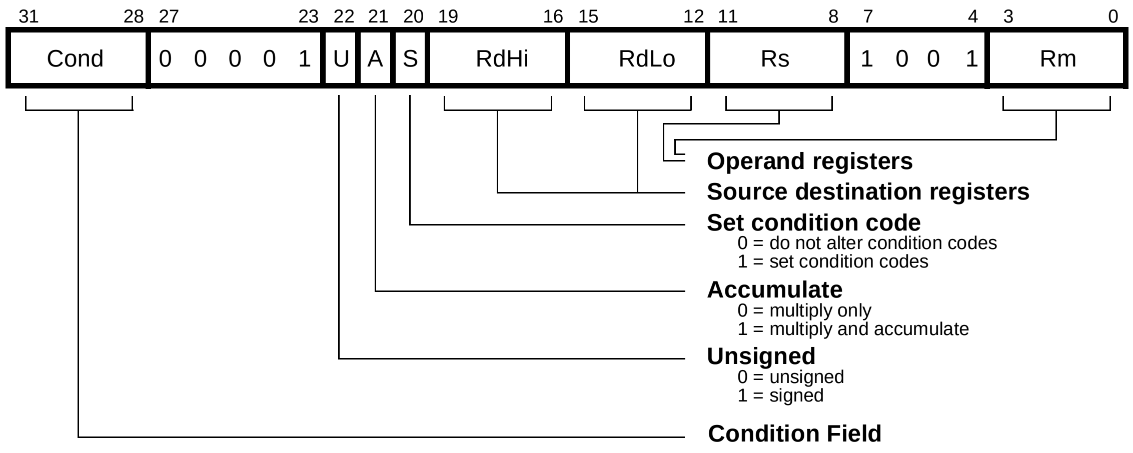
```
MUL             R1,R2,R3 ; R1:=R2*R3
MLAEQS          R1,R2,R3,R4; Conditionally R1:=R2*R3+R4,
                         ; setting condition codes.
```

Multiply Long and Multiply-Accumulate Long (MULL,MLAL)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 22.

The multiply long instructions perform integer multiplication on two 32 bit operands and produce 64 bit results. Signed and unsigned multiplication each with optional accumulate give rise to four variations.

Figure 22. Multiply Long Instructions



The multiply forms (UMULL and SMULL) take two 32 bit numbers and multiply them to produce a 64 bit result of the form RdHi,RdLo := Rm * Rs. The lower 32 bits of the 64 bit result are written to RdLo, the upper 32 bits of the result are written to RdHi.

The multiply-accumulate forms (UMLAL and SMLAL) take two 32 bit numbers, multiply them and add a 64 bit number to produce a 64 bit result of the form RdHi,RdLo := Rm * Rs + RdHi,RdLo. The lower 32 bits of the 64 bit number to add is read from RdLo. The upper 32 bits of the 64 bit number to add is read from RdHi. The lower 32 bits of the 64 bit result are written to RdLo. The upper 32 bits of the 64 bit result are written to RdHi.

The UMULL and UMLAL instructions treat all of their operands as unsigned binary numbers and write an unsigned 64 bit result. The SMULL and SMLAL instructions treat all of their operands as two's-complement signed numbers and write a two's-complement signed 64 bit result.

Operand restrictions

- R15 must not be used as an operand or as a destination register.
- RdHi, RdLo, and Rm must all specify different registers.

CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N and Z flags are set correctly on the result (N is equal to bit 63 of the result, Z is set if and only if all 64 bits of the result are zero). Both the C and V flags are set to meaningless values.

Instruction cycle times

MULL takes 1S + (m+1)I and MLAL 1S + (m+2)I cycles to execute, where *m* is the number of 8 bit multiplier array cycles required to complete the multiply, which is controlled by the value of the multiplier operand specified by Rs.

Its possible values are as follows:

For signed instructions SMULL, SMLAL:

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

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

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

4 in all other cases.

For unsigned instructions UMULL, UMLAL:

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 in all other cases.

S and I are as defined in Cycle Types.

Assembler syntax

Table 9. Assembler Syntax Descriptions

Mnemonic	Description	Purpose
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	Unsigned Multiply Long	32 x 32 = 64
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	Unsigned Multiply & Accumulate Long	32 x 32 + 64 = 64
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	Signed Multiply Long	32 x 32 = 64
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	Signed Multiply & Accumulate Long	32 x 32 + 64 = 64

where:

{cond} two-character condition mnemonic. See Table 6.

{S} set condition codes if S present

RdLo, RdHi, Rm, Rs are expressions evaluating to a register number other than R15.

Examples

```
UMULL          R1,R4,R2,R3; R4,R1:=R2*R3
UMLALS         R1,R5,R2,R3; R5,R1:=R2*R3+R5,R1 also setting
                        ; condition codes
```

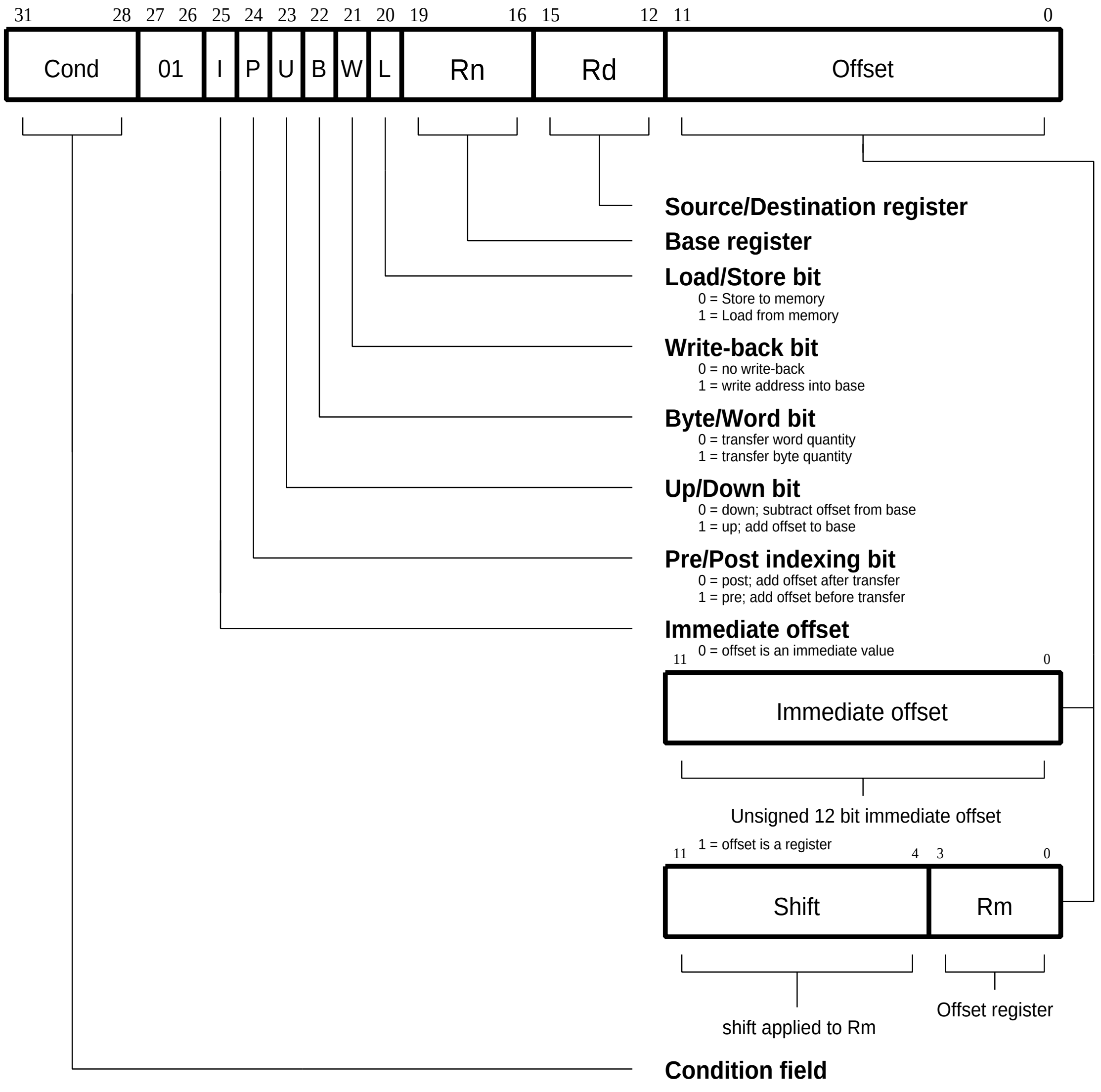
Single Data Transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 23.

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register.

The result of this calculation may be written back into the base register if auto-indexing is required.

Figure 23. Single Data Transfer Instructions



Offsets and auto-indexing

The offset from the base may be either a 12 bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces non-privileged mode for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this hardware.

Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See Shifts.

Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM7TDMI register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the **BIGEND** control signal. The two possible configurations are described below.

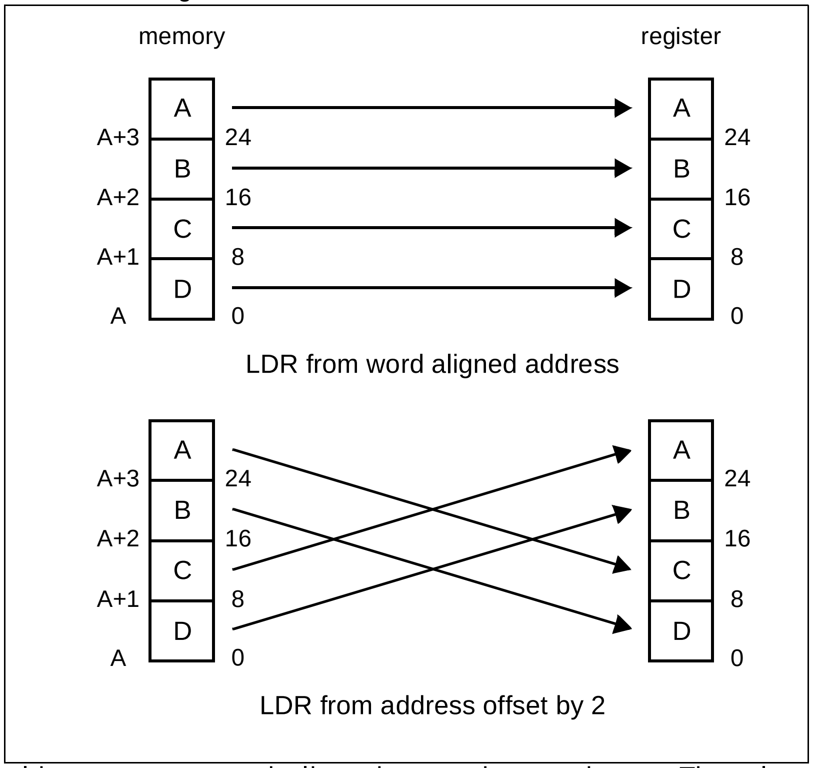
Little endian configuration

A byte load (LDRB) expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros. Please see Figure 5.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits. This is illustrated in Figure 24.

Figure 24. Little Endian Offset Addressing



A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

Big endian configuration

A byte load (LDRB) expects the data on data bus inputs 31 through 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register and the remaining bits of the register are filled with zeros. Please see Figure 4.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 15 through 8.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

Use of R15

Write-back must not be specified if R15 is specified as the base register (Rn). When using R15 as the base register you must remember it contains an address 8 bytes on from the address of the current instruction.

R15 must not be specified as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 12.

Restriction on the use of base register

When configured for late aborts, the following example code is difficult to unwind as the base register, Rn, gets updated before the abort handler starts. Sometimes it may be impossible to calculate the initial value.

After an abort, the following example code is difficult to unwind as the base register, Rn, gets updated before the abort handler starts. Sometimes it may be impossible to calculate the initial value.

Example:

```
LDRR0,[R1],R1
```

Therefore a post-indexed LDR or STR where Rm is the same register as Rn should not be used.

Data aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor **ABORT** input HIGH whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Instruction cycle times

Normal LDR instructions take 1S + 1N + 1I and LDR PC take 2S + 2N +1I incremental cycles, where S,N and I are as defined in Cycle Types.

STR instructions take 2N incremental cycles to execute.

Assembler syntax

`<LDR|STR>{cond}{B}{T} Rd,<Address>`

where:

LDR load from memory into a register

STR store from a register into memory

{cond} two-character condition mnemonic. See Table 6.

{B} if B is present then byte transfer, otherwise word transfer

{T} if T is present the W bit will be set in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.

Rd is an expression evaluating to a valid register number.

Rn and Rm are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7TDMI pipelining. In this case base write-back should not be specified.

<Address>can be:

1. An expression which generates an address:

   `<expression>`

   The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

2. A pre-indexed addressing specification:

`[Rn]`	offset of zero
`[Rn,<#expression>]{!}`	offset of <expression> bytes
`[Rn,{+/-}Rm{,<shift>}]{!}`	offset of +/- contents of index register, shifted by <shift>

3. A post-indexed addressing specification:

`[Rn],<#expression>`	offset of <expression> bytes
`[Rn],{+/-}Rm{,<shift>}`	offset of +/- contents of index register, shifted as by <shift>.

<shift> general shift operation (see data processing instructions) but you cannot specify the shift amount by a register.

{!} writes back the base register (set the W bit) if! is present.

Examples

```
STR   R1,[R2,R4]!       ; Store R1 at R2+R4 (both of which are
                        ; registers) and write back address to
                        ; R2.
STR   R1,[R2],R4        ; Store R1 at R2 and write back
                        ; R2+R4 to R2.
LDR   R1,[R2,#16]       ; Load R1 from contents of R2+16, but
                        ; don't write back.
LDR   R1,[R2,R3,LSL#2]  ; Load R1 from contents of R2+R3*4.
LDREQBR1,[R6,#5]        ; Conditionally load byte at R6+5 into
                        ; R1 bits 0 to 7, filling bits 8 to 31
                        ; with zeros.
STR   R1,PLACE          ; Generate PC relative offset to
                        ; address PLACE.
      •
PLACE
```

Halfword and Signed Data Transfer(LDRH/STRH/LDRSB/LDRSH)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Table 25, below, and Figure 26.

These instructions are used to load or store half-words of data and also load sign-extended bytes or half-words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if auto-indexing is required.

Figure 25. Halfword and Signed Data Transfer with Register Offset

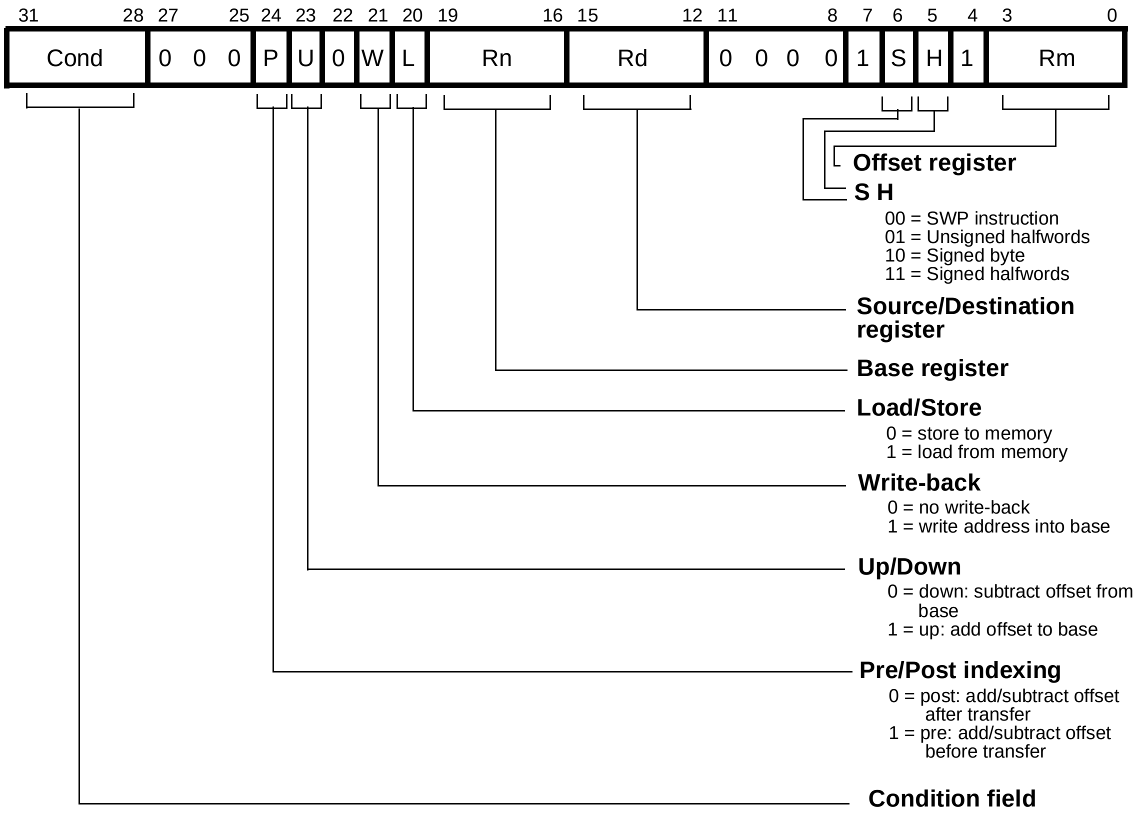
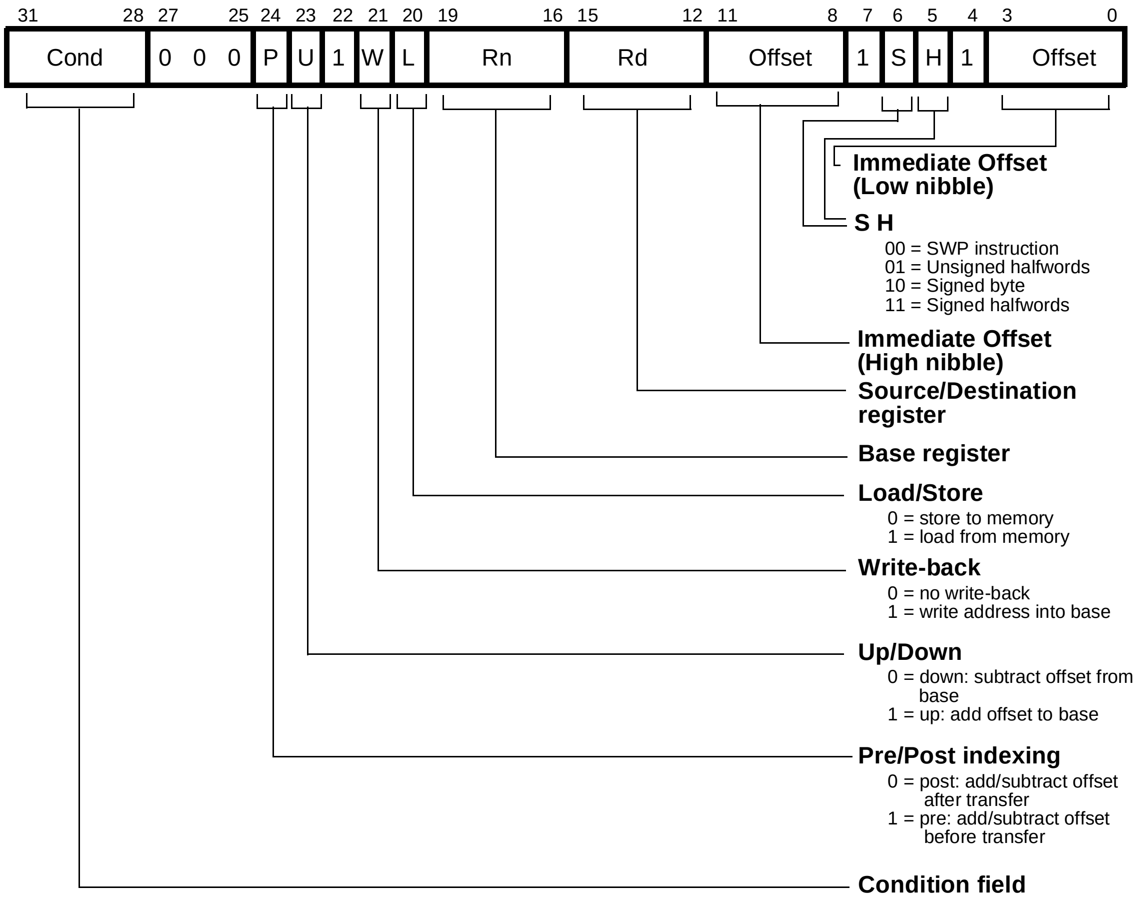


Figure 26. Halfword and Signed Data Transfer With Immediate Offset



Offsets and auto-indexing

The offset from the base may be either a 8-bit unsigned binary immediate value in the instruction, or a second register. The 8-bit offset is formed by concatenating bits 11 to 8 and bits 3 to 0 of the instruction word, such that bit 11 becomes the MSB and bit 0 becomes the LSB. The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base register is used as the transfer address.

The W bit gives optional auto-increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained if necessary by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base.

The Write-back bit should not be set high (W=1) when post-indexed addressing is selected.

Halfword load and stores

Setting S=0 and H=1 may be used to transfer unsigned Half-words between an ARM7TDMI register and memory.

The action of LDRH and STRH instructions is influenced by the BIGEND control signal. The two possible configurations are described in the section below.

Signed byte and halfword loads

The S bit controls the loading of sign-extended data. When S=1 the H bit selects between Bytes (H=0) and Half-words (H=1). The L bit should not be set low (Store) when Signed (S=1) operations have been selected.

The LDRSB instruction loads the selected Byte into bits 7 to 0 of the destination register and bits 31 to 8 of the destination register are set to the value of bit 7, the sign bit.

The LDRSH instruction loads the selected Half-word into bits 15 to 0 of the destination register and bits 31 to 16 of the destination register are set to the value of bit 15, the sign bit.

The action of the LDRSB and LDRSH instructions is influenced by the BIGEND control signal. The two possible configurations are described in the following section.

Endianness and byte/halfword selection

Little endian configuration

A signed byte load (LDRSB) expects data on data bus inputs 7 through to 0 if the supplied address is on a word boundary, on data bus inputs 15 through to 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bit of the destination register, and the remaining bits of the register are filled with the sign bit, bit 7 of the byte. Please see Figure 5

A halfword load (LDRSH or LDRH) expects data on data bus inputs 15 through to 0 if the supplied address is on a word boundary and on data bus inputs 31 through to 16 if it is a halfword boundary, (A[1]=1).The supplied address should always be on a halfword boundary. If bit 0 of the supplied address is HIGH then the ARM7TDMI will load an unpredictable value. The selected halfword is placed in the bottom 16 bits of the destination register. For unsigned half-words (LDRH), the top 16 bits of the register are filled with zeros and for signed half-words (LDRSH) the top 16 bits are filled with the sign bit, bit 15 of the halfword.

A halfword store (STRH) repeats the bottom 16 bits of the source register twice across the data bus outputs 31 through to 0. The external memory system should activate the appropriate halfword subsystem to store the data. Note that the address must be halfword aligned, if bit 0 of the address is HIGH this will cause unpredictable behaviour.

Big endian configuration

A signed byte load (LDRSB) expects data on data bus inputs 31 through to 24 if the supplied address is on a word boundary, on data bus inputs 23 through to 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bit of the destination register, and the remaining bits of the register are filled with the sign bit, bit 7 of the byte. Please see Figure 4.

A halfword load (LDRSH or LDRH) expects data on data bus inputs 31 through to 16 if the supplied address is on a word boundary and on data bus inputs 15 through to 0 if it is a halfword boundary, (A[1]=1). The supplied address should always be on a halfword boundary. If bit 0 of the supplied address is HIGH then the ARM7TDMI will load an unpredictable value. The selected halfword is placed in the bottom 16 bits of the destination register. For unsigned half-words (LDRH), the top 16 bits of the register are filled with zeros and for signed half-words (LDRSH) the top 16 bits are filled with the sign bit, bit 15 of the halfword.

A halfword store (STRH) repeats the bottom 16 bits of the source register twice across the data bus outputs 31 through to 0. The external memory system should activate the appropriate halfword subsystem to store the data. Note that the address must be halfword aligned, if bit 0 of the address is HIGH this will cause unpredictable behaviour.

Use of R15

Write-back should not be specified if R15 is specified as the base register (Rn). When using R15 as the base register you must remember it contains an address 8 bytes on from the address of the current instruction.

R15 should not be specified as the register offset (Rm).

When R15 is the source register (Rd) of a Half-word store (STRH) instruction, the stored address will be address of the instruction plus 12.

Data aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from the main memory. The memory manager can signal a problem by taking the processor ABORT input HIGH whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Instruction cycle times

Normal LDR(H,SH,SB) instructions take 1S + 1N + 1I

LDR(H,SH,SB) PC take 2S + 2N + 1I incremental cycles.

S,N and I are defined in Cycle Types.

STRH instructions take 2N incremental cycles to execute.

Assembler syntax

`<LDR|STR>{cond}<H|SH|SB> Rd,<address>`

LDR load from memory into a register

STR Store from a register into memory

{cond} two-character condition mnemonic. See Table 6.

H Transfer halfword quantity

SB Load sign extended byte (Only valid for LDR)

SH Load sign extended halfword (Only valid for LDR)

Rd is an expression evaluating to a valid register number.

<address> can be:

1. An expression which generates an address:

   `<expression>`

   The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

2. A pre-indexed addressing specification:

[Rn]	offset of zero
[Rn,<#expression>]{!}	offset of <expression> bytes
[Rn,{+/-}Rm]{!}	offset of +/- contents of index register

3. A post-indexed addressing specification:

   [Rn],<#expression> offset of <expression> bytes

   [Rn],{+/-}Rm offset of +/- contents of index register.

   Rn and Rm are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7TDMI pipelining. In this case base write-back should not be specified.

{!} writes back the base register (set the W bit) if ! is present

Examples

```
LDRH  R1,[R2,-R3]!; Load R1 from the contents of the
               ; halfword address contained in
               ; R2-R3 (both of which are registers)
               ; and write back address to R2
STRH  R3,[R4,#14]; Store the halfword in R3 at R14+14
               ; but don't write back.
LDRSB R8,[R2],#-223; Load R8 with the sign extended
               ; contents of the byte address
               ; contained in R2 and write back
               ; R2-223 to R2.
LDRNESHR11,[R0]; conditionally load R11 with the sign
               ; extended contents of the halfword
               ; address contained in R0.
HERE           ; Generate PC relative offset to
               ; address FRED.
               ; Store the halfword in R5 at address
               ; FRED.
STRH  R5, [PC, #(FRED-HERE-8)]
      .
FRED
```

Block Data Transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 27.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

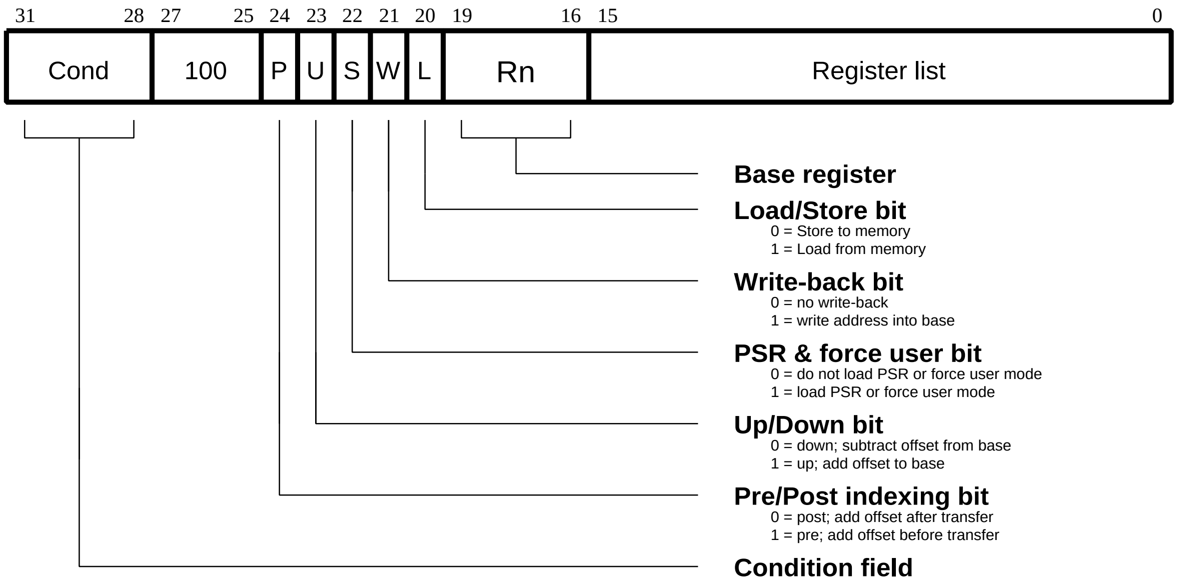
The register list

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 12.

Figure 27. Block Data Transfer Instructions



31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down bit
0 = down; subtract offset from base
1 = up; add offset to base
Pre/Post indexing bit
0 = post; add offset after transfer
1 = pre; add offset before transfer
Condition field

Addressing modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of the modified base is required (W=1). Figure 28, Figure 29, Figure 30 and Figure 31 show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 0x1000 unless it was also in the transfer list of a load multiple register instruction, when it would have been overwritten with the loaded value.

Address alignment

The address should normally be a word aligned quantity and non-word aligned addresses do not affect the instruction. However, the bottom 2 bits of the address will appear on **A[1:0]** and might be interpreted by the memory system.

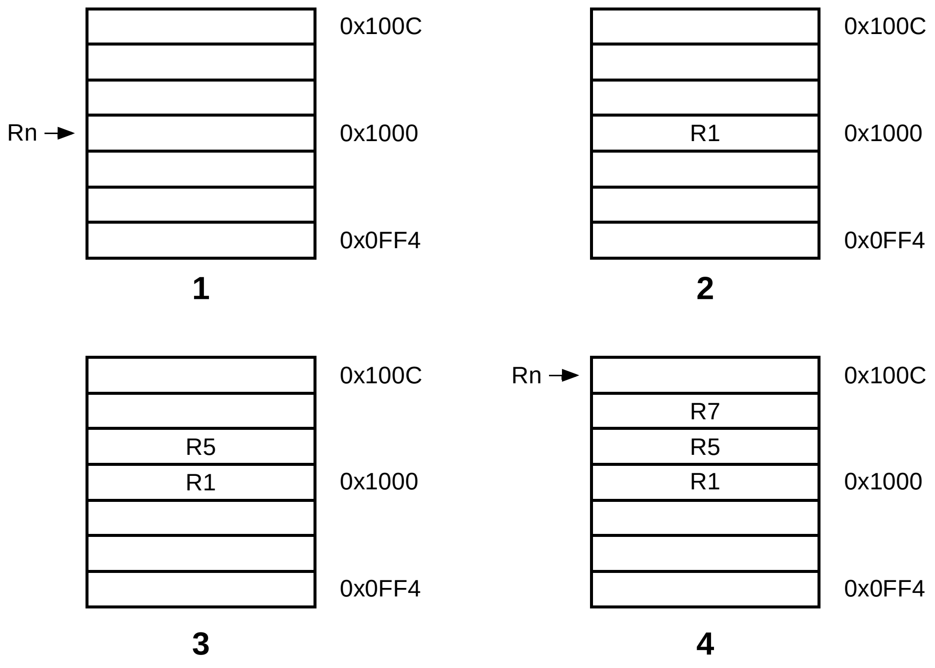


Figure 28. Post-increment Addressing

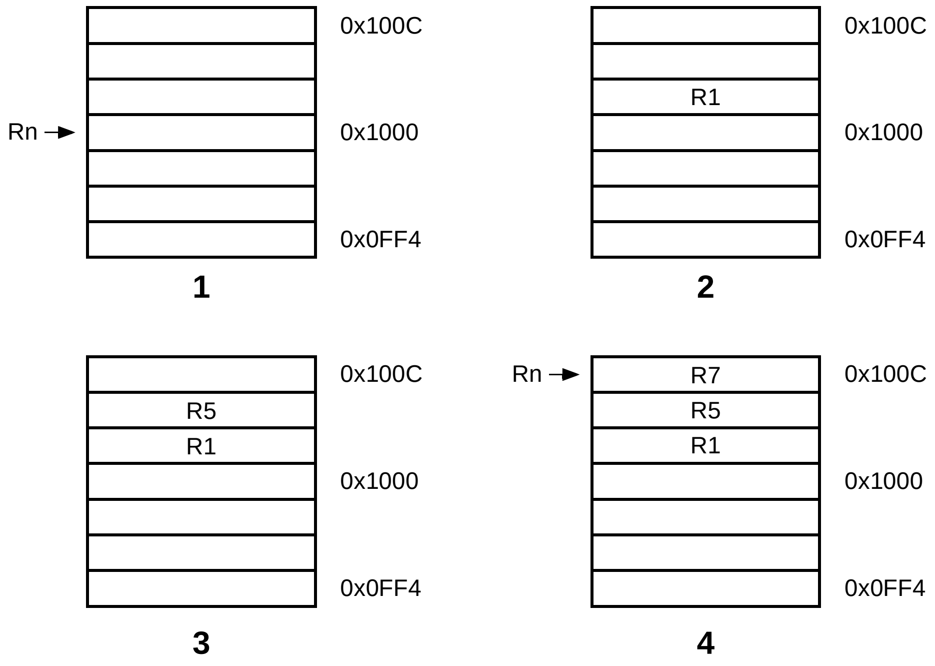


Figure 29. Pre-increment Addressing

Figure 30. Post-decrement Addressing

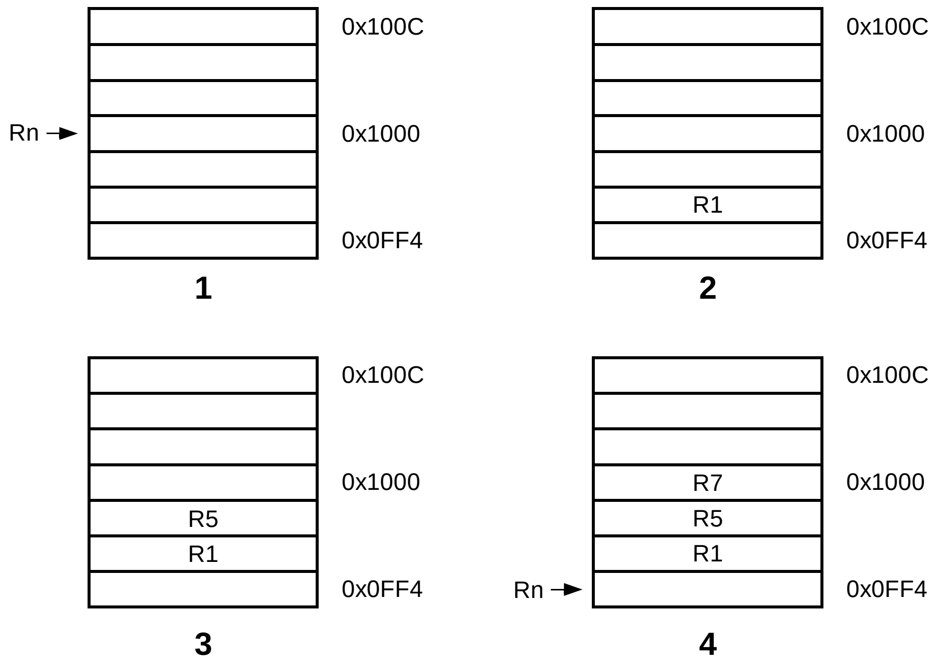
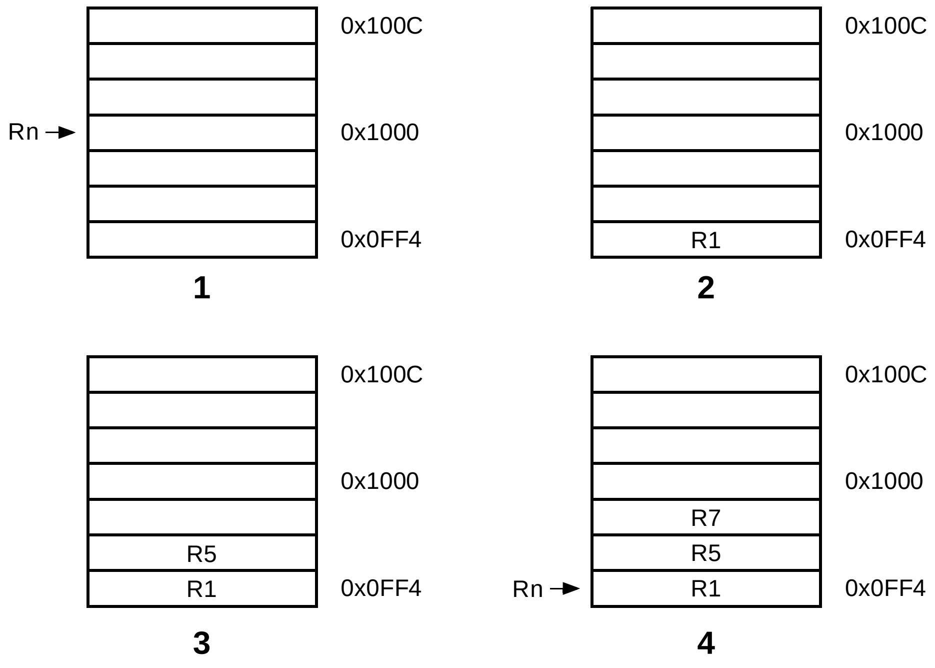


Figure 31. Pre-decrement Addressing



Use of the S bit

When the S bit is set in a LDM/STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode.

LDM with R15 in transfer list and S bit set (Mode changes)

If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

STM with R15 in transfer list and S bit set (User bank transfer)

The registers transferred are taken from the User bank rather than the bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back should not be used when this mechanism is employed.

R15 not in list and S bit set (User bank transfer)

For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back should not be used when this mechanism is employed.

When the instruction is LDM, care must be taken not to read from a banked register during the following cycle (inserting a dummy instruction such as MOV R0, R0 after the LDM will ensure safety).

Use of R15 as the base

R15 should not be used as the base register in any LDM or STM instruction.

Inclusion of the base in the register list

When write-back is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. A LDM will always overwrite the updated base if the base is in the list.

Data aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **ABORT** signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM7TDMI is to be used in a virtual memory system.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM7TDMI takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When ARM7TDMI detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

1. Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have overwritten registers. The PC is always the last register to be written and so will always be preserved.
2. The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

Instruction cycle times

Normal LDM instructions take nS + 1N + 1I and LDM PC takes (n+1)S + 2N + 1I incremental cycles, where S,N and I are as defined in Cycle Types. STM instructions take (n-1)S + 2N incremental cycleto execute, where *n* is the number of words transferred.

Assembler syntax

```
<LDM|STM>{cond}<FD|ED|FA|EA|IA|IB|DA|DB>
Rn{!},<Rlist>{^}
```

where:

{cond} two character condition mnemonic. See Table 6.

Rn is an expression evaluating to a valid register number

<Rlist> is a list of registers and register ranges enclosed in {} (e.g. {R0,R2-R7,R10}).

{!} if present requests write-back (W=1), otherwise W=0

{^} if present set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalence between the names and the values of the bits in the instruction are shown in the following table

Table 10. Addressing Mode Names

Name	Stack	Other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a “full” or “empty” stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

Examples

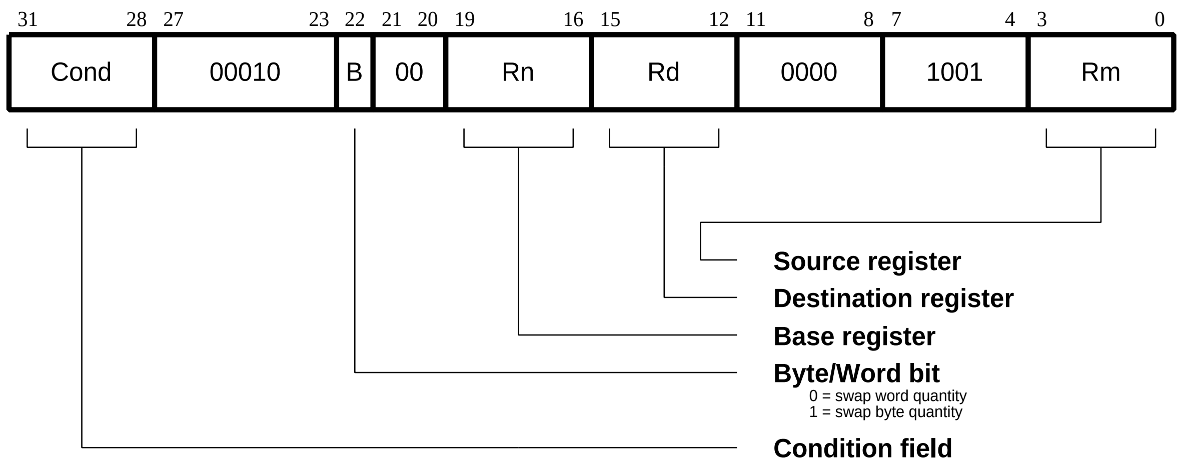
```
LDMFD SP!,{R0,R1,R2}    ; Unstack 3 registers.
STMIA R0,{R0-R15}       ; Save all registers.
LDMFD SP!,{R15}         ; R15 <- (SP),CPSR unchanged.
LDMFD SP!,{R15}^        ; R15 <- (SP), CPSR <- SPSR_mode
                        ; (allowed only in privileged modes).
STMFD R13,{R0-R14}^     ; Save user mode regs on stack
                        ; (allowed only in privileged modes).
```

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

```
STMED SP!,{R0-R3,R14}   ; Save R0 to R3 to use as workspace
                        ; and R14 for returning.
BL    somewhere         ; This nested call will overwrite R14
LDMED SP!,{R0-R3,R15}   ; restore workspace and return.
```

Single Data Swap (SWP)

Figure 32. Swap Instruction



The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 32.

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are "locked" together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

The **LOCK** output goes HIGH for the duration of the read and write operations to signal to the external memory manager that they are locked together, and should be allowed to complete without interruption. This is important in multiprocessor systems where the swap instruction is the only indivisible instruction which may be used to implement semaphores; control of the memory must not be removed from a processor while it is performing a locked operation.

Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM7TDMI register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

Use of R15

Do not use R15 as an operand (Rd, Rn or Rs) in a SWP instruction.

Data aborts

If the address used for the swap is unacceptable to a memory management system, the memory manager can flag the problem by driving ABORT HIGH. This can happen on either the read or the write cycle (or both), and in either case, the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Instruction cycle times

Swap instructions take 1S + 2N +1I incremental cycles to execute, where S,N and I are as defined in Cycle Types.

Assembler syntax

```
    <SWP>{cond}{B} Rd,Rm,[Rn]
```

{cond} two-character condition mnemonic. See Table 6.

{B} if B is present then byte transfer, otherwise word transfer

Rd,Rm,Rn are expressions evaluating to valid register numbers

Examples

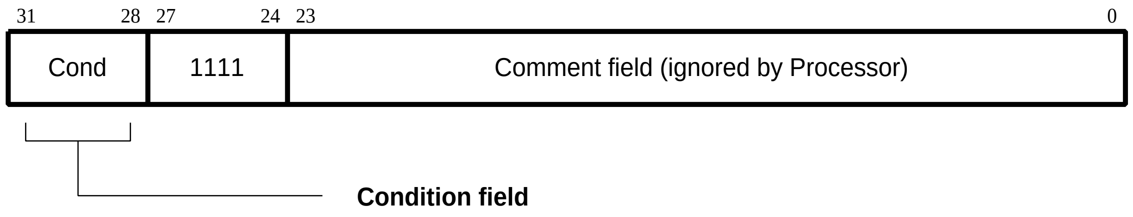
```
SWP   R0,R1,[R2]     ; Load R0 with the word addressed by R2, and
                     ; store R1 at R2.
SWPB  R2,R3,[R4]     ; Load R2 with the byte addressed by R4, and
                     ; store bits 0 to 7 of R3 at R4.
SWPEQ R0,R0,[R1]     ; Conditionally swap the contents of the
                     ; word addressed by R1 with R0.
```

Software Interrupt (SWI)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 37, below.

The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

Figure 33. Software Interrupt Instruction



Return from the supervisor

The PC is saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS PC,R14_svc will return to the calling program and restore the CPSR.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address and SPSR.

Comment field

The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

Instruction cycle times

Software interrupt instructions take 2S + 1N incremental cycles to execute, where S and N are as defined in Cycle Types

Assembler syntax

```
SWI{cond} <expression>
```

{cond} two character condition mnemonic, Table 6.

<expression> is evaluated and placed in the comment field (which is ignored by ARM7TDMI).

Examples

```
SWI   ReadC              ; Get next character from read stream.
SWI   WriteI+”k”         ; Output a “k” to the write stream.
SWINE 0                  ; Conditionally call supervisor
                         ; with 0 in comment field.
```

Supervisor code

The previous examples assume that suitable supervisor code exists, for instance:

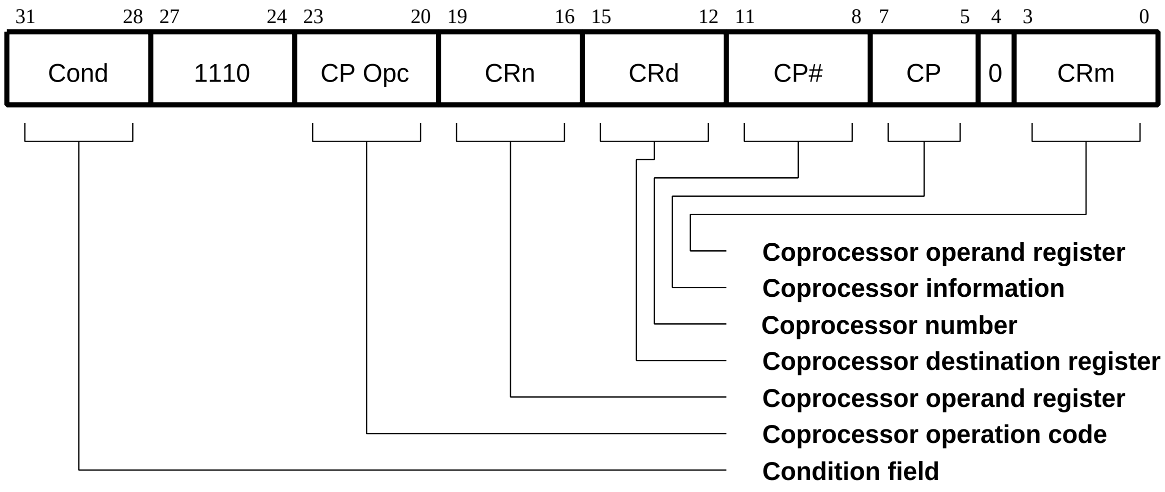
```
0x08 B Supervisor      ; SWI entry point
EntryTable             ; addresses of supervisor routines
       DCD ZeroRtn
       DCD ReadCRtn
       DCD WriteIRtn
       . . .
Zero   EQU  0
ReadC  EQU  256
WriteI EQU  512

Supervisor

; SWI has routine required in bits 8-23 and data (if any) in
; bits 0-7.
; Assumes R13_svc points to a suitable stack

STMFD R13,{R0-R2,R14}    ; Save work registers and return
                         ; address.
LDR   R0,[R14,#-4]       ; Get SWI instruction.
BIC   R0,R0,#0xFF000000  ; Clear top 8 bits.
MOV   R1,R0,LSR#8        ; Get routine offset.
ADR   R2,EntryTable      ; Get start address of entry table.
LDR   R15,[R2,R1,LSL#2]  ; Branch to appropriate routine.

      WriteIRtn          ; Enter with character in R0 bits 0-7.
         .  .  .  .  .  .
LDMFD R13,{R0-R2,R15}^   ; Restore workspace and return,
                         ; restoring processor mode and flags.
```

Coprocessor Data Operations (CDP)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 34.

This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to ARM7TDMI, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other activity, allowing the coprocessor and ARM7TDMI to perform independent tasks in parallel.

Figure 34. Coprocessor Data Operation Instruction



The coprocessor fields

Only bit 4 and bits 24 to 31 are significant to ARM7TDMI. The remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

Instruction cycle times

Coprocessor data operations take 1S + bI incremental cycles to execute, where *b* is the number of cycles spent in the coprocessor busy-wait loop.

S and I are as defined in Cycle Types.

Assembler syntax

```
CDP{cond}
p#,<expression1>,cd,cn,cm{,<expression2>}
```

{cond} two character condition mnemonic. See Table 6.

p# the unique number of the required coprocessor

<expression1> evaluated to a constant and placed in the CP Opc field

cd, cn and cm evaluate to the valid coprocessor register numbers CRd, CRn and CRm respectively

<expression2> where present is evaluated to a constant and placed in the CP field

Examples

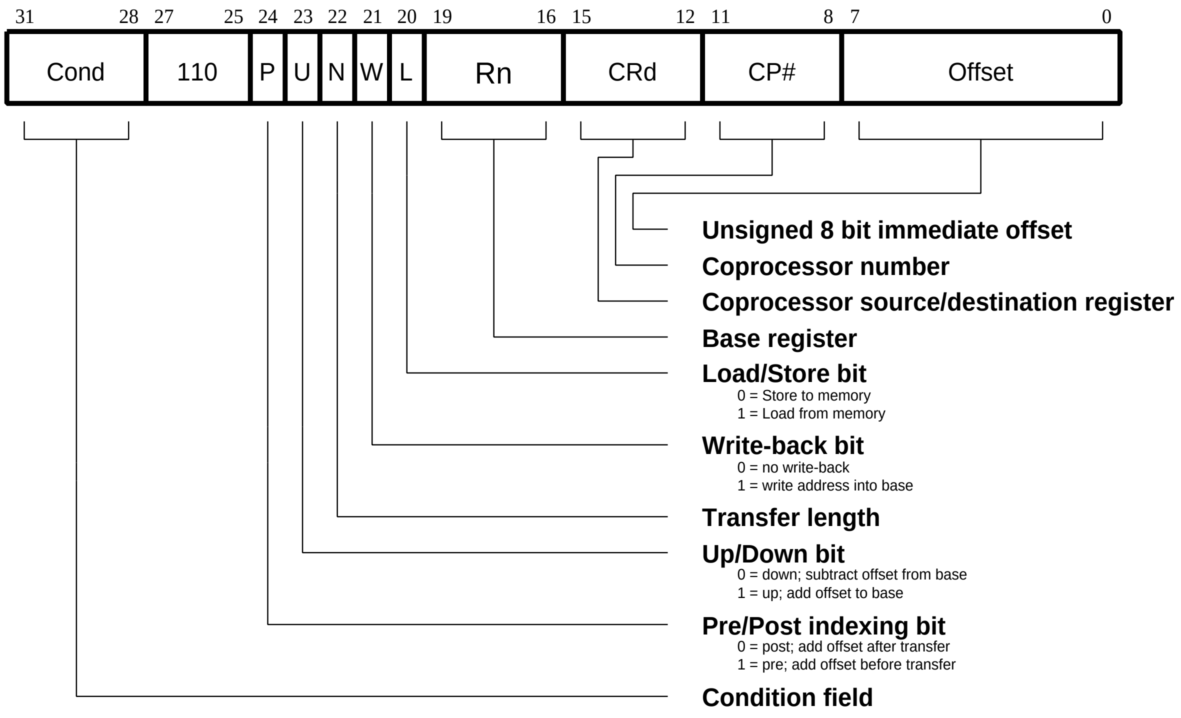
```
CDP   p1,10,c1,c2,c3    ; Request coproc 1 to do operation 10
                        ; on CR2 and CR3, and put the result
                        ; in CR1.
CDPEQ p2,5,c1,c2,c3,2   ; If Z flag is set request coproc 2
                        ; to do operation 5 (type 2) on CR2
                        ; and CR3,and put the result in CR1.
```

Coprocessor Data Transfers (LDC, STC)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 35.

This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessors's registers directly to memory. ARM7TDMI is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

Figure 35. Coprocessor Data Transfer Instructions



The coprocessor fields

The CP# field is used to identify the coprocessor which is required to supply or accept the data, and a coprocessor will only respond if its number matches the contents of this field.

The CRd field and the N bit contain information for the coprocessor which may be interpreted in different ways by different coprocessors, but by convention CRd is the register to be transferred (or the first register where more than one is to be transferred), and the N bit is used to choose one of two transfer length options. For instance N=0 could select the transfer of a single register, and N=1 could select the transfer of all the registers for context switching.

Addressing modes

ARM7TDMI is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note, however, that the immediate offsets are 8 bits wide and specify word offsets for coprocessor data transfers, whereas they are 12 bits wide and specify byte offsets for single data transfers.

The 8 bit unsigned immediate offset is shifted left 2 bits and either added to (U=1) or subtracted from (U=0) the base register (Rn); this calculation may be performed either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0). Note that post-indexed addressing modes require explicit setting of the W bit, unlike LDR and STR which always write-back when post-indexed.

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

Address alignment

The base address should normally be a word aligned quantity. The bottom 2 bits of the address will appear on **A[1:0]** and might be interpreted by the memory system.

Use of R15

If Rn is R15, the value used will be the address of the instruction plus 8 bytes. Base write-back to R15 must not be specified.

Data aborts

If the address is legal but the memory manager generates an abort, the data trap will be taken. The write-back of the modified base will take place, but all other processor state will be preserved. The coprocessor is partly responsible for ensuring that the data transfer can be restarted after the cause of the abort has been resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

Instruction cycle times

Coprocessor data transfer instructions take (n-1)S + 2N + bI incremental cycles to execute, where:

n is the number of words transferred.

b is the number of cycles spent in the coprocessor busy-wait loop.

S, N and I are as defined in Cycle Types.

Assembler syntax

```
<LDC|STC>{cond}{L} p#,cd,<Address>
```

LDC load from memory to coprocessor

STC store from coprocessor to memory

{L} when present perform long transfer (N=1), otherwise perform short transfer (N=0)

{cond} two character condition mnemonic. See Table 6.

p# the unique number of the required coprocessor

cd is an expression evaluating to a valid coprocessor register number that is placed in the CRd field

<Address> can be:

1. An expression which generates an address:

   `<expression>`

   The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.
2. A pre-indexed addressing specification:

`[Rn]`	offset of zero
`[Rn,<#expression>]{!}`	offset of <expression> bytes

3. A post-indexed addressing specification:

`[Rn],<#expression>`	offset of <expression> bytes
{!}	write back the base register (set the W bit) if! is present
Rn	is an expression evaluating to a valid ARM7TDMI register number.

Note: If Rn is R15, the assembler will subtract 8 from the offset value to allow for ARM7TDMI pipelining.

Examples

```
LDC   p1,c2,table; Load c2 of coproc 1 from address
               ; table, using a PC relative address.
STCEQLp2,c3,[R5,#24]!; Conditionally store c3 of coproc 2
               ; into an address 24 bytes up from R5,
               ; write this address back to R5, and use
               ; long transfer option (probably to
               ; store multiple words).
```

Note: Although the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

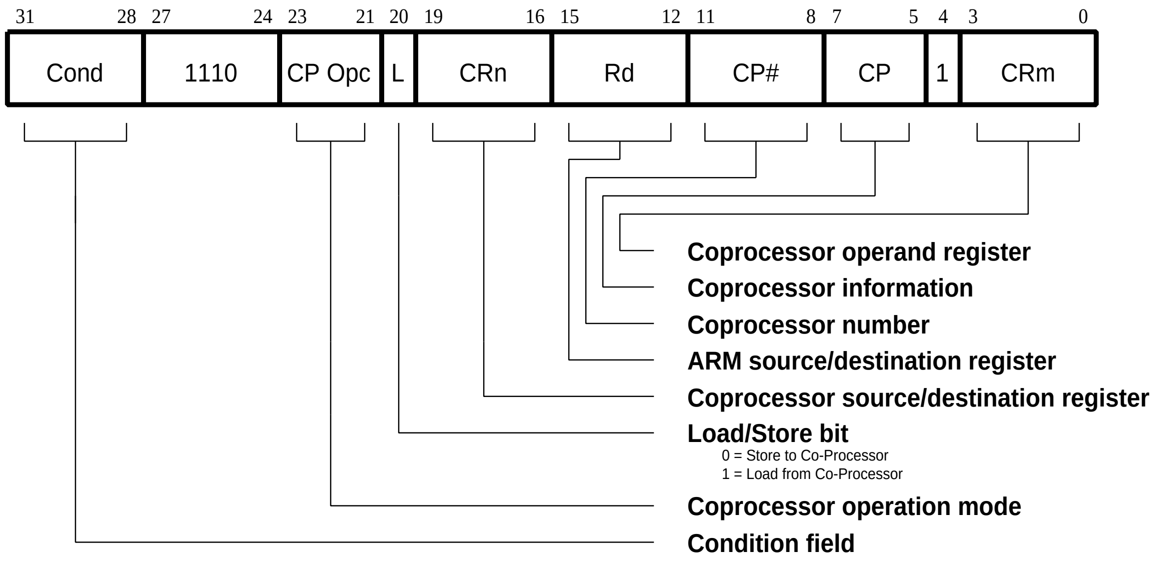
Coprocessor Register Transfers (MRC, MCR)

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction encoding is shown in Figure 36.

This class of instruction is used to communicate information directly between ARM7TDMI and a coprocessor. An example of a coprocessor to ARM7TDMI register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32 bit integer within the coprocessor, and the result is then transferred to ARM7TDMI register. A FLOAT of a 32 bit value in ARM7TDMI register into a floating point value within the coprocessor illustrates the use of ARM7TDMI register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the ARM7TDMI CPSR flags. As an example, the result of a comparison of two floating point values within a coprocessor can be moved to the CPSR to control the subsequent flow of execution.

Figure 36. Coprocessor Register Transfer Instructions


31 28 27 24 23 21 20 19 16 15 12 11 8 7 5 4 3 0
Cond
1110
CP Opc
L
CRn
Rd
CP#
CP
1
CRm
Coprocessor operand register
Coprocessor information
Coprocessor number
ARM source/destination register
Coprocessor source/destination register
Load/Store bit
0 = Store to Co-Processor
1 = Load from Co-Processor
Coprocessor operation mode
Condition field

The coprocessor fields

The CP# field is used, as for all coprocessor instructions, to specify which coprocessor is being called upon.

The CP Opc, CRn, CP and CRm fields are used only by the coprocessor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the coprocessor functionality is incompatible with this one. The conventional interpretation is that the CP Opc and CP fields specify the operation the coprocessor is required to perform, CRn is the coprocessor register which is the source or destination of the transferred information, and CRm is a second coprocessor register which may be involved in some way which depends on the particular operation specified.

Transfers to R15

When a coprocessor register transfer to ARM7TDMI has R15 as the destination, bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other CPSR bits are unaffected by the transfer.

Transfers from R15

A coprocessor register transfer from ARM7TDMI with R15 as the source register will store the PC+12.

Instruction cycle times

MRC instructions take 1S + (b+1)I +1C incremental cycles to execute, where S, I and C are as defined in Cycle Types.

MCR instructions take 1S + bI +1C incremental cycles to execute, where *b* is the number of cycles spent in the coprocessor busy-wait loop.

Assembler syntax

`<MCR|MRC>{cond}`
`p#,<expression1>,Rd,cn,cm{,<expression2>}`

MRC move from coprocessor to ARM7TDMI register (L=1)

MCR move from ARM7TDMI register to coprocessor (L=0)

{cond} two character condition mnemonic. See Table 6.

p# the unique number of the required coprocessor

<expression1> evaluated to a constant and placed in the CP Opc field

Rd is an expression evaluating to a valid ARM7TDMI register number

cn and cm are expressions evaluating to the valid coprocessor register numbers CRn and CRm respectively

<expression2> where present is evaluated to a constant and placed in the CP field

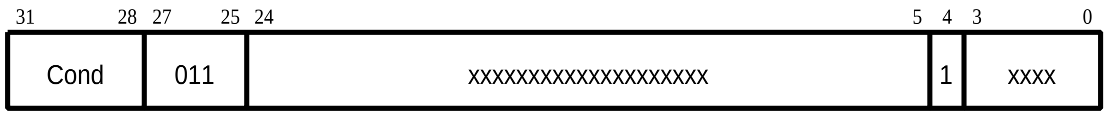
Examples

```
MRC   p2,5,R3,c5,c6  ; Request coproc 2 to perform operation 5
                     ; on c5 and c6, and transfer the (single
                     ; 32 bit word) result back to R3.

MCR   p6,0,R4,c5,c6  ; Request coproc 6 to perform operation 0
                     ; on R4 and place the result in c6.

MRCEQ p3,9,R3,c5,c6,2; Conditionally request coproc 3 to
                     ; perform operation 9 (type 2) on c5 and
                     ; c6, and transfer the result back to R3.
```

Undefined Instruction

The instruction is only executed if the condition is true. The various conditions are defined in Table 6. The instruction format is shown in Figure 37.

Figure 37. Undefined Instruction



If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering this instruction to any coprocessors which may be present, and all coprocessors must refuse to accept it by driving **CPA** and **CPB** HIGH.

Instruction cycle times

This instruction takes 2S + 1I + 1N cycles, where S, N and I are as defined in Cycle Types.

Assembler syntax

The assembler has no mnemonics for generating this instruction. If it is adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, this instruction must not be used.

Instruction Set Examples

The following examples show ways in which the basic ARM7TDMI instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some), mostly they just save code.

Using the conditional instructions

Using conditionals for logical OR

```
CMP   Rn,#p          ; If Rn=p OR Rm=q THEN GOTO Label.
BEQ   Label
CMP   Rm,#q
BEQ   Label
```

This can be replaced by

```
CMP   Rn,#p
CMPNE Rm,#q          ; If condition not satisfied try
                     ; other test.
BEQ   Label
```

Absolute value

```
TEQ   Rn,#0          ; Test sign
RSBMI Rn,Rn,#0       ; and 2's complement if necessary.
```

Multiplication by 4, 5 or 6 (run time)

```
MOV   Rc,Ra,LSL#2    ; Multiply by 4,
CMP   Rb,#5          ; test value,
ADDCS Rc,Rc,Ra       ; complete multiply by 5,
ADDHI Rc,Rc,Ra       ; complete multiply by 6.
```

Combining discrete and range tests

```
TEQ   Rc,#127        ; Discrete test,
CMPNE Rc,#” ”-1      ; range test
MOVLS Rc,#”.”        ; IF   Rc<=” ” OR Rc=ASCII(127)
                     ; THEN Rc:=”.”
```

Division and remainder

A number of divide routines for specific applications are provided in source form as part of the ANSI C library provided with the ARM Cross Development Toolkit, available from your supplier. A short general purpose divide routine follows.

```
                              ; Enter with numbers in Ra and Rb.
                              ;
      MOV      Rcnt,#1        ; Bit to control the division.
Div1  CMP      Rb,#0x80000000 ; Move Rb until greater than Ra.
      CMPCC    Rb,Ra
      MOVCC    Rb,Rb,ASL#1
      MOVCC    Rcnt,Rcnt,ASL#1
      BCC      Div1
      MOV      Rc,#0
Div2  CMP      Ra,Rb          ; Test for possible subtraction.
      SUBCS    Ra,Ra,Rb       ; Subtract if ok,
      ADDCS    Rc,Rc,Rcnt     ; put relevant bit into result
      MOVS     Rcnt,Rcnt,LSR#1; shift control bit
      MOVNE    Rb,Rb,LSR#1    ; halve unless finished.
      BNE      Div2
                              ;
                              ; Divide result in Rc,
                                    ; remainder in Ra.
```

Overflow detection in the ARM7TDMI

1. Overflow in unsigned multiply with a 32 bit result

```
         UMULL    Rd,Rt,Rm,Rn    ;3 to 6 cycles
         TEQ      Rt,#0          ;+1 cycle and a register
         BNE      overflow
```

2. Overflow in signed multiply with a 32 bit result

```
         SMULL    Rd,Rt,Rm,Rn    ;3 to 6 cycles
         TEQ      Rt,Rd ASR#31   ;+1 cycle and a register
         BNE      overflow
```

3. Overflow in unsigned multiply accumulate with a 32 bit result

```
         UMLAL    Rd,Rt,Rm,Rn    ;4 to 7 cycles
         TEQ      Rt,#0          ;+1 cycle and a register
         BNE      overflow
```

4. Overflow in signed multiply accumulate with a 32 bit result

```
         SMLAL    Rd,Rt,Rm,Rn    ;4 to 7 cycles
         TEQ      Rt,Rd, ASR#31  ;+1 cycle and a register
         BNE      overflow
```

5. Overflow in unsigned multiply accumulate with a 64 bit result

```
         UMULL    Rl,Rh,Rm,Rn    ;3 to 6 cycles
         ADDS     Rl,Rl,Ra1      ;lower accumulate
         ADC      Rh,Rh,Ra2      ;upper accumulate
         BCS      overflow       ;1 cycle and 2 registers
```

6. Overflow in signed multiply accumulate with a 64 bit result

```
         SMULL    Rl,Rh,Rm,Rn    ;3 to 6 cycles
         ADDS     Rl,Rl,Ra1      ;lower accumulate
         ADC      Rh,Rh,Ra2      ;upper accumulate
         BVS      overflow       ;1 cycle and 2 registers
```

Note: Overflow checking is not applicable to unsigned and signed multiplies with a 64-bit result, since overflow does not occur in such calculations.

Pseudo-random binary sequence generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive-OR feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32 bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition), so this example uses a 33 bit register with taps at bits 33 and 20. The basic algorithm is newbit:=bit 33 eor bit 20, shift left the 33 bit number and put in newbit at the bottom; this operation is performed for all the newbits needed (i.e. 32 bits). The entire operation can be done in 5 S cycles:

```
                        ; Enter with seed in Ra (32 bits),
                          Rb (1 bit in Rb lsb), uses Rc.
                        ;
TST   Rb,Rb,LSR#1       ; Top bit into carry
MOVS  Rc,Ra,RRX         ; 33 bit rotate right
ADC   Rb,Rb,Rb          ; carry into lsb of Rb
EOR   Rc,Rc,Ra,LSL#12   ; (involved!)
EOR   Ra,Rc,Rc,LSR#20   ; (similarly involved!)
                        ; new seed in Ra, Rb as before
```

Multiplication by constant using the barrel shifter

Multiplication by 2^n (1,2,4,8,16,32..)

```
MOVRa, Rb, LSL #n
```

Multiplication by 2^n+1 (3,5,9,17..)

```
ADDRa,Ra,Ra,LSL #n
```

Multiplication by 2^n-1 (3,7,15..)

```
RSBRa,Ra,Ra,LSL #n
```

Multiplication by 6

```
ADDRa,Ra,Ra,LSL #1; multiply by 3

MOVRa,Ra,LSL#1; and then by 2
```

Multiply by 10 and add in extra number

```
ADDRa,Ra,Ra,LSL#2; multiply by 5

ADDRa,Rc,Ra,LSL#1; multiply by 2 and add in next digit
```

General recursive method for Rb := Ra*C, C a constant:

1. If C even, say C = 2^n*D, D odd:

```
D=1:      MOV   Rb,Ra,LSL #n
D<>1:     {Rb := Ra*D}
          MOV      Rb,Rb,LSL #n
```

2. If C MOD 4 = 1, say C = 2^n*D+1, D odd, n>1:

```
D=1:      ADD   Rb,Ra,Ra,LSL #n
D<>1:     {Rb := Ra*D}
          ADD      Rb,Ra,Rb,LSL #n
```

3. If C MOD 4 = 3, say C = 2^n*D-1, D odd, n>1:

```
D=1:      RSB   Rb,Ra,Ra,LSL #n
D<>1:     {Rb := Ra*D}
          RSB      Rb,Ra,Rb,LSL #n
```

This is not quite optimal, but close. An example of its non-optimality is multiply by 45 which is done by:

```
RSB       Rb,Ra,Ra,LSL#2 ; multiply by 3
RSB       Rb,Ra,Rb,LSL#2 ; multiply by 4*3-1 = 11
ADD       Rb,Ra,Rb,LSL# 2; multiply by 4*11+1 = 45
```

rather than by:

```
ADD       Rb,Ra,Ra,LSL#3 ; multiply by 9
ADD       Rb,Rb,Rb,LSL#2 ; multiply by 5*9 = 45
```

Loading a word from an unknown alignment

```
                     ; enter with address in Ra (32 bits)
                     ; uses Rb, Rc; result in Rd.
                     ; Note d must be less than c e.g. 0,1
                     ;
BIC   Rb,Ra,#3       ; get word aligned address
LDMIA Rb,{Rd,Rc}     ; get 64 bits containing answer
AND   Rb,Ra,#3       ; correction factor in bytes
MOVS  Rb,Rb,LSL#3    ; ...now in bits and test if aligned
MOVNE Rd,Rd,LSR Rb   ; produce bottom of result word
                     ; (if not aligned)
RSBNE Rb,Rb,#32      ; get other shift amount
ORRNE Rd,Rd,Rc,LSL Rb; combine two halves to get result
```

Thumb Instruction Set

This chapter describes the THUMB instruction set.

Format Summary

The THUMB instruction set formats are shown in the following figure.

Figure 38. THUMB Instruction Set Formats

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			Move shifted register
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			Add/subtract
3	0	0	1	Op		Rd			Offset8								Move/compare/add /subtract immediate
4	0	1	0	0	0	0	Op				Rs			Rd			ALU operations
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			Hi register operations /branch exchange
6	0	1	0	0	1	Rd			Word8								PC-relative load
7	0	1	0	1	L	B	0	Ro			Rb			Rd			Load/store with register offset
8	0	1	0	1	H	S	1	Ro			Rb			Rd			Load/store sign-extended byte/halfword
9	0	1	1	B	L	Offset5					Rb			Rd			Load/store with immediate offset
10	1	0	0	0	L	Offset5					Rb			Rd			Load/store halfword
11	1	0	0	1	L	Rd			Word8								SP-relative load/store
12	1	0	1	0	SP	Rd			Word8								Load address
13	1	0	1	1	0	0	0	0	S	SWord7							Add offset to stack pointer
14	1	0	1	1	L	1	0	R	Rlist								Push/pop registers
15	1	1	0	0	L	Rb			Rlist								Multiple load/store
16	1	1	0	1	Cond				Soffset8								Conditional branch
17	1	1	0	1	1	1	1	1	Value8								Software Interrupt
18	1	1	1	0	0	Offset11											Unconditional branch
19	1	1	1	1	H	Offset											Long branch with link
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Opcode Summary

The following table summarizes the THUMB instruction set. For further information about a particular instruction please refer to the sections listed in the right-most column.

Table 11. THUMB Instruction Set Opcodes

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set	See Page:
ADC	Add with Carry	✔		✔	84
ADD	Add	✔	✔	✔(1)	81, 83, 86, 100, 101
AND	AND	✔		✔	84
ASR	Arithmetic Shift Right	✔		✔	80, 84
B	Unconditional branch	✔			108
B*xx*	Conditional branch	✔			105
BIC	Bit Clear	✔		✔	84
BL	Branch and Link				109
BX	Branch and Exchange	✔	✔		86
CMN	Compare Negative	✔		✔	84
CMP	Compare	✔	✔	✔	83, 84, 86
EOR	EOR	✔		✔	84
LDMIA	Load multiple	✔			104
LDR	Load word	✔			89, 90, 94, 98
LDRB	Load byte	✔			90, 94
LDRH	Load halfword	✔			92, 96
LSL	Logical Shift Left	✔		✔	80, 84
LDSB	Load sign-extended byte	✔			92
LDSH	Load sign-extended half-word	✔			92
LSR	Logical Shift Right	✔		✔	80, 84
MOV	Move register	✔	✔	✔(2)	83, 86
MUL	Multiply	✔		✔	84
MVN	Move Negative register	✔		✔	84
NEG	Negate	✔		✔	84
ORR	OR	✔		✔	84
POP	Pop registers	✔			102
PUSH	Push registers	✔			102
ROR	Rotate Right	✔		✔	84
SBC	Subtract with Carry	✔		✔	84
STMIA	Store Multiple	✔			104
STR	Store word	✔			90, 94, 98
STRB	Store byte	✔			90
STRH	Store halfword	✔			92, 96
SWI	Software Interrupt				107
SUB	Subtract	✔		✔	81, 83
TST	Test bits	✔		✔	84

Notes: 1. The condition codes are unaffected by the format 5, 12 and 13 versions of this instruction.
2. The condition codes are unaffected by the format 5 version of this instruction.

Format 1: move shifted register

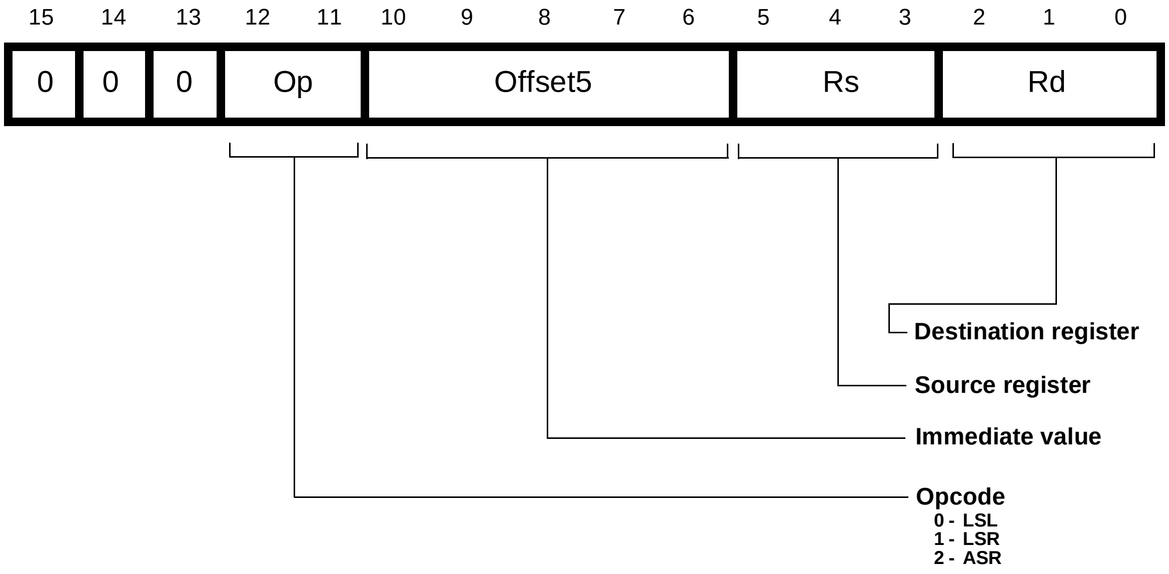


Figure 39. Format 1

Operation

These instructions move a shifted value between Lo registers. The THUMB assembler syntax is shown in Table 12.

Note: All instructions in this group set the CPSR condition codes

Table 12. Summary of Format 1 Instructions

OP	THUMB assembler	ARM equivalent	Action
00	LSL Rd, Rs, #Offset5	MOVS Rd, Rs, LSL #Offset5	Shift Rs left by a 5-bit immediate value and store the result in Rd.
01	LSR Rd, Rs, #Offset5	MOVS Rd, Rs, LSR #Offset5	Perform logical shift right on Rs by a 5-bit immediate value and store the result in Rd.
10	ASR Rd, Rs, #Offset5	MOVS Rd, Rs, ASR #Offset5	Perform arithmetic shift right on Rs by a 5-bit immediate value and store the result in Rd.

Instruction cycle times

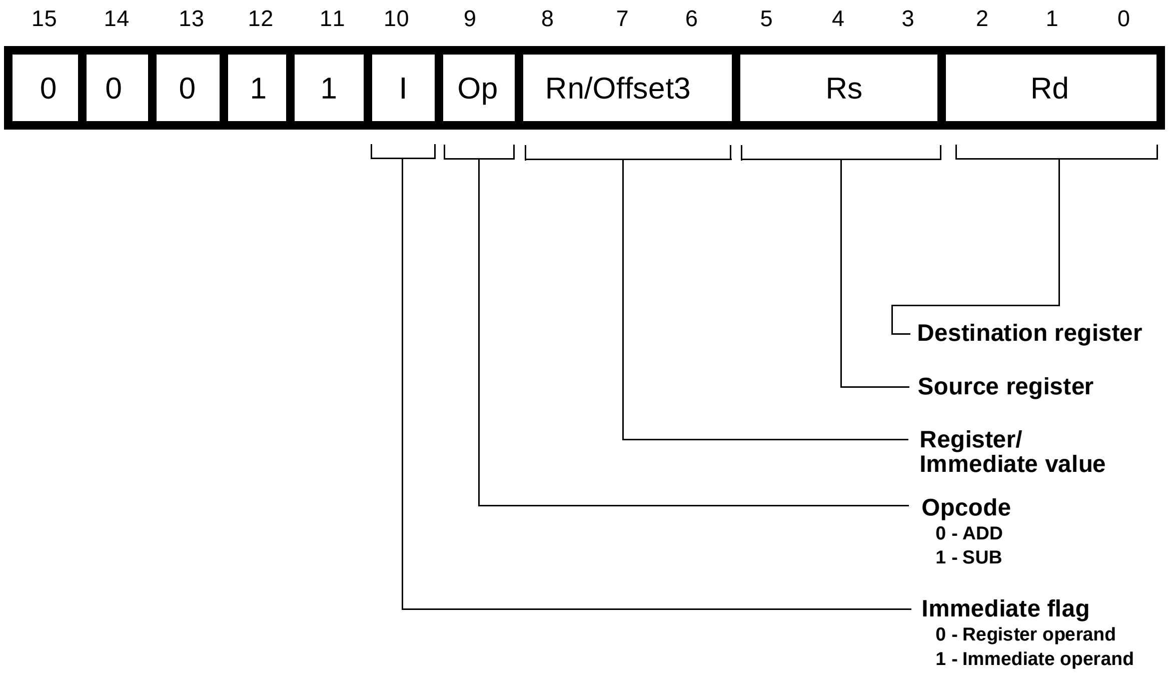
All instructions in this format have an equivalent ARM instruction as shown in Table 12. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
LSR   R2, R5, #27    ; Logical shift right the contents
                     ; of R5 by 27 and store the result in R2.
                     ; Set condition codes on the result.
```

Format 2: add/subtract

Figure 40. Format 2



Operation

These instructions allow the contents of a Lo register or a 3-bit immediate value to be added to or subtracted from a Lo register. The THUMB assembler syntax is shown in Table 13.

Note: All instructions in this group set the CPSR condition codes

Table 13. Summary of Format 2 Instructions

Op	I	THUMB assembler	ARM equivalent	Action
0	0	ADD Rd, Rs, Rn	ADDS Rd, Rs, Rn	Add contents of Rn to contents of Rs. Place result in Rd.
0	1	ADD Rd, Rs, #Offset3	ADDS Rd, Rs, #Offset3	Add 3-bit immediate value to contents of Rs. Place result in Rd.
1	0	SUB Rd, Rs, Rn	SUBS Rd, Rs, Rn	Subtract contents of Rn from contents of Rs. Place result in Rd.
1	1	SUB Rd, Rs, #Offset3	SUBS Rd, Rs, #Offset3	Subtract 3-bit immediate value from contents of Rs. Place result in Rd.

Instruction cycle times

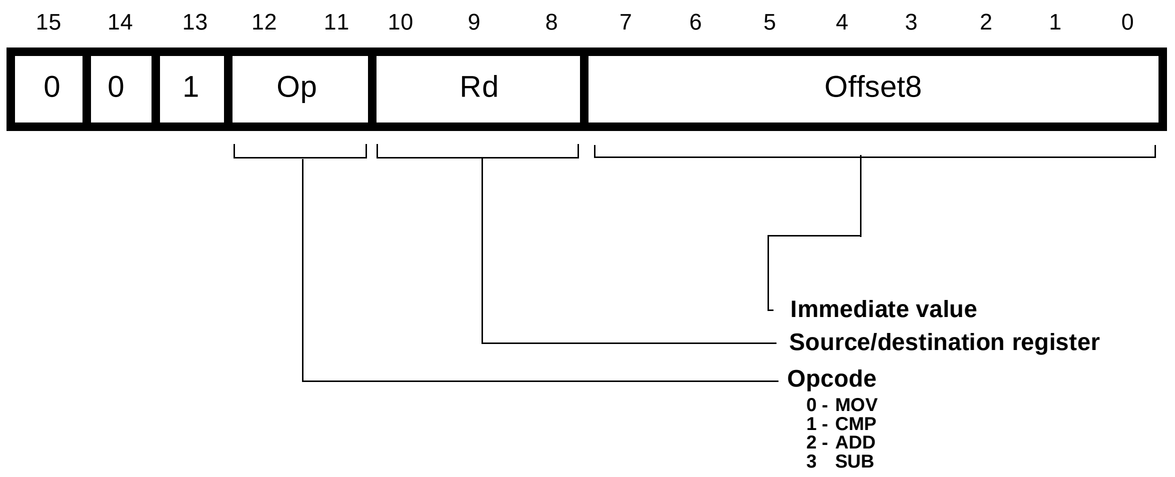
All instructions in this format have an equivalent ARM instruction as shown in Table 13. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175

Examples

```
ADD   R0, R3, R4     ; R0 := R3 + R4 and set condition codes on
                     ; the result.

SUB   R6, R2, #6     ; R6 := R2 - 6 and set condition codes.
```

Format 3: move/compare/add/subtract immediate

Figure 41. Format 3



Operations

The instructions in this group perform operations between a Lo register and an 8-bit immediate value.

The THUMB assembler syntax is shown in Table 14.

Note: All instructions in this group set the CPSR condition codes.

Table 14. Summary of Format 3 Instructions

Op	THUMB assembler	ARM equivalent	Action
00	MOV Rd, #Offset8	MOVS Rd, #Offset8	Move 8-bit immediate value into Rd.
01	CMP Rd, #Offset8	CMP Rd, #Offset8	Compare contents of Rd with 8-bit immediate value.
10	ADD Rd, #Offset8	ADDS Rd, Rd, #Offset8	Add 8-bit immediate value to contents of Rd and place the result in Rd.
11	SUB Rd, #Offset8	SUBS Rd, Rd, #Offset8	Subtract 8-bit immediate value from contents of Rd and place the result in Rd.

Instruction cycle times

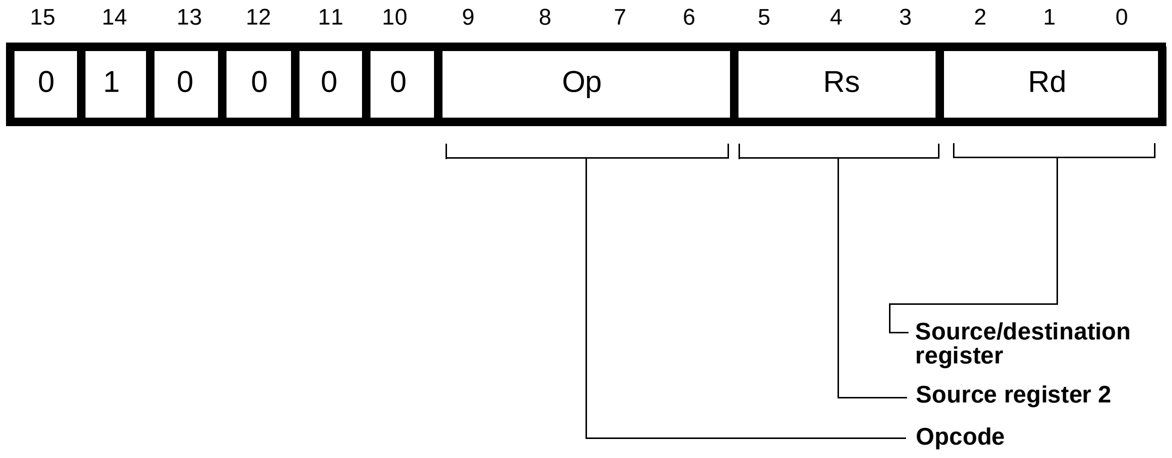
All instructions in this format have an equivalent ARM instruction as shown in Table 14. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
MOV   R0, #128       ; R0 := 128 and set condition codes

CMP   R2, #62        ; Set condition codes on R2 - 62

ADD   R1, #255       ; R1 := R1 + 255 and set condition
                     ; codes

SUB   R6, #145       ; R6 := R6 - 145 and set condition
                     ; codes
```

Format 4: ALU operations

Figure 42. Format 4



Operation

The following instructions perform ALU operations on a Lo register pair.

Note: All instructions in this group set the CPSR condition codes.

Table 15. Summary of Format 4 Instructions

OP	THUMB assembler	ARM equivalent	Action
0000	AND Rd, Rs	ANDS Rd, Rd, Rs	Rd:= Rd AND Rs
0001	EOR Rd, Rs	EORS Rd, Rd, Rs	Rd:= Rd EOR Rs
0010	LSL Rd, Rs	MOVS Rd, Rd, LSL Rs	Rd := Rd << Rs
0011	LSR Rd, Rs	MOVS Rd, Rd, LSR Rs	Rd := Rd >> Rs
0100	ASR Rd, Rs	MOVS Rd, Rd, ASR Rs	Rd := Rd ASR Rs
0101	ADC Rd, Rs	ADCS Rd, Rd, Rs	Rd := Rd + Rs + C-bit
0110	SBC Rd, Rs	SBCS Rd, Rd, Rs	Rd := Rd - Rs - NOT C-bit
0111	ROR Rd, Rs	MOVS Rd, Rd, ROR Rs	Rd := Rd ROR Rs
1000	TST Rd, Rs	TST Rd, Rs	Set condition codes on Rd AND Rs
1001	NEG Rd, Rs	RSBS Rd, Rs, #0	Rd = -Rs
1010	CMP Rd, Rs	CMP Rd, Rs	Set condition codes on Rd - Rs
1011	CMN Rd, Rs	CMN Rd, Rs	Set condition codes on Rd + Rs
1100	ORR Rd, Rs	ORRS Rd, Rd, Rs	Rd := Rd OR Rs
1101	MUL Rd, Rs	MULS Rd, Rs, Rd	Rd := Rs * Rd
1110	BIC Rd, Rs	BICS Rd, Rd, Rs	Rd := Rd AND NOT Rs
1111	MVN Rd, Rs	MVNS Rd, Rs	Rd := NOT Rs

Instruction cycle times

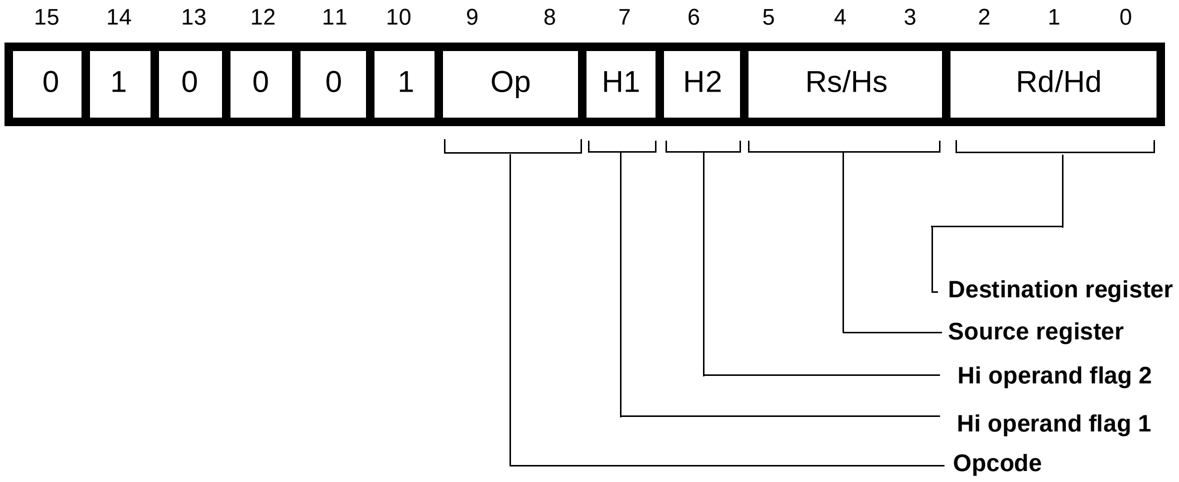
All instructions in this format have an equivalent ARM instruction as shown in Table 15. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
EOR   R3, R4   ; R3 := R3 EOR R4 and set condition codes

ROR   R1, R0   ; Rotate Right R1 by the value in R0, store
               ; the result in R1 and set condition codes

NEG   R5, R3   ; Subtract the contents of R3 from zero,
               ; store the result in R5. Set condition codes
               ; ie R5 = -R3

CMP   R2, R6   ; Set the condition codes on the result of
               ; R2 - R6

MUL   R0, R7   ; R0 := R7 * R0 and set condition codes
```

Format 5: Hi register operations/branch exchange

Figure 43. Format 5



Operation

There are four sets of instructions in this group. The first three allow ADD, CMP and MOV operations to be performed between Lo and Hi registers, or a pair of Hi registers. The fourth, BX, allows a Branch to be performed which may also be used to switch processor state.

The THUMB assembler syntax is shown in Table 16.

Note: In this group only CMP (Op = 01) sets the CPSR condition codes.

The action of H1= 0, H2 = 0 for Op = 00 (ADD), Op =01 (CMP) and Op = 10 (MOV) is undefined, and should not be used

Table 16. Summary of Format 5 Instructions

Op	H1	H2	THUMB assembler	ARM equivalent	Action
00	0	1	ADD Rd, Hs	ADD Rd, Rd, Hs	Add a register in the range 8-15 to a register in the range 0-7.
00	1	0	ADD Hd, Rs	ADD Hd, Hd, Rs	Add a register in the range 0-7 to a register in the range 8-15.
00	1	1	ADD Hd, Hs	ADD Hd, Hd, Hs	Add two registers in the range 8-15
01	0	1	CMP Rd, Hs	CMP Rd, Hs	Compare a register in the range 0-7 with a register in the range 8-15. Set the condition code flags on the result.
01	1	0	CMP Hd, Rs	CMP Hd, Rs	Compare a register in the range 8-15 with a register in the range 0-7. Set the condition code flags on the result.
01	1	1	CMP Hd, Hs	CMP Hd, Hs	Compare two registers in the range 8-15. Set the condition code flags on the result.
10	0	1	MOV Rd, Hs	MOV Rd, Hs	Move a value from a register in the range 8-15 to a register in the range 0-7.
10	1	0	MOV Hd, Rs	MOV Hd, Rs	Move a value from a register in the range 0-7 to a register in the range 8-15.
10	1	1	MOV Hd, Hs	MOV Hd, Hs	Move a value between two registers in the range 8-15.
11	0	0	BX Rs	BX Rs	Perform branch (plus optional state change) to address in a register in the range 0-7.
11	0	1	BX Hs	BX Hs	Perform branch (plus optional state change) to address in a register in the range 8-15.

Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in Table 16. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

The BX instruction

BX performs a Branch to a routine whose start address is specified in a Lo or Hi register.

Bit 0 of the address determines the processor state on entry to the routine:

Bit 0 = 0 causes the processor to enter ARM state.

Bit 0 = 1 causes the processor to enter THUMB state.

Note: The action of H1 = 1 for this instruction is undefined, and should not be used.

Examples

Hi register operations

```
      ADD      PC, R5   ; PC := PC + R5 but don't set the
                        ; condition codes.

      CMP      R4, R12  ; Set the condition codes on the
                        ; result of R4 - R12.

      MOV      R15, R14 ; Move R14 (LR) into R15 (PC)
                        ; but don't set the condition codes,
                        ; eg. return from subroutine.
```

Branch and exchange

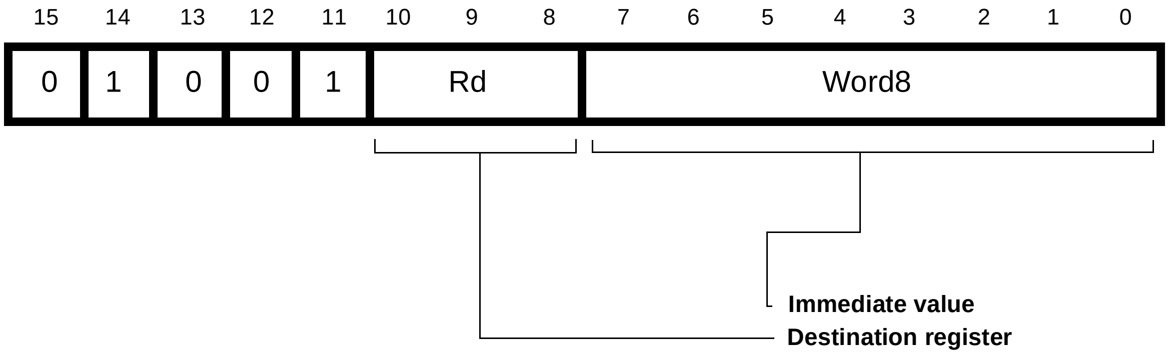
```
                        ; Switch from THUMB to ARM state.

      ADR      R1,outofTHUMB
                        ; Load address of outofTHUMB
                        ; into R1.
      MOV      R11,R1
      BX       R11      ; Transfer the contents of R11 into
                        ; the PC.
                        ; Bit 0 of R11 determines whether
                        ; ARM or THUMB state is entered, ie.
                        ; ARM state here.
        ...
      ALIGN
      CODE32
outofTHUMB
                        ; Now processing ARM instructions...
```

Using R15 as an operand

If R15 is used as an operand, the value will be the address of the instruction + 4 with bit 0 cleared. Executing a BX PC in THUMB state from a non-word aligned address will result in unpredictable execution.

Format 6: PC-relative load

Figure 44. Format 6



Operation

This instruction loads a word from an address specified as a 10-bit immediate offset from the PC.

The THUMB assembler syntax is shown below.

Table 17. Summary of PC-Relative Load Instruction

THUMB assembler	ARM equivalent	Action
LDR Rd, [PC, #Imm]	LDR Rd, [R15, #Imm]	Add unsigned offset (255 words, 1020 bytes) in Imm to the current value of the PC. Load the word from the resulting address into Rd.

Note: The value specified by #Imm is a full 10-bit address, but must always be word-aligned (ie with bits 1:0 set to 0), since the assembler places #Imm >> 2 in field Word8.

Note: The value of the PC will be 4 bytes greater than the address of this instruction, but bit 1 of the PC is forced to 0 to ensure it is word aligned.

Instruction cycle times

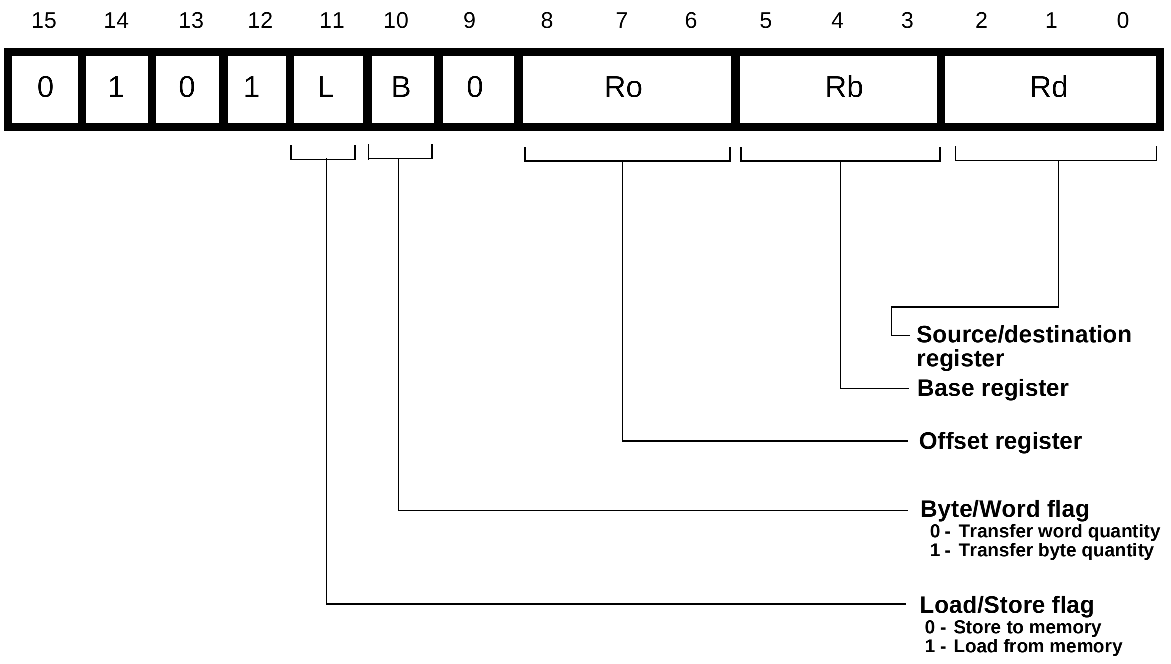
All instructions in this format have an equivalent ARM instruction as shown in Table 17. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
LDR R3,[PC,#844]     ; Load into R3 the word found at the
                     ; address formed by adding 844 to PC.
                     ; bit[1] of PC is forced to zero.
                     ; Note that the THUMB opcode will contain
                     ; 211 as the Word8 value.
```

Format 7: load/store with register offset

Figure 45. Format 7



Operation

These instructions transfer byte or word values between registers and memory. Memory addresses are pre-indexed using an offset register in the range 0-7.

The THUMB assembler syntax is shown in Table 18

Table 18. Summary of Format 7 Instructions

L	B	THUMB assembler	ARM equivalent	Action
0	0	STR Rd, [Rb, Ro]	STR Rd, [Rb, Ro]	Pre-indexed word store: Calculate the target address by adding together the value in Rb and the value in Ro. Store the contents of Rd at the address.

Table 18. Summary of Format 7 Instructions (Continued)

L	B	THUMB assembler	ARM equivalent	Action
0	1	STRB Rd, [Rb, Ro]	STRB Rd, [Rb, Ro]	Pre-indexed byte store: Calculate the target address by adding together the value in Rb and the value in Ro. Store the byte value in Rd at the resulting address.
1	0	LDR Rd, [Rb, Ro]	LDR Rd, [Rb, Ro]	Pre-indexed word load: Calculate the source address by adding together the value in Rb and the value in Ro. Load the contents of the address into Rd.
1	1	LDRB Rd, [Rb, Ro]	LDRB Rd, [Rb, Ro]	Pre-indexed byte load: Calculate the source address by adding together the value in Rb and the value in Ro. Load the byte value at the resulting address.

Instruction cycle times

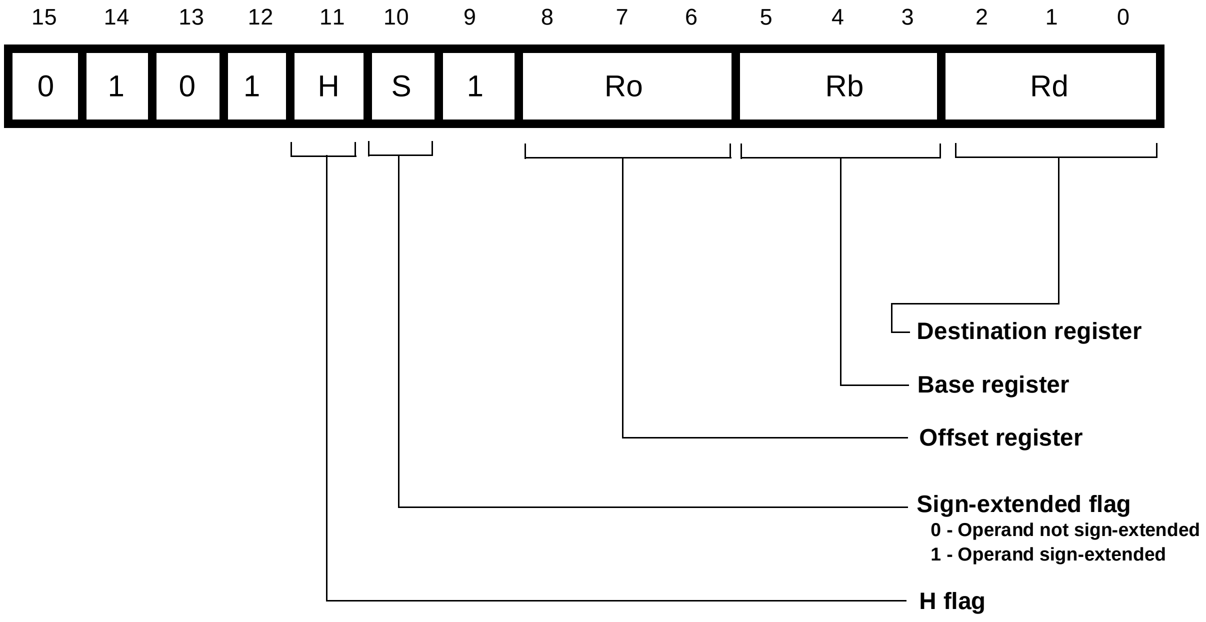
All instructions in this format have an equivalent ARM instruction as shown in Table 18. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
STR   R3, [R2,R6]    ; Store word in R3 at the address
                     ; formed by adding R6 to R2.

LDRB  R2, [R0,R7]    ; Load into R2 the byte found at
                     ; the address formed by adding
                     ; R7 to R0.
```

Format 8: load/store sign-extended byte/halfword

Figure 46. Format 8



Operation

These instructions load optionally sign-extended bytes or halfwords, and store halfwords. The THUMB assembler syntax is shown below.

Table 19. Summary of Format 8 Instructions

S	H	THUMB assembler	ARM equivalent	Action
0	0	STRH Rd, [Rb, Ro]	STRH Rd, [Rb, Ro]	Store halfword: Add Ro to base address in Rb. Store bits 0-15 of Rd at the resulting address.
0	1	LDRH Rd, [Rb, Ro]	LDRH Rd, [Rb, Ro]	Load halfword: Add Ro to base address in Rb. Load bits 0-15 of Rd from the resulting address, and set bits 16-31 of Rd to 0.
1	0	LDSB Rd, [Rb, Ro]	LDRSB Rd, [Rb, Ro]	Load sign-extended byte: Add Ro to base address in Rb. Load bits 0-7 of Rd from the resulting address, and set bits 8-31 of Rd to bit 7.
1	1	LDSH Rd, [Rb, Ro]	LDRSH Rd, [Rb, Ro]	Load sign-extended halfword: Add Ro to base address in Rb. Load bits 0-15 of Rd from the resulting address, and set bits 16-31 of Rd to bit 15.

Instruction cycle times

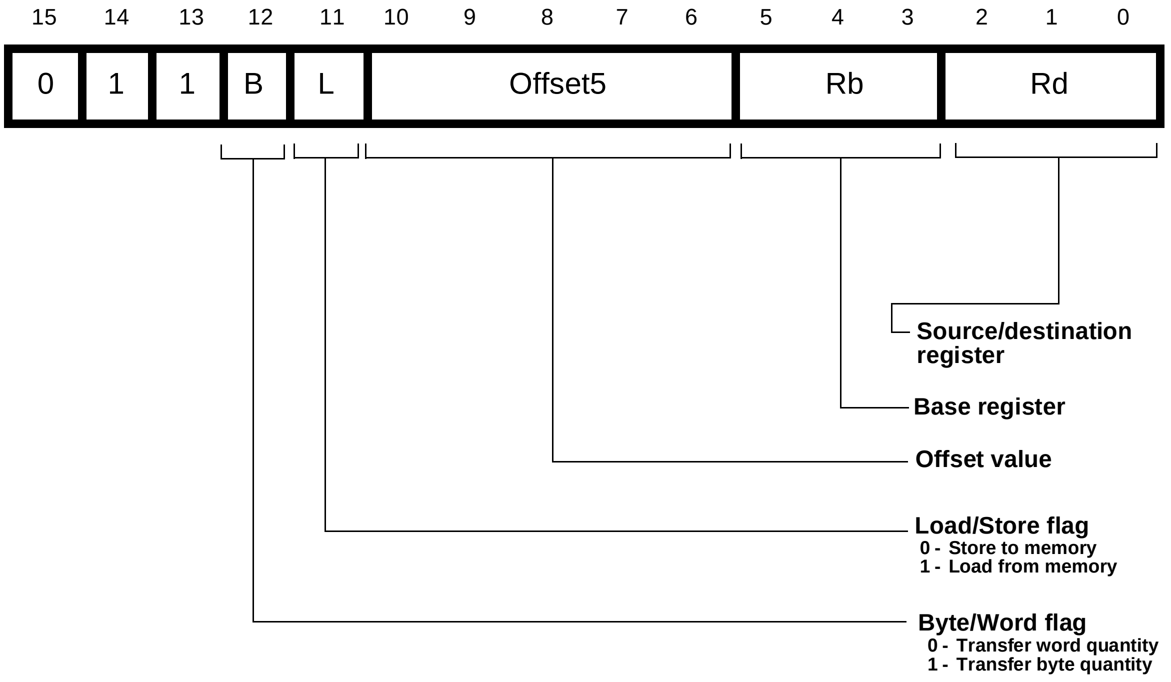
All instructions in this format have an equivalent ARM instruction as shown in Table 19. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
STRH  R4, [R3, R0]   ; Store the lower 16 bits of R4 at the
                     ; address formed by adding R0 to R3.

LDSB  R2, [R7, R1]   ; Load into R2 the sign extended byte
                     ; found at the address formed by adding
                     ; R1 to R7.

LDSH  R3, [R4, R2]   ; Load into R3 the sign extended halfword
                     ; found at the address formed by adding
                     ; R2 to R4.
```

Format 9: load/store with immediate offset

Figure 47. Format 9



Operation

These instructions transfer byte or word values between registers and memory using an immediate 5 or 7-bit offset.

The THUMB assembler syntax is shown in Table 20.

Table 20. Summary of Format 9 Instructions

L	B	THUMB assembler	ARM equivalent	Action
0	0	STR Rd, [Rb, #Imm]	STR Rd, [Rb, #Imm]	Calculate the target address by adding together the value in Rb and Imm. Store the contents of Rd at the address.
1	0	LDR Rd, [Rb, #Imm]	LDR Rd, [Rb, #Imm]	Calculate the source address by adding together the value in Rb and Imm. Load Rd from the address.
0	1	STRB Rd, [Rb, #Imm]	STRB Rd, [Rb, #Imm]	Calculate the target address by adding together the value in Rb and Imm. Store the byte value in Rd at the address.
1	1	LDRB Rd, [Rb, #Imm]	LDRB Rd, [Rb, #Imm]	Calculate source address by adding together the value in Rb and Imm. Load the byte value at the address into Rd.

Note: For word accesses (B = 0), the value specified by #Imm is a full 7-bit address, but must be word-aligned (ie with bits 1:0 set to 0), since the assembler places #Imm >> 2 in the Offset5 field.

Instruction cycle times

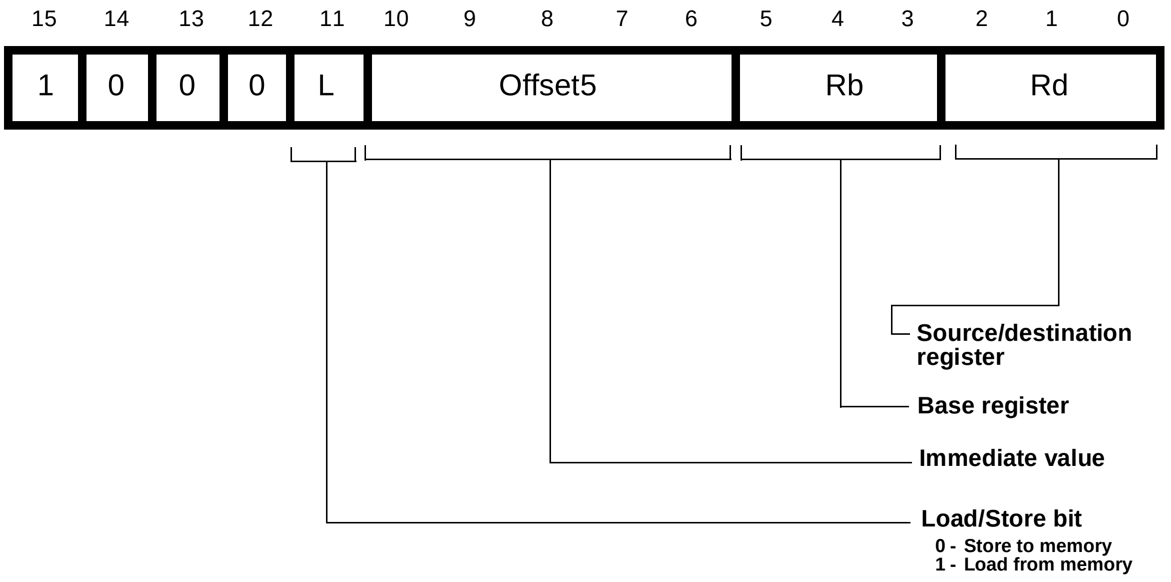
All instructions in this format have an equivalent ARM instruction as shown in Table 20. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
LDR   R2, [R5,#116]  ; Load into R2 the word found at the
                     ; address formed by adding 116 to R5.
                     ; Note that the THUMB opcode will
                     ; contain 29 as the Offset5 value.

STRB  R1, [R0,#13]   ; Store the lower 8 bits of R1 at the
                     ; address formed by adding 13 to R0.
                     ; Note that the THUMB opcode will
                     ; contain 13 as the Offset5 value.
```

Format 10: load/store halfword

Figure 48. Format 10



Operation

These instructions transfer halfword values between a Lo register and memory. Addresses are pre-indexed, using a 6-bit immediate value.

The THUMB assembler syntax is shown in Table 21.

Table 21. Halfword Data Transfer Instructions

L	THUMB assembler	ARM equivalent	Action
0	STRH Rd, [Rb, #Imm]	STRH Rd, [Rb, #Imm]	Add #Imm to base address in Rb and store bits 0-15 of Rd at the resulting address.
1	LDRH Rd, [Rb, #Imm]	LDRH Rd, [Rb, #Imm]	Add #Imm to base address in Rb. Load bits 0-15 from the resulting address into Rd and set bits 16-31 to zero.

Note: #Imm is a full 6-bit address but must be halfword-aligned (ie with bit 0 set to 0) since the assembler places #Imm >> 1 in the Offset5 field.

Instruction cycle times

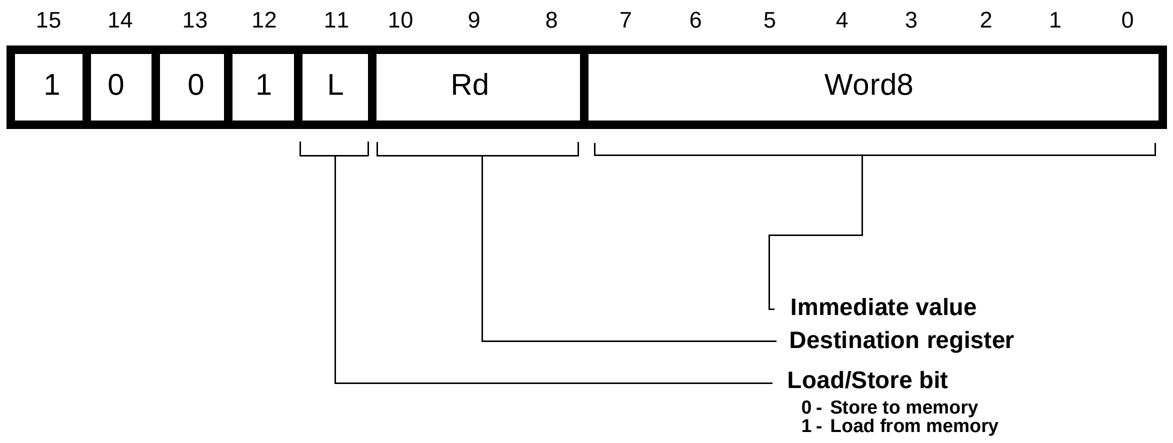
All instructions in this format have an equivalent ARM instruction as shown in Table 21. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
STRH  R6, [R1, #56]  ; Store the lower 16 bits of R4 at
                     ; the address formed by adding 56
                     ; R1.
                     ; Note that the THUMB opcode will
                     ; contain 28 as the Offset5 value.

LDRH  R4, [R7, #4]   ; Load into R4 the halfword found at
                     ; the address formed by adding 4 to R7.
                     ; Note that the THUMB opcode will contain
                     ; 2 as the Offset5 value.
```

Format 11: SP-relative load/store

Figure 49. Format 11



Operation

The instructions in this group perform an SP-relative load or store.The THUMB assembler syntax is shown in the following table.

Table 22. SP-Relative Load/Store Instructions

L	THUMB assembler	ARM equivalent	Action
0	STR Rd, [SP, #Imm]	STR Rd, [R13 #Imm]	Add unsigned offset (255 words, 1020 bytes) in Imm to the current value of the SP (R7). Store the contents of Rd at the resulting address.
1	LDR Rd, [SP, #Imm]	LDR Rd, [R13 #Imm]	Add unsigned offset (255 words, 1020 bytes) in Imm to the current value of the SP (R7). Load the word from the resulting address into Rd.

Note: The offset supplied in #Imm is a full 10-bit address, but must always be word-aligned (ie bits 1:0 set to 0), since the assembler places #Imm >> 2 in the Word8 field.

Instruction cycle times

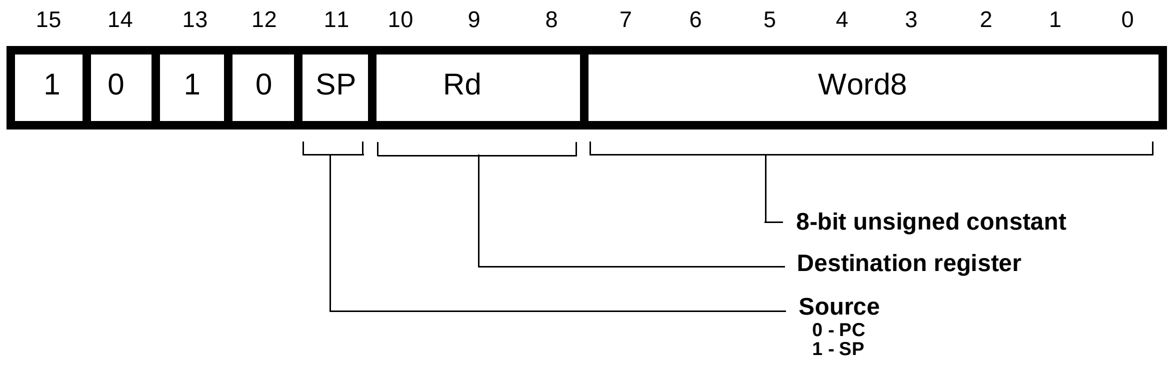
All instructions in this format have an equivalent ARM instruction as shown in Table 22. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
STR   R4, [SP,#492]  ; Store the contents of R4 at the address
                     ; formed by adding 492 to SP (R13).
                     ; Note that the THUMB opcode will contain
                     ; 123 as the Word8 value.
```

Format 12: load address

Figure 50. Format 12



Operation

These instructions calculate an address by adding an 10-bit constant to either the PC or the SP, and load the resulting address into a register.

The THUMB assembler syntax is shown in the following table

Table 23. Load Address

SP	THUMB assembler	ARM equivalent	Action
0	ADD Rd, PC, #Imm	ADD Rd, R15, #Imm	Add #Imm to the current value of the program counter (PC) and load the result into Rd.
1	ADD Rd, SP, #Imm	ADD Rd, R13, #Imm	Add #Imm to the current value of the stack pointer (SP) and load the result into Rd.

Note: The value specified by #Imm is a full 10-bit value, but this must be word-aligned (ie with bits 1:0 set to 0) since the assembler places #Imm >> 2 in field Word8.

Where the PC is used as the source register (SP = 0), bit 1 of the PC is always read as 0. The value of the PC will be 4 bytes greater than the address of the instruction before bit 1 is forced to 0.

The CPSR condition codes are unaffected by these instructions.

Instruction cycle times

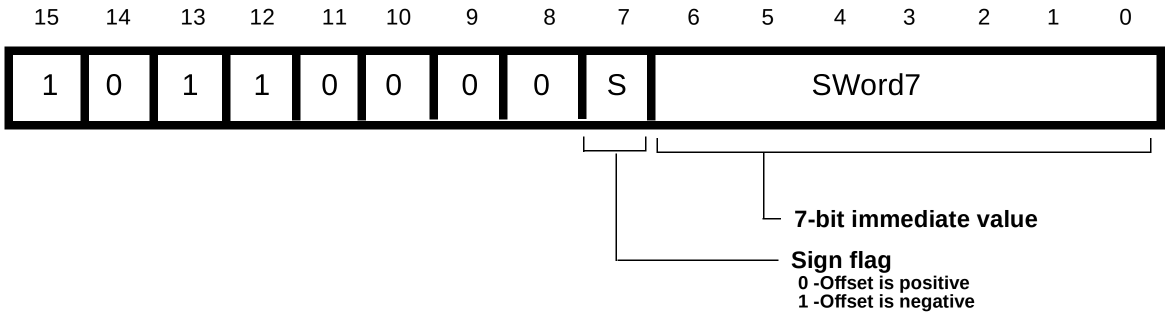
All instructions in this format have an equivalent ARM instruction as shown in Table 23. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
ADD   R2, PC, #572   ; R2 := PC + 572, but don't set the
                     ; condition codes. bit[1] of PC is
                     ; forced to zero.
                     ; Note that the THUMB opcode will
                     ; contain 143 as the Word8 value.

ADD   R6, SP, #212   ; R6 := SP (R13) + 212, but don't
                     ; set the condition codes.
                     ; Note that the THUMB opcode will
                     ; contain 53 as the Word8 value.
```

Format 13: add offset to Stack Pointer

Figure 51. Format 13



Operation

This instruction adds a 9-bit signed constant to the stack pointer. The following table shows the THUMB assembler syntax.

Table 24. The ADD SP Instruction

S	THUMB assembler	ARM equivalent	Action
0	ADD SP, #Imm	ADD R13, R13, #Imm	Add #Imm to the stack pointer (SP).
1	ADD SP, #-Imm	SUB R13, R13, #Imm	Add #-Imm to the stack pointer (SP).

Note: The offset specified by #Imm can be up to -/+ 508, but must be word-aligned (ie with bits 1:0 set to 0) since the assembler converts #Imm to an 8-bit sign + magnitude number before placing it in field SWord7.

Note: The condition codes are not set by this instruction.

Instruction cycle times

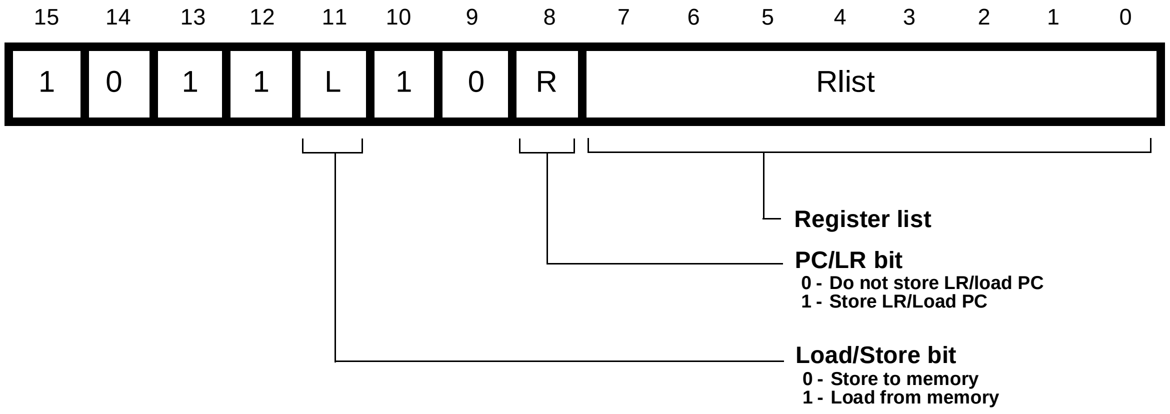
All instructions in this format have an equivalent ARM instruction as shown in Table 24. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175

Examples

```
ADD   SP, #268       ; SP (R13) := SP + 268, but don't set
                     ; the condition codes.
                     ; Note that the THUMB opcode will
                     ; contain 67 as the Word7 value and S=0.

ADD   SP, #-104      ; SP (R13) := SP - 104, but don't set
                     ; the condition codes.
                     ; Note that the THUMB opcode will contain
                     ; 26 as the Word7 value and S=1.
```

Format 14: push/pop registers

Figure 52. Format 14



Operation

The instructions in this group allow registers 0-7 and optionally LR to be pushed onto the stack, and registers 0-7 and optionally PC to be popped off the stack.

The THUMB assembler syntax is shown in Table 25.

Note: The stack is always assumed to be Full Descending.

Table 25. PUSH and POP Instructions

L	R	THUMB assembler	ARM equivalent	Action
0	0	PUSH { Rlist }	STMDB R13!, { Rlist }	Push the registers specified by Rlist onto the stack. Update the stack pointer.
0	1	PUSH { Rlist, LR }	STMDB R13!, { Rlist, R14 }	Push the Link Register and the registers specified by Rlist (if any) onto the stack. Update the stack pointer.
1	0	POP { Rlist }	LDMIA R13!, { Rlist }	Pop values off the stack into the registers specified by Rlist. Update the stack pointer.
1	1	POP { Rlist, PC }	LDMIA R13!, { Rlist, R15 }	Pop values off the stack and load into the registers specified by Rlist. Pop the PC off the stack. Update the stack pointer.

Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in Table 25. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Examples

```
PUSH  {R0-R4,LR}     ; Store R0,R1,R2,R3,R4 and R14 (LR) at
                     ; the stack pointed to by R13 (SP) and
                     ; update R13.
                     ; Useful at start of a sub-routine to
                     ; save workspace and return address.

POP   {R2,R6,PC}     ; Load R2,R6 and R15 (PC) from the stack
                     ; pointed to by R13 (SP) and update R13.
                     ; Useful to restore workspace and return
                     ; from sub-routine.
```

Format 15: multiple load/store

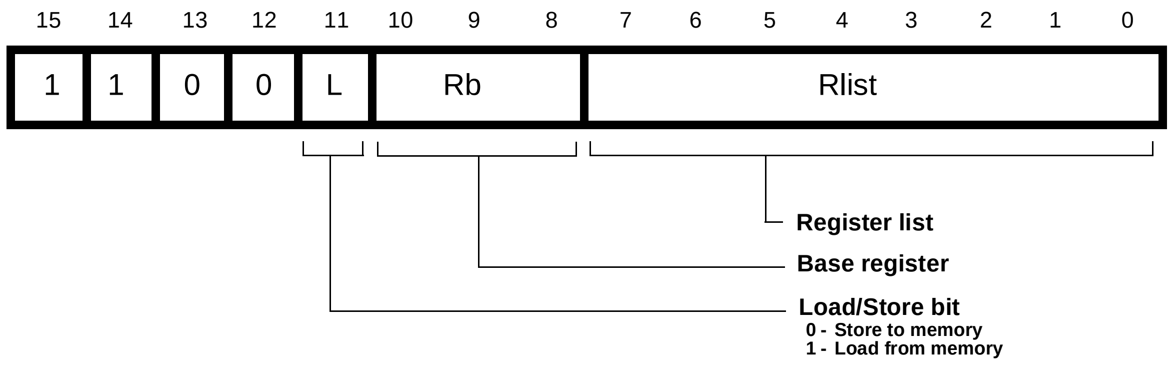


Figure 53. Format 15

Operation

These instructions allow multiple loading and storing of Lo registers. The THUMB assembler syntax is shown in the following table.

Table 26. The Multiple Load/Store Instructions

L	THUMB assembler	ARM equivalent	Action
0	STMIA Rb!, { Rlist }	STMIA Rb!, { Rlist }	Store the registers specified by Rlist, starting at the base address in Rb. Write back the new base address.
1	LDMIA Rb!, { Rlist }	LDMIA Rb!, { Rlist }	Load the registers specified by Rlist, starting at the base address in Rb. Write back the new base address.

Instruction cycle times

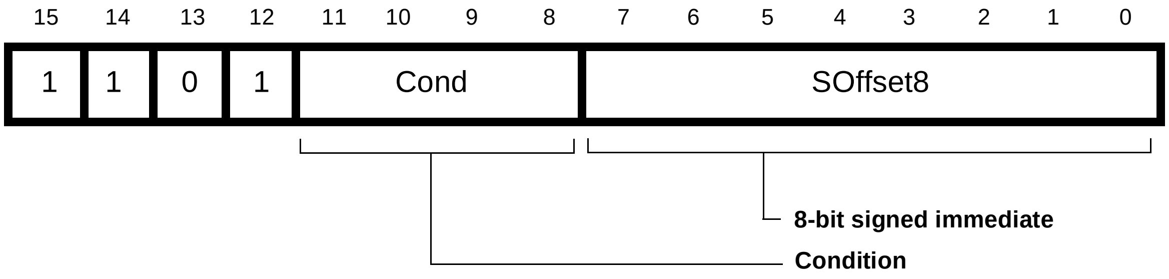
All instructions in this format have an equivalent ARM instruction as shown in Table 26. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175

Examples

```
STMIA R0!, {R3-R7}   ; Store the contents of registers R3-R7
                     ; starting at the address specified in
                     ; R0, incrementing the addresses for each
                     ; word.
                     ; Write back the updated value of R0.
```

Format 16: conditional branch

Figure 54. Format 16



Operation

The instructions in this group all perform a conditional Branch depending on the state of the CPSR condition codes. The branch offset must take account of the prefetch operation, which causes the PC to be 1 word (4 bytes) ahead of the current instruction.

The THUMB assembler syntax is shown in the following table.

Table 27. The Conditional Branch Instructions

Cond	THUMB assembler	ARM equivalent	Action
0000	BEQ label	BEQ label	Branch if Z set (equal)
0001	BNE label	BNE label	Branch if Z clear (not equal)
0010	BCS label	BCS label	Branch if C set (unsigned higher or same)
0011	BCC label	BCC label	Branch if C clear (unsigned lower)
0100	BMI label	BMI label	Branch if N set (negative)
0101	BPL label	BPL label	Branch if N clear (positive or zero)
0110	BVS label	BVS label	Branch if V set (overflow)
0111	BVC label	BVC label	Branch if V clear (no overflow)
1000	BHI label	BHI label	Branch if C set and Z clear (unsigned higher)
1001	BLS label	BLS label	Branch if C clear or Z set (unsigned lower or same)
1010	BGE label	BGE label	Branch if N set and V set, or N clear and V clear (greater or equal)
1011	BLT label	BLT label	Branch if N set and V clear, or N clear and V set (less than)
1100	BGT label	BGT label	Branch if Z clear, and either N set and V set or N clear and V clear (greater than)
1101	BLE label	BLE label	Branch if Z set, or N set and V clear, or N clear and V set (less than or equal)

Note: While label specifies a full 9-bit two's complement address, this must always be halfword-aligned (ie with bit 0 set to 0) since the assembler actually places label >> 1 in field SOffset8.

Note: Cond = 1110 is undefined, and should not be used. Cond = 1111 creates the SWI instruction: see Format 17: software interrupt on page 107.

Instruction cycle times

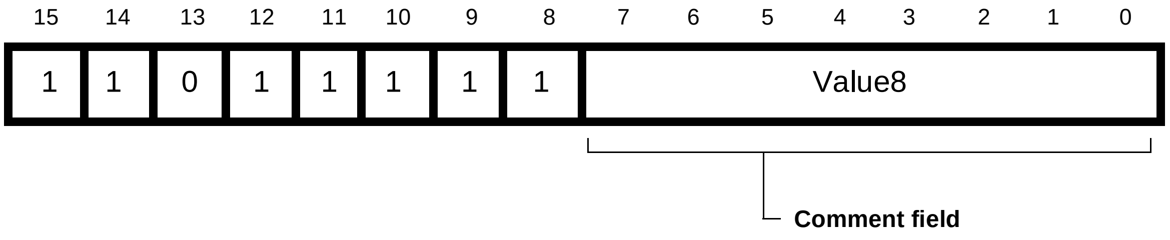
All instructions in this format have an equivalent ARM instruction as shown in Table 27. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175

Examples

```
      CMP R0, #45    ; Branch to ’over’ if R0 > 45.
      BGT over       ; Note that the THUMB opcode will contain
      ...            ; the number of halfwords to offset.
      ...
      ...
over  ...            ; Must be halfword aligned.
      ...
```

Format 17: software interrupt

Figure 55. Format 17



Operation

The SWI instruction performs a software interrupt. On taking the SWI, the processor switches into ARM state and enters Supervisor (SVC) mode.

The THUMB assembler syntax for this instruction is shown below

Table 28. The SWI Instruction

THUMB assembler	ARM equivalent	Action
SWI Value8	SWI Value8	Perform Software Interrupt: Move the address of the next instruction into LR, move CPSR to SPSR, load the SWI vector address (0x8) into the PC. Switch to ARM state and enter SVC mode.

Note: Value8 is used solely by the SWI handler: it is ignored by the processor.

Instruction cycle times

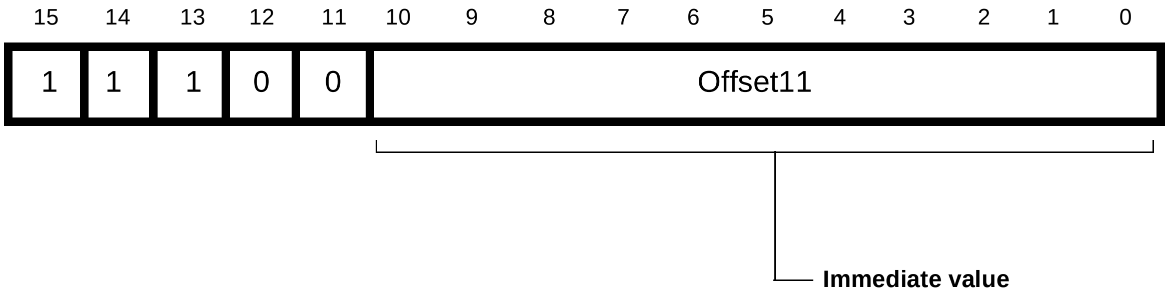
All instructions in this format have an equivalent ARM instruction as shown in Table 28. The instruction cycle times for the THUMB instruction are identical to that of the equivalent ARM instruction. For more information on instruction cycle times, please refer to Instruction Cycle Operations on page 175

Examples

```
SWI 18          ; Take the software interrupt exception.
                ; Enter Supervisor mode with 18 as the
                ; requested SWI number.
```

Format 18: unconditional branch

Figure 56. Format 18



Operation

This instruction performs a PC-relative Branch. The THUMB assembler syntax is shown below. The branch offset must take account of the prefetch operation, which causes the PC to be 1 word (4 bytes) ahead of the current instruction

Table 29. Summary of Branch Instruction

THUMB assembler	ARM equivalent	Action
B label	BAL label (halfword offset)	Branch PC relative +/- Offset11 << 1, where label is PC +/- 2048 bytes.

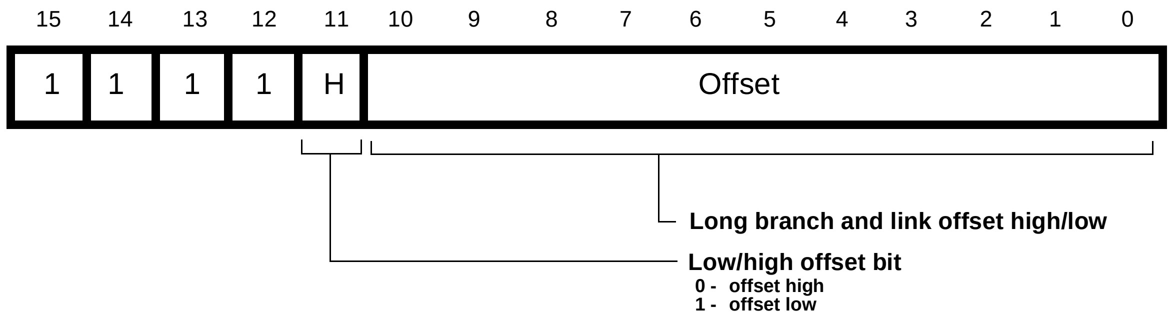
Note: The address specified by label is a full 12-bit two's complement address, but must always be halfword aligned (ie bit 0 set to 0), since the assembler places label >> 1 in the Offset11 field.

Examples

```
here  B here         ; Branch onto itself.
                     ; Assembles to 0xE7FE.
                     ; (Note effect of PC offset).
      B jimmy        ; Branch to 'jimmy'.
       ...           ; Note that the THUMB opcode will
                     ; contain the number of halfwords
                     ; to offset.
jimmy   ...          ; Must be halfword aligned.
```

Format 19: long branch with link

Figure 57. Format 19



Operation

This format specifies a long branch with link.

The assembler splits the 23-bit two's complement half-word offset specifed by the label into two 11-bit halves, ignoring bit 0 (which must be 0), and creates two THUMB instructions.

Instruction 1 (H = 0)

In the first instruction the Offset field contains the upper 11 bits of the target address. This is shifted left by 12 bits and added to the current PC address. The resulting address is placed in LR.

Instruction 2 (H =1)

In the second instruction the Offset field contains an 11-bit representation lower half of the target address. This is shifted left by 1 bit and added to LR. LR, which now contains the full 23-bit address, is placed in PC, the address of the instruction following the BL is placed in LR and bit 0 of LR is set.

The branch offset must take account of the prefetch operation, which causes the PC to be 1 word (4 bytes) ahead of the current instruction

Instruction cycle times

This instruction format does not have an equivalent ARM instruction. For details of the instruction cycle times, please refer to Instruction Cycle Operations on page 175.

Table 30. The BL Instruction

H	THUMB assembler	ARM equivalent	Action
0	BL label	none	LR := PC + OffsetHigh << 12

1			temp := next instruction address
			PC := LR + OffsetLow << 1
			LR := temp | 1

Examples

```
      BL faraway     ; Unconditionally Branch to 'faraway'
next  ...            ; and place following instruction
                     ; address, ie ’next’, in R14,the Link
                     ; Register and set bit 0 of LR high.
                     ; Note that the THUMB opcodes will
                     ; contain the number of halfwords to
                     ; offset.
faraway ...          ; Must be Half-word aligned.
```

Instruction Set Examples

The following examples show ways in which the THUMB instructions may be used to generate small and efficient code. Each example also shows the ARM equivalent so these may be compared.

Multiplication by a constant using shifts and adds

The following shows code to multiply by various constants using 1, 2 or 3 Thumb instructions alongside the ARM equivalents. For other constants it is generally better to use the built-in MUL instruction rather than using a sequence of 4 or more instructions.

```
Thumb                     ARM
```

1. Multiplication by 2^n (1,2,4,8,...)

```
LSL Ra, Rb, LSL #n        MOV Ra, Rb, LSL #n
```

2. Multiplication by 2^n+1 (3,5,9,17,...)

```
LSL Rt, Rb, #n            ADD Ra, Rb, Rb, LSL #n
ADD Ra, Rt, Rb
```

3. Multiplication by 2^n-1 (3,7,15,...)

```
LSL Rt, Rb, #n            RSB Ra, Rb, Rb, LSL #n
SUB Ra, Rt, Rb
```

4. Multiplication by -2^n (-2, -4, -8, ...)

```
LSL Ra, Rb, #n            MOV Ra, Rb, LSL #n
MVN Ra, Ra                RSB Ra, Ra, #0
```

5. Multiplication by -2^n-1 (-3, -7, -15, ...)

```
LSL Rt, Rb, #n            SUB Ra, Rb, Rb, LSL #n
SUB Ra, Rb, Rt
```

6. Multiplication by any C = {2^n+1, 2^n-1, -2^n or -2^n-1} * 2^n

   Effectively this is any of the multiplications in 2 to 5 followed by a final shift.

   This allows the following additional constants to be multiplied.

   6, 10, 12, 14, 18, 20, 24, 28, 30, 34, 36, 40, 48, 56, 60, 62 .....

```
(2..5)                    (2..5)
LSL Ra, Ra, #n            MOV Ra, Ra, LSL #n
```

General purpose signed divide

This example shows a general purpose signed divide and remainder routine in both Thumb and ARM code.

Thumb code

```
signed_divide
; Signed divide of R1 by R0: returns quotient in R0,
; remainder in R1

; Get abs value of R0 into R3
      ASR R2, R0, #31 ; Get 0 or -1 in R2 depending on sign of R0
      EOR R0, R2      ; EOR with -1 (0xFFFFFFFF) if negative
      SUB R3, R0, R2  ; and ADD 1 (SUB -1) to get abs value


; SUB always sets flag so go & report division by 0 if necessary
;     BEQ divide_by_zero

; Get abs value of R1 by xoring with 0xFFFFFFFF and adding 1
; if negative
      ASR R0, R1, #31 ; Get 0 or -1 in R3 depending on sign of R1
      EOR R1, R0      ; EOR with -1 (0xFFFFFFFF) if negative
      SUB R1, R0      ; and ADD 1 (SUB -1) to get abs value

; Save signs (0 or -1 in R0 & R2) for later use in determining ; sign of quotient & remainder.
      PUSH {R0, R2}

; Justification, shift 1 bit at a time until divisor (R0 value) ; is just <= than dividend
(R1 value). To do this shift dividend ; right by 1 and stop as soon as shifted value becomes
>.
      LSR R0, R1, #1
      MOV R2, R3
      B   %FT0

just_l    LSL  R2, #1
0     CMP      R2, R0
          BLS  just_l

          MOV  R0, #0         ; Set accumulator to 0
          B    %FT0           ; Branch into division loop

div_l     LSR  R2, #1
0     CMP      R1, R2         ; Test subtract
          BCC  %FT0
          SUB  R1, R2         ; If successful do a real
                              ; subtract
0         ADC  R0, R0         ; Shift result and add 1 if
                              ; subtract succeeded

CMP   R2, R3   ; Terminate when R2 == R3 (ie we have just
BNE   div_l    ; tested subtracting the 'ones' value).
```

```
; Now fixup the signs of the quotient (R0) and remainder (R1)
POP   {R2, R3} ; Get dividend/divisor signs back

EOR   R3, R2   ; Result sign
EOR   R0, R3   ; Negate if result sign = -1
SUB   R0, R3

EOR   R1, R2   ; Negate remainder if dividend sign = -1
SUB   R1, R2

MOV   pc, lr
```

ARM code

```
signed_divide
; effectively zero a4 as top bit will be shifted out later
      ANDS    a4, a1, #&80000000
      RSBMI   a1, a1, #0
      EORS    ip, a4, a2, ASR #32
; ip bit 31 = sign of result
; ip bit 30 = sign of a2
      RSBCS   a2, a2, #0

; central part is identical code to udiv
; (without MOV a4, #0 which comes for free as part of signed
; entry sequence)
      MOVS    a3, a1
      BEQ     divide_by_zero

just_l
; justification stage shifts 1 bit at a time
      CMP     a3, a2, LSR #1
      MOVLS   a3, a3, LSL #1
; NB: LSL #1 is always OK if LS succeeds
      BLO     s_loop

div_l
      CMP     a2, a3
      ADC     a4, a4, a4
      SUBCS   a2, a2, a3

      TEQ     a3, a1
      MOVNE   a3, a3, LSR #1
      BNE     s_loop2
      MOV     a1, a4

      MOVS    ip, ip, ASL #1
      RSBCS   a1, a1, #0
      RSBMI   a2, a2, #0

      MOV      pc, lr
```

Division by a constant

The ARM instruction set was designed following a RISC philosophy. One of the consequences of this is that the ARM core has no divide instruction, so divides must be performed using a subroutine. This means that divides can be quite slow, but this is not a major issue as divide performance is rarely critical for applications.

It is possible to do better than the general divide in the special case when the divisor is a constant. The divc.c example shows how the divide-by-constant technique works by generating ARM assembler code for divide-by-constant.

In the special case when dividing by 2^n, a simple right shift is all that is required.

There is a small caveat which concerns the handling of signed and unsigned numbers. For signed numbers, an arithmetic right shift is required, as this performs sign extension (to handle negative numbers correctly). In contrast, unsigned numbers require a 0-filled logical shift right:

```
MOV    a2, a1, lsr #5      ; unsigned division by 32
MOV    a2, a1, asr #10     ; signed division by 1024
```

Explanation of divide-by-constant ARM code

The divide-by-constant technique basically does a multiply in place of the divide. Given that:

x/y == x * (1/y)

consider the underlined portion as a 0.32 fixed-point number (truncating any bits past the most significant 32). 0.32 means 0 bits before the decimal point and 32 after it.

== (x * (2^32/y)) / 2^32

the underlined portion here is a 32.0 bit fixed-point number:

== (x * (2^32/y)) >> 32

This is effectively returning the top 32-bits of the 64-bit product of x and (2^32/y).

If y is a constant, then (2^32/y) is also a constant.

For certain y, the reciprocal (2^32/y) is a repeating pattern in binary:

```
  y                (2^32/y)

  2     10000000000000000000000000000000    #
  3     01010101010101010101010101010101    *
  4     01000000000000000000000000000000    #
  5     00110011001100110011001100110011    *
  6     00101010101010101010101010101010    *
  7     00100100100100100100100100100100    *
  8     00100000000000000000000000000000    #
  9     00011100011100011100011100011100    *
 10     00011001100110011001100110011001    *
 11     00010111010001011101000101110100
 12     00010101010101010101010101010101    *
 13     00010011101100010011101100010011
 14     00010010010010010010010010010010    *
 15     00010001000100010001000100010001    *
 16     00010000000000000000000000000000    #
 17     00001111000011110000111100001111    *
 18     00001110001110001110001110001110    *
 19     00001101011110010100001101011110
 20     00001100110011001100110011001100    *
 21     00001100001100001100001100001100    *
 22     00001011101000101110100010111010
```

```
23      00001011001000010110010000101100
24      00001010101010101010101010101010    *
25      00001010001111010111000010100011
```

The lines marked with a ’#’ are the special cases 2^n, which have already been dealt with. The lines marked with a ’*’ have a simple repeating pattern.

Note how regular the patterns are for y=2^n+2^m or y=2^n-2^m (for n>m):

n	m	(2^n+2^m)	n	m	(2^n-2^m)
1	0	3	1	0	1
2	0	5	2	1	2
2	1	6	2	0	3
3	0	9	3	2	4
3	1	10	3	1	6
3	2	12	3	0	7
4	0	17	4	3	8
4	1	18	4	2	12
4	2	20	4	1	14
4	3	24	4	0	15
5	0	33	5	4	16
5	1	34	5	3	24
5	2	36	5	2	28
5	3	40	5	1	30
5	4	48	5	0	31

For the repeating patterns, it is a relatively easy matter to calculate the product by using a multiply-by-constant method.

The result can be calculated in a small number of instructions by taking advantage of the repetition in the pattern.

The actual multiply is slightly unusual due to the need to return the top 32 bits of the 64-bit result. It efficient to calculate just the top 32 bits.

Consider this fragment of the divide-by-ten code (x is the input dividend as used in the above equations):

```
SUB  a1,  x,  x, lsr #2   ; a1 = x*%0.11000000000000000000000000000000
ADD  a1, a1, a1, lsr #4   ; a1 = x*%0.11001100000000000000000000000000
ADD  a1, a1, a1, lsr #8   ; a1 = x*%0.11001100110011000000000000000000
ADD  a1, a1, a1, lsr #16  ; a1 = x*%0.11001100110011001100110011001100
MOV  a1, a1, lsr #3       ; a1 = x*%0.00011001100110011001100110011001
```

The SUB calculates (for example):

a1 = x - x/4

= x - x*%0.01

= x*%0.11

Therefore, just five instructions are needed to perform the multiply.

A small problem is caused by calculating just the top 32 bits, as this ignores any carry from the low 32 bits of the 64-bit product. Fortunately, this can be corrected. A correct divide would round down, so the remainder can be calculated by:

x - (x/10)*10 = 0..9

By making good use of the ARM's barrel shifter, it takes just two ARM instructions to perform this multiply-by-10 and subtract. In the case when (x/10) is too small by 1 (if carry has been lost), the remainder will be in the range 10..19, in which case corrections must be applied. This test would require a compare-with-10 instruction, but this can be combined with other operations to save an instruction (see below).

When a lost carry is detected, both the quotient and remainder must be fixed up (one instruction each).

The following fragments should explain the full divide-by-10 code.

ARM code

```
div10
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10             ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1              ; fix-up quotient
    ADDMI  a2, a2, #10             ; fix-up remainder
    MOV    pc, lr
```

The optimisation which eliminates the compare-with-10 instruction is to keep (x-10) for use in the subtraction to calculate the remainder. This means that compare-with-0 is required instead, which is easily achieved by adding an S (to set the flags) to the SUB opcode. This also means that the subtraction has to be undone if no rounding error occurred (which is why the ADDMI instruction is used).

THUMB code

```
udiv10
; takes argument in a1
; returns quotient in a1, remainder in a2
      MOV     a2, a1
      LSR     a3, a1, #2
      SUB     a1, a3
      LSR     a3, a1, #4
      ADD     a1, a3
      LSR     a3, a1, #8
      ADD     a1, a3
      LSR     a3, a1, #16
      ADD     a1, a3
      LSR     a1, #3
      ASL     a3, a1, #2
      ADD     a3, a1
      ASL     a3, #1
      SUB     a2, a3
      CMP     a2, #10
      BLT     %FT0
      ADD     a1, #1
      SUB     a2, #10
0
      MOV     pc, lr
```

Memory Interface

This chapter describes the ARM7TDMI memory interface.

Overview

ARM7TDMI's memory interface consists of the following basic elements:

- 32-bit address bus

  This specifies to memory the location to be used for the transfer.
- 32-bit data bus

  Instructions and data are transferred across this bus. Data may be word, halfword or byte wide in size.

  ARM7TDMI includes a bidirectional data bus, **D[31:0**], plus separate unidirectional data busses, **DIN[31:0]** and **DOUT[31:0]**. Most of the text in this chapter describes the bus behaviour assuming that the bidirectional is in use. However, the behaviour applies equally to the unidirectional busses.
- Control signals

  These specify, for example, the size of the data to be transferred, and the direction of the transfer together with providing privileged information.

This collection of signals allow ARM7TDMI to be simply interfaced to DRAM, SRAM and ROM. To fully exploit page mode access to DRAM, information is provided on whether or not the memory accesses are sequential. In general, interfacing to static memories is much simpler than interfacing to dynamic memory.

Cycle Types

All memory transfer cycles can be placed in one of four categories:

1. Non-sequential cycle. ARM7TDMI requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.
2. Sequential cycle. ARM7TDMI requests a transfer to or from an address which is either the same as the address in the preceding cycle, or is one word or halfword after the preceding address.
3. Internal cycle. ARM7TDMI does not require a transfer, as it is performing an internal function and no useful prefetching can be performed at the same time.
4. Coprocessor register transfer. ARM7TDMI wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

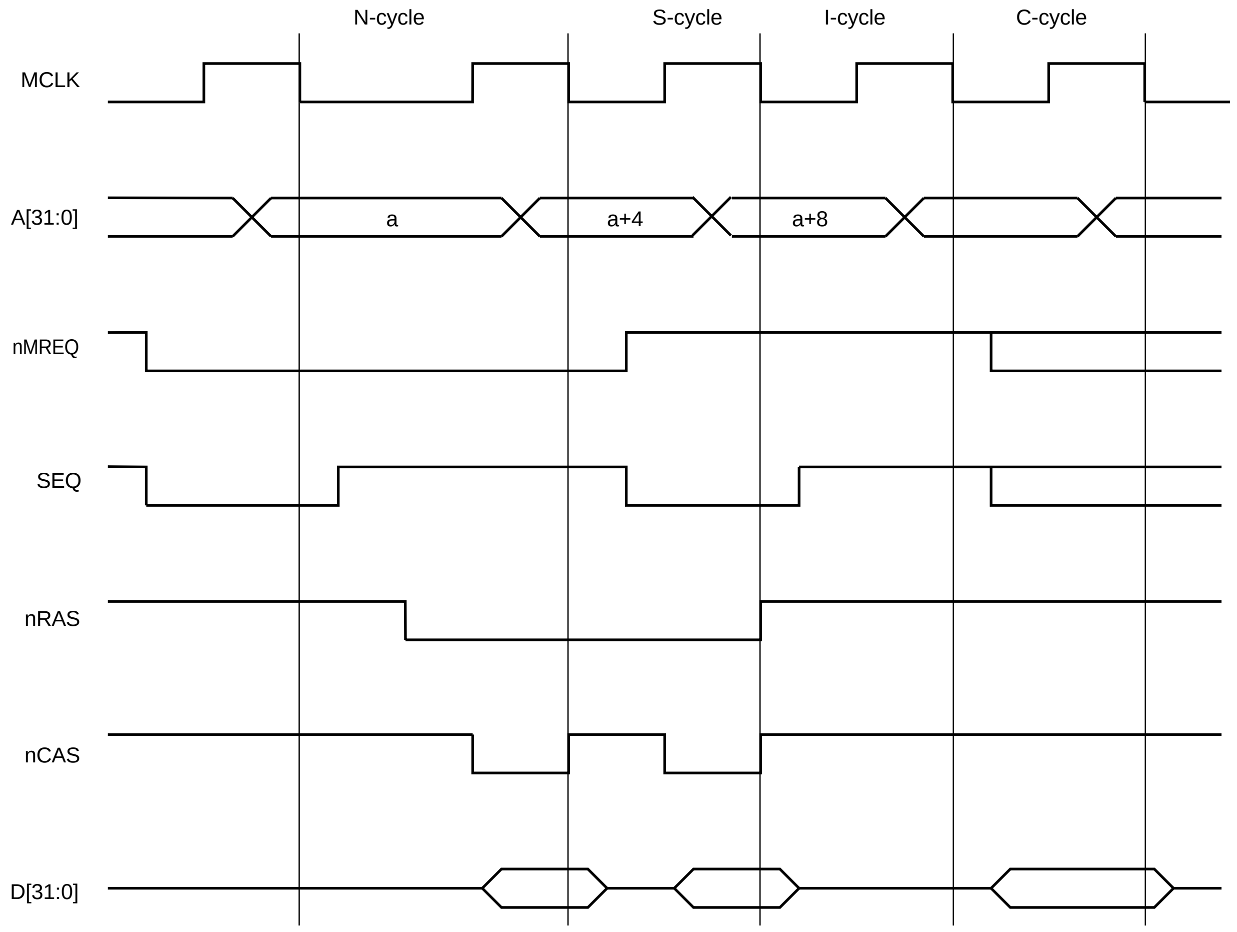
These four classes are distinguishable to the memory system by inspection of the **nMREQ** and **SEQ** control lines (see Table 31). These control lines are generated during phase 1 of the cycle before the cycle whose characteristics they forecast, and this pipelining of the control information gives the memory system sufficient time to decide whether or not it can use a page mode access.

Table 31. Memory Cycle Types

nMREQ	SEQ	Cycle type
0	0	Non-sequential (N-cycle)
0	1	Sequential (S-cycle)
1	0	Internal (I-cycle)
1	1	Coprocessor register transfer (C-cycle)

Figure 58 shows the pipelining of the control signals, and suggests how the DRAM address strobes (**nRAS** and **nCAS**) might be timed to use page mode for S-cycles. Note that the N-cycle is longer than the other cycles. This is to allow for the DRAM precharge and row access time, and is not an ARM7TDMI requirement.

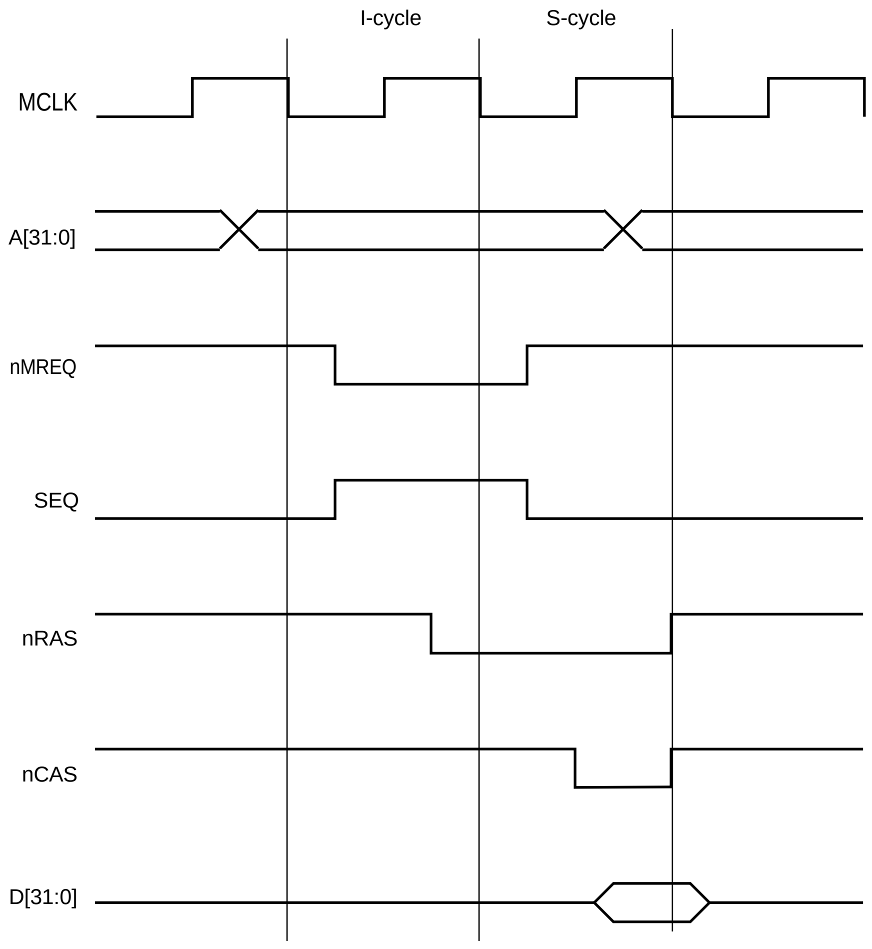
Figure 58. ARM Memory Cycle Timing



When an S-cycle follows an N-cycle, the address will always be one word or halfword greater than the address used in the N-cycle. This address (marked “a” in the above diagram) should be checked to ensure that it is not the last in the DRAM page before the memory system commits to the S-cycle. If it is at the page end, the S-cycle cannot be performed in page mode and the memory system will have to perform a full access.

The processor clock must be stretched to match the full access. When an S-cycle follows an I-cycle, the address will be the same as that used in the I-cycle. This fact may be used to start the DRAM access during the preceding cycle, which enables the S-cycle to run at page mode speed whilst performing a full DRAM access. This is shown in Figure 59.

Figure 59. Memory Cycle Optimization



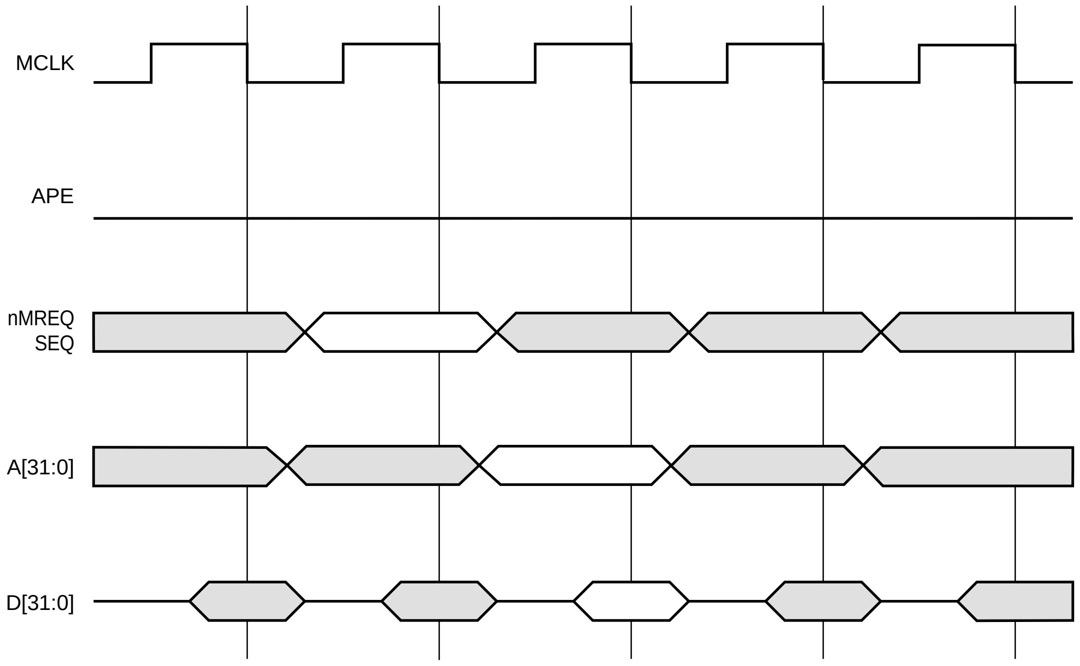
Address Timing

ARM7TDMI's address bus can operate in one of two configurations - pipelined or depipelined, and this is controlled by the **APE** input signal. The configurability is provided to ease the design in of ARM7TDMI to both SRAM and DRAM based systems.

It is a requirement SRAMs and ROMs that the address be held stable throughout the memory cycle. In a system containing SRAM and ROM only, **APE** may be tied permanently LOW, producing the desired address timing. This is shown in Figure 60.

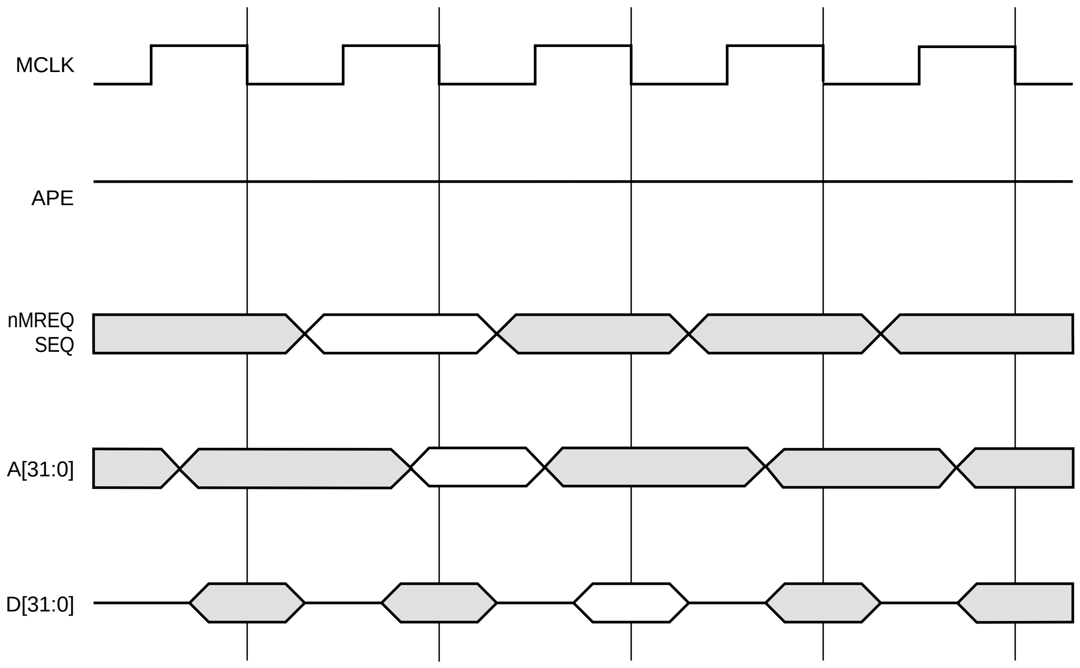
Note: APE affects the timing of the address bus **A[31:0]**, plus **nRW**, **MAS[1:0]**, **LOCK**, **nOPC** and **nTRANS**.

Figure 60. ARM7TDMI De-Pipelined Addresses



In a DRAM based system, it is desirable to obtain the address from ARM7TDMI as early as possible. When **APE** is HIGH, ARM7TDMI's address becomes valid in the **MCLK** high phase before the memory cycle to which it refers. This timing allows longer for address decoding and the generation of DRAM control signals. Figure 61 shows the effect on the timing when **APE** is HIGH.

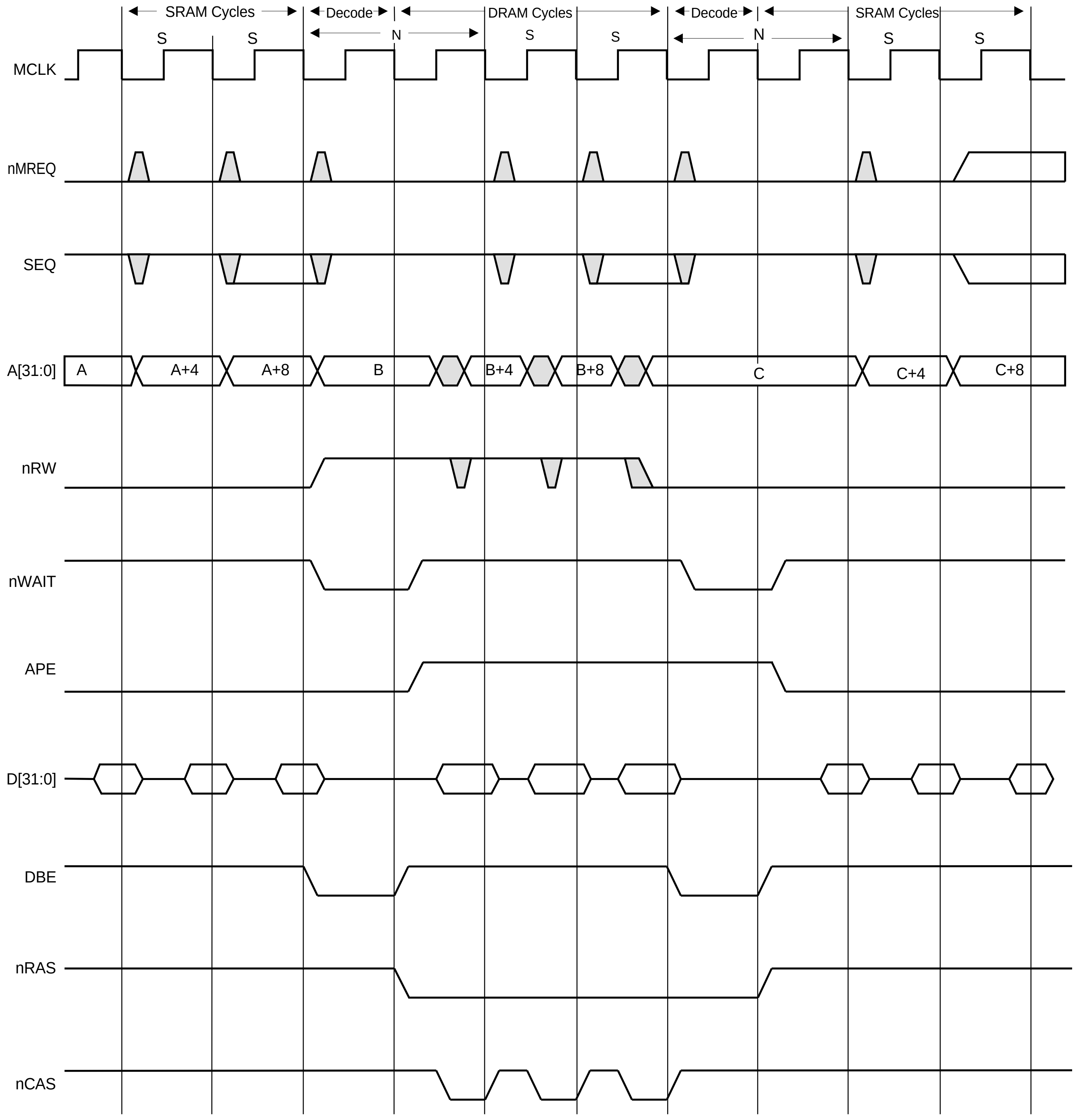
Figure 61. ARM7TDMI Pipelined Addresses



Many systems will contain a mixture of DRAM and SRAM/ROM. To cater for the different address timing requirements, **APE** may be safely changed during the low phase of **MCLK**. Typically, **APE** would be held at one level during a burst of sequential accesses to one type of memory. When a non-sequential access occurs, the timing of most systems enforce a wait state to allow for address decoding. As a result of the address decode, **APE** can be driven to the correct value for the particular bank of memory being accessed. The value of **APE** can be held until the memory control signals denote another non-sequential access.

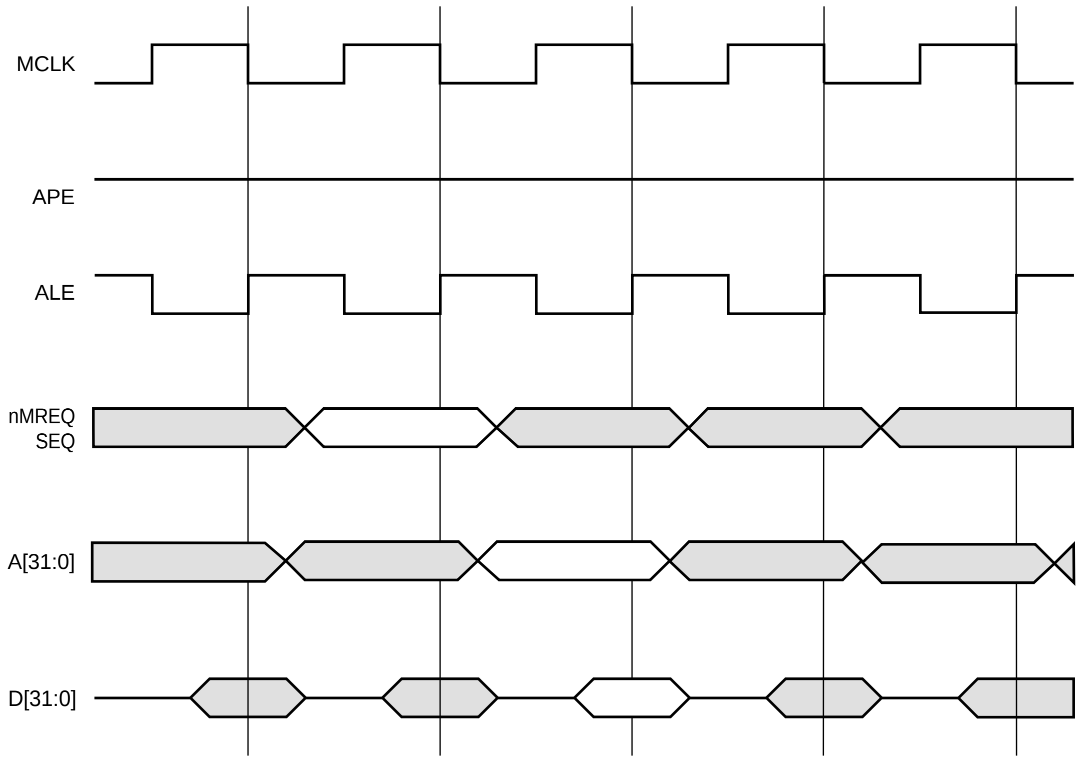
By way of an example, Figure 62 shows a combination of accesses to a mixed DRAM / SRAM system. Here, the SRAM has zero wait states, and the DRAM has a 2:1 N-cycle / S-cycle ratio. A single wait state is inserted for address decode when a non-sequential access occurs. Typical, externally generated DRAM control signals are also shown.

Figure 62. Typical System Timing



Previous ARM processors included the **ALE** signal, and this is retained for backwards compatibility. This signal also allows the address timing to be modified to achieve the same results as **APE**, but in an asynchronous manner. To obtain clean **MCLK** low timing of the address bus by this mechanism, **ALE** must be driven HIGH with the falling edge of **MCLK**, and LOW with the rising edge of **MCLK**. **ALE** can simply be the inverse of **MCLK** but the delay from **MCLK** to **ALE** must be carefully controlled such that the *Tald* timing constraint is achieved. Figure 63 shows how **ALE** can be used to achieve SRAM compatible address timing. Refer to Timing Diagrams on page 189 for details of the exact timing constraints.

Figure 63. SRAM Compatible AddressTiming



Note: If **ALE** is to be used to change address timing, then **APE** must be tied HIGH. Similarly, if **APE** is to be used, **ALE** must be tied HIGH.

Data Transfer Size

In an ARM7TDMI system, words, halfwords or bytes may be transferred between the processor and the memory. The size of the transaction taking place is determined by the **MAS[1:0]** pins. These are encoded as follows:

MAS[1:0]	00	Byte
	01	halfword
	10	word
	11	reserved

The processor always produces a byte address, but instructions are either words (4 bytes) or halfwords (2 bytes), and data can be any size. Note that when word instructions are fetched from memory, **A[1:0]** are undefined and when halfword instructions are fetched, **A[0]** is undefined. The **MAS[1:0]** outputs share the same timing as the address bus and thus can be modified by the use of **ALE** and **APE** as described in Address Timing on page 120.

When a data read of byte or halfword size is performed (eg LDRB), the memory system may safely ignore the fact that the request is for a sub-word sized quantity and present the whole word. ARM7TDMI will always correctly extract the addressed byte or halfword from the data. The memory system may also choose just to supply the addressed byte or halfword. This may be desirable in order to save power or to simplify the decode logic.

When a byte or halfword write occurs (eg STRH), ARM7TDMI will broadcast the byte or halfword across the whole of the bus. The memory system must then decode **A[1:0]** to enable writing only to the addressed byte or halfword.

One way of implementing the byte decode in a DRAM system is to separate the 32-bit wide block of DRAM into four byte wide banks, and generate the column address strobes independently as shown in Figure 64.

When the processor is configured for Little Endian operation, byte 0 of the memory system should be connected to data lines 7 through 0 (**D[7:0]**) and strobed by **nCAS0**. **nCAS1** drives the bank connected to data lines 15 though 8, and so on. This has the added advantage of reducing the load on each column strobe driver, which improves the precision of this time-critical-signal.

In the Big Endian case, byte 0 of the memory system should be connected to data lines 31 through 24.

Instruction Fetch

ARM7TDMI will perform 32- 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 **TBIT** signal. When this is LOW, the processor is in ARM state and 32-bit instructions are fetched. When **TBIT** is HIGH, the processor is in THUMB state and 16-bit instructions are fetched. The size of the data being fetched is also indicated on the **MAS[1:0]** bits, as described above.

When the processor is in ARM state, 32-bit instructions are fetched on **D[31:0]**. When the processor is in THUMB state, 16-bit instructions are fetched from either the upper, **D[31:16**], or the lower **D[15:0**] half of the bus. This is determined by the endianism of the memory system, as configured by the **BIGEND** input, and the state of **A[1]**. Table 32 shows which half of the data bus is sampled in the different configurations.

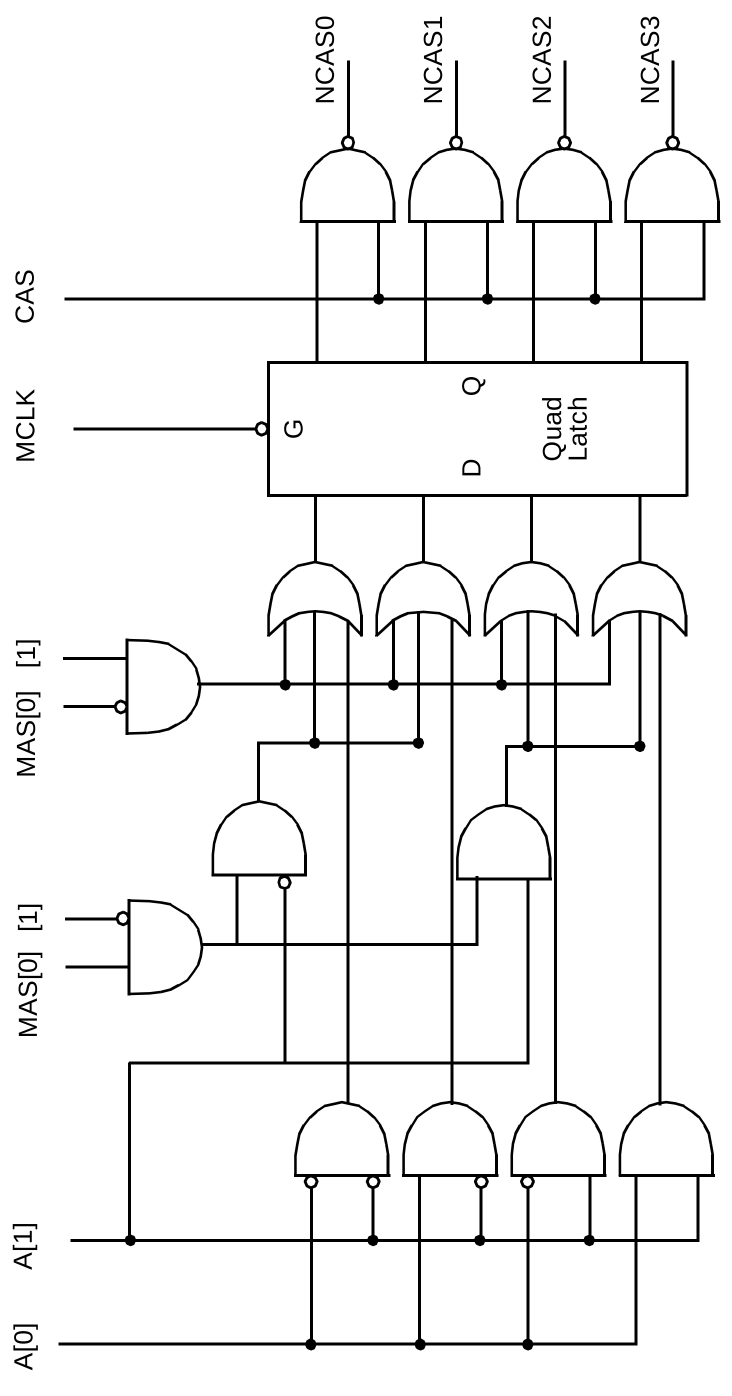
Table 32. Endianism Effect on Instruction Position

	Endianism	
	Little BIGEND = 0	**Big BIGEND = 1**
A[1] = 0	D[15:0]	D[31:16]
A[1] = 1	D[31:16]	D[15:0]

When a 16-bit instruction is fetched, ARM7TDMI ignores the unused half of the data bus.

Table 32 describes instructions fetched from the bidirectional data bus (i.e. **BUSEN** is LOW). When the unidirectional data busses are in use (i.e. **BUSEN** is HIGH), data will be fetched from the corresponding half of the **DIN[31:0]** bus.

Figure 64. Decoding Byte Accesses to Memory



Memory Management

The ARM7TDMI address bus may be processed by an address translation unit before being presented to the memory, and ARM7TDMI is capable of running a virtual memory system. The **ABORT** input to the processor may be used by the memory manager to inform ARM7TDMI of page faults. Various other signals enable different page protection levels to be supported:

1. **nRW** can be used by the memory manager to protect pages from being written to.
2. **nTRANS** indicates whether the processor is in user or a privileged mode, and may be used to protect system pages from the user, or to support completely separate mappings for the system and the user.

Address translation will normally only be necessary on an N-cycle, and this fact may be exploited to reduce power consumption in the memory manager and avoid the translation delay at other times. The times when translation is necessary can be deduced by keeping track of the cycle types that the processor uses.

Locked Operations

The ARM instruction set of ARM7TDMI includes a data swap (SWP) instruction that allows the contents of a memory location to be swapped with the contents of a processor register. This instruction is implemented as an uninterruptable pair of accesses; the first access reads the contents of the memory, and the second writes the register data to the memory. These accesses must be treated as a contiguous operation by the memory controller to prevent another device from changing the affected memory location before the swap is completed. ARM7TDMI drives the **LOCK** signal HIGH for the duration of the swap operation to warn the memory controller not to give the memory to another device.

Stretching Access Times

All memory timing is defined by **MCLK**, and long access times can be accommodated by stretching this clock. It is usual to stretch the LOW period of **MCLK**, as this allows the memory manager to abort the operation if the access is eventually unsuccessful.

Either **MCLK** can be stretched before it is applied to ARM7TDMI, or the **nWAIT** input can be used together with a free-running **MCLK**. Taking **nWAIT** LOW has the same effect as stretching the LOW period of **MCLK**, and **nWAIT** must only change when **MCLK** is LOW.

ARM7TDMI does not contain any dynamic logic which relies upon regular clocking to maintain its internal state. Therefore there is no limit upon the maximum period for which **MCLK** may be stretched, or **nWAIT** held LOW.

The ARM Data Bus

To ease the connection of ARM7TDMI to sub-word sized memory systems, input data and instructions may be latched on a byte by byte basis. This is achieved by use of the **BL[3:0]** input signals where **BL[3]** controls the latching of the data present on **D[31:24]** of the data bus and so on.

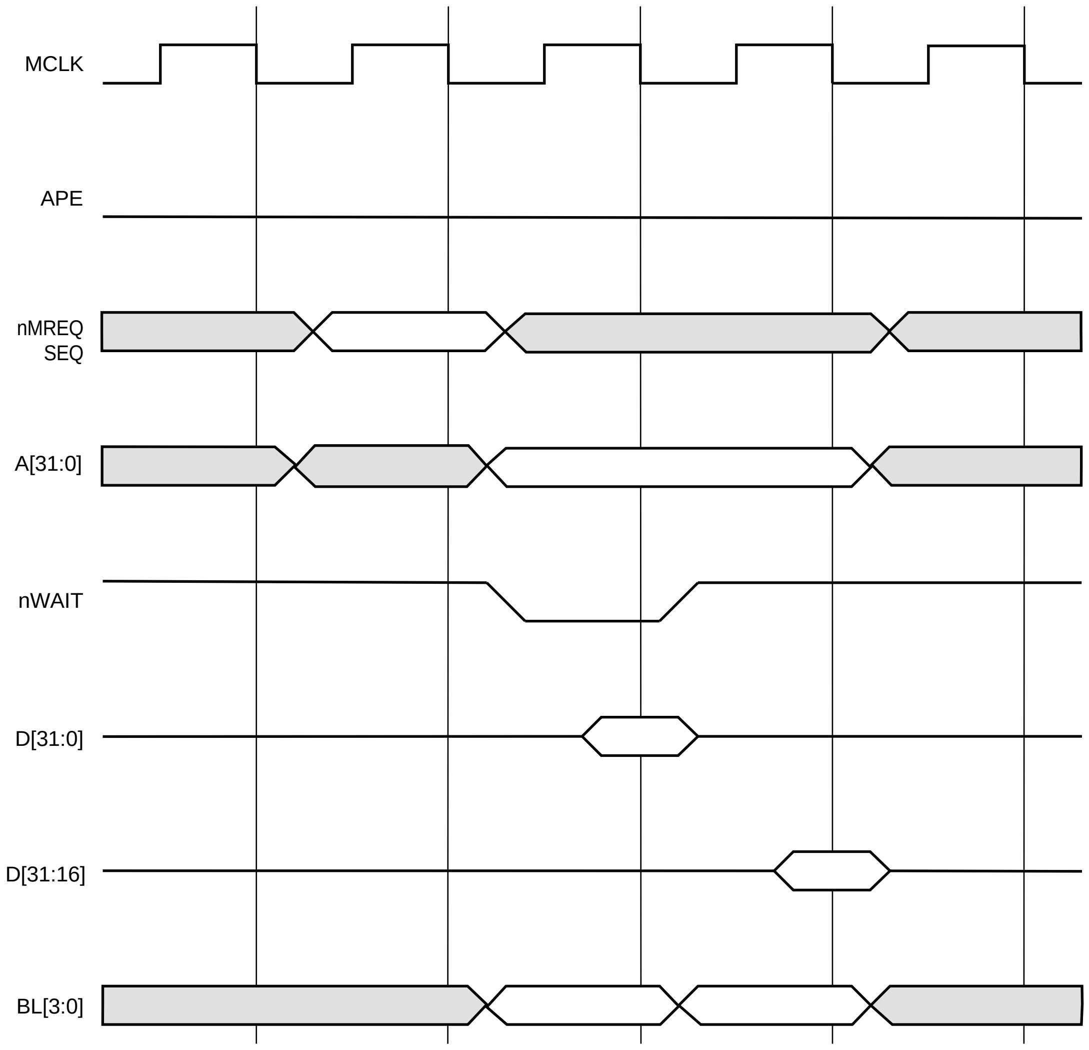
In a memory system containing word wide memory only, **BL[3:0]** may be tied HIGH. For sub word wide memory systems, **BL[3:0]** are used to latch the data as it is read out of memory. For example, a word access to halfword wide memory must take place in two memory cycles. In the first cycle, the data for **D[15:0]** is obtained from the memory and latched into the processor on the falling edge of **MCLK** when **BL[1:0]** are both HIGH. In the second cycle, the data for **D[31:16]** is latched into the processor on the falling edge of **MCLK** when **BL[3:2]** are both HIGH.

A memory access like this is shown in Figure 65. Here, a word access is performed from halfword wide memory in two cycles.In the first, the data read is applied to the lower half of the bus, in the second cycle the read data is applied to the upper half of the bus. Since two memory cycles were required, **nWAIT** is used to stretch the internal processor clock. However, **nWAIT** does not effect the operation of the data latches. In this way, data may be extracted from memory word, halfword or byte at a time, and the memory may have as many wait states as required. In any multi-cycle memory access, **nWAIT** is held LOW until the final quantum of data is latched.

In this example, **BL[3:0]** were driven to value 0x3 in the first cycle so that only the latches on **D[15:0]** were opened. In fact, **BL[3:0]** could have been driven to value 0xF and all the latches opened. Since in the second cycle, the latches on **D[31:16]** were written with the correct data, this would not have effected the processor's operation.

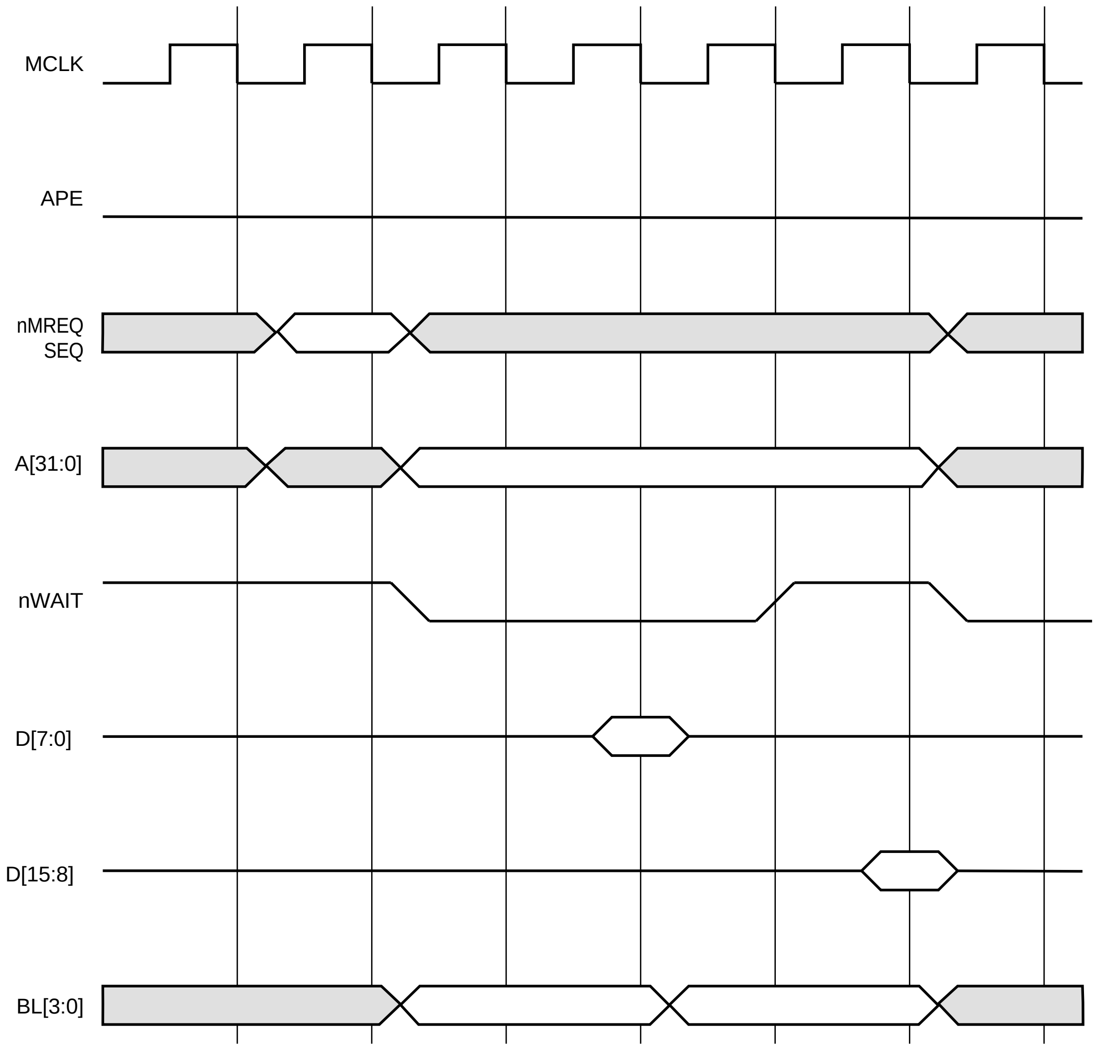
Note: BL[3:0] should all be HIGH during store cycles.

Figure 65. Memory Access



As a further example, a halfword load from 2-wait state byte wide memory is shown in Figure 66. Here, each memory access takes two cycles. In the first, access, **BL[3:0]** are driven to value 0xF. The correct data is latched from **D[7:0]** whilst unknown data is latched from **D[31:8]**. In the second access, the byte for **D[15:8]** is latched and so the halfword on **D[15:0]** has been correctly read from the memory. The fact that internally **D[31:16]** are unknown does not matter because internally the processor will extract only the halfword it is interested in.

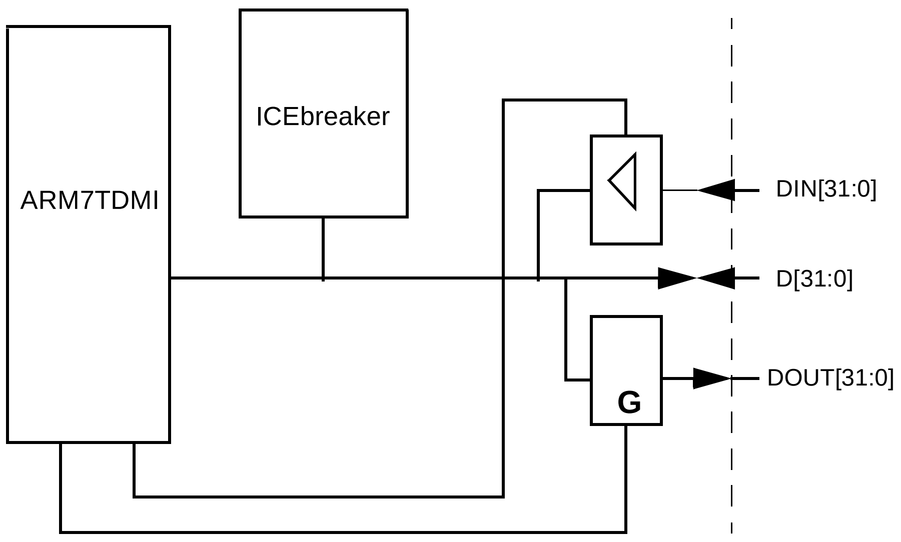
Figure 66. Two-Cycle Memory Access



The External Data Bus

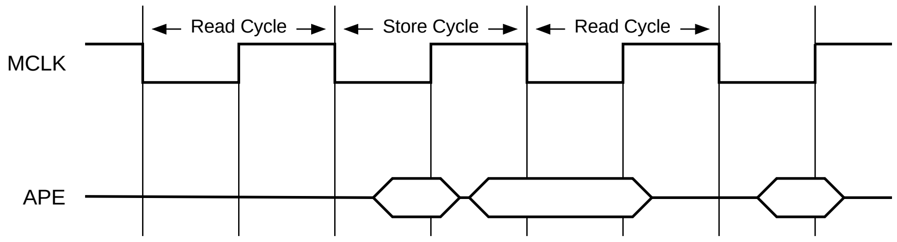
ARM7TDMI has a bidirectional data bus, **D[31:0]**. However, since some ASIC design methodologies prohibit the use of bidirectional buses, unidirectional data in, **DIN[31:0]**, and data out, **DOUT[31:0]**, busses are also provided. The logical arrangement of these buses is shown inFigure 67.

Figure 67. ARM7TDMI External Bus Arrangement



When the bidirectional data bus is being used, the unidirectional busses must be disabled by driving **BUSEN** LOW. The timing of the bus for three cycles, load-store-load, is shown in Figure 68.

Figure 68. Bidirectional Bus Timing



The unidirectional data bus

When the unidirectional data busses are being used, (i.e. when **BUSEN** is HIGH), the bidirectional bus, **D[31:0]**, must be left unconnected.

When **BUSEN** is HIGH, all instructions and input data are presented on the input data bus, **DIN[31:0]**. The timing of this data is similar to that of the bidirectional bus when in input mode. Data must be set up and held to the falling edge of **MCLK**. For the exact timing requirements refer to Timing Diagrams on page 189.

In this configuration, all output data is presented on **DOUT[31:0]**. The value on this bus only changes when the processor performs a store cycle. Again, the timing of the data is similar to that of the bidirectional data bus. The value on **DOUT[31:0]** changes off the falling edge of **MCLK**.

The bus timing of a read-write-read cycle combination is shown in Figure 69.

When **BUSEN** is LOW, the buffer between **DIN[31:0]** and **D[31:0]** is disabled. Any data presented on **DIN[31:0]** is ignored. Also, when **BUSEN** is low, the value on **DOUT[31:0]** is forced to 0x00000000.

Typically, the unidirectional busses would be used internally in ASIC embedded applications. Externally, most systems still require a bidirectional data bus to interface to external memory. Figure 70 shows how the unidirectional busses may be joined up at the pads of an ASIC to connect to an external bidirectional bus.

Figure 69. Unidirectional Bus Timing

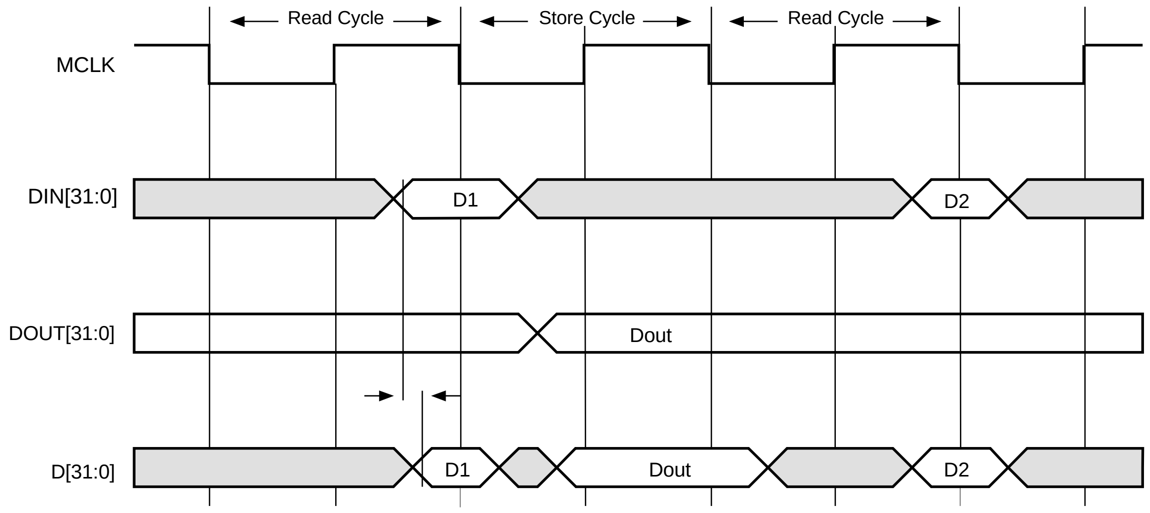
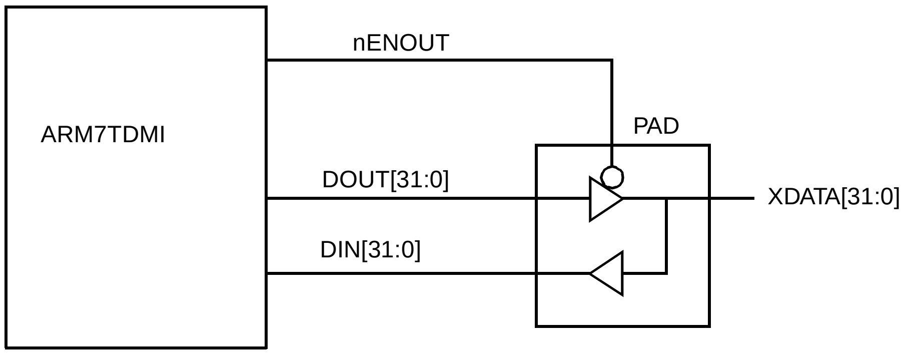


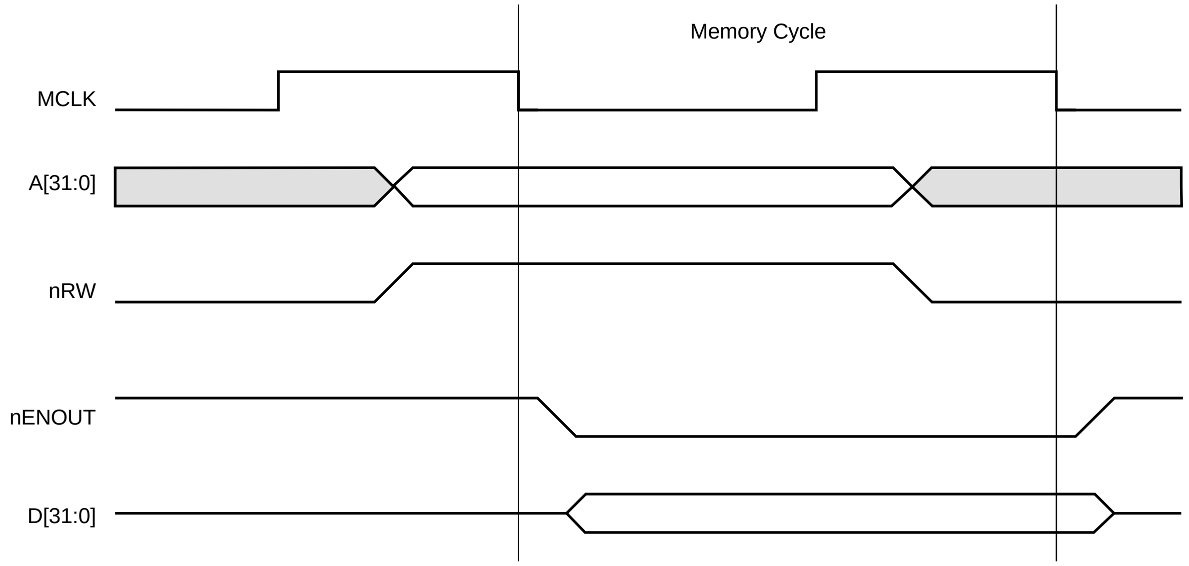
Figure 70. External Connection of Unidirectional Busses



The bidirectional data bus

ARM7TDMI has a bidirectional data bus, **D[31:0]**. Most of the time, the ARM reads from memory and so this bus is configured to input. During write cycles however, the ARM7TDMI must output data. During phase 2 of the previous cycle, the signal **nRW** is driven HIGH to indicate a write cycle. During the actual cycle, **nENOUT** is driven LOW to indicate that the ARM7TDMI is driving **D[31:0]** as an output. Figure 71 shows this bus timing (**DBE** has been tied HIGH in this example). Figure 73 on page 133 shows the circuit which exists in ARM7TDMI for controlling exactly when the external bus is driven out.

Figure 71. Data Write Bus Cycle



The ARM7TDMI macrocell has an additional bus control signal, **nENIN,** which allows the external system to manually tristate the bus. In the simplest systems, **nENIN** can be tied LOW and **nENOUT** can be ignored. However, in many applications when the external data bus is a shared resource, greater control may be required. In this situation, **nENIN** can be used to delay when the external bus is driven. Note that for backwards compatibility, **DBE** is also included. At the macrocell level, **DBE** and **nENIN** have almost identical functionality and in most applications one can be tied off.

The Section Example system: The ARM7TDMI Testchip on page 133 describes how ARM7TDMI may be interfaced to an external data bus, using ARM7TDMI Testchip as an example.

ARM7TDMI has another output control signal called **TBE**. This signal is normally only used during test and must be tied HIGH when not in use. When driven LOW, **TBE** forces all three-stateable outputs to high impedance. It is as if both **DBE** and **ABE** have been driven LOW, causing the data bus, the address bus, and all other signals normally controlled by **ABE** to become high impedance. Note, however, that there is no scan cell on **TBE**. Thus, **TBE** is completely independent of scan data and may be used to put the outputs into a high impedance state while scan testing takes place.

Table 33 below, shows the tri-state control of ARM7TDMI's outputs.

Signals without ✔ in the **ABE**, **DBE** or **TBE** column cannot be driven to the high impedance state:

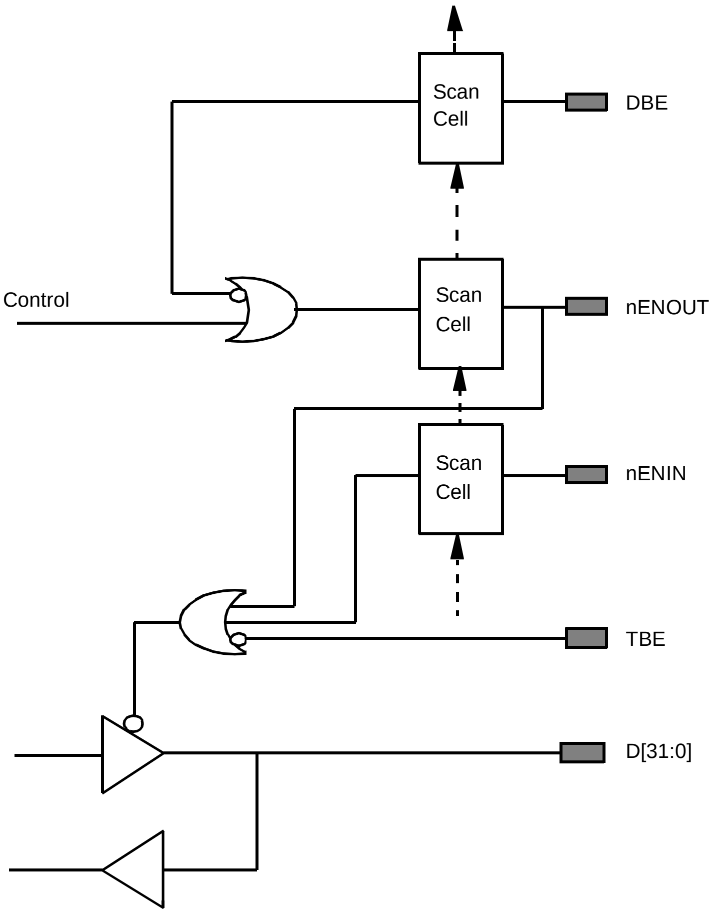
Table 33. Output Enable Control Summary

ARM7TDMI output	ABE	DBE	TBE
A[31:0]	✔		✔
D[31:0]		✔	✔
nRW	✔		✔
LOCK	✔		✔
MAS[1:0]	✔		✔
nOPC	✔		✔
nTRANS	✔		✔
DBGACK			

Table 33. Output Enable Control Summary

ARM7TDMI output	ABE	DBE	TBE
ECLK			
nCPI			
nENOUT			
nEXEC			
nM[4:0]			
TBIT			
nMREQ			
SDOUTMS			
SDOUTDATA			
SEQ			
DOUT[31:0]			

Figure 72. ARM7TDMI Data Bus Control Circuit



Example system: The ARM7TDMI Testchip

Connecting ARM7TDMI's data bus, **D[31:0]** to an external shared bus requires some simple additional logic. This will vary from application to application. As an example, the following describes how the ARM7TDMI macrocell was connected to the bi-directional data bus pads of the ARM7TDMI testchip.

In this application, care must be taken to prevent bus clash on **D[31:0]** when the data bus drive changes direction. The timing of **nENIN**, and the pad control signals must be arranged so that when the core starts to drive out, the pad drive onto **D[31:0]** switches off before the core starts to drive. Similarly, when the bus switches back to input, the core must stop driving before the pad switches on.

All this can be achieved using a simple non-overlapping clock generator. The actual circuit implemented in the ARM7TDMI testchip is shown in Figure 73. Note that at the core level, **TBE** and **DBE** are tied HIGH (inactive). This is because in a packaged part, there is no need to ever manually force the internal buses into a high impedance state. Note also that at the pad level, the signal **EDBE** is factored into the bus control logic. This allows the external memory controller to arbitrate the bus and asynchronously disable ARM7TDMI testchip if required.

Figure 73. The ARM7TDMI Testchip Data Bus Circuit

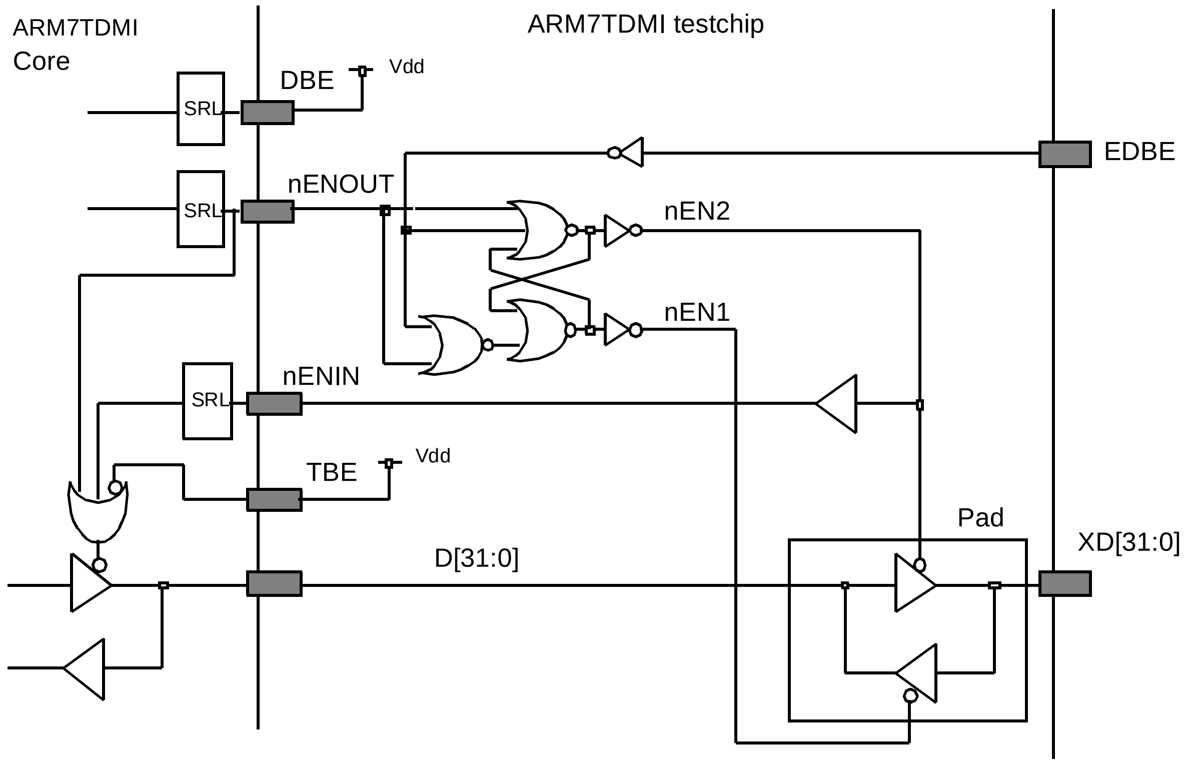
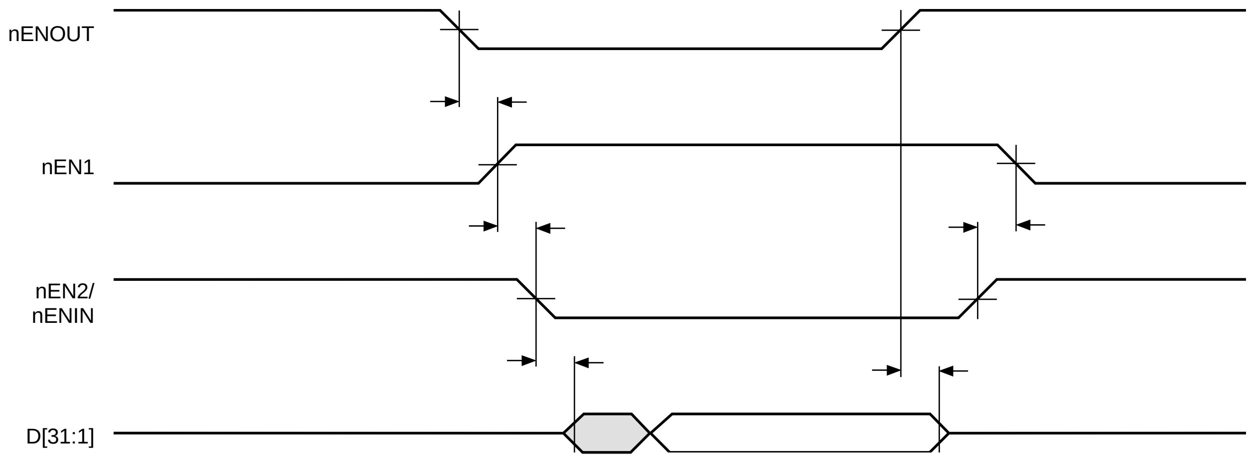


Figure 74 shows how the various control signals interact. Under normal conditions, when the data bus is configured as input, **nENOUT** is HIGH, **nEN1** is LOW, and **nEN2**/**nENIN** is HIGH. Thus the pads drive **XD[31:0]** onto **D[31:0]**.

When a write cycle occurs, **nRW** is driven HIGH to indicate a write during phase 2 of the previous cycle, (ie, with the address). During phase 1 of the actual cycle, **nENOUT** is driven LOW to indicate that ARM7TDMI is about to drive the bus. The falling edge of this signal makes **nEN1** go HIGH, which disables the input half pad from driving **D[31:0]**. This in turn makes **nEN2** go LOW, which enables the output half of the pad so that the ARM7TDMI is now driving the external data bus, **XD[31:0]**. **nEN2** is then buffered and driven back into the core on **nENIN,** so that finally the ARM7TDMI macrocell drives **D[31:0]**. The delay between all the signals ensures that there is no clash on the data bus as it changes direction from input to output.

Figure 74. Data Bus Control Signal Timing



When the bus turns around to the other direction at the end of the cycle, the various control signals switch the other way. Again, the non-overlap ensures that there is never a bus clash. This time, **nENOUT** is driven HIGH to denote that ARM7TDMI no longer needs to drive the bus and the core's output is immediately switched off. This causes **nEN2** to disable the output half of the pad which in turn causes **nEN1** to switch on the input half. Thus, the bus is back to its original input configuration.

Note that the data out time of ARM7TDMI is not directly determined by **nENOUT** and **nENIN**, and so delaying exactly when the bus is driven will not affect the propagation delay. Please refer to Timing Diagrams on page 189 for timing details.

Coprocessor Interface

The functionality of the ARM7TDMI instruction set can be extended by adding external coprocessors. This chapter describes the ARM7TDMI coprocessor interface.

Overview

The functionality of the ARM7TDMI instruction set may be extended by the addition of up to 16 external coprocessors. When the coprocessor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions. Adding the coprocessor will then increase the system performance in a software compatible way. Note that some coprocessor numbers have already been assigned. Contact ARM Ltd for up-to-date information.

Interface Signals

Three dedicated signals control the coprocessor interface, **nCPI**, **CPA** and **CPB**. The **CPA** and **CPB** inputs should be driven HIGH except when they are being used for handshaking.

Coprocessor present/absent

ARM7TDMI takes **nCPI** LOW whenever it starts to execute a coprocessor (or undefined) instruction. (This will not happen if the instruction fails to be executed because of the condition codes.) Each coprocessor will have a copy of the instruction, and can inspect the CP# field to see which coprocessor it is for. Every coprocessor in a system must have a unique number and if that number matches the contents of the CP# field the coprocessor should drive the **CPA** (coprocessor absent) line LOW. If no coprocessor has a number which matches the CP# field, **CPA** and **CPB** will remain HIGH, and ARM7TDMI will take the undefined instruction trap. Otherwise ARM7TDMI observes the **CPA** line going LOW, and waits until the coprocessor is not busy.

Busy-waiting

If **CPA** goes LOW, ARM7TDMI will watch the **CPB** (coprocessor busy) line. Only the coprocessor which is driving **CPA** LOW is allowed to drive **CPB** LOW, and it should do so when it is ready to complete the instruction. ARM7TDMI will busy-wait while **CPB** is HIGH, unless an enabled interrupt occurs, in which case it will break off from the coprocessor handshake to process the interrupt. Normally ARM7TDMI will return from processing the interrupt to retry the coprocessor instruction.

When **CPB** goes LOW, the instruction continues to completion. This will involve data transfers taking place between the coprocessor and either ARM7TDMI or memory, except in the case of coprocessor data operations which complete immediately the coprocessor ceases to be busy.

All three interface signals are sampled by both ARM7TDMI and the coprocessor(s) on the rising edge of **MCLK**. If all three are LOW, the instruction is committed to execution, and if transfers are involved they will start on the next cycle. If **nCPI** has gone HIGH after being LOW, and before the instruction is committed, ARM7TDMI has broken off from the busy-wait state to service an interrupt. The instruction may be restarted later, but other coprocessor instructions may come sooner, and the instruction should be discarded.

Pipeline following

In order to respond correctly when a coprocessor instruction arises, each coprocessor must have a copy of the instruction. All ARM7TDMI instructions are fetched from memory via the main data bus, and coprocessors are connected to this bus, so they can keep copies of all instructions as they go into the ARM7TDMI pipeline. The **nOPC** signal indicates when an instruction fetch is taking place, and **MCLK** gives the timing of the transfer, so these may be used together to load an instruction pipeline within the coprocessor.

Data transfer cycles

Once the coprocessor has gone not-busy in a data transfer instruction, it must supply or accept data at the ARM7TDMI bus rate (defined by **MCLK**). It can deduce the direction of transfer by inspection of the L bit in the instruction, but must only drive the bus when permitted to by **DBE** being HIGH. The coprocessor is responsible for determining the number of words to be transferred; ARM7TDMI will continue to increment the address by one word per transfer until the coprocessor tells it to stop. The termination condition is indicated by the coprocessor driving **CPA** and **CPB** HIGH.

There is no limit in principle to the number of words which one coprocessor data transfer can move, but by convention no coprocessor should allow more than 16 words in one instruction. More than this would worsen the worst case ARM7TDMI interrupt latency, as the instruction is not interruptible once the transfers have commenced. At 16 words, this instruction is comparable with a block transfer of 16 registers, and therefore does not affect the worst case latency.

Register Transfer Cycle

The coprocessor register transfer cycle is the one case when ARM7TDMI requires the data bus without requiring the memory to be active. The memory system is informed that the bus is required by ARM7TDMI taking both **nMREQ** and **SEQ** HIGH. When the bus is free, **DBE** should be taken HIGH to allow ARM7TDMI or the coprocessor to drive the bus, and an **MCLK** cycle times the transfer.

Privileged Instructions

The coprocessor may restrict certain instructions for use in privileged modes only. To do this, the coprocessor will have to track the **nTRANS** output.

As an example of the use of this facility, consider the case of a floating point coprocessor (FPU) in a multi-tasking system. The operating system could save all the floating point registers on every task switch, but this is inefficient in a typical system where only one or two tasks will use floating point operations. Instead, there could be a privileged instruction which turns the FPU on or off. When a task switch happens, the operating system can turn the FPU off without saving its registers. If the new task attempts an FPU operation, the FPU will appear to be absent, causing an undefined instruction trap. The operating system will then realise that the new task requires the FPU, so it will re-enable it and save FPU registers. The task can then use the FPU as normal. If, however, the new task never attempts an FPU operation (as will be the case for most tasks), the state saving overhead will have been avoided.

Idempotency

A consequence of the implementation of the coprocessor interface, with the interruptible busy-wait state, is that all instructions may be interrupted at any point up to the time when the coprocessor goes not-busy. If so interrupted, the instruction will normally be restarted from the beginning after the interrupt has been processed. It is therefore essential that any action taken by the coprocessor before it goes not-busy must be idempotent, ie must be repeatable with identical results.

For example, consider a FIX operation in a floating point coprocessor which returns the integer result to an ARM7TDMI register. The coprocessor must stay busy while it performs the floating point to fixed point conversion, as ARM7TDMI will expect to receive the integer value on the cycle immediately following that where it goes not-busy. The coprocessor must therefore preserve the original floating point value and not corrupt it during the conversion, because it will be required again if an interrupt arises during the busy period.

The coprocessor data operation class of instruction is not generally subject to idempotency considerations, as the processing activity can take place after the coprocessor goes not-busy. There is no need for ARM7TDMI to be held up until the result is generated, because the result is confined to stay within the coprocessor.

Undefined Instructions

Undefined instructions are treated by ARM7TDMI as coprocessor instructions. All coprocessors must be absent (ie **CPA** and **CPB** must be HIGH) when an undefined instruction is presented. ARM7TDMI will then take the undefined instruction trap. Note that the coprocessor need only look at bit 27 of the instruction to differentiate undefined instructions (which all have 0 in bit 27) from coprocessor instructions (which all have 1 in bit 27)

Note that when in THUMB state, coprocessor instructions are not supported but undefined instructions are. Thus, all coprocessors must monitor the state of the **TBIT** output from ARM7TDMI. When ARM7TDMI is in THUMB state, coprocessors must appear absent (ie they must drive **CPA** and **CPB** HIGH) and the instructions seen on the data bus must be ignored. In this way, coprocessors will not erroneously execute THUMB instructions, and all undefined instructions will be handled correctly.

Debug Interface

This chapter describes the ARM7TDMI advanced debug interface.

Overview

The ARM7TDMI debug interface is based on IEEE Std. 1149.1- 1990, “Standard Test Access Port and Boundary-Scan Architecture”. Please refer to this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

ARM7TDMI 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 either on a given instruction fetch (breakpoint) or data access (watchpoint), or asynchronously by a debug-request. When this happens, ARM7TDMI is said to be in *debug state*. At this point, the core’s internal state and the system’s external state may be examined. Once examination is complete, the core and system state may be restored and program execution resumed.

ARM7TDMI 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 *ICEBreaker*. 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.

ARM7TDMI’s internal state is examined via a JTAG-style serial interface, which allows instructions to be serially inserted into the core’s pipeline without using the external data bus. Thus, when in debug state, a store-multiple (STM) could be inserted into the instruction pipeline and this would dump the contents of ARM7TDMI’s registers. This data can be serially shifted out without affecting the rest of the system.

Debug Systems

The ARM7TDMI 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 ARM7TDMI. Such a system typically has three parts:

1. The Debug Host

   This is a computer, for example a PC, running a software debugger such as ARMSD. 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".

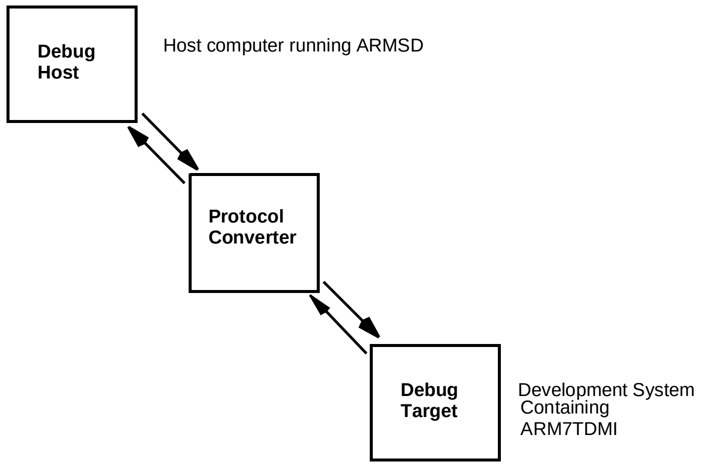
2. The Protocol Converter

   The Debug Host will be connected to the ARM7TDMI development system via an interface (an RS232, for example). The messages broadcast over this connection must be converted to the interface signals of the ARM7TDMI, and this function is performed by the protocol converter.

3. ARM7TDMI

   ARM7TDMI, with hardware extensions to ease debugging, is the lowest level of the system. The debug extensions allow the user to stall the core from program execution, examine its internal state and the state of the memory system, and then resume program execution.

Figure 75. Typical Debug System


Debug Host
Host computer running ARMSD
Protocol Converter
Debug Target
Development System Containing ARM7TDMI

The anatomy of ARM7TDMI is shown in Figure 77. The major blocks are:

ARM7TDMI This is the CPU core, with hardware support for debug.

ICEBreaker This is a set of registers and comparators used to generate debug exceptions (eg breakpoints). This unit is described in ICEBreaker Module on page 163.

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

The Debug Host and the Protocol Converter are system dependent. The rest of this chapter describes the ARM7TDMI's hardware debug extensions.

Debug Interface Signals

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

- **BREAKPT** and **DBGRQ**
  with which the system requests ARM7TDMI to enter debug state.
- **DBGACK**
  which ARM7TDMI uses to flag back to the system that it is in debug state.

Entry into debug state

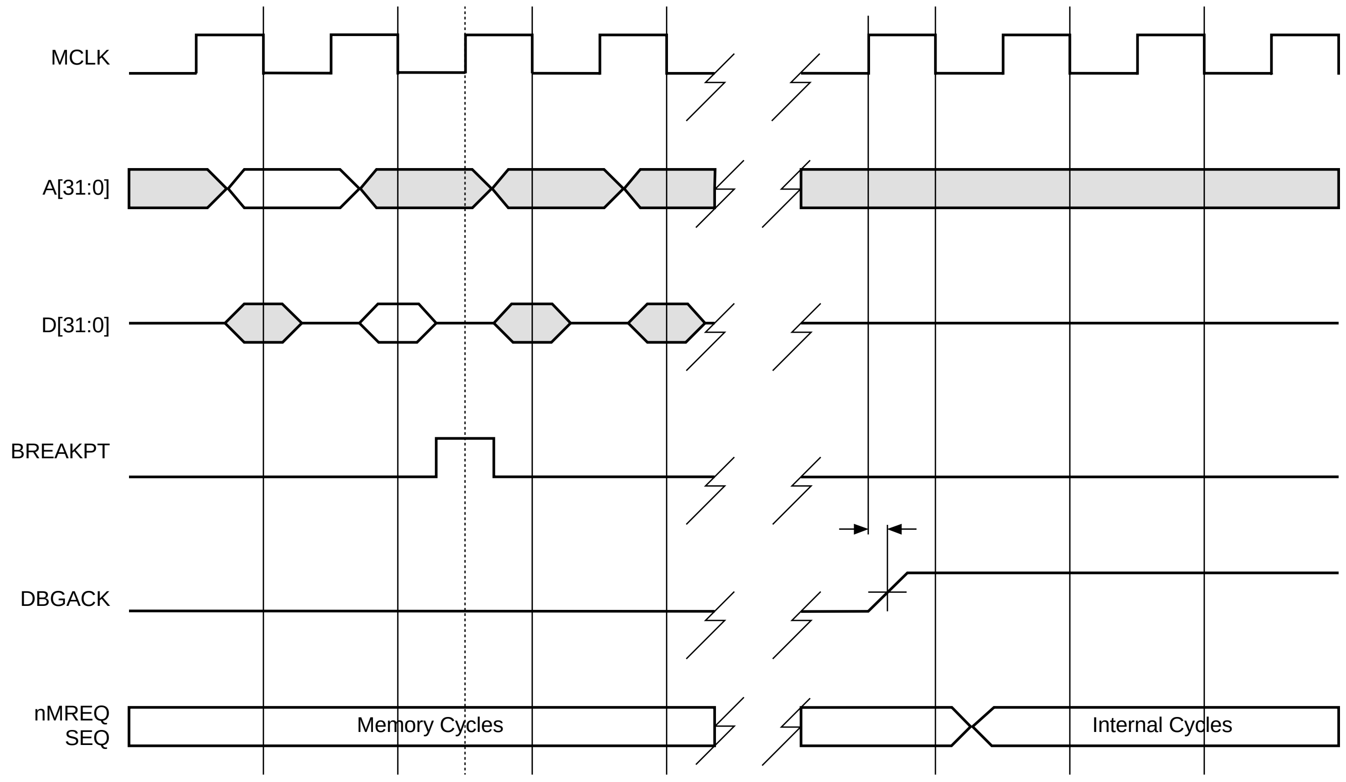
ARM7TDMI is forced into debug state after a breakpoint, watchpoint or debug-request has occurred.

Conditions under which a breakpoint or watchpoint occur can be programmed using ICEBreaker. Alternatively, external logic can monitor the address and data bus, and flag breakpoints and watchpoints via the **BREAKPT** pin.

The timing is the same for externally generated breakpoints and watchpoints. Data must always be valid around the falling edge of **MCLK**. If this data is an instruction to be breakpointed, the **BREAKPT** signal must be HIGH around the next rising edge of **MCLK**. Similarly, if the data is for a load or store, this can be marked as watchpointed by asserting **BREAKPT** around the next rising edge of **MCLK**.

When a breakpoint or watchpoint is generated, there may be a delay before ARM7TDMI enters debug state. When it does, the **DBGACK** signal is asserted in the HIGH phase of **MCLK**. The timing for an externally generated breakpoint is shown in Figure 76.

Figure 76. Debug State Entry



Entry into debug state on breakpoint

After an instruction has been breakpointed, the core does not enter debug state immediately. Instructions are marked as being breakpointed as they enter ARM7TDMI's instruction pipeline.

Thus ARM7TDMI only enters debug state when (and if) the instruction reaches the pipeline's execute stage.

A breakpointed instruction may not cause ARM7TDMI to enter debug state for one of two reasons:

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

When a breakpointed conditional instruction reaches the execute stage of the pipeline, the breakpoint is *always* taken and ARM7TDMI enters debug state, regardless of whether the condition was met.

Breakpointed instructions *do not* get executed: instead, ARM7TDMI enters debug state. Thus, when the internal state is examined, the state *before* the breakpointed instruction is seen. Once examination is complete, the breakpoint should be removed and program execution restarted from the previously breakpointed instruction.

Entry into debug state on watchpoint

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

Watchpoints can be thought of as being similar to data aborts. The difference is however that if a data abort occurs, although the instruction completes, all subsequent changes to ARM7TDMI's state are prevented. This allows the cause of the abort to be cured by the abort handler, and the instruction re-executed. This is not so in the case of a watchpoint. Here, the instruction completes and all changes to the core's state occur (ie load data is written into the destination registers, and base write-back occurs). Thus the instruction does not need to be restarted.

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

Entry into debug state on debug-request

ARM7TDMI may also be forced into debug state on debug request. This can be done either through ICEBreaker programming (see "ICEBreaker Module" on page 163) or be the assertion of the **DBGRQ** pin. This pin is an asynchronous input and is thus synchronised by logic inside ARM7TDMI before it takes effect. Following synchronisation, the core will normally enter debug state at the end of the current instruction. However, if the current instruction is a busy-waiting access to a coprocessor, the instruction terminates and ARM7TDMI enters debug state immediately (this is similar to the action of **nIRQ** and **nFIQ**).

Action of ARM7TDMI in debug state

Once ARM7TDMI is in debug state, **nMREQ** and **SEQ** are forced to indicate internal cycles. This allows the rest of the memory system to ignore ARM7TDMI and function as normal. Since the rest of the system continues operation, ARM7TDMI must be forced to ignore aborts and interrupts.

The **BIGEND** signal should not be changed by the system during debug. If it changes, not only will there be a synchronisation problem, but the programmer's view of ARM7TDMI will change without the debugger's knowledge. **nRESET** must also be held stable during debug. If the system applies reset to ARM7TDMI (ie. **nRESET** is driven LOW) then ARM7TDMI's state will change without the debugger's knowledge.

The **BL[3:0]** signals must remain HIGH while ARM7TDMI is clocked by **DCLK** in debug state to ensure all of the data in the scan cells is correctly latched by the internal logic.

When instructions are executed in debug state, ARM7TDMI outputs (except **nMREQ** and **SEQ**) will change asynchronously to the memory system. For example, every time a new instruction is scanned into the pipeline, the address bus will change. Although this is asynchronous it should not affect the system, since **nMREQ** and **SEQ** are forced to indicate internal cycles regardless of what the rest of ARM7TDMI is doing. The memory controller must be designed to ensure that this asynchronous behaviour does not affect the rest of the system.

Scan Chains and JTAG Interface

There are three JTAG style scan chains inside ARM7TDMI. These allow testing, debugging and ICEBreaker programming. The scan chains are controlled from a JTAG style TAP (Test Access Port) controller. For further details of the JTAG specification, please refer to IEEE Standard 1149.1 - 1990 "Standard Test Access Port and Boundary-Scan Architecture". 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 control signals provided for this scan chain are described later.

Note: The scan cells are not fully JTAG compliant. The following sections describe the limitations on their use.

Scan limitations

The three scan paths are referred to as scan chain 0, 1 and 2: these are shown in Figure 77.

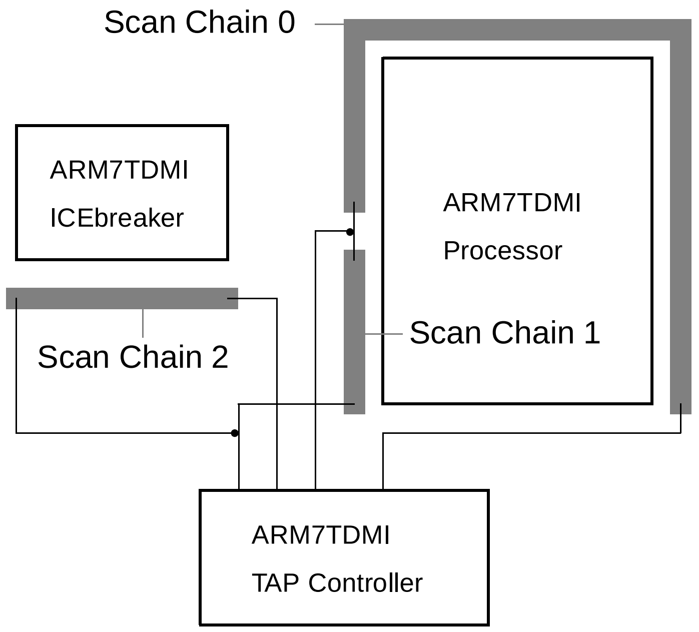
Scan chain 0
Scan chain 0 allows access to the entire periphery of the ARM7TDMI core, including the data bus. The scan chain functions allow inter-device testing (EXTEST) and serial testing of the core (INTEST).

The order of the scan chain (from **SDIN** to **SDOUTMS**) is: data bus bits 0 through 31, the control signals, followed by the address bus bits 31 through 0.

Scan chain 1
Scan chain 1 is a subset of the signals that are accessible through scan chain 0. Access to the core's data bus **D[31:0]**, and the **BREAKPT** signal is available serially. There are 33 bits in this scan chain, the order being (from serial data in to out): data bus bits 0 through 31, followed by **BREAKPT**.

Scan Chain 2
This scan chain simply allows access to the ICEBreaker registers. Refer to ICEBreaker Module on page 163 for details.

Figure 77. ARM7TDMI Scan Chain Arrangement

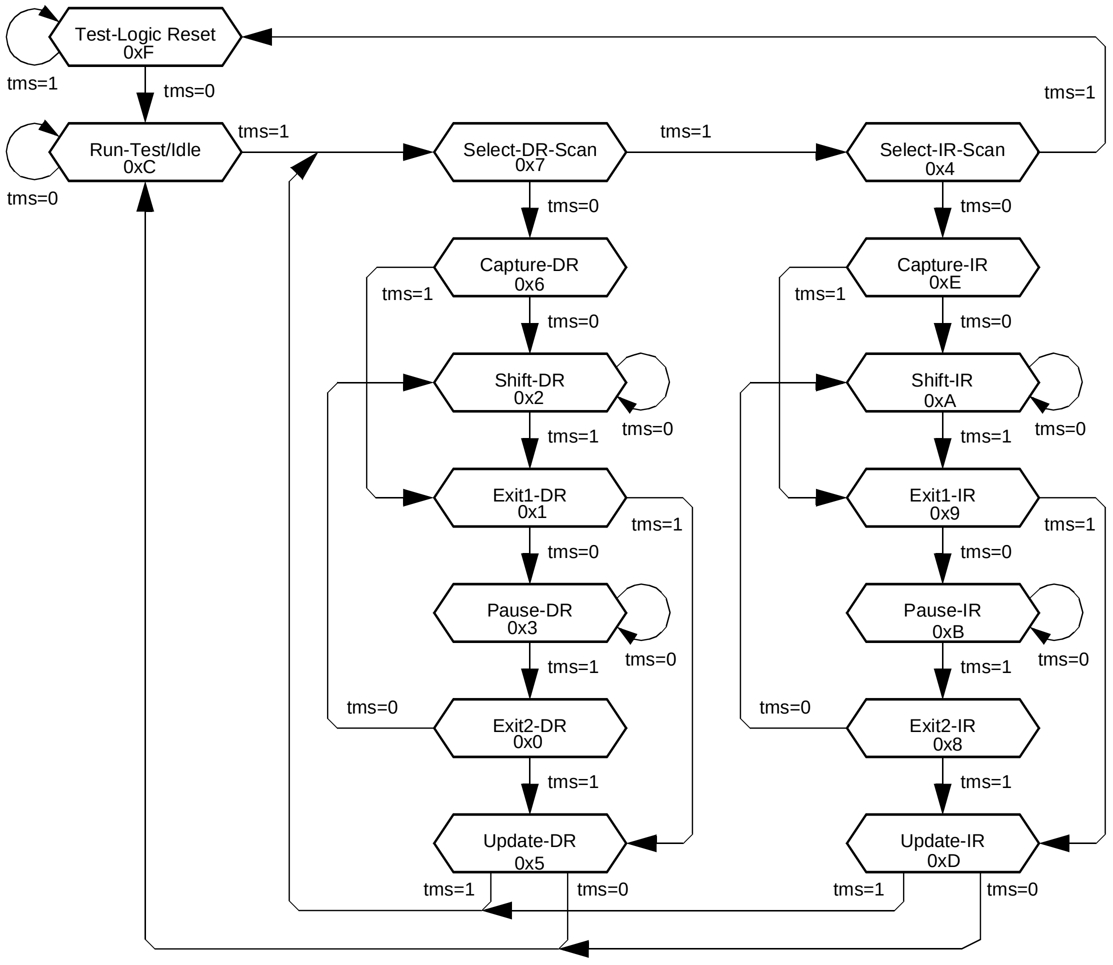


The JTAG state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure 78 shows the state transitions that occur in the TAP controller.

The state numbers are also shown on the diagram. These are output from ARM7TDMI on the **TAPSM[3:0]** bits.

Figure 78. Test Access port (TAP) controller state transitions



Reset

The boundary-scan 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 boundary scan 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 that a clock on **TCK** is not necessary to reset the device.

The action of reset is as follows:

1. System mode is selected (ie 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 will be clocked out of **TDO**.

Pullup Resistors

The IEEE 1149.1 standard effectively requires that **TDI** and **TMS** should have internal pullup resistors. In order to minimise static current draw, these resistors are *not* fitted to ARM7TDMI. Accordingly, the 4 inputs to the test interface (the above 3 signals plus **TCK**) must all be driven to good logic levels to achieve normal circuit operation.

Instruction Register

The instruction register is 4 bits in length.

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

Public Instructions

The following public instructions are supported:

Table 34. Public Instructions

Instruction	Binary Code
EXTEST	0000
SCAN_N	0010
INTEST	1100
IDCODE	1110
BYPASS	1111
CLAMP	0101
HIGHZ	0111
CLAMPZ	1001
SAMPLE/PRELOAD	0011
RESTART	0100

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 1000 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 4 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.

Single-step operation is possible using the INTEST instruction.

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. See ARM7TDMI device identification (ID) code register on page 147 for the details of the ID register format.

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. Note that the first bit shifted out will be a zero. The bypass register is not affected in the UPDATE-DR state. Note that 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. Note that 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, the Address bus, **A[31:0]**, the data bus, **D[31:0]**, plus **nRW**, **nOPC**, **LOCK**, **MAS[1:0]** and **nTRANS** are all driven to the high impedance state and the external **HIGHZ** signal is driven HIGH. This is as if the signal **TBE** had been driven LOW.

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. Note that 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, 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 a 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. Note that the first bit shifted out will be a zero. The bypass register is not affected in the UPDATE-DR state.

SAMPLE/PRELOAD (0011)

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

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 resynchronise back to the memory system once the RUN-TEST/IDLE state is entered.

Test Data Registers

There are 6 test data registers which may be connected between **TDI** and **TDO**. They are: Bypass Register, ID Code Register, Scan Chain Select Register, Scan chain 0, 1 or 2. These are now described in detail.

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 the CAPTURE-DR state.

ARM7TDMI device identification (ID) code register

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

Length: 32 bits. The format of the ID register is as follows:

31 28	27 12	11 1	0
Version	**Part Number**	**Manufacturer Identity**	**1**

Please contact your supplier for the correct Device Identification Code.

Operating mode:

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

There is no parallel output from the ID register.

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

Instruction register

Purpose: Changes the current TAP instruction.

Length: 4 bits

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

During the CAPTURE-IR state, the value 0001 binary is loaded into this register. This is shifted out during SHIFT-IR (lsb first), while a new instruction is shifted in (lsb first). During the UPDATE-IR state, the value in the instruction register becomes the current instruction. On reset, IDCODE becomes the current instruction.

Scan chain select register

Purpose: Changes the current active scan chain.

Length: 4 bits

Operating mode: After SCAN_N has been selected as the current instruction, when in the SHIFT-DR state, the Scan Chain Select Register is selected as the serial path between **TDI** and **TDO**.

During the CAPTURE-DR state, the value 1000 binary is loaded into this register. This is shifted out during SHIFT-DR (lsb first), while a new value is shifted in (lsb first). During the UPDATE-DR state, the value in the register selects a scan chain to become the currently active scan chain. All further instructions such as INTEST then apply to that scan chain.

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

The number of the currently selected scan chain is reflected on the **SCREG[3:0]** outputs. The TAP controller may be used to drive external scan chains in addition to those within the ARM7TDMI macrocell. The external scan chain must be assigned a number and control signals for it can be derived from **SCREG[3:0]**, **IR[3:0]**, **TAPSM[3:0]**, **TCK1** and **TCK2**.

The list of scan chain numbers allocated by ARM are shown in Table 35. An external scan chain may take any other number.The serial data stream to be applied to the external scan chain is made present on **SDINBS**, the serial data back from the scan chain must be presented to the TAP controller on the **SDOUTBS** input. The scan chain present between **SDINBS** and **SDOUTBS** 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[3:0]**

Table 35. Scan Chain Number Allocation

Scan Chain Number	Function
0	Macrocell scan test
1	Debug
2	ICEbreaker programming
3	External boundary scan
4	Reserved
8	Reserved

Scan chains 0,1 and 2

These allow serial access to the core logic, and to ICE-Breaker for programming purposes. They are described in detail below.

Scan chain 0 and 1

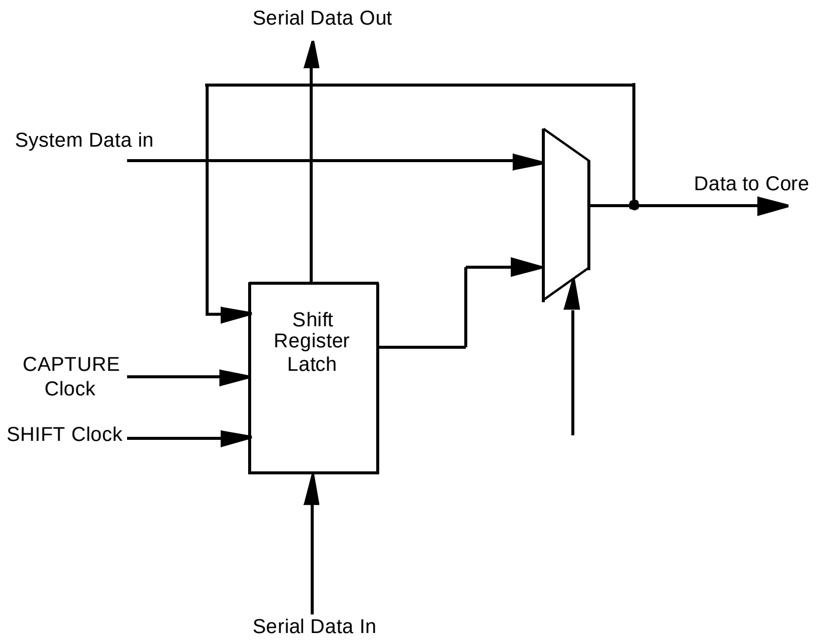
Purpose: Allows access to the processor core for test and debug.

Length: Scan chain 0: 105 bits
Scan chain 1: 33 bits

Each scan chain cell is fairly simple, and consists of a serial register and a multiplexer. The scan cells perform two basic functions, *capture* and *shift*.

For input cells, the capture stage involves copying the value of the system input to the core into the serial register. During shift, this value is output serially. The value applied to the core from an input cell is either the system input or the contents of the serial register, and this is controlled by the multiplexer.

Figure 79. Input Scan Cell



For output cells, capture involves placing the value of a core's output into the serial register. During shift, this value is serially output as before. The value applied to the system from an output cell is either the core output, or the contents of the serial register.

All the control signals for the scan cells are generated internally by the TAP controller. The action of the TAP controller is determined by the current instruction, and the state of the TAP state machine. This is described below.

There are three basic modes of operation of the scan chains, INTEST, EXTEST and SYSTEM, and these are selected by the various TAP controller instructions. In SYSTEM mode, the scan cells are idle. System data is applied to inputs, and core outputs are applied to the system. In INTEST mode, the core is internally tested. The data serially scanned in is applied to the core, and the resulting outputs are captured in the output cells and scanned out. In EXTEST mode, data is scanned onto the core's outputs and applied to the external system. System input data is captured in the input cells and then shifted out.

Note: The scan cells are not fully JTAG compliant in that they do not have an *Update* stage. Therefore, while data is being moved around the scan chain, the contents of the scan cell is not isolated from the output. Thus the output from the scan cell to the core or to the external system could change on every scan clock.

This does not affect ARM7TDMI since its internal state does not change until it is clocked. However, the rest of the system needs to be aware that every output could change asynchronously as data is moved around the scan chain. External logic must ensure that this does not harm the rest of the system.

Scan chain 0

Scan chain 0 is intended primarily for inter-device testing (EXTEST), and testing the core (INTEST). Scan chain 0 is selected via the SCAN_N instruction: see SCAN_N (0010) on page 145.

INTEST allows serial testing of the core. The TAP Controller must be placed in INTEST mode after scan chain 0 has been selected. During CAPTURE-DR, the current outputs from the core'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 1 cycle in RUN-TEST/IDLE. The whole operation may then be repeated.

For details of the core's clocks during test and debug, see ARM7TDMI Core Clocks on page 151.

EXTEST allows inter-device testing, useful for verifying the connections between devices on a circuit board. The TAP Controller must be placed in EXTEST mode after scan chain 0 has been selected. During CAPTURE-DR, the current inputs to the core'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 UPDATE-DR, the value shifted into the data bus **D[31:0]** scan cells appears on the outputs. For all other outputs, the value appears as the data is shifted round. Note, during RUN-TEST/IDLE, the core is not clocked. The operation may then be repeated.

Scan chain 1

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

Note that this scan chain is 33 bits long - 32 bits for the data value, plus the scan cell on the **BREAKPT** core input. This 33rd bit serves four purposes:

1. Under normal INTEST test conditions, it allows a known value to be scanned into the **BREAKPT** input.
2. During EXTEST test conditions, the value applied to the **BREAKPT** input from the system can be captured.
3. While debugging, the value placed in the 33rd bit determines whether ARM7TDMI synchronises back to system speed before executing the instruction. See*System speed access on page 156* for further details.
4. After ARM7TDMI has entered debug state, the first time this bit is captured and scanned out, its value tells the debugger whether the core entered debug state due to a breakpoint (bit 33 LOW), or a watchpoint (bit 33 HIGH).

Scan chain 2

Purpose: Allows ICEBreaker's registers to be accessed. The order of the scan chain, from **TDI** to **TDO** is: read/write, register address bits 4 to 0, followed by data value bits 31 to 0. See <body>Figure 84.

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 ICEBreaker register to be accessed. During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read). Refer to ICEBreaker Module on page 163 for further details.

Scan chain 3

Purpose: Allows ARM7TDMI 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 ARM7TDMI. Typically this would be used for a scan chain around the pad ring of a packaged device. The following control signals are provided which are generated only when scan chain 3 has been selected. These outputs are inactive at all other times.

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

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

SHCLKBS, **SHCLK2BS**
These are non-overlapping clocks generated in the SHIFT-DR state used to clock the master and slave element of the scan cells respectively. When the state machine is not in the SHIFT-DR state, both these clocks are LOW.

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

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

ARM7TDMI Core Clocks

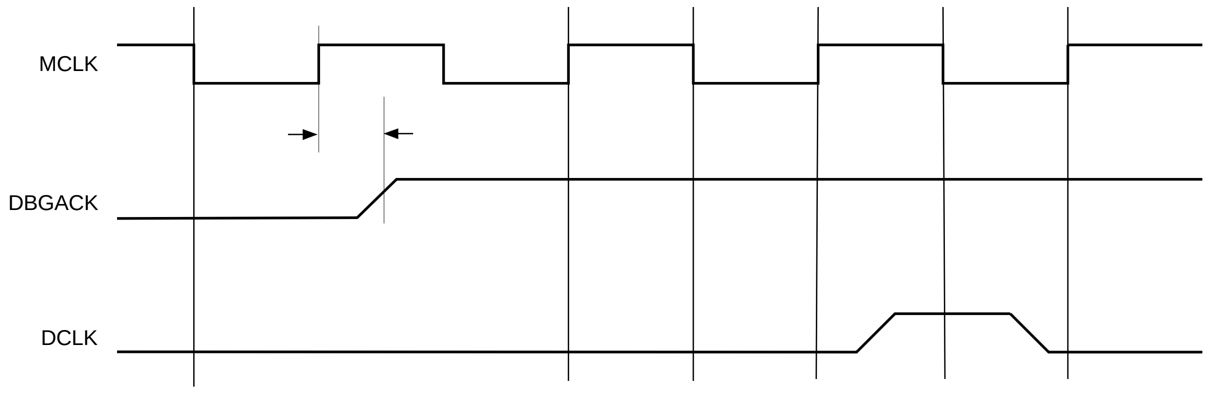
ARM7TDMI has two clocks, the memory clock, **MCLK**, and an internally **TCK** generated clock, **DCLK**. During normal operation, the core is clocked by **MCLK**, and internal logic holds **DCLK** LOW. When ARM7TDMI is in the debug state, the core is clocked by **DCLK** under control of the TAP state machine, and **MCLK** may free run. The selected clock is output on the signal **ECLK** for use by the external system. Note that when the CPU core is being debugged and is running from **DCLK**, **nWAIT** has no effect.

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

Clock switch during debug

When ARM7TDMI enters debug state, it must switch from **MCLK** to **DCLK**. This is handled automatically by logic in the ARM7TDMI. On entry to debug state, ARM7TDMI asserts **DBGACK** in the HIGH phase of **MCLK**. The switch between the two clocks occurs on the next falling edge of **MCLK**. This is shown in Figure 80.

Figure 80. Clock Switching on Entry to Debug State



ARM7TDMI is forced to use **DCLK** as the primary clock until debugging is complete. On exit from debug, the core must be allowed to synchronise back to **MCLK**. This must be done in the following sequence. The final instruction of the debug sequence must be shifted into the data bus scan chain and clocked in by asserting **DCLK**. At this point, BYPASS must be clocked into the TAP instruction register. ARM7TDMI will now automatically resynchronise back to **MCLK** and start fetching instructions from memory at **MCLK** speed. Please refer also to *Exit from debug state on page 153*.

Clock switch during test

When under serial test conditions—ie when test patterns are being applied to the ARM7TDMI core through the JTAG interface—ARM7TDMI must be clocked using **DCLK**. Entry into test is less automatic than debug and some care must be taken.

On the way into test, **MCLK** must be held LOW. The TAP controller can now be used to serially test ARM7TDMI. If scan chain 0 and **INTEST** are selected, **DCLK** is generated while the state machine is in the RUN-TEST/IDLE state. During EXTEST, **DCLK** is not generated.

On exit from test, BYPASS must be selected as the TAP controller instruction. When this is done, **MCLK** 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 BYPASS and then cause a system reset, or to insert MOV PC, #0 into the instruction pipeline before switching back.

Determining the Core and System State

When ARM7TDMI is in debug state, the core and system's state may be examined. This is done by forcing load and store multiples into the instruction pipeline.

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

Determining the core's state

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

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

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

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

From this point on, the processor's 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 bus. These values can then be sampled and shifted out.

Note: The above use of R0 as the base register for the STM is for illustration only, any register could be used.

After determining the values in the current bank of registers, it may be desirable to access the 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 had 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:

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

All these instructions are said to execute at *debug speed*. Debug speed is much slower than system speed since between each core clock, 33 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 ARM7TDMI 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 legally be scanned into the instruction pipeline for execution:

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

Determining system state

In order to meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously to it. Thus, ARM7TDMI must be forced to synchronise back to system speed. This is controlled by the 33rd bit of scan chain 1.

Any instruction may be placed in scan chain 1 with bit 33 (the **BREAKPT** bit) LOW. This instruction will then be executed at debug speed. To execute an instruction at system speed, the instruction prior to it must be scanned into scan chain 1 with bit 33 set HIGH.

After the system speed instruction has been scanned into the data bus and clocked into the pipeline, the BYPASS instruction must be loaded into the TAP controller. This will cause ARM7TDMI to automatically synchronise back to **MCLK** (the system clock), execute the instruction at system speed, and then re-enter debug state and switch itself back to the internally generated **DCLK**. When the instruction has completed, **DBGACK** will be HIGH and the core will have switched back to **DCLK**. At this point, INTEST can be selected in the TAP controller, and debugging can resume.

In order to determine that a system speed instruction has completed, the debugger must look at both **DBGACK** and **nMREQ**. In order to access memory, ARM7TDMI drives **nMREQ** LOW after it has synchronised back to system speed. This transition is used by the memory controller to arbitrate whether ARM7TDMI can have the bus in the next cycle. If the bus is not available, ARM7TDMI may have its clock stalled indefinitely. Therefore, the only way to tell that the memory access has completed, is to examine the state of both **nMREQ** and **DBGACK**. When both are HIGH, the access has completed. Usually, the debugger would be using ICEBreaker to control debugging, and by reading ICEBreaker's status register, the state of **nMREQ** and **DBGACK** can be determined. Refer to ICEBreaker Module on page 163 for more details.

By the use of system speed load multiples and debug speed store multiples, the state of the system's memory can be fed back to the debug host.

There are restrictions on which instructions may have the 33rd bit set. The only valid instructions on which to set this bit are loads, stores, load multiple and store multiple. See also *<Reference><body> Exit from debug state<body>*. When ARM7TDMI returns to debug state after a system speed access, bit 33 of scan chain 1 is set HIGH. This gives the debugger information about why the core entered debug state the first time this scan chain is read.

Exit from debug state

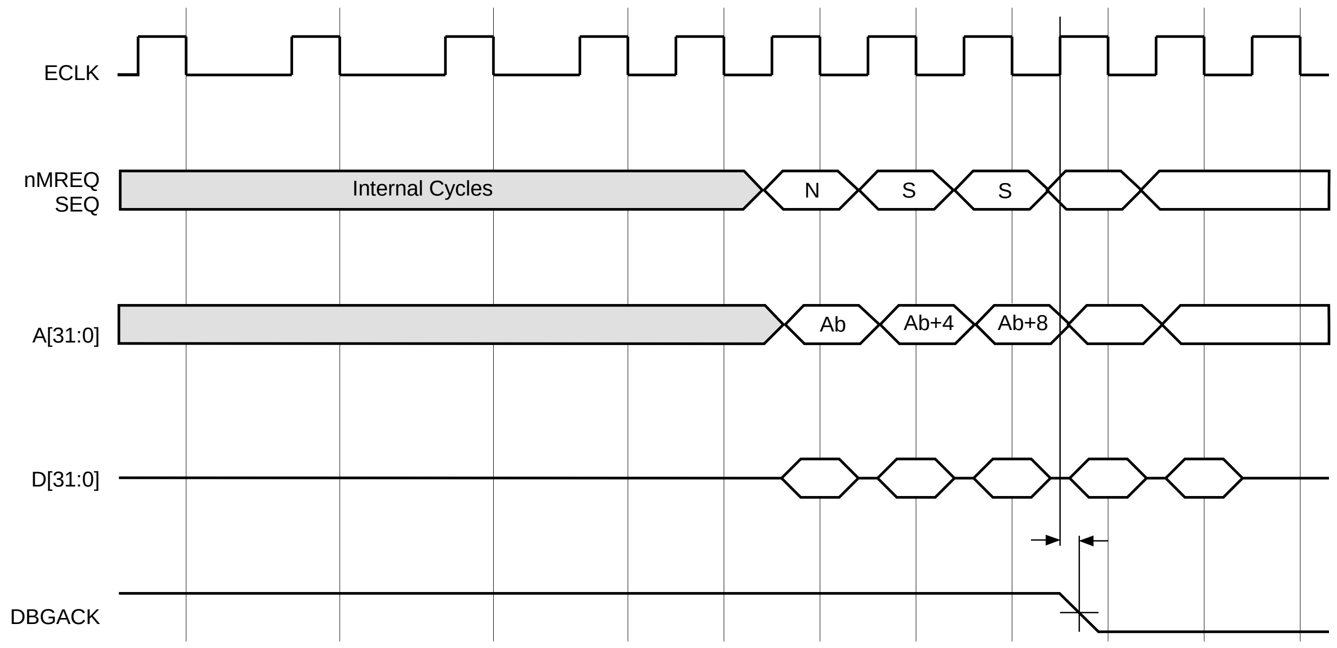
Leaving debug state involves restoring ARM7TDMI's internal state, causing a branch to the next instruction to be executed, and synchronising back to **MCLK**. After restoring internal state, a branch instruction must be loaded into the pipeline. See *The PC's Behaviour During Debug on page 155* for details on calculating the branch.

Bit 33 of scan chain 1 is used to force ARM7TDMI to resynchronise back to **MCLK**. The penultimate instruction of the debug sequence is scanned in with bit 33 set HIGH. The final instruction of the debug sequence is the branch, and this is scanned in with bit 33 LOW. The core is then clocked to load the branch into the pipeline. Now, the RESTART instruction is selected in the TAP controller. When the state machine enters the RUN-TEST/IDLE state, the scan chain will revert back to system mode and clock resynchronisation to **MCLK** will occur within ARM7TDMI. ARM7TDMI will then resume normal operation, fetching instructions from memory. This delay, until the state machine is in the RUN-TEST/IDLE state, allows conditions to be set up in other devices in a multiprocessor system without taking immediate effect. Then, when the RUN-TEST/IDLE state is entered, all the processors resume operation simultaneously.

The function of **DBGACK** is to tell the rest of the system when ARM7TDMI is in debug state. This can be used to inhibit peripherals such as watchdog timers which have real time characteristics. Also, **DBGACK** can be used to mask out memory accesses which are caused by the debugging process. For example, when ARM7TDMI 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. Therefore, on exit from debug state, the pipeline must be refilled to its previous state. Thus, because of the debugging process, more memory accesses occur than would normally be expected. Any system peripheral which may be sensitive to the number of memory accesses can be inhibited through the use of **DBGACK**.

For example, imagine a fictitious peripheral that simply counts the number of memory cycles. This device should return the same answer after a program has been run both with and without debugging. Figure 81 shows the behaviour of ARM7TDMI on exit from the debug state.

Figure 81. Debug Exit Sequence



It can be seen from Figure 76 that the final memory access occurs in the cycle *after* **DBGACK** goes HIGH, and this is the point at which the cycle counter should be disabled. Figure 81 shows that the first memory access that the cycle counter has not seen before occurs in the cycle after **DBGACK** goes LOW, and so this is the point at which the counter should be re-enabled.

Note that when a system speed access from debug state occurs, ARM7TDMI temporarily drops out of debug state, and so **DBGACK** can go LOW. If there are peripherals which are sensitive to the number of memory accesses, they must be led to believe that ARM7TDMI is still in debug state. By programming the ICEBreaker control register, the value on **DBGACK** can be forced to be HIGH. See ICE-Breaker Module on page 163 for more details.

The PC's Behaviour During Debug

In order that ARM7TDMI may be forced 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 five cases: breakpoint, watchpoint, watchpoint when another exception occurs, debug request and system speed access.

Breakpoint

Entry to the debug state from a breakpoint advances the PC by 4 addresses, or 16 bytes. Each instruction executed in debug state advances the PC by 1 address, or 4 bytes. The normal way to exit from debug state after a breakpoint is to remove the breakpoint, and branch back to the previously breakpointed address.

For example, if ARM7TDMI entered debug state from a breakpoint set on a given address and 2 debug speed instructions were executed, a branch of -7 addresses must occur (4 for debug entry, +2 for the instructions, +1 for the final branch). The following sequence shows the data scanned into scan chain 1. This is msb first, and so the first digit is the value placed in the **BREAKPT** bit, followed by the instruction data.

```
0 E0802000; ADD R2, R0, R0
1 E1826001; ORR R6, R2, R1
0 EAFFFFF9; B -7 (2's complement)
```

Note that once in debug state, a minimum of two instructions must be executed before the branch, although these may both be NOPs (MOV R0, R0). For small branches, the final branch could be replaced with a subtract with the PC as the destination (SUB PC, PC, #28 in the above example).

Watchpoints

Returning to program execution after entering debug state from a watchpoint is done in the same way as the procedure described above. Debug entry adds 4 addresses to the PC, and every instruction adds 1 address. The difference is that since the instruction that caused the watchpoint has executed, the program returns to the next instruction.

Watchpoint with another exception

If a watchpointed access simultaneously causes a data abort, ARM7TDMI 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. ARM7TDMI will enter debug state in the exception's mode, and so the debugger must check to see whether this happened. The debugger can deduce whether an exception occurred by looking at the current and previous mode (in the CPSR and SPSR), and the value of the PC. If an exception did take place, the user should be given the choice of whether to service the exception before debugging.

Exiting debug state if an exception occurred is slightly different from the other cases. Here, entry to debug state causes the PC to be incremented by 3 addresses rather than 4, and this must be taken into account in the return branch calculation. For example, suppose that an abort occurred on a watchpointed access and 10 instructions had been executed to determine this. The following sequence could be used to return to program execution.

```
0 E1A00000; MOV R0, R0
1 E1A00000; MOV R0, R0
0 EAFFFFF0; B -16
```

This will force a branch back to the abort vector, causing the instruction at that location to be refetched and executed. Note that after the abort service routine, the instruction which caused the abort and watchpoint will be reexecuted. This will cause the watchpoint to be generated and hence ARM7TDMI will enter debug state again.

Debug request

Entry into debug state via a debug request is similar to a breakpoint. However, unlike a breakpoint, the last instruction will have completed execution and so must not be refetched on exit from debug state. Therefore, it can be thought that entry to debug state adds 3 addresses to the PC, and every instruction executed in debug state adds 1.

For example, suppose that the user has invoked a debug request, and decides to return to program execution straight away. The following sequence could be used:

```
0 E1A00000; MOV R0, R0
1 E1A00000; MOV R0, R0
0 EAFFFFFA; B -6
```

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

System speed access

If a system speed access is performed during debug state, the value of the PC is increased by 3 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, ARM7TDMI enters abort mode before returning to debug state.

This is similar to an aborted watchpoint except that the problem is much harder to fix, because the abort was not caused by an instruction in the main program, and the PC does not point to the instruction which caused the abort. An abort handler usually looks at the PC to determine the instruction which caused the abort, and hence the abort address. In this case, the value of the PC is invalid, but the debugger should know what location was being accessed. Thus the debugger can be written to help the abort handler fix the memory system.

Summary of return address calculations

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

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

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

Priorities / Exceptions

Because the normal program flow is broken when a breakpoint or a debug request occurs, debug can be thought of as being another type of exception. Some of the interaction with other exceptions has been described above. This section summarises the priorities.

Breakpoint with prefetch abort

When a breakpointed instruction fetch causes a prefetch abort, the abort is taken and the breakpoint is disregarded. Normally, prefetch aborts occur when, for example, an access is made to a virtual address which does not physically exist, and the returned data is therefore invalid. In such a case the operating system's normal action will be to swap in the page of memory and return to the previously invalid address. This time, when the instruction is fetched, and providing the breakpoint is activated (it may be data dependent), ARM7TDMI will enter debug state.

Thus the prefetch abort takes higher priority than the breakpoint.

Interrupts

When ARM7TDMI enters debug state, interrupts are automatically disabled. If interrupts are disabled during debug, ARM7TDMI will never be forced into an interrupt mode. Interrupts only have this effect on watchpointed accesses. They are ignored at all times on breakpoints.

If an interrupt was pending during the instruction prior to entering debug state, ARM7TDMI will enter debug state in the mode of the interrupt. Thus, on entry to debug state, the debugger cannot assume that ARM7TDMI will be in the expected mode of the user's program. It must check the PC, the CPSR and the SPSR to fully determine the reason for the exception.

Thus, debug takes higher priority than the interrupt, although ARM7TDMI *remembers* that an interrupt has occurred.

Data aborts

As described above, when a data abort occurs on a watchpointed access, ARM7TDMI enters debug state in abort mode. Thus the watchpoint has higher priority than the abort, although, as in the case of interrupt, ARM7TDMI remembers that the abort happened.

Scan Interface Timing

Figure 82. Scan General Timing

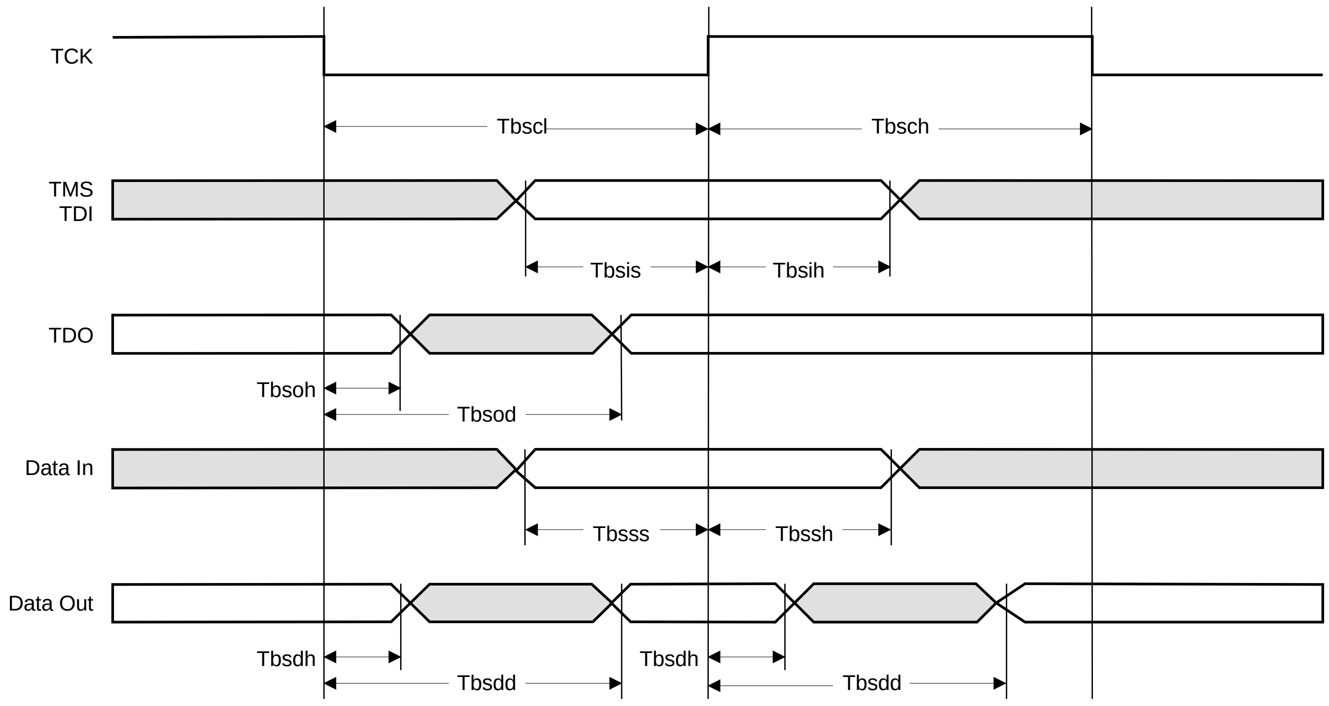


Table 36. ARM7TDMI Scan Interface Timing

Symbol	Parameter	Min	Typ	Max	Notes
Tbscl	**TCK** low period	15.1			
Tbsch	**TCK** high period	15.1			
Tbsis	**TDI**,**TMS** setup to [TCr]	0			
Tbsih	**TDI**,**TMS** hold from [TCr]	0.9			
Tbsoh	**TDO** hold time	2.4			2
Tbsod	TCr to **TDO** valid			16.4	2
Tbsss	I/O signal setup to [TCr]	3.6			1
Tbssh	I/O signal hold from [TCr]	7.6			1
Tbsdh	data output hold time	2.4			2
Tbsdd	TCf to data output valid			17.1	2
Tbsr	Reset period	25			
Tbse	Output Enable time			16.4	2
Tbsz	Output Disable time			14.7	2

Notes:

1. For correct data latching, the I/O signals (from the core and the pads) must be setup and held with respect to the rising edge of **TCK** in the CAPTURE-DR state of the INTEST and EXTEST instructions.
2. Assumes that the data outputs are loaded with the AC test loads (see AC parameter specification).

All delays are provisional and assume a process which achieves 33MHz **MCLK** maximum operating frequency.

In the above table all units are ns.

Table 37. Macrocell Scan Signals and Pins

No	Signal	Type
1	D[0]	I/O
2	D[1]	I/O
3	D[2]	I/O
4	D[3]	I/O
5	D[4]	I/O
6	D[5]	I/O
7	D[6]	I/O
8	D[7]	I/O
9	D[8]	I/O
10	D[9]	I/O
11	D[10]	I/O
12	D[11]	I/O
13	D[12]	I/O
14	D[13]	I/O
15	D[14]	I/O
16	D[15]	I/O
17	D[16]	I/O
18	D[17]	I/O
19	D[18]	I/O
20	D[19]	I/O
21	D[20]	I/O
22	D[21]	I/O
23	D[22]	I/O
24	D[23]	I/O
25	D[24]	I/O
26	D[25]	I/O
27	D[26]	I/O
28	D[27]	I/O
29	D[28]	I/O
30	D[29]	I/O
31	D[30]	I/O
32	D[31]	I/O
33	BREAKPT	I
34	NENIN	I
35	NENOUT	O
36	LOCK	O
37	BIGEND	I
38	DBE	I
39	MAS[0]	O
40	MAS[1]	O
41	BL[0]	I
42	BL[1]	I
43	BL[2]	I
44	BL[3]	I
45	DCTL **	O
46	nRW	O
47	DBGACK	O
48	CGENDBGACK	O
49	nFIQ	I
50	nIRQ	I
51	nRESET	I
52	ISYNC	I
53	DBGRQ	I
54	ABORT	I
55	CPA	I
56	nOPC	O
57	IFEN	I
58	nCPI	O
59	nMREQ	O
60	SEQ	O
61	nTRANS	O
62	CPB	I
63	nM[4]	O
64	nM[3]	O
65	nM[2]	O
66	nM[1]	O
67	nM[0]	O
68	nEXEC	O
69	ALE	I
70	ABE	I
71	APE	I
72	TBIT	O
73	nWAIT	I
74	A[31]	O
75	A[30]	O
76	A[29]	O
77	A[28]	O
78	A[27]	O
79	A[26]	O
80	A[25]	O
81	A[24]	O
82	A[23]	O
83	A[22]	O
84	A[21]	O
85	A[20]	O
86	A[19]	O
87	A[18]	O
88	A[17]	O
89	A[16]	O
90	A[15]	O
91	A[14]	O
92	A[13]	O
93	A[12]	O
94	A[11]	O

Table 37. Macrocell Scan Signals and Pins

No	Signal	Type
95	A[10]	O
96	A[9]	O
97	A[8]	O
98	A[7]	O
99	A[6]	O
100	A[5]	O
101	A[4]	O
102	A[3]	O
103	A[2]	O
104	A[1]	O
105	A[0]	O

KeyI - Input

O - Output

I/O - Input/Output

Note: DCTL is not described in this datasheet. **DCTL** is an output from the processor used to control the unidirectional data out latch, **DOUT[31:0]**. This signal is not visible from the periphery of ARM7TDMI.

Debug Timing

Table 38. ARM7TDMI debug interface timing

Symbol	Parameter	Min	Max
Ttdbgd	TCK falling to DBGACK, DBGRQI changing		13.3
Ttpfd	TCKf to TAP outputs		10.0
Ttpfh	TAP outputs hold time from TCKf	2.4	
Ttprd	TCKr to TAP outputs		8.0
Ttprh	TAP outputs hold time from TCKr	2.4	
Ttckr	TCK to TCK1, TCK2 rising		7.8
Ttckf	TCK to TCK1, TCK2 falling		6.1
Tecapd	TCK to ECAPCLK changing		8.2
Tdckf	DCLK induced: TCKf to various outputs valid		23.8
Tdckfh	DCLK induced: Various outputs hold from TCKf	6.0	
Tdckr	DCLK induced: TCKr to various outputs valid		26.6
Tdckrh	DCLK induced: Various outputs hold from TCKr	6.0	
Ttrstd	nTRSTf to TAP outputs valid		8.5
Ttrsts	nTRSTr setup to TCKr	2.3	
Tsdtd	SDOUTBS to TDO valid		10.0
Tclkbs	TCK to Boundary Scan Clocks		8.2
Tshbsr	TCK to SHCLKBS, SHCLK2BS rising		5.7
Tshbsf	TCK to SHCLKBS, SHCLK2BS falling		4.0

Notes:

- All delays are provisional and assume a process which achieves 33MHz **MCLK** maximum operating frequency.
- Assumes that the data outputs are loaded with the AC test loads (see AC parameter specification).
- All units are ns.

ICEBreaker Module

This chapter describes the ARM7TDMI ICEBreaker module.

Note: The name ICEbreaker has changed. It is now known as the EmbeddedICE macrocell. Future versions of the datasheet will reflect this change.

Overview

The ARM7TDMI-ICEBreaker module, hereafter referred to simply as *ICEBreaker*, provides integrated on-chip debug support for the ARM7TDMI core.

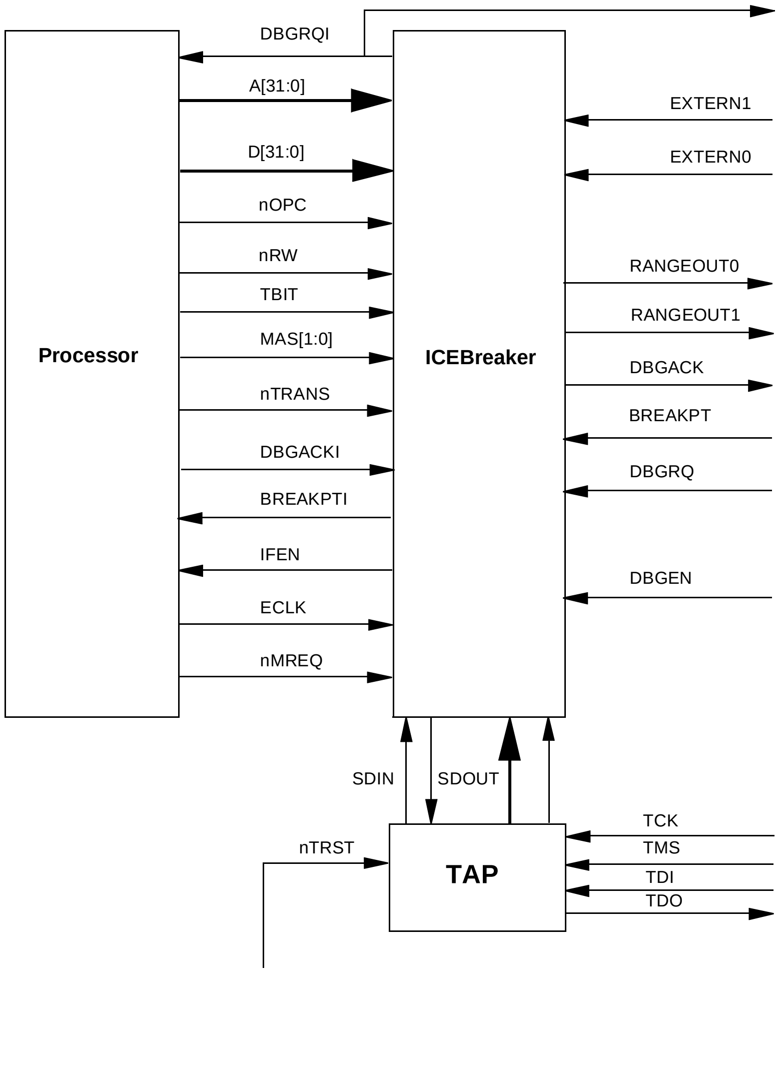
ICEBreaker is programmed in a serial fashion using the ARM7TDMI TAP controller. It consists of two real-time watchpoint units, together with a control and status register. One or both of the watchpoint units can be programmed to halt the execution of instructions by the ARM7TDMI core via its **BREAKPT** signal. Execution is halted when a match occurs between the values programmed into ICEBreaker and the values currently appearing on the address bus, data bus and various control signals. Any bit can be masked so that its value does not affect the comparison.

Figure 83 shows the relationship between the core, ICEBreaker and the TAP controller. Either watchpoint unit can be configured to be a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). Watchpoints and breakpoints can be made to be data-dependent.

Two independent registers, Debug Control and Debug Status, provide overall control of ICEBreaker's operation.

Note: Only those signals that are pertinent to ICEBreaker are shown.

Figure 83. ARM7TDMI Block Diagram



The Watchpoint Registers

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

1. Address Value and Address Mask
2. Data Value and Data Mask
3. Control Value and Control Mask

Each register is independently programmable, and has its own address: see Table 39.

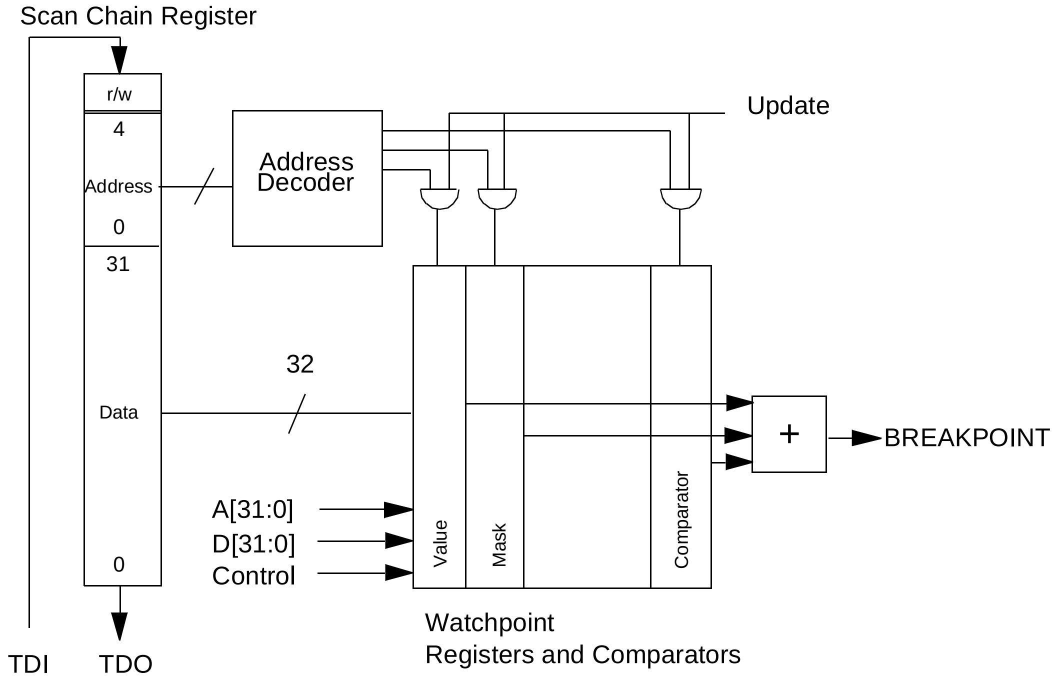
Table 39. Function and Mapping of ICEBreaker Registers

Address	**Width**	**Function**
00000	3	Debug Control
00001	5	Debug Status
00100	6	Debug Comms Control Register
00101	32	Debug Comms Data Register
01000	32	Watchpoint 0 Address Value
01001	32	Watchpoint 0 Address Mask
01010	32	Watchpoint 0 Data Value
01011	32	Watchpoint 0 Data Mask
01100	9	Watchpoint 0 Control Value
01101	8	Watchpoint 0 Control Mask
10000	32	Watchpoint 1Address Value
10001	32	Watchpoint 1 Address Mask
10010	32	Watchpoint 1 Data Value
10011	32	Watchpoint 1 Data Mask
10100	9	Watchpoint 1 Control Value
10101	8	Watchpoint 1 Control Mask

Programming and reading watchpoint registers

A register is programmed by scanning data into the ICE-Breaker scan chain (scan chain 2). The scan chain consists of a 38-bit shift register comprising a 32-bit data field, a 5-bit address field and a read/write bit. This is shown in Figure 84.

Figure 84. ICEBreaker Block Diagram



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

A register is read by scanning its address into the address field and a 0 into the read/write bit. The 32-bit data field is ignored.

The register addresses are shown in Table 39.

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

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

Using the mask registers

For each Value register in a register pair, there is a Mask register of the same format. Setting a bit to 1 in the Mask register has the effect of making the corresponding bit in the Value register disregarded in the comparison.

For example, if a watchpoint is required 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 field ignored.

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

The control registers

The Control Value and Control Mask registers are mapped identically in the lower eight bits, as shown below. Bit 8 of the control value register is the **ENABLE** bit, which cannot be masked.

Figure 85. Watchpoint Control Value and Mask Format

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

The bits have the following functions:

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

MAS[1:0] compares against the **MAS[1:0]** signal from the core in order to detect the size of bus activity.

The encoding is shown in the following table.

Table 40. MAS [1:0] Signal Encoding

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

nOPC is used to detect whether the current cycle is an instruction fetch (**nOPC** = 0) or a data access (**nOPC** = 1).

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

EXTERN is an external input to ICEBreaker which 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 the chain output of another watchpoint in order to implement, for example, debugger requests of the form “breakpoint on address YYY only when in process XXX”.

In the ARM7TDMI-ICEBreaker, 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 ARM7TDMI-ICEBreaker, the **RANGEOUT** output of Watchpoint 1 is connected to the **RANGE** input of Watchpoint 0. This allows the two watchpoints to be coupled for detecting conditions that occur simultaneously, eg for range-checking.

ENABLE If a watchpoint match occurs, the **BREAKPT** signal will only be asserted when the **ENABLE** bit is set. This bit only exists in the value register: it cannot be masked.

For each of the bits 8:0 in the Control Value register, there is a corresponding bit in the Control Mask register. This removes the dependency on particular signals.

Programming Breakpoints

Breakpoints can be classified as hardware breakpoints or software breakpoints.

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

Software breakpoints:
Monitor a particular bit pattern being fetched from any address. One ICEBreaker watchpoint can thus be used to support any number of software breakpoints. Software breakpoints can normally only be set in RAM because an instruction has to be replaced by the special bit pattern chosen to cause a software breakpoint.

Hardware breakpoints

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

1. Program its Address Value register with the address of the instruction to be breakpointed.
2. For a breakpoint in ARM state, program bits [1:0] of the Address Mask register to 1. For a breakpoint in THUMB state, program bit 0 of the Address Mask to 1. In both cases the remaining bits are set to 0.
3. Program the Data Value register only if you require a data-dependent breakpoint: ie only if the actual instruction code fetched must be matched as well as the address. If the data value is not required, program the Data Mask register to 0xFFFFFFFF (all bits to1), otherwise program it to0x00000000.
4. Program the Control Value register with **nOPC** = 0.
5. Program the Control Mask register with **nOPC** =0, all other bits to 1.
6. If you need to make the distinction between user and non-user mode instruction fetches, program the **nTRANS** Value and Mask bits as above.
7. If required, program the **EXTERN**, **RANGE** and **CHAIN** bits in the same way.

Software breakpoints

To make a watchpoint unit cause software breakpoints (ie on instruction fetches of a particular bit pattern):

1. Program its Address Mask register to 0xFFFFFFFF (all bits set to 1) so that the address is disregarded.
2. Program the Data Value register with the particular bit pattern that has been chosen to represent a software breakpoint.

   If a THUMB software breakpoint is being programmed, the 16-bit pattern must be repeated in both halves of the Data Value register. For example, if the bit pattern is 0xDFFF, then 0xDFFFDFFF must be programmed. When a 16-bit instruction is fetched, ICEbreaker only compares the valid half of the data bus against the contents of the Data Value register. In this way, a single Watchpoint register can be used to catch software breakpoints on both the upper and lower halves of the data bus.
3. Program the Data Mask register to 0x00000000.
4. Program the Control Value register with **nOPC** = 0.
5. Program the Control Mask register with **nOPC** = 0, all other bits to 1.
6. If you wish to make the distinction between user and non-user mode instruction fetches, program the **nTRANS** bit in the Control Value and Control Mask registers accordingly.
7. If required, program the **EXTERN**, **RANGE** and **CHAIN** bits in the same way.

Note: The address value register need not be programmed.

Setting the breakpoint
To set the software breakpoint:

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

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

Programming Watchpoints

To make a watchpoint unit cause watchpoints (ie on data accesses):

1. Program its Address Value register with the address of the data access to be watchpointed.
2. Program the Address Mask register to 0x00000000.
3. Program the Data Value register only if you require a data-dependent watchpoint; i.e. only if the actual data value read or written must be matched as well as the address. If the data value is irrelevant, program the Data Mask register to 0xFFFFFFFF (all bits set to 1) otherwise program it to 0x00000000.
4. Program the Control Value register with **nOPC** = 1, **nRW** = 0 for a read or **nRW** = 1 for a write, **MAS[1:0]** with the value corresponding to the appropriate data size.
5. Program the Control Mask register with **nOPC** = 0, **nRW** = 0, **MAS[1:0]** = 0, all other bits to 1. Note that **nRW** or **MAS[1:0]** may be set to 1 if both reads and writes or data size accesses are to be watchpointed respectively.
6. If you wish to make the distinction between user and non-user mode data accesses, program the **nTRANS** bit in the Control Value and Control Mask registers accordingly.
7. If required, program the **EXTERN**, **RANGE** and **CHAIN** bits in the same way.

Note: The above are just examples of how to program the watchpoint register to generate breakpoints and watchpoints; many other ways of programming the registers are possible. For instance, simple range breakpoints can be provided by setting one or more of the address mask bits.

The Debug Control Register

The Debug Control Register is 3 bits wide. If the register is accessed for a write (with the read/write bit HIGH), the control bits are written. If the register is accessed for a read (with the read/write bit LOW), the control bits are read.

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

Figure 0-1. Debug Control Register Format

2	1	0
INTDIS	DBGRQ	DBGACK

Bits 1 and 0 allow the values on **DBGRQ** and **DBGACK** to be forced.

As shown in Figure 87, the value stored in bit 1 of the control register is synchronised and then ORed with the external **DBGRQ** before being applied to the processor. The output of this OR gate is the signal **DBGRQI** which is brought out externally from the macrocell.

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

In the case of **DBGACK**, the value of **DBGACK** from the core is ORed with the value held in bit 0 to generate the external value of **DBGACK** seen at the periphery of ARM7TDMI. This allows the debug system to signal to the rest of the system that the core is still being debugged even when system-speed accesses are being performed (in which case the internal **DBGACK** signal from the core will be LOW).

If Bit 2 (**INTDIS**) is asserted, the interrupt enable signal (**IFEN**) of the core is forced LOW. Thus all interrupts (IRQ and FIQ) are disabled during debugging (**DBGACK** =1) or if the **INTDIS** bit is asserted. The **IFEN** signal is driven according to the following table:

Table 41. IFEN Signal Control

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

Debug Status Register

The Debug Status Register is 5 bits wide. If it is accessed for a write (with the read/write bit set HIGH), the status bits are written. If it is accessed for a read (with the read/write bit LOW), the status bits are read.

Figure 86. Debug Status Register Format

4	3	2	1	0
TBIT	nMREQ	IFEN	DBGRQ	DBGACK

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

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

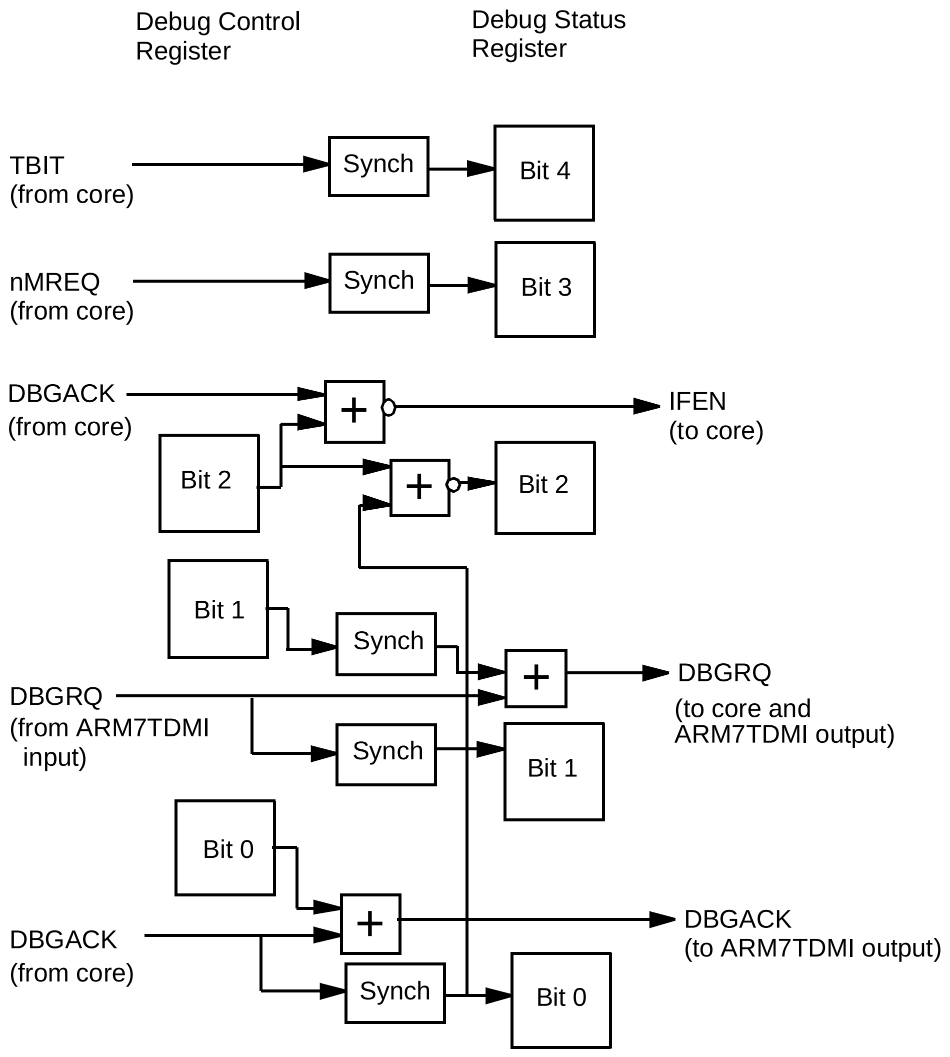
Bit 2 allows the state of the core interrupt enable signal (**IFEN**) to be read. Since the capture clock for the scan chain may be asynchronous to the processor clock, the **DBGACK** output from the core is synchronised before being used to generate the **IFEN** status bit.

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

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

The structure of the debug status register bits is shown in Figure 87.

Figure 87. Structure of TBIT, NMREQ, DBGACK, DBGRQ and INTDIS Bits



Coupling Breakpoints and Watchpoints

Watchpoint units 1 and 0 can be coupled together via the **CHAIN** and **RANGE** inputs. The use of **CHAIN** enables watchpoint 0 to be triggered only if watchpoint 1 has previously matched. The use of **RANGE** enables simple range checking to be performed by combining the outputs of both watchpoints.

Example

Let

A_v[31:0] be the value in the Address Value Register

A_m[31:0] be the value in the Address Mask Register

A[31:0] be the Address Bus from the ARM7TDMI

D_v[31:0] be the value in the Data Value Register

D_m[31:0] be the value in the Data Mask Register

D[31:0] be the Data Bus from the ARM7TDMI

C_v[8:0] be the value in the Control Value Register

C_m[7:0] be the value in the Control Mask Register

C[9:0] be the combined Control Bus from the ARM7TDMI, other watchpoint registers and the EXTERN signal.

CHAINOUT signal

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

WHEN (({A_v[31:0],C_v[4:0]} XNOR {A[31:0],C[4:0]}) OR {A_m[31:0],C_m[4:0]} == 0xFFFFFFFFF)

CHAINOUT = ((({D_v[31:0],C_v[6:4]} XNOR {D[31:0],C[7:5]}) OR {D_m[31:0],C_m[7:5]}) == 0x7FFFFFFFF)

The **CHAINOUT** output of watchpoint register 1 provides the **CHAIN** input to Watchpoint 0. This allows for quite complicated configurations of breakpoints and watchpoints.

Take for example the request by a debugger to breakpoint on the instruction at location YYY when running process XXX in a multiprocess system.

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

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

RANGEOUT signal

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

RANGEOUT = ((({A_v[31:0],C_v[4:0]} XNOR {A[31:0],C[4:0]}) OR {A_m[31:0],C_m[4:0]}) == 0xFFFFFFFFF) AND ((({D_v[31:0],C_v[7:5]} XNOR {D[31:0],C[7:5]}) OR {D_m[31:0],C_m[7:5]}) == 0x7FFFFFFFF)

The **RANGEOUT** output of watchpoint register 1 provides the **RANGE** input to watchpoint register 0. This allows two breakpoints to be coupled together to form range breakpoints. Note that selectable ranges are restricted to being powers of 2. This is best illustrated by an example.

Example

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

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

If Watchpoint 0 matches but Watchpoint 1 does not (ie the **RANGE** input to Watchpoint 0 is 0), the breakpoint will be

Disabling ICEBreaker

ICEBreaker may be disabled by wiring the **DBGEN** input LOW.

When **DBGEN** is LOW, **BREAKPT** and **DBGRQ** to the core are forced LOW, **DBGACK** from the ARM7TDMI is also forced LOW and the **IFEN** input to the core is forced HIGH, enabling interrupts to be detected by ARM7TDMI.

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

ICEBreaker Timing

The **EXTERN1** and **EXTERN0** inputs are sampled by ICE-Breaker on the falling edge of **ECLK**. Sufficient set-up and hold time must therefore be allowed for these signals.

Programming Restriction

The ICEBreaker watchpoint units should only be programmed when the clock to the core is stopped. This can be achieved by putting the core into the debug state.

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

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

Note: This restriction does not apply in any event to the Debug Control or Status Registers.

Debug Communications Channel

ARM7TDMI's ICEbreaker 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 synchronised handshaking between the processor and the asynchronous debugger. These registers live in fixed locations in ICEbreaker's memory map (as shown in Table 39) and are accessed from the processor via MCR and MRC instructions to coprocessor 14.

Debug comms channel registers

The Debug Comms Control register is read only and allows synchronised hanshaking between the processor and the debugger

Figure 88. Debug Comms Control Register

31	30	29	28	...	1	0
0	0	0	1	...	W	R

The function of each register bit is described below:

Bits 31:28 contain a fixed pattern which denote the ICE-breaker version number, in this case 0001.

Bit 1 denotes whether the Comms Data Write register (from the processor's point of view) is free. From the processor's point of view, if the Comms Data Write register is free (W=0) then new data may be written. If it is not free (W=1), then the processor must poll until W=0. From the debugger's point of view, if W=1 then 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. From the processor's point of view, if R=1, then there is some new data which may be read via an MRC instruction. From the debugger's point of view, if R=0 then the Comms Data Read register is free and new data may be placed there through the scan chain. If R=1, then 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 CP14, 0, Rd, C0, C0
```

Returns the Debug Comms Control register into Rd

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

Writes the value in Rn to the Comms Data Write register

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

Returns the Debug Data Read register into Rd

Since the THUMB instruction set does not contain coprocessor instructions, it is recommended that these are accessed via SWI instructions when in THUMB state.

Communications via the comms channel

Communication between the debugger and the processor occurs as follows. When the processor wishes to send a message to ICEbreaker, it first checks that the Comms Data Write register is free for use. This is done by reading the Debug Comms Control register to check that the W bit is clear. If it is clear then the Comms Data Write register is empty and a message is written by a register transfer to the coprocessor. The action of this data transfer automatically sets the W bit. If on reading the W bit it is found to be set, then this implys that previously written data has not been picked up by the debugger and thus the processor must poll until the W bit is clear.

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. When the debugger polls this register it sees a synchronised version of both the R and W bit. When the debugger sees that the W bit is set it can read the Comms Data Write register and scan the data out. The action of reading this data register clears the W bit of the Debug Comms Control register. At this point, the communications process may begin again.

Message transfer from the debugger to the processor is carried out in a similar fashion. Here, the debugger polls the R bit of the Debug Comms Control register. If the R bit is low then the Data Read register is free and so data can be placed there for the processor to read. If the R bit is set, then previously deposited data has not yet been collected and so the debugger must wait.

When the Comms Data Read register is free, data is written there via the scan chain. 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 synchronised version. If the R bit is set then this denotes that there is data waiting to be collected, and this can be read via a CPRT load. 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, this denotes that the data has been taken and the process may now be repeated.

Instruction Cycle Operations

This chapter describes the ARM7TDMI instruction cycle operations.

Introduction

In the following tables **nMREQ** and **SEQ** (which are pipelined up to one cycle ahead of the cycle to which they apply) are shown in the cycle in which they appear, so they predict the type of the *next* cycle. The address, **MAS[1:0]**, **nRW**, **nOPC, nTRANS** and **TBIT** (which appear up to half a cycle ahead) are shown in the cycle to which they apply. The address is incremented for prefetching of instructions in most cases. Since the instruction width is 4 bytes in ARM state and 2 bytes in THUMB state, the increment will vary accordingly. Hence the letter L is used to indicate instruction length (4 bytes in ARM state and 2 bytes in THUMB state). Similarly, **MAS[1:0]** will indicate the width of the instruction fetch, i=2 in ARM state and i=1 in THUMB state representing word and halfword accesses respectively.

Branch and Branch with Link

A branch instruction calculates the branch destination in the first cycle, whilst performing a prefetch from the current PC. This prefetch is done in all cases, since by the time the decision to take the branch has been reached it is already too late to prevent the prefetch.

During the second cycle a fetch is performed from the branch destination, and the return address is stored in register 14 if the link bit is set.

The third cycle performs a fetch from the destination + L, refilling the instruction pipeline, and if the branch is with link R14 is modified (4 is subtracted from it) to simplify return from `SUB PC,R14,#4 to MOV PC,R14`. This makes the `STM..{R14} LDM..{PC}` type of subroutine work correctly. The cycle timings are shown below in Table 42.

Table 42. Branch Instruction Cycle Operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1	pc+2L	i	0	(pc + 2L)	0	0	0
2	alu	i	0	(alu)	0	1	0
3	alu+L	i	0	(alu + L)	0	1	0
	alu+2L						

pc	is the address of the branch instruction
alu	is an address calculated by ARM7TDMI

(alu) are the contents of that address

Note: This applies to branches in ARM and THUMB state, and to Branch with Link in ARM state only.

THUMB Branch with Link

A THUMB Branch with Link operation consists of two consecutive THUMB instructions, see Format 19: long branch with link .

The first instruction acts like a simple data operation, taking a single cycle to add the PC to the upper part of the offset, storing the result in Register 14 (LR).

The second instruction acts in a similar fashion to the ARM Branch with Link instruction, thus its first cycle calculates the final branch destination whilst performing a prefetch from the current PC.

The second cycle of the second instruction performs a fetch from the branch destination and the return address is stored in R14.

The third cycle of the second instruction performs a fetch from the destination +2, refilling the instruction pipeline and R14 is modified (2 subtracted from it) to simplify the return to `MOV PC, R14`. This makes the `PUSH {..,LR} ; POP {..,PC}` type of subroutine work correctly.

The cycle timings of the complete operation are shown in Table 43.

Table 43. THUMB Long Branch with Link

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1	pc + 4	1	0	(pc + 4)	0	1	0
2	pc + 6	1	0	(pc + 6)	0	0	0
3	alu	1	0	(alu)	0	1	0
4	alu + 2	1	0	(alu + 2)	0	1	0
	alu + 4						

pcis the address of the first instruction of the operation.

Branch and Exchange (BX)

A Branch and Exchange operation takes 3 cycles and is similar to a Branch.

In the first cycle, the branch destination and the new core state are extracted from the register source, whilst performing a prefetch from the current PC. This prefetch is performed in all cases, since by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch.

During the second cycle, a fetch is performed from the branch destination using the new instruction width, dependent on the state that has been selected.

The third cycle performs a fetch from the destination +2 or +4 dependent on the new specified state, refilling the instruction pipeline. The cycle timings are shown in Table 44.

Table 44. Branch and Exchange Instruction Cycle Operations

Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	noPC	TBIT
1	pc + 2W	I	0	(pc + 2W)	0	0	0	T
2	alu	i	0	(alu)	0	1	0	t
3	alu+w	i	0	(alu+w)	0	1	0	t
	alu + 2w							

Notes:

1. W and w represent the instruction width before and after the BX respectively. In ARM state the width equals 4 bytes and in THUMB state the width equals 2 bytes. For example, when changing from ARM to THUMB state, W would equal 4 and w would equal 2.
2. I and i represent the memory access size before and after the BX respectively. In ARM state, the MAS[1:0] is 2 and in THUMB state MAS[1:0] is 1. When changing from THUMB to ARM state, I would equal 1 and i would equal 2.
3. T and t represent the state of the TBIT before and after the BX respectively. In ARM state TBIT is 0 and in THUMB state TBIT is 1. When changing from ARM to THUMB state, T would equal 0 and t would equal 1.

Data Operations

A data operation executes in a single datapath cycle except where the shift is determined by the contents of a register. A register is read onto the A bus, and a second register or the immediate field onto the B bus. The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction, and the result (when required) is written to the destination register. (Compares and tests do not produce results, only the ALU status flags are affected.)

An instruction prefetch occurs at the same time as the above operation, and the program counter is incremented.

When the shift length is specified by a register, an additional datapath cycle occurs before the above operation to copy the bottom 8 bits of that register into a holding latch in the barrel shifter. The instruction prefetch will occur during this first cycle, and the operation cycle will be internal (ie will not request memory). This internal cycle can be merged with the following sequential access by the memory manager as the address remains stable through both cycles.

The PC may be one or more of the register operands. When it is the destination, external bus activity may be affected. If the result is written to the PC, the contents of the instruction pipeline are invalidated, and the address for the next instruction prefetch is taken from the ALU rather than the address incrementer. The instruction pipeline is refilled before any further execution takes place, and during this time exceptions are locked out.

PSR Transfer operations exhibit the same timing characteristics as the data operations except that the PC is never used as a source or destination register. The cycle timings are shown below Table 45.

Table 45. Data Operation Instruction Cycle Operations

	Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
normal	1	pc+2L	i	0	(pc+2L)	0	1	0
		pc+3L						
dest=pc	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	i	0	(alu)	0	1	0
	3	alu+L	i	0	(alu+L)	0	1	0
		alu+2L						
shift(Rs)	1	pc+2L	i	0	(pc+2L)	1	0	0
	2	pc+3L	i	0	-	0	1	1
		pc+3L						
shift(Rs)	1	pc+8	2	0	(pc+8)	1	0	0
dest=pc	2	pc+12	2	0	-	0	0	1
	3	alu	2	0	(alu)	0	1	0
	4	alu+4	2	0	(alu+4)	0	1	0
		alu+8						

Note: Shifted register with destination equals PC is not possible in THUMB state.

Multiply and Multiply Accumulate

The multiply instructions make use of special hardware which implements integer multiplication with early termination. All cycles except the first are internal.

The cycle timings are shown in the following four tables, where *m* is the number of cycles required by the multiplication algorithm; see Instruction Speed Summary on page 188.

Table 46. Multiply Instruction Cycle Operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+2L	0	i	(pc+2L)	1	0	0
2	pc+3L	0	i	-	1	0	1
•	pc+3L	0	i	-	1	0	1
m	pc+3L	0	i	-	1	0	1
m+1	pc+3L	0	i	-	0	1	1
	pc+3L						

Table 47. Multiply-Accumulate Instruction Cycle Operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+8	0	2	(pc+8)	1	0	0
2	pc+8	0	2	-	1	0	1
•	pc+12	0	2	-	1	0	1
m	pc+12	0	2	-	1	0	1
m+1	pc+12	0	2	-	1	0	1
m+2	pc+12	0	2	-	0	1	1
	pc+12						

Table 48. Multiply Long Instruction Cycle Operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+2L	0	i	(pc+2L)	1	0	0
2	pc+3L	0	i	-	1	0	1
•	pc+3L	0	i	-	1	0	1
m	pc+3L	0	i	-	1	0	1
m+1	pc+3L	0	i	-	1	0	1
m+2	pc+3L	0	i	-	0	1	1
	pc+3L						

Table 49. Multiply-Accumulate Long Instruction Cycle Operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+8	0	2	(pc+8)	1	0	0
2	pc+8	0	2	-	1	0	1
•	pc+12	0	2	-	1	0	1
m	pc+12	0	2	-	1	0	1
m+1	pc+12	0	2	-	1	0	1
m+2	pc+12	0	2	-	1	0	1
m+3	pc+12	0	2	-	0	1	1
	pc+12						

Note: Multiply-Accumulate is not possible in THUMB state.

Load Register

The first cycle of a load register instruction performs the address calculation. The data is fetched from memory during the second cycle, and the base register modification is performed during this cycle (if required). During the third cycle the data is transferred to the destination register, and external memory is unused. This third cycle may normally be merged with the following prefetch to form one memory N-cycle. The cycle timings are shown below in Table 50.

Either the base or the destination (or both) may be the PC, and the prefetch sequence will be changed if the PC is affected by the instruction.

The data fetch may abort, and in this case the destination modification is prevented.

Table 50. Load Register Instruction Cycle Operations

Cycle		Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC	nTRANS
normal	1	pc+2L	i	0	(pc+2L)	0	0	0	c
	2	alu	b/h/w	0	(alu)	1	0	1	d
	3	pc+3L	i	0	-	0	1	1	c
		pc+3L							
dest=pc	1	pc+8	2	0	(pc+8)	0	0	0	c
	2	alu		0	pc’	1	0	1	d
	3	pc+12	2	0	-	0	0	1	c
	4	pc’	2	0	(pc’)	0	1	0	c
	5	pc’+4	2	0	(pc’+4)	0	1	0	c
		pc’+8							

b, h and w are byte, halfword and word as defined in Table 40.

c represents current mode-dependent value.

d will either be 0 if the T bit has been specified in the instruction (eg. LDRT), or c at all other times.

Note: Destination equals PC is not possible in THUMB state.

Store Register

The first cycle of a store register is similar to the first cycle of load register. During the second cycle the base modification is performed, and at the same time the data is written to memory. There is no third cycle.

The cycle timings are shown below in Table 51.

Table 51. Store Register Instruction Cycle Operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC	nTRANS
1	pc+2L	i	0	(pc+2L)	0	0	0	c
2	alu	b/h/w	1	Rd	0	0	1	d
	pc+3L							

b, h and w are byte, halfword and word as defined in Table 40.

c represents current mode-dependent value

d will either be 0 if the T bit has been specified in the instruction (eg. SDRT), or c at all other times.

Load Multiple Registers

The first cycle of LDM is used to calculate the address of the first word to be transferred, whilst performing a prefetch from memory. The second cycle fetches the first word, and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is latched internally in case it is needed to patch up after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The cycle timings are shown in Table 52.

The last cycle may be merged with the next instruction prefetch to form a single memory N-cycle.

If an abort occurs, the instruction continues to completion, but all register writing after the abort is prevented. The final cycle is altered to restore the modified base register (which may have been overwritten by the load activity before the abort occurred).

When the PC is in the list of registers to be loaded the current instruction pipeline must be invalidated.

Note: The PC is always the last register to be loaded, so an abort at any point will prevent the PC from being overwritten.

Note: LDM with destination = PC cannot be executed in THUMB state. However `POP{Rlist,PC}` equates to an LDM with destination=PC.

Table 52. Load Multiple Registers Instruction Cycle Operations

Cycle		Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1 register	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	2	0	(alu)	1	0	1
	3	pc+3L	i	0	-	0	1	1
		pc+3L						
1 register	1	pc+2L	i	0	(pc+2L)	0	0	0
dest=pc	2	alu	2	0	pc'	1	0	1
	3	pc+3L	i	0	-	0	0	1
	4	pc'	i	0	(pc')	0	1	0
	5	pc'+L	i	0	(pc'+L)	0	1	0
		pc'+2L						
n registers	1	pc+2L	i	0	(pc+2L)	0	0	0
(n>1)	2	alu	2	0	(alu)	0	1	1
	•	alu+•	2	0	(alu+•)	0	1	1
	n	alu+•	2	0	(alu+•)	0	1	1
	n+1	alu+•	2	0	(alu+•)	1	0	1
	n+2	pc+3L	i	0	-	0	1	1
		pc+3L						
n registers	1	pc+2L	i	0	(pc+2L)	0	0	0
(n>1)	2	alu	2	0	(alu)	0	1	1
incl pc	•	alu+•	2	0	(alu+•)	0	1	1
	n	alu+•	2	0	(alu+•)	0	1	1
	n+1	alu+•	2	0	pc'	1	0	1
	n+2	pc+3L	i	0	-	0	0	1
	n+3	pc'	i	0	(pc')	0	1	0
	n+4	pc'+L	i	0	(pc'+L)	0	1	0
		pc'+2L						

Store Multiple Registers

Store multiple proceeds very much as load multiple, without the final cycle. The restart problem is much more straightforward here, as there is no wholesale overwriting of registers. The cycle timings are shown in Table 53 below.

Table 53. Store Multiple Registers Instruction Cycle Operations

Cycle		Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1 register	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	2	1	Ra	0	0	1
		pc+3L						
n registers	1	pc+8	i	0	(pc+2L)	0	0	0
(n>1)	2	alu	2	1	Ra	0	1	1
	•	alu+•	2	1	R•	0	1	1
	n	alu+•	2	1	R•	0	1	1
	n+1	alu+•	2	1	R•	0	0	1
		pc+12						

Data Swap

This is similar to the load and store register instructions, but the actual swap takes place in cycles 2 and 3. In the second cycle, the data is fetched from external memory. In the third cycle, the contents of the source register are written out to the external memory. The data read in cycle 2 is written into the destination register during the fourth cycle. The cycle timings are shown below in Table 54.

The **LOCK** output of ARM7TDMI is driven HIGH for the duration of the swap operation (cycles 2 and 3) to indicate that both cycles should be allowed to complete without interruption.

The data swapped may be a byte or word quantity (b/w).

The swap operation may be aborted in either the read or write cycle, and in both cases the destination register will not be affected.

Table 54. Data Swap Instruction Cycle Operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC	LOCK
1	pc+8	2	0	(pc+8)	0	0	0	0
2	Rn	b/w	0	(Rn)	0	0	1	1
3	Rn	b/w	1	Rm	1	0	1	1
4	pc+12	2	0	-	0	1	1	0
	pc+12							

b and w are byte and word as defined in Table 40.

Note: Data swap cannot be executed in THUMB state.

Software Interrupt and Exception Entry

Exceptions (and software interrupts) force the PC to a particular value and refill the instruction pipeline from there. During the first cycle the forced address is constructed, and a mode change may take place. The return address is moved to R14 and the CPSR to SPSR_svc.

During the second cycle the return address is modified to facilitate return, though this modification is less useful than in the case of branch with link.

The third cycle is required only to complete the refilling of the instruction pipeline. The cycle timings are shown below in Table 55.

Table 55. Software Interrupt Instruction Cycle Operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC	nTRANS	Mode	TBIT
1	pc+2L	i	0	(pc+2L)	0	0	0	C	old mode	T
2	Xn	2	0	(Xn)	0	1	0	1	exception mode	0
3	Xn+4	2	0	(Xn+4)	0	1	0	1	exception mode	0
	Xn+8									

C represents the current mode-dependent value.

T represents the current state-dependent value

pc for software interrupts is the address of the SWI instruction.
for exceptions is the address of the instruction following the last one to be executed before entering the exception.
for prefetch aborts is the address of the aborting instruction.
for data aborts is the address of the instruction following the one which attempted the aborted data transfer.

Xn is the appropriate trap address.

Coprocessor Data Operation

A coprocessor data operation is a request from ARM7TDMI for the coprocessor to initiate some action. The action need not be completed for some time, but the coprocessor must commit to doing it before driving **CPB** LOW.

If the coprocessor can never do the requested task, it should leave **CPA** and **CPB** HIGH. If it can do the task, but can't commit right now, it should drive **CPA** LOW but leave **CPB** HIGH until it can commit. ARM7TDMI will busy-wait until **CPB** goes LOW. The cycle timings are shown in Table 56.

Table 56. Coprocessor Data Operation Instruction Cycle Operations

Cycle		Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	0	2	(pc+8)	0	0	0	0	0	0
		pc+12									
not ready	1	pc+8	0	2	(pc+8)	1	0	0	0	0	1
	2	pc+8	0	2	-	1	0	1	0	0	1
	•	pc+8	0	2	-	1	0	1	0	0	1
	n	pc+8	0	2	-	0	0	1	0	0	0
		pc+12									

Note: This operation cannot occur in THUMB state.

Coprocessor Data Transfer (from memory to coprocessor)

Here the coprocessor should commit to the transfer only when it is ready to accept the data. When **CPB** goes LOW, ARM7TDMI will produce addresses and expect the coprocessor to take the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred, and indicates the last transfer cycle by driving **CPA** and **CPB** HIGH.

ARM7TDMI spends the first cycle (and any busy-wait cycles) generating the transfer address, and performs the write-back of the address base during the transfer cycles. The cycle timings are shown in Table 57.

Table 57. Coprocessor Data Transfer Instruction Cycle Operations

Cycles		Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
1 register	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
ready	2	alu	2	0	(alu)	0	0	1	1	1	1
		pc+12									
1 register	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
not ready	2	pc+8	2	0	-	1	0	1	0	0	1
	•	pc+8	2	0	-	1	0	1	0	0	1
	n	pc+8	2	0	-	0	0	1	0	0	0
	n+1	alu	2	0	(alu)	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	2	0	(alu)	0	1	1	1	0	0
ready	•	alu+•	2	0	(alu+•)	0	1	1	1	0	0
	n	alu+•	2	0	(alu+•)	0	1	1	1	0	0
	n+1	alu+•	2	0	(alu+•)	0	0	1	1	1	1
		pc+12									
m registers	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
(m>1)	2	pc+8	2	0	-	1	0	1	0	0	1
not ready	•	pc+8	2	0	-	1	0	1	0	0	1
	n	pc+8	2	0	-	0	0	1	0	0	0
	n+1	alu	2	0	(alu)	0	1	1	1	0	0
	•	alu+•		0	(alu+•)	0	1	1	1	0	0
	n+m	alu+•	2	0	(alu+•)	0	1	1	1	0	0
	n+m+1	alu+•	2	0	(alu+•)	0	0	1	1	1	1
		pc+12									

Note: This operation cannot occur in THUMB state.

Coprocessor Data Transfer (from coprocessor to memory)

The ARM7TDMI controls these instructions exactly as for memory to coprocessor transfers, with the one exception that the **nRW** line is inverted during the transfer cycle. The cycle timings are show in Table 58.

Table 58. Coprocessor Data Transfer Instruction Cycle Operations

Cycle		Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
1 register	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
ready	2	alu	2	1	CPdata	0	0	1	1	1	1
		pc+12									
1 register	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
not ready	2	pc+8	2	0	-	1	0	1	0	0	1
	•	pc+8	2	0	-	1	0	1	0	0	1
	n	pc+8	2	0	-	0	0	1	0	0	0
	n+1	alu	2	1	CPdata	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	2	1	CPdata	0	1	1	1	0	0
ready	•	alu+•	2	1	CPdata	0	1	1	1	0	0
	n	alu+•	2	1	CPdata	0	1	1	1	0	0
	n+1	alu+•	2	1	CPdata	0	0	1	1	1	1
		pc+12									
m registers	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
(m>1)	2	pc+8	2	0	-	1	0	1	0	0	1
not ready	•	pc+8	2	0	-	1	0	1	0	0	1
	n	pc+8	2	0	-	0	0	1	0	0	0
	n+1	alu	2	1	CPdata	0	1	1	1	0	0
	•	alu+•	2	1	CPdata	0	1	1	1	0	0
	n+m	alu+•	2	1	CPdata	0	1	1	1	0	0
	n+m+1	alu+•	2	1	CPdata	0	0	1	1	1	1
		pc+12									

Note: This operation cannot occur in THUMB state.

Coprocessor Register Transfer (Load from coprocessor)

Here the busy-wait cycles are much as above, but the transfer is limited to one data word, and ARM7TDMI puts the word into the destination register in the third cycle. The third cycle may be merged with the following prefetch cycle into one memory N-cycle as with all ARM7TDMI register load instructions. The cycle timings are shown in Table 59.

Table 59. Coprocessor Register Transfer (Load from Coprocessor)

Cycle		Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	2	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	2	0	CPdata	1	0	1	1	1	1
	3	pc+12	2	0	-	0	1	1	1	-	-
		pc+12									
not ready	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
	2	pc+8	2	0	-	1	0	1	0	0	1
	•	pc+8	2	0	-	1	0	1	0	0	1
	n	pc+8	2	0	-	1	1	1	0	0	0
	n+1	pc+12	2	0	CPdata	1	0	1	1	1	1
	n+2	pc+12	2	0	-	0	1	1	1	-	-
		pc+12									

Note: This operation cannot occur in THUMB state.

Coprocessor Register Transfer (Store to coprocessor)

As for the load from coprocessor, except that the last cycle is omitted. The cycle timings are shown in Table 60.

Table 60. Coprocessor Register Transfer (Store to Coprocessor)

Cycle		Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	2	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	2	1	Rd	0	0	1	1	1	1
		pc+12									
not ready	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
	2	pc+8	2	0	-	1	0	1	0	0	1
	•	pc+8	2	0	-	1	0	1	0	0	1
	n	pc+8	2	0	-	1	1	1	0	0	0
	n+1	pc+12	2	1	Rd	0	0	1	1	1	1
		pc+12									

Note: This operation cannot occur in THUMB state.

Undefined Instructions and Coprocessor Absent

When a coprocessor detects a coprocessor instruction which it cannot perform, and this must include all undefined instructions, it must not drive **CPA** or **CPB** LOW. These will remain HIGH, causing the undefined instruction trap to be taken. Cycle timings are shown in Table 61.

Table 61. Undefined Instruction Cycle Operations

Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB	nTRANS	Mode	TBIT
1	pc+2L	i	0	(pc+2L)	1	0	0	0	1	1	C	Old	T
2	pc+2L	i	0	-	0	0	0	1	1	1	C	Old	T
3	Xn	2	0	(Xn)	0	1	0	1	1	1	1	00100	0
4	Xn+4	2	0	(Xn+4)	0	1	0	1	1	1	1	00100	0
	Xn+8												

C represents the current mode-dependent value.

T represents the current state-dependent value.

Note: Coprocessor Instructions cannot occur in THUMB state.

Unexecuted Instructions

Any instruction whose condition code is not met will fail to execute. It will add one cycle to the execution time of the code segment in which it is embedded (see Table 62).

Table 62. Unexecuted Instruction Cycle Operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1	pc+2L	i	0	(pc+2L)	0	1	0
	pc+3L						

Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in Table 63. These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

n is the number of words transferred

m is 1 if bits [32:8] of the multiplier operand are all zero or one.
2 if bits[32:16] of the multiplier operand are all zero or one.
3if bits[31:24] of the multiplier operand are all zero or all one.
4 otherwise.

b is the number of cycles spent in the coprocessor busy-wait loop.

If the condition is not met all the instructions take one S-cycle. The cycle types N, S, I, and C are defined in Memory Interface on page 117.

Table 63. ARM Instruction Speed Summary

Instruction	Cycle count	Additional
Data Processing	1S	+ 1I for SHIFT(Rs) + 1S + 1N if R15 written
MSR, MRS	1S	
LDR	1S+1N+1I	+ 1S + 1N if R15 loaded
STR	2N	
LDM	nS+1N+1I	+ 1S + 1N if R15 loaded
STM	(n-1)S+2N	
SWP	1S+2N+1I	
B,BL	2S+1N	
SWI, trap	2S+1N	
MUL	1S+mI	
MLA	1S+(m+1)I	
MULL	1S+(m+1)I	
MLAL	1S+(m+2)I	
CDP	1S+bI	
LDC,STC	(n-1)S+2N+bI	
MCR	1N+bI+1C	
MRC	1S+(b+1)I+1C	

Timing Diagrams

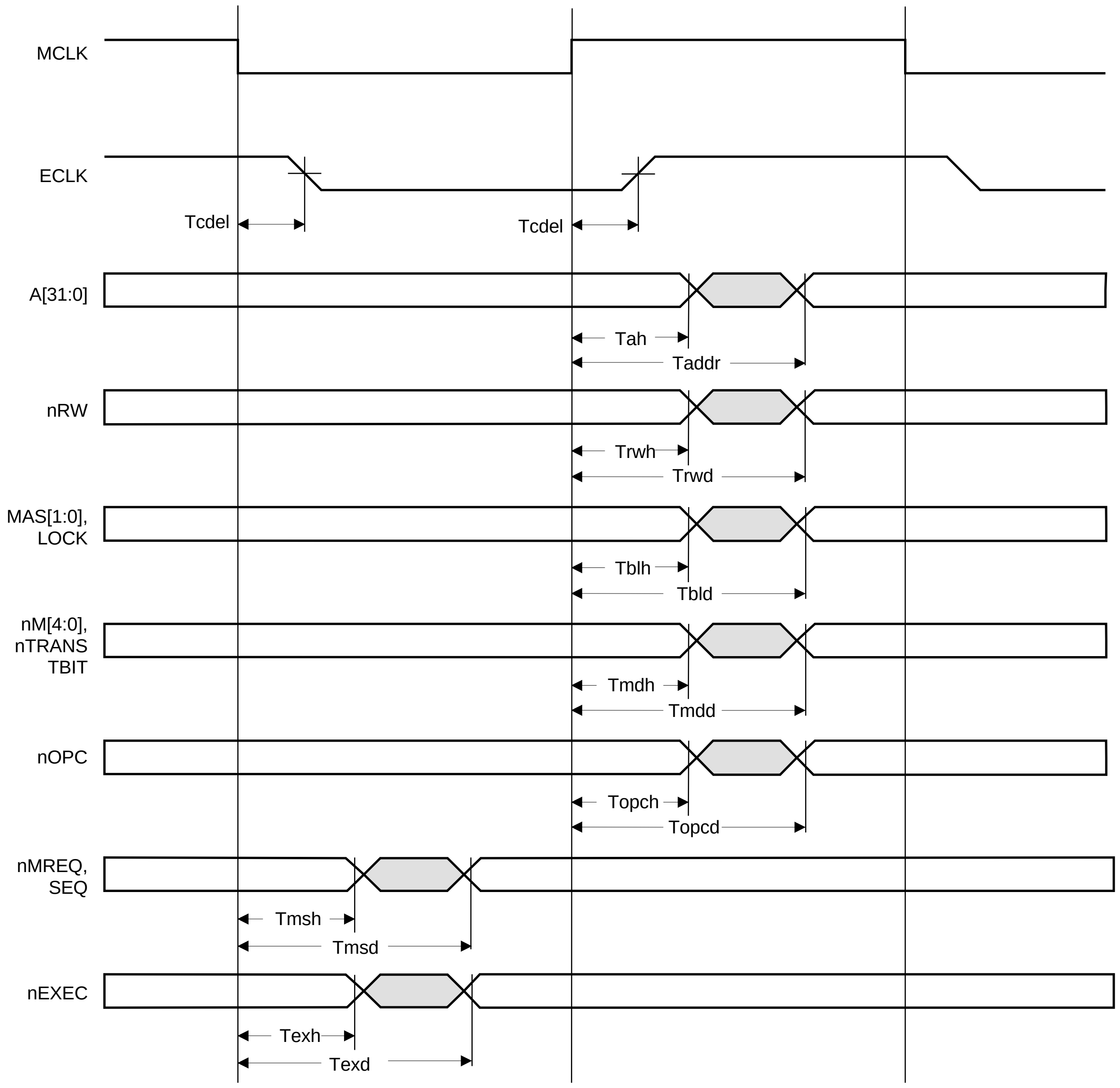
This sections presents the timing diagrams for the ARM7TDMI Core.

The delays shown in these timing diagrams are all process specific. For the corresponding characterized values, refer to one of the following datasheets:

- ARM7TDMI Embedded Core ATC50 Electrical Characteristics
  (0.5 micron three-layer-metal CMOS process intended for use with a supply voltage of 3.3V ± 0.3V, previously known as AT55K)
- ARM7TDMI Embedded Core ATC50/E^2 Electrical Characteristics
  (0.5 micron three-layer-metal CMOS/NVM process intended for use with a supply voltage of 3.3V ± 0.3V, previously known as AT55.8K)
- ARM7TDMI Embedded Core ATC35 Electrical Characteristics
  (0.35 micron three-layer-metal CMOS process intended for use with a supply voltage of 3.3V ± 0.3V, previously known as AT56K)

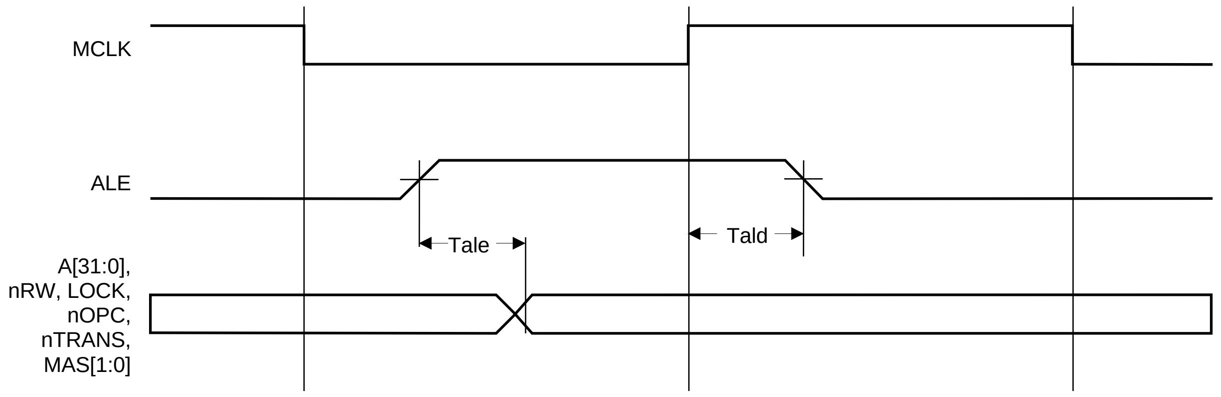
Timing Diagrams

Figure 89. General Timings



Note: **nWAIT, APE, ALE** and **ABE** are all **HIGH** during the cycle shown. Tcdel is the delay (on either edge) from **MCLK** changing to **ECLK** changing.

Figure 90. ALE Address Control



Note: Tald is the time by which **ALE** must be driven LOW in order to latch the current address in phase 2. If **ALE** is driven low after Tald, then a new address will be latched.

Figure 91. APE Address Control

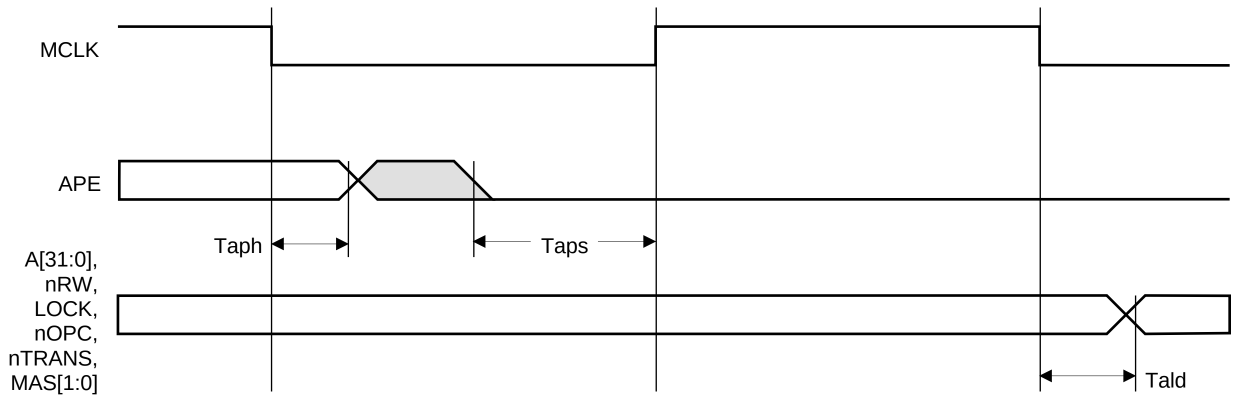


Figure 92. ABE Address Control

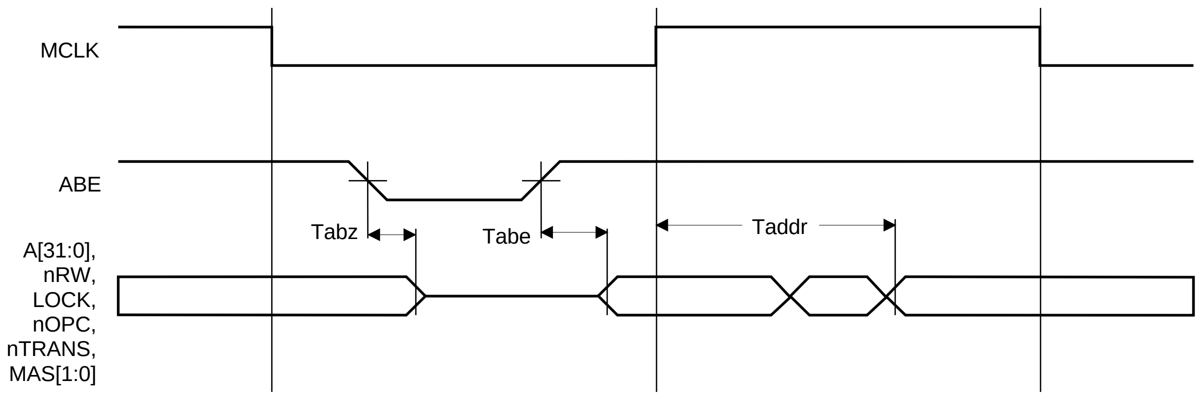
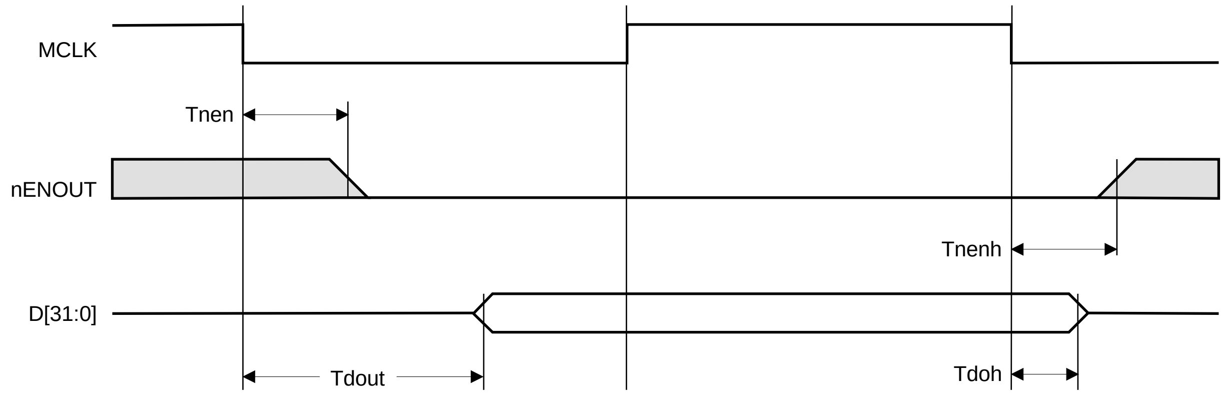
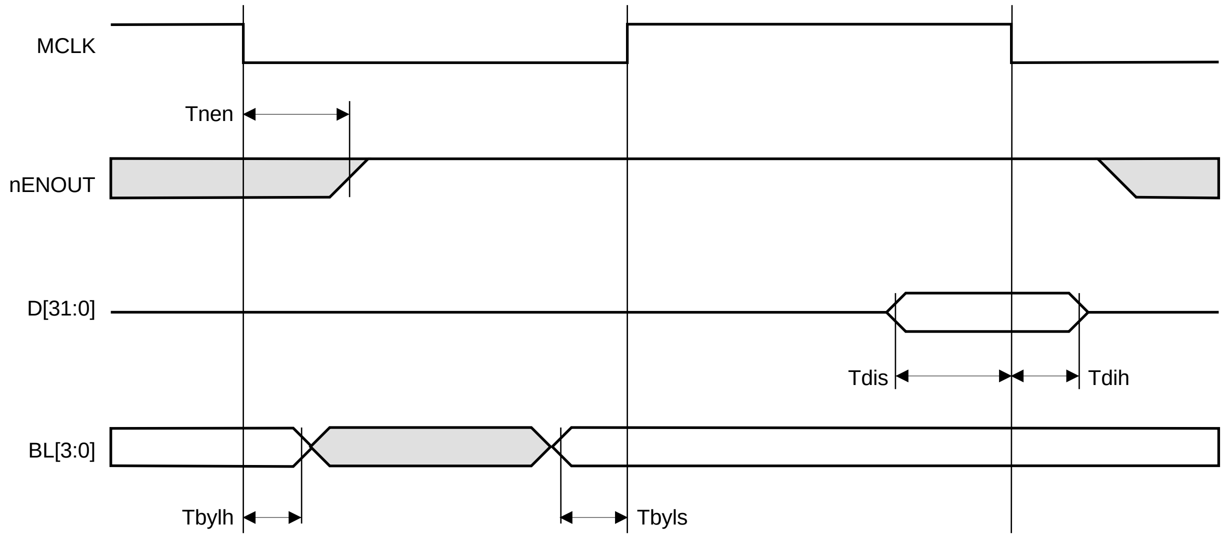


Figure 93. Bidirectional Data Write Cycle



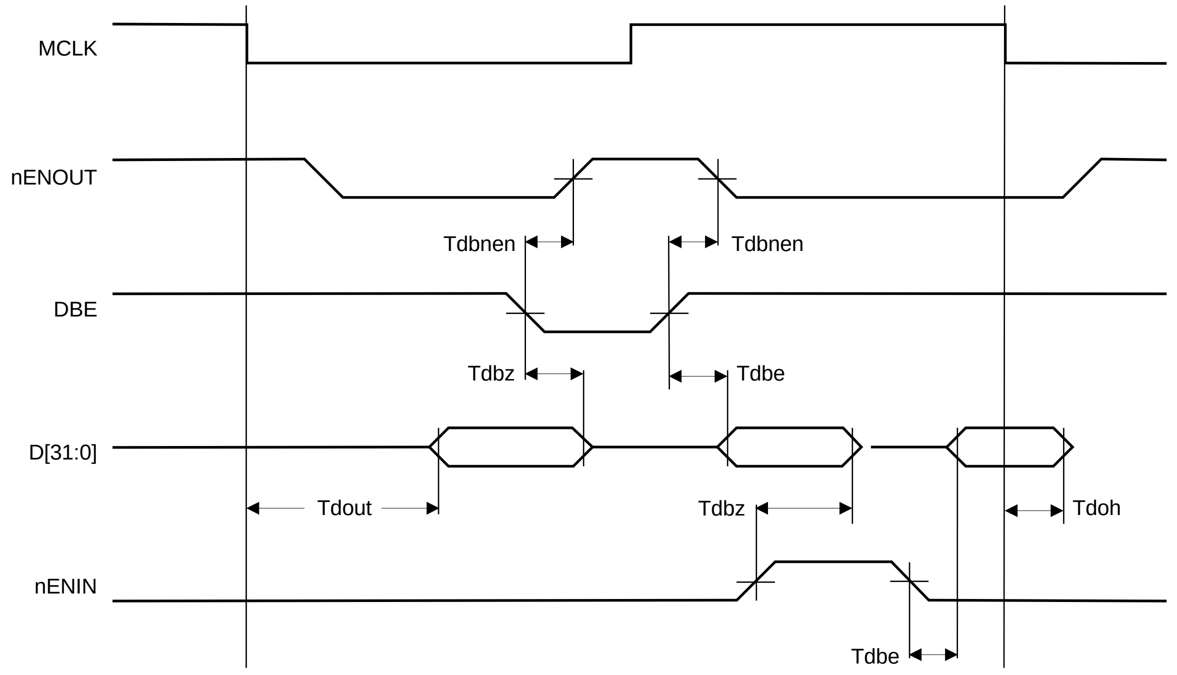
Note: **DBE** is HIGH and **nENIN** is LOW during the cycle shown.

Figure 94. Bidirectional Data Read Cycle



Note: **DBE** is HIGH and **nENIN** is LOW during the cycle shown.

Figure 95. Data Bus Control



Note: The cycle shown is a data write cycle since **nENOUT** was driven LOW during phase 1. Here, **DBE** has first been used to modify the behaviour of the data bus, and then **nENIN**.

Figure 96. Output 3-State Time

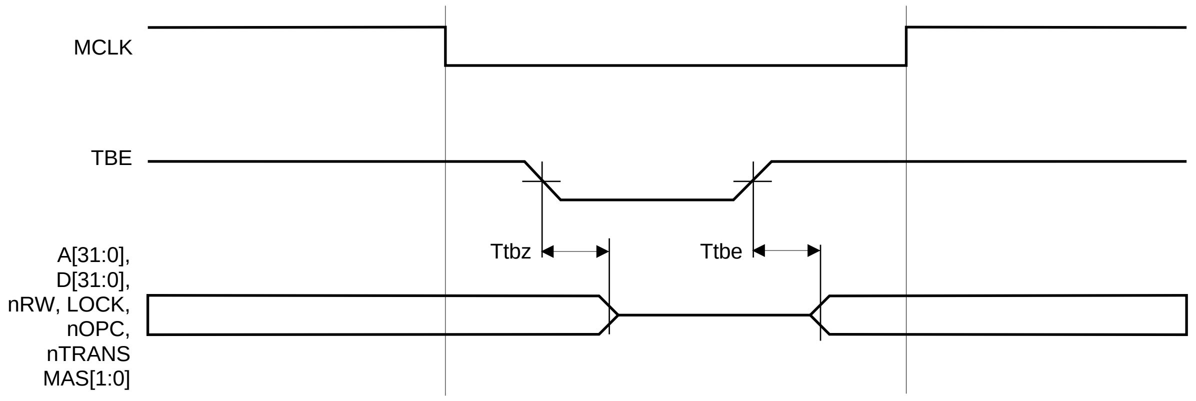


Figure 97. Unidirectional Data Write Cycle

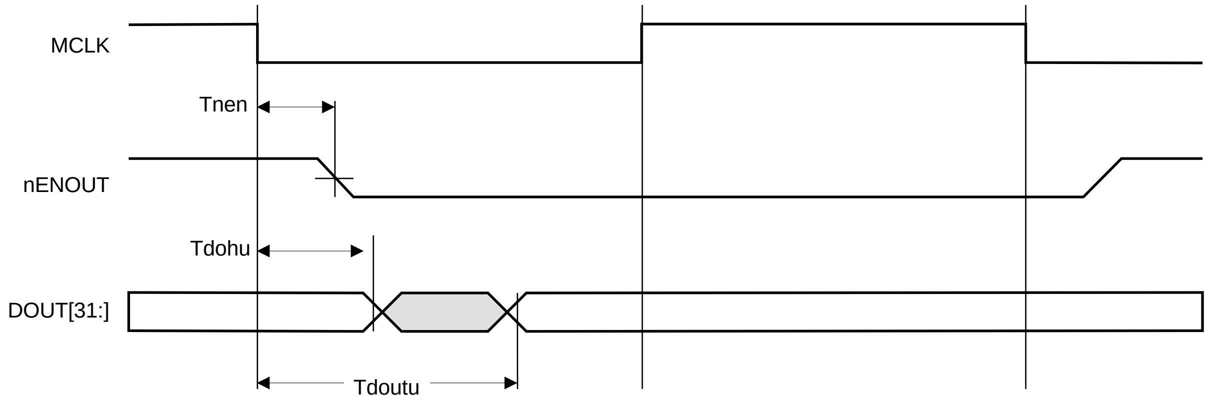


Figure 98. Unidirectional Data Read Cycle

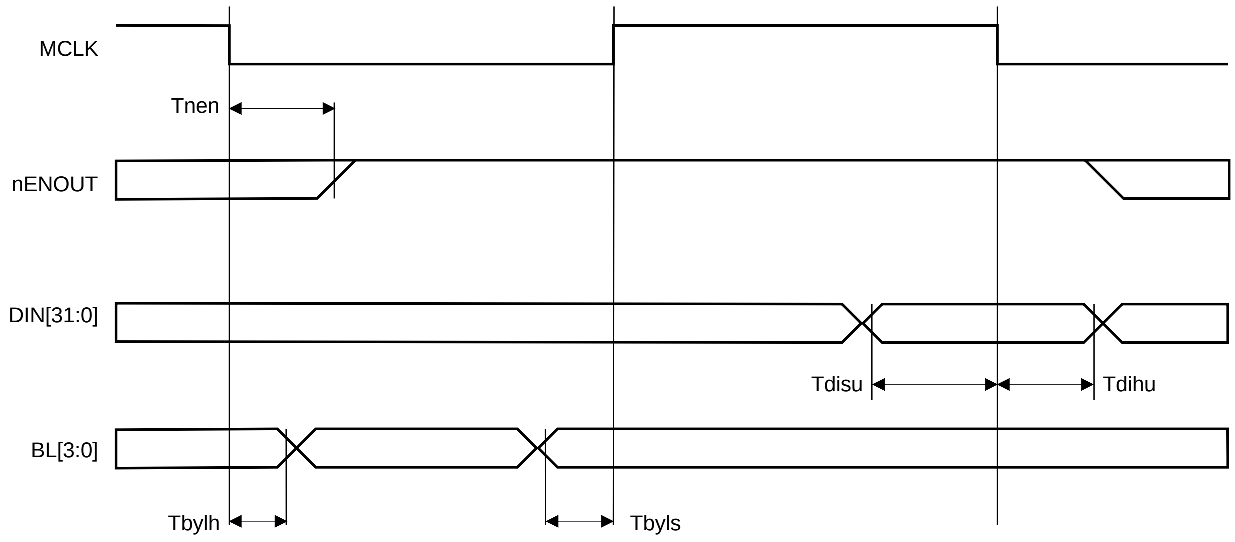


Figure 99. Configuration Pin Timing

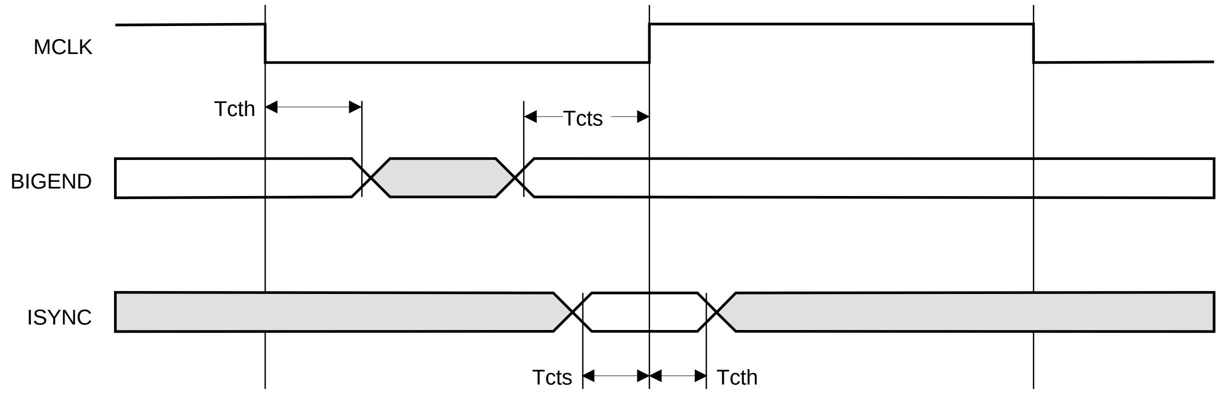
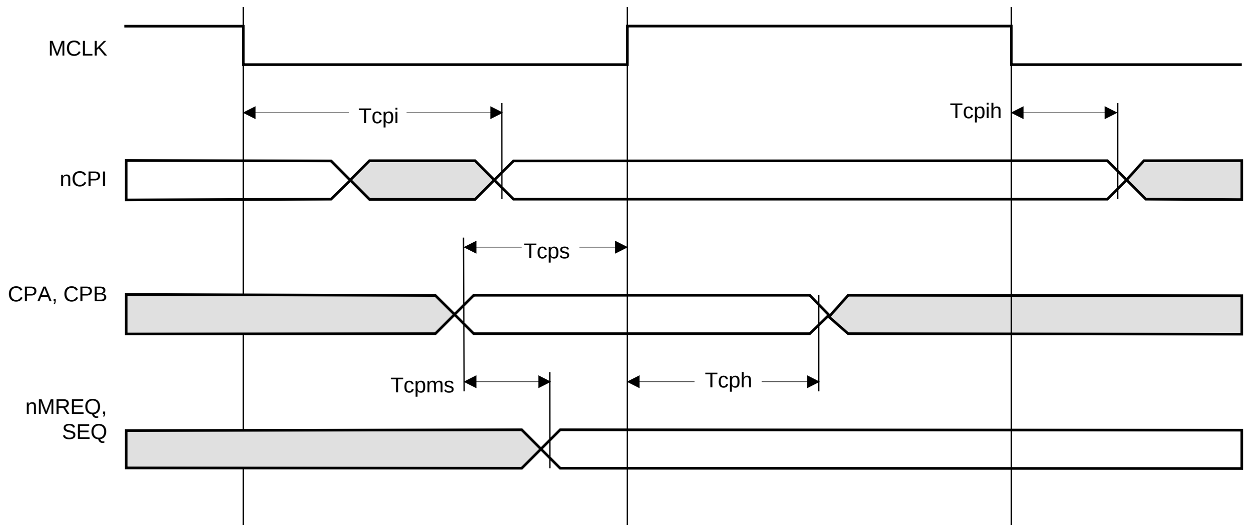
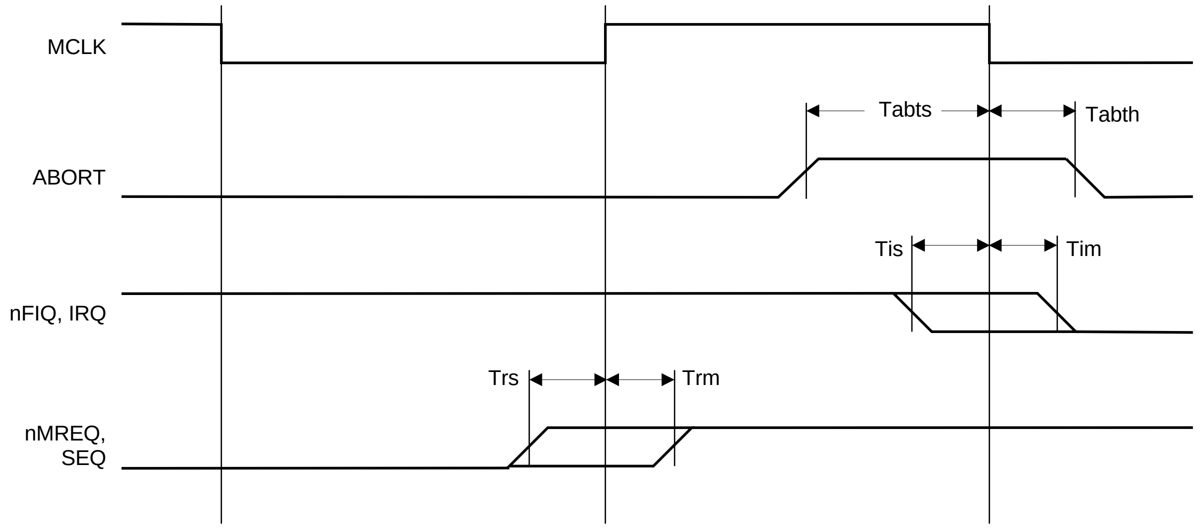


Figure 100. Coprocessor Timing



Note: Normally, **nMREQ** and **SEQ** become valid Tmsd after the falling edge of **MCLK**. In this cycle the ARM has been busy-waiting, waiting for a coprocessor to complete the instruction. If **CPA** and **CPB** change during phase 1, the timing of **nMREQ** and **SEQ** will depend on Tcpms. Most systems should be able to generate **CPA** and **CPB** during the previous phase 2, and so the timing of **nMREQ** and **SEQ** will always be Tmsd.

Figure 101. Exception Timing



Note: Tis/Trs guarantee recognition of the interrupt (or reset) source by the corresponding clock edge. Tim/Trm guarantee non-recognition by that clock edge. These inputs may be applied fully asynchronously where the exact cycle of recognition is unimportant.

Figure 102. Debug Timing

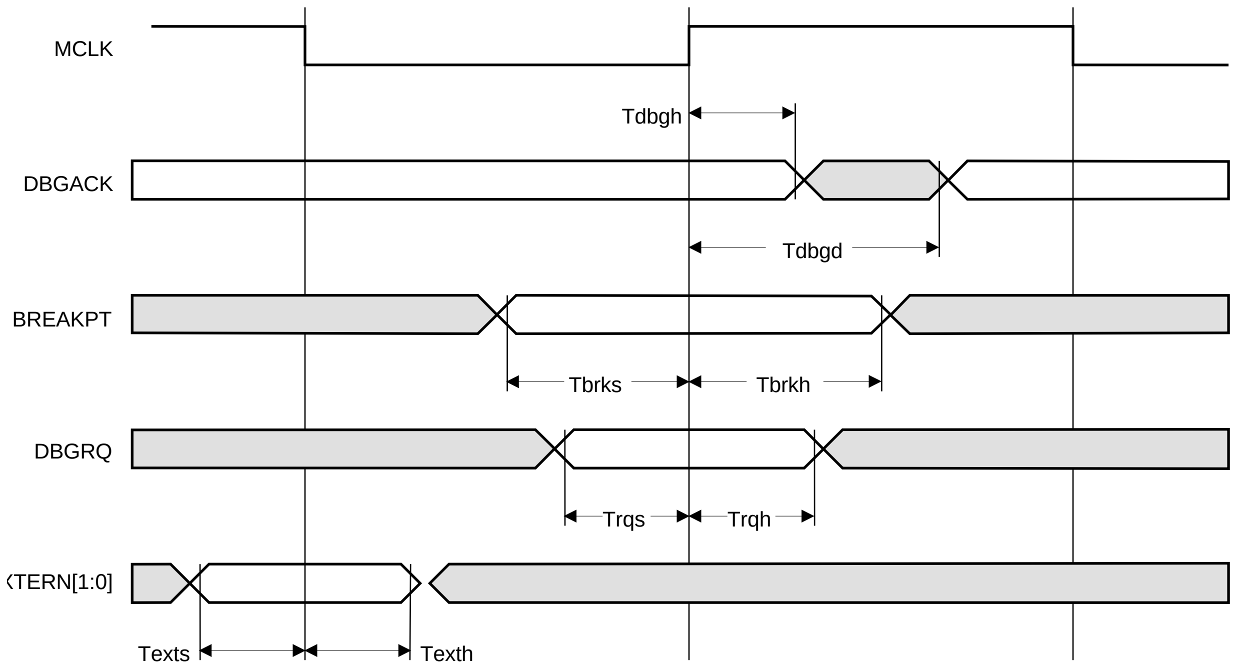
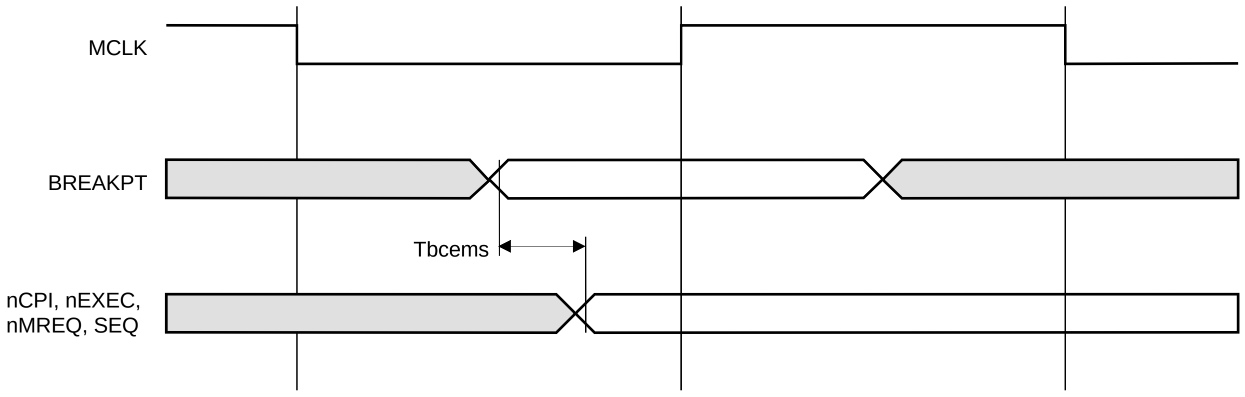


Figure 103. Breakpoint Timing



Note: **BREAKPT** changing in the LOW phase of **MCLK** to signal a watchpointed store can affect **nCPI**, **nEXEC**, **nMREQ**, and **SEQ** in the LOW phase of **MCLK**.

Figure 104. TCK-ECLK Relationship

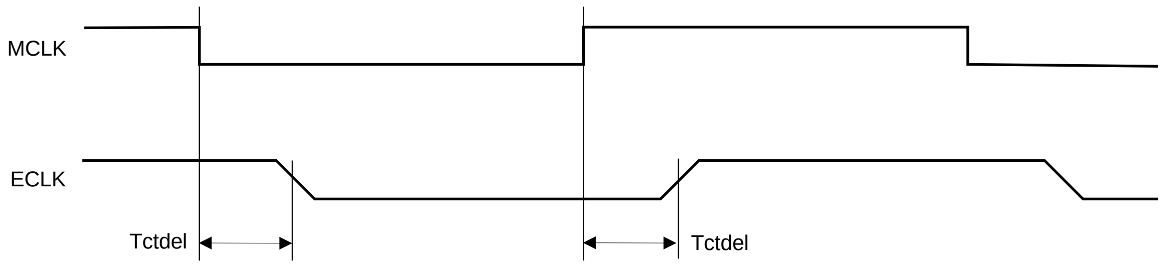
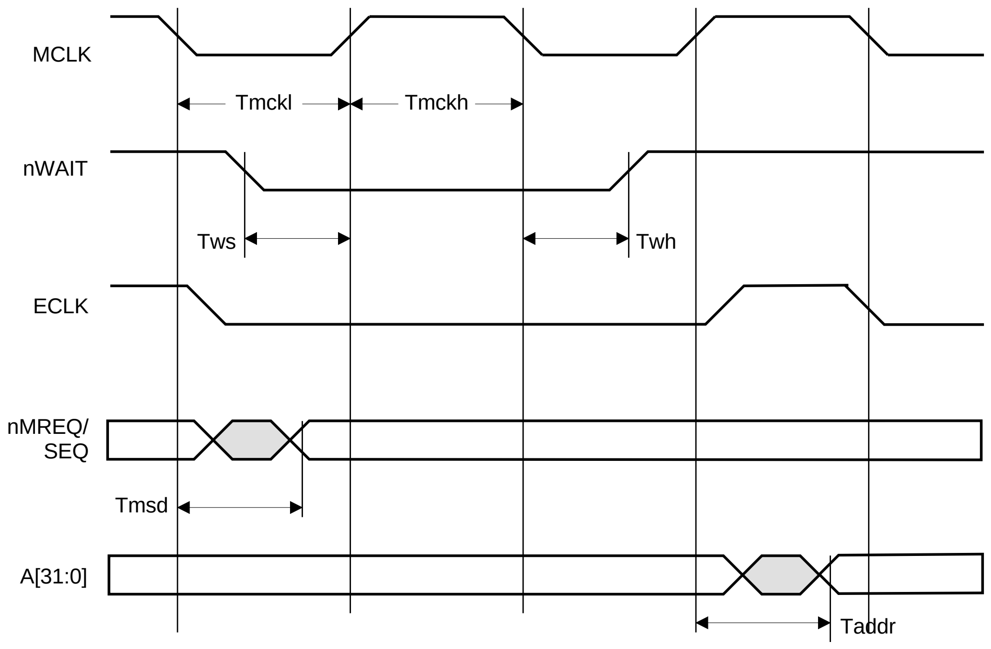


Figure 105. MCLK Timing



Note: The ARM core is not clocked by the HIGH phase of **MCLK** enveloped by **nWAIT**. Thus, during the cycles shown, **nMREQ** and **SEQ** change once, during the first LOW phase of **MCLK**, and **A[31:0]** change once, during the second HIGH phase of **MCLK**. For reference, ph2 is shown. This is the internal clock from which the core times all its activity. This signal is included to show how the high phase of the external **MCLK** has been removed from the internal core clock.

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 441-0311
FAX (408) 487-2600

Europe
Atmel U.K., Ltd.
Coliseum Business Centre
Riverside Way
Camberley, Surrey GU15 3YL
England
TEL (44) 1276-686677
FAX (44) 1276-686697

Asia
Atmel Asia, Ltd.
Room 1219
Chinachem Golden Plaza
77 Mody Road
Tsimshatsui East
Kowloon, Hong Kong
TEL (852) 27219778
FAX (852) 27221369

Japan
Atmel Japan K.K.
Tonetsu Shinkawa Bldg., 9F
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Atmel Colorado Springs
1150 E. Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Atmel Rousset
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4 42 53 60 00
FAX (33) 4 42 53 60 01

Fax-on-Demand
North America:
1-(800) 292-8635
International:
1-(408) 441-0732

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

BBS
1-(408) 436-4309

Rev. 0673B–01/99