ARM 7TDMI

Data Sheet

Document Number:ARM DDI 0029E

Issued: August 1995

Change Log

Issue	Date	By	Change
A (Draft 0.1)	Sept 1994	EH/BJH	Created.
(Draft 0.2)	Oct 1994	EH	First pass review comments added.
B	Dec 1994	EH/AW	First formal release
C	Dec 1994	AW	Further review comments
	Mar 1995	AW	Reissued with open access status. No change to the content.
D draft1	Mar 1995	AW	Changes in line with the ARM7TDM datasheet. Further technical changes.
D	Mar 1995	AW	Review comments added.
E	Aug 1995	AP	Signals added plus minor changes.

Key:

Open Access	No confidentiality

To enable document tracking, the document number has two codes:

Major release

-	Pre-release
A	First release
B	Second release
etc	etc

Draft Status

	Full and complete
draft1	First Draft
draft2	Second Draft
etc	etc
E	Embargoed (date given)

TOC

Contents

Contents

1 Introduction

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

1.1 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.

1.2 ARM7TDMI Architecture

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.

1.2.1 The THUMB Concept

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.

1.2.2 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.

1.3 ARM7TDMI Block Diagram

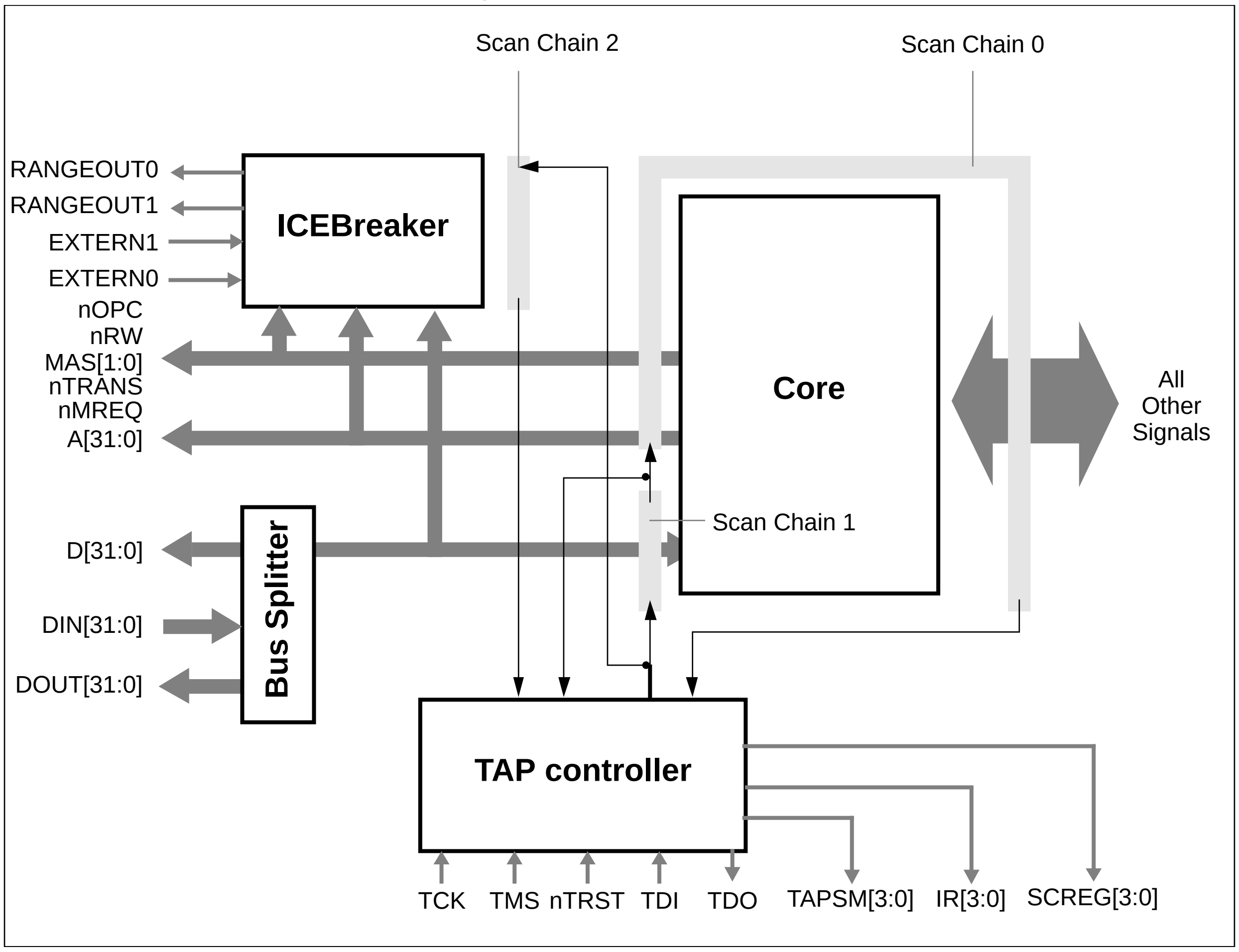


Figure 1-1: ARM7TDMI block diagram

1.4 ARM7TDMI Core Diagram

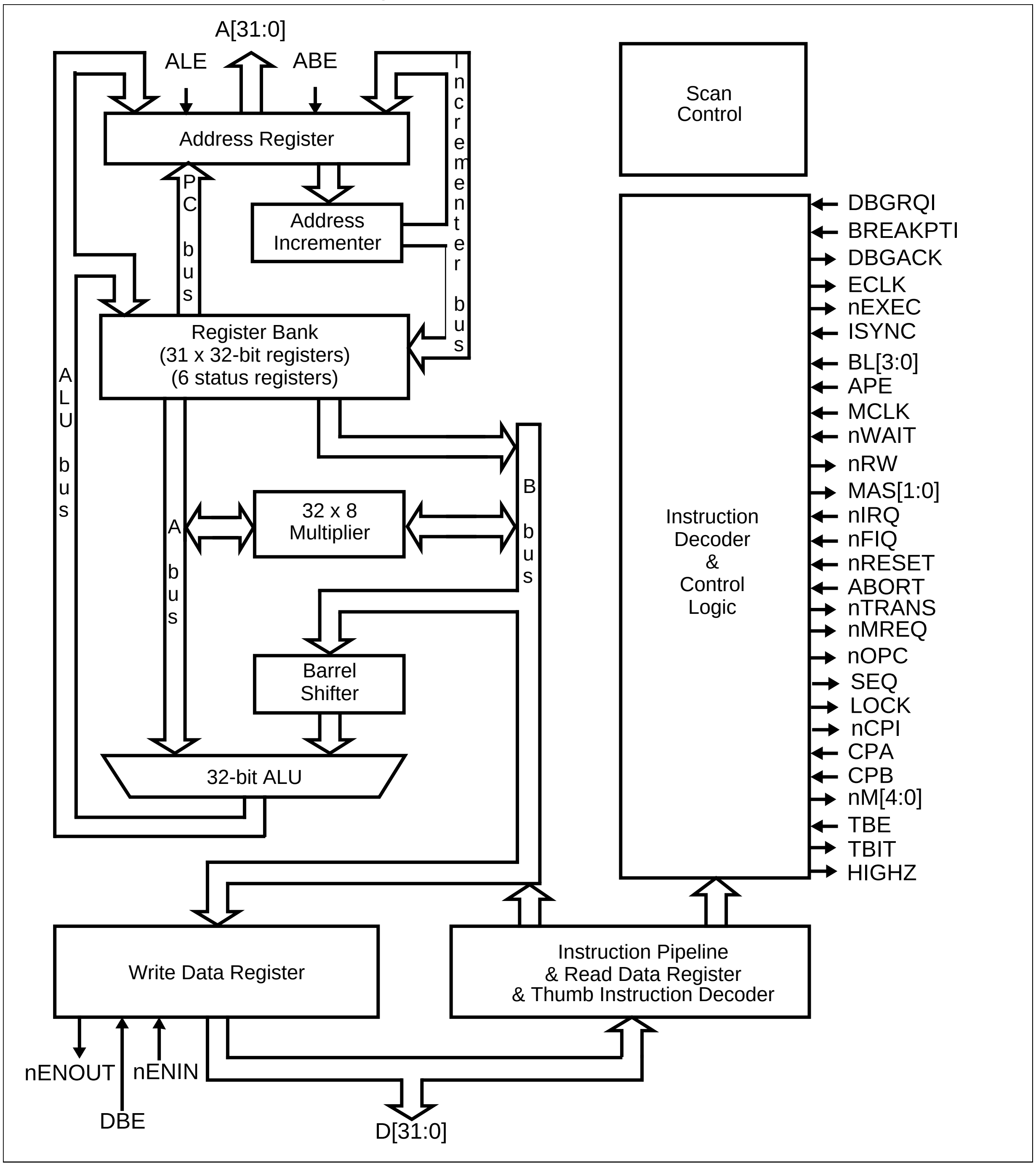


Figure 1-2: ARM7TDMI core

1.5 ARM7TDMI Functional Diagram

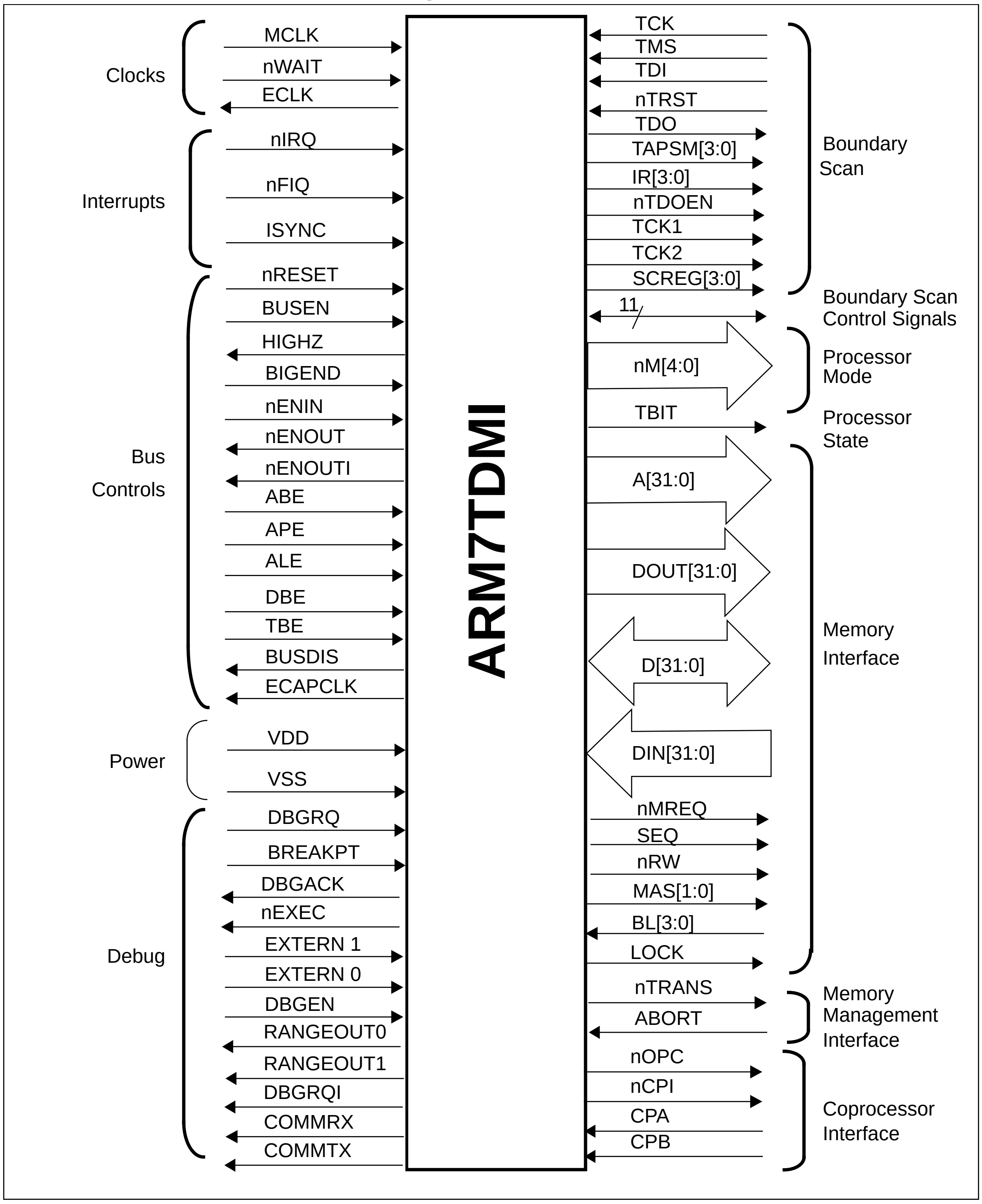


Figure 1-3: ARM7TDMI functional diagram

2 Signal Description

This chapter lists and describes the signals for the ARM7TDMI.

2.1 Signal Description

The following table lists and describes all the signals for the ARM7TDMI.

Transistor sizes

For a 0.6 µm ARM7TDMI:

INV4 driver has transistor sizes of	p = 22.32 µm/0.6 µm
	N = 12.6 µm/0.6 µm

INV8 driver has transistor sizes of	p = 44.64 µm/0.6 µm
	N = 25.2 µm/0.6 µm

Key to signal types

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

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.

Table 2-1: Signal Description

Name	Type	Description
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 ➲ *Chapter 6, Memory Interface* 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 ➲ *Chapter 6, Memory Interface* 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 breakpointed. 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 ➲ *Chapter 9, ICEBreaker Module*.
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 ➲ *Chapter 6, Memory Interface*. 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 ➲*9.11 Debug Communications Channel* on page 9-14 for more information on the debug comms channel.

Table 2-1: 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 *9.11 Debug Communications Channel* on page 9-14 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 *Chapter 6, Memory Interface*.
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 *Chapter 9, ICEBreaker Module* for details.

Table 2-1: Signal Description (Continued)

Name	Type	Description
DBGRQI Internal debug request	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 ➲ *Chapter 9, ICEBreaker Module* 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.
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 ARM7DM 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.

Table 2-1: Signal Description (Continued)

Name	Type	Description
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 *Chapter 8, Debug Interface* 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 *8.8 Public Instructions* on page 8-9. 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 2-1: 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 ➲ *Chapter 6, Memory Interface*.
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 ➲ *Chapter 6, Memory Interface*.
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.

Table 2-1: Signal Description (Continued)

Name	Type	Description
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.
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 ➲ *Chapter 8, Debug Interface.*

Table 2-1: Signal Description (Continued)

Name	Type	Description
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 2-1: 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 *8.4.2 The JTAG state machine* on page 8-8. 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.

Table 2-1: Signal Description (Continued)

Name	Type	Description
VDD Power supply.	P	These connections provide power to the device.
VSS Ground.	P	These connections are the ground reference for all signals.

Table 2-1: Signal Description (Continued)

3 Programmer's Model

This chapter describes the two operating states of the ARM7TDMI.

3.1 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.*

3.2 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.).

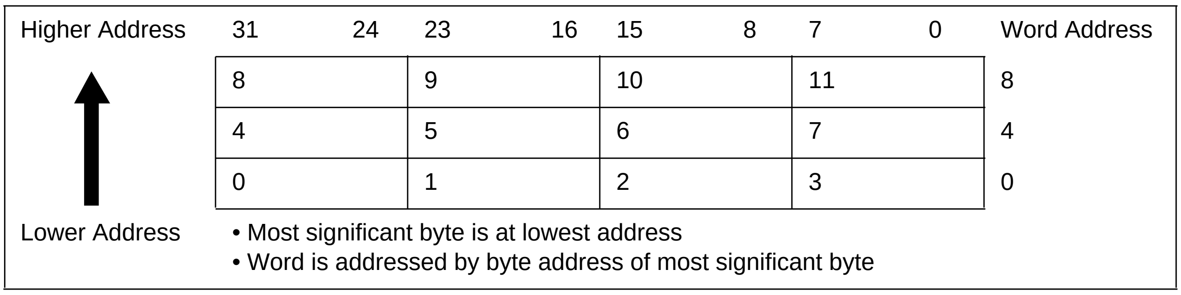
In this case, the PC is placed in the exception mode's link register, and execution commences at the exception's vector address.

3.3 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.

3.3.1 Big endian format

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



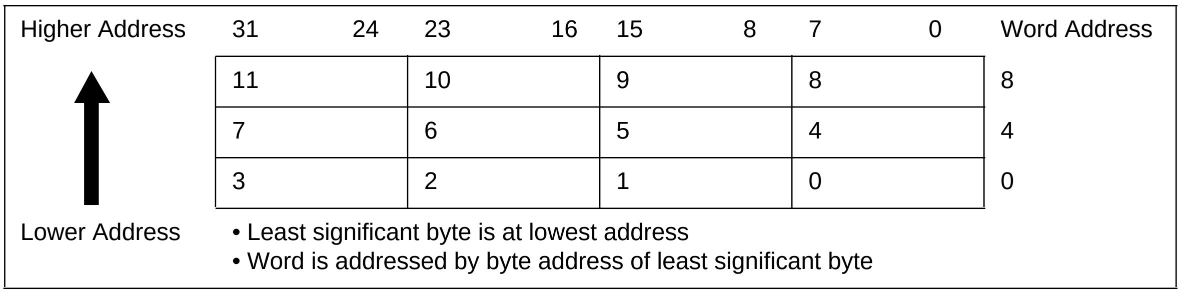
Higher Address	31 — 24	23 — 16	15 — 8	7 — 0	Word Address
	8	9	10	11	8
	4	5	6	7	4
Lower Address	0	1	2	3	0

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

Figure 3-1: Big endian addresses of bytes within words

3.3.2 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.



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

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

Figure 3-2: Little endian addresses of bytes within words

3.4 Instruction Length

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

3.5 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.

3.6 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.

3.7 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.

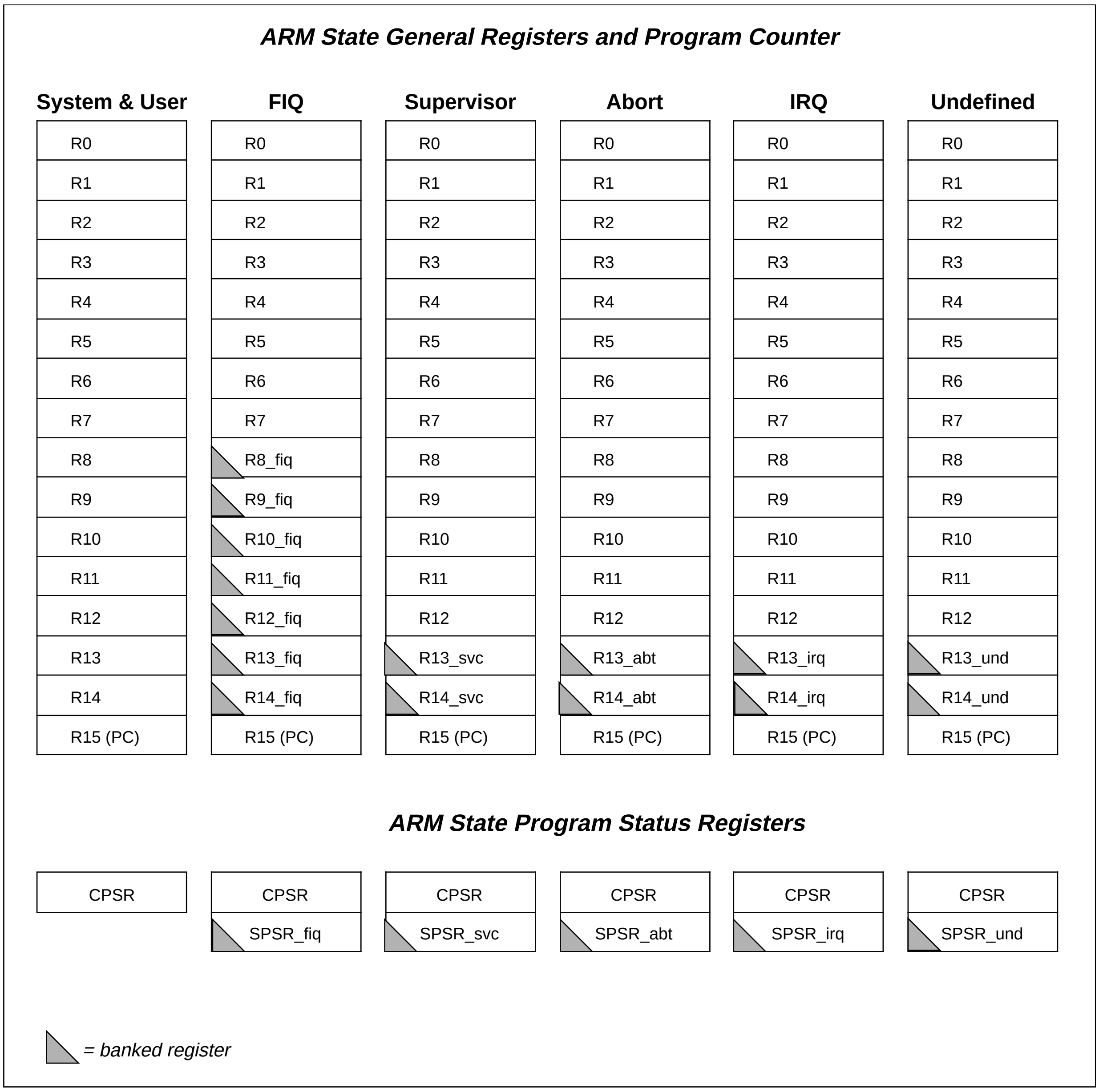
3.7.1 The ARM state register set

In ARM state, 16 general registers and one or two status registers are visible at any one time. In privileged (non-User) modes, mode-specific banked registers are switched in. *Figure 3-3: Register organization in ARM state* 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.



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*

Figure 3-3: Register organization in ARM state

3.7.2 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 3-4: Register organization in THUMB state*.

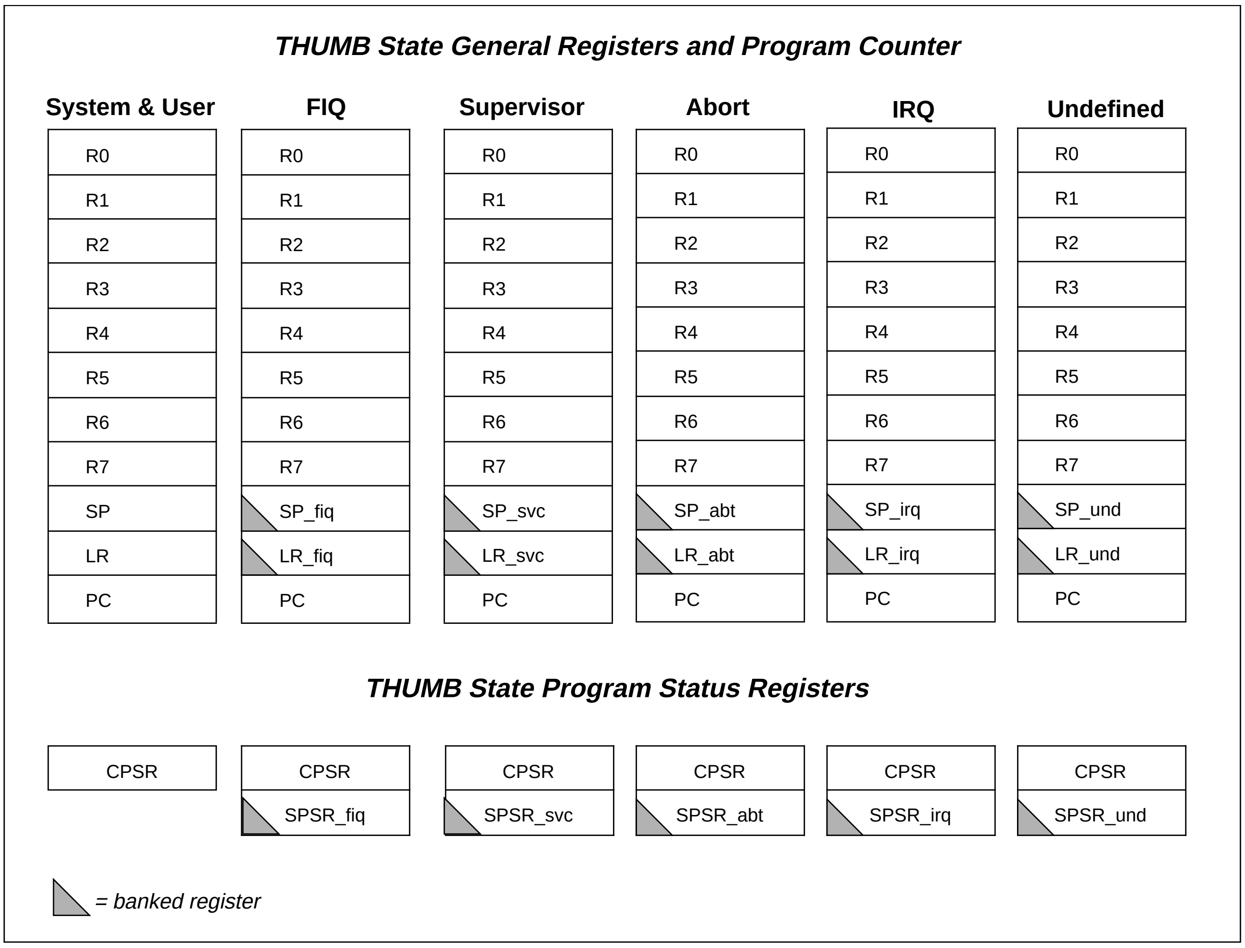


Figure 3-4: Register organization in THUMB state

3.7.3 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 3-5: Mapping of THUMB state registers onto ARM state registers*.

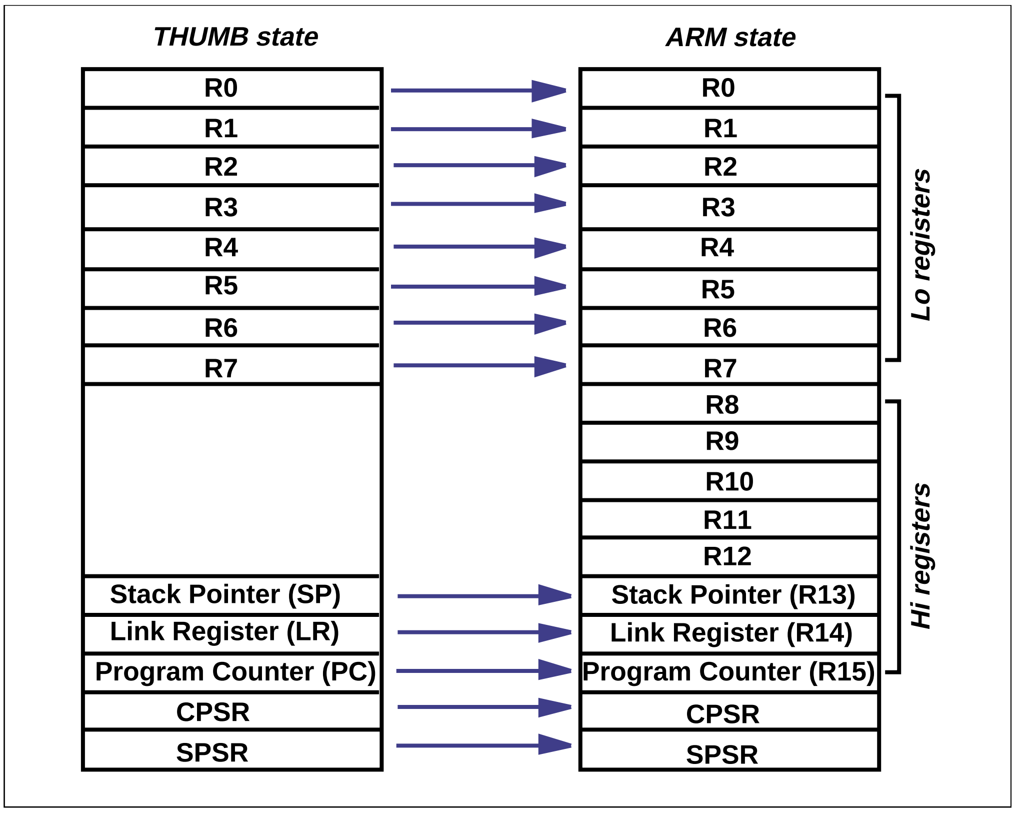


Figure 3-5: Mapping of THUMB state registers onto ARM state registers

3.7.4 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 *5.5 Format 5: Hi register operations/ branch exchange* on page 5-13.

3.8 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 3-6: Program status register format.*

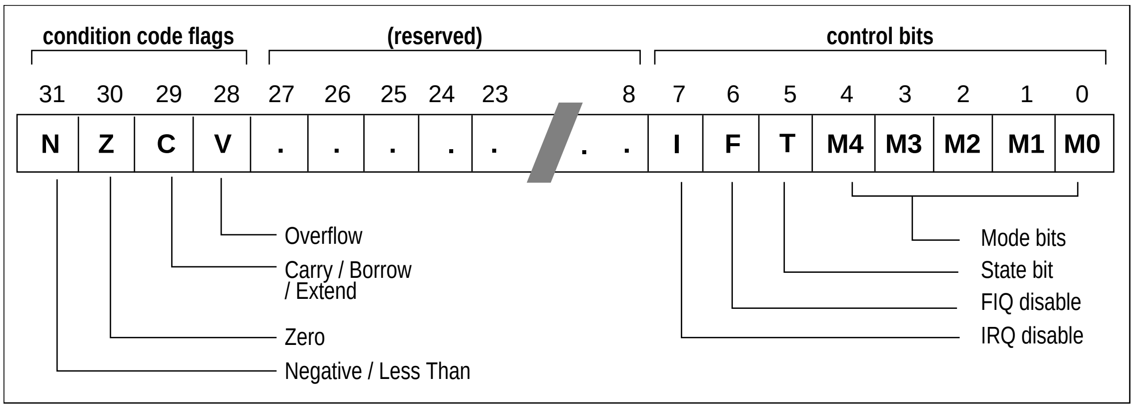


Figure 3-6: Program status register format

3.8.1 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 *4.2 The Condition Field* on page 4-5 for details.

In THUMB state, only the Branch instruction is capable of conditional execution: see *5.17 Format 17: software interrupt* on page 5-38

3.8.2 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 3-1: PSR mode bit values* on page 3-9. 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.

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

Table 3-1: PSR mode bit values

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.

3.9 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 *3.9.10 Exception priorities* on page 3-14.

3.9.1 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 3-2: Exception entry/exit* on page 3-11 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.

3.9.2 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.*

3.9.3 Exception entry/exit summary

➲*Table 3-2: Exception entry/exit* summarises the PC value preserved in the relevant R14 on exception entry, and the recommended instruction for exiting the exception handler.

	Return Instruction	**Previous State** ARM R14_x	**Previous State** THUMB R14_x	**Notes**
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

Table 3-2: Exception entry/exit

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.

3.9.4 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.

3.9.5 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
```

3.9.6 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.

3.9.7 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.

3.9.8 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.

3.9.9 Exception vectors

The following table shows the exception vector addresses.

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

Table 3-3: Exception vectors

3.9.10 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.

3.10 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.

3.11 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.

4 ARM Instruction Set

This chapter describes the ARM instruction set.

4.1 Instruction Set Summary

4.1.1 Format summary

The ARM instruction set formats are shown below.

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	

Figure 4-1: ARM instruction set formats

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.*

4.1.2 Instruction summary

Mnemonic	Instruction	Action	See Section:
ADC	Add with carry	Rd := Rn + Op2 + Carry	4.5
ADD	Add	Rd := Rn + Op2	4.5
AND	AND	Rd := Rn AND Op2	4.5
B	Branch	R15 := address	4.4
BIC	Bit Clear	Rd := Rn AND NOT Op2	4.5
BL	Branch with Link	R14 := R15, R15 := address	4.4
BX	Branch and Exchange	R15 := Rn, T bit := Rn[0]	4.3
CDP	Coprocesor Data Processing	(Coprocessor-specific)	4.14
CMN	Compare Negative	CPSR flags := Rn + Op2	4.5
CMP	Compare	CPSR flags := Rn - Op2	4.5
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)	4.5
LDC	Load coprocessor from memory	Coprocessor load	4.15
LDM	Load multiple registers	Stack manipulation (Pop)	4.11
LDR	Load register from memory	Rd := (address)	4.9, 4.10
MCR	Move CPU register to coprocessor register	cRn := rRn {<op>cRm}	4.16
MLA	Multiply Accumulate	Rd := (Rm * Rs) + Rn	4.7, 4.8
MOV	Move register or constant	Rd : = Op2	4.5
MRC	Move from coprocessor register to CPU register	Rn := cRn {<op>cRm}	4.16
MRS	Move PSR status/flags to register	Rn := PSR	4.6
MSR	Move register to PSR status/flags	PSR := Rm	4.6
MUL	Multiply	Rd := Rm * Rs	4.7, 4.8
MVN	Move negative register	Rd := 0xFFFFFFFF EOR Op2	4.5
ORR	OR	Rd := Rn OR Op2	4.5

Table 4-1: The ARM Instruction set

Mnemonic	Instruction	Action	See Section:
RSB	Reverse Subtract	Rd := Op2 - Rn	4.5
RSC	Reverse Subtract with Carry	Rd := Op2 - Rn - 1 + Carry	4.5
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry	4.5
STC	Store coprocessor register to memory	address := CRn	4.15
STM	Store Multiple	Stack manipulation (Push)	4.11
STR	Store register to memory	<address> := Rd	4.9, 4.10
SUB	Subtract	Rd := Rn - Op2	4.5
SWI	Software Interrupt	OS call	4.13
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm	4.12
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2	4.5
TST	Test bits	CPSR flags := Rn AND Op2	4.5

Table 4-1: The ARM Instruction set (Continued)

4.2 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 4-2: Condition code summary*. 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.

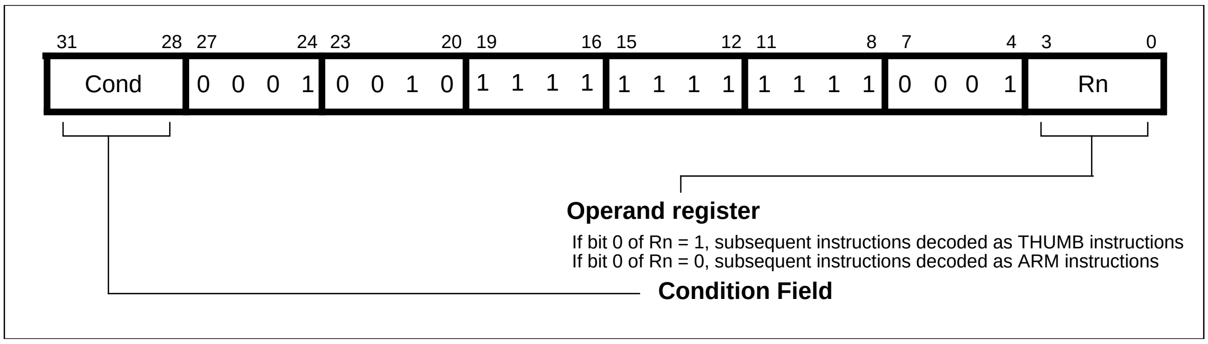
Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear 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

Table 4-2: Condition code summary

4.3 Branch and Exchange (BX)

This instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5.

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.



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

Figure 4-2: Branch and Exchange instructions

4.3.1 Instruction cycle times

The BX instruction takes 2S + 1N cycles to execute, where S and N are as defined in *6.2 Cycle Types* on page 6-2.

4.3.2 Assembler syntax

BX - branch and exchange.

```
BX{cond} Rn
```

{cond}	Two character condition mnemonic. See *Table 4-2: Condition code summary* on page 4-5.
Rn	is an expression evaluating to a valid register number.

4.3.3 Using R15 as an operand

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

4.3.4 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

      .
      .
```

4.4 Branch and Branch with Link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-3: Branch instructions*, below.

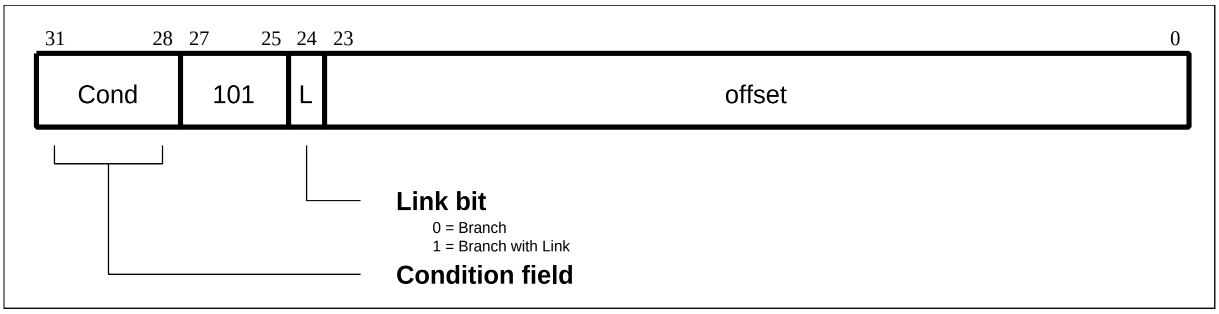


Figure 4-3: Branch instructions

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.

4.4.1 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.

4.4.2 Instruction cycle times

Branch and Branch with Link instructions take 2S + 1N incremental cycles, where S and N are as defined in *6.2 Cycle Types* on page 6-2.

4.4.3 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 4-2: Condition code summary* on page 4-5. If absent then AL (ALways) will be used.
<expression>	is the destination. The assembler calculates the offset.

4.4.4 Examples

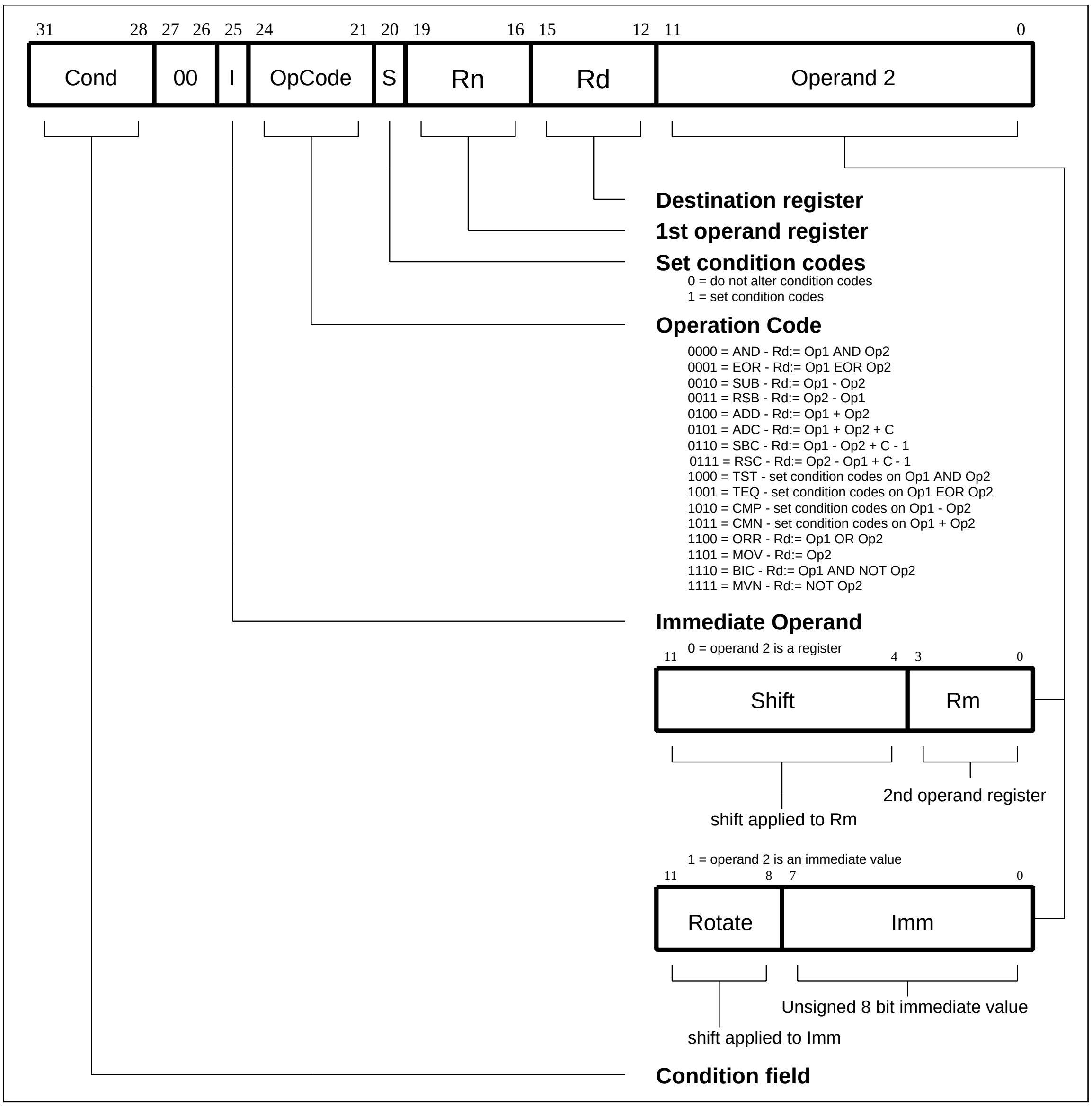
```
here  BAL   here     ; assembles to 0xEAFFFFFE (note effect of
                     ; PC offset).
      B     there    ; Always condition used as default.
      CMP   R1,#0    ; Compare R1 with zero and branch to fred
                     ; if R1 was zero, otherwise continue
      BEQ   fred    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

4.5 Data Processing

The data processing instruction is only executed if the condition is true. The conditions are defined in *Table 4-2: Condition code summary* on page 4-5.

The instruction encoding is shown in *Figure 4-4: Data processing instructions* below.



```
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
```

Figure 4-4: Data processing instructions

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 4-3: ARM Data processing instructions* on page 4-11**.**

4.5.1 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.

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)

Table 4-3: ARM Data processing instructions

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).

4.5.2 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 4-5: ARM shift operations*.

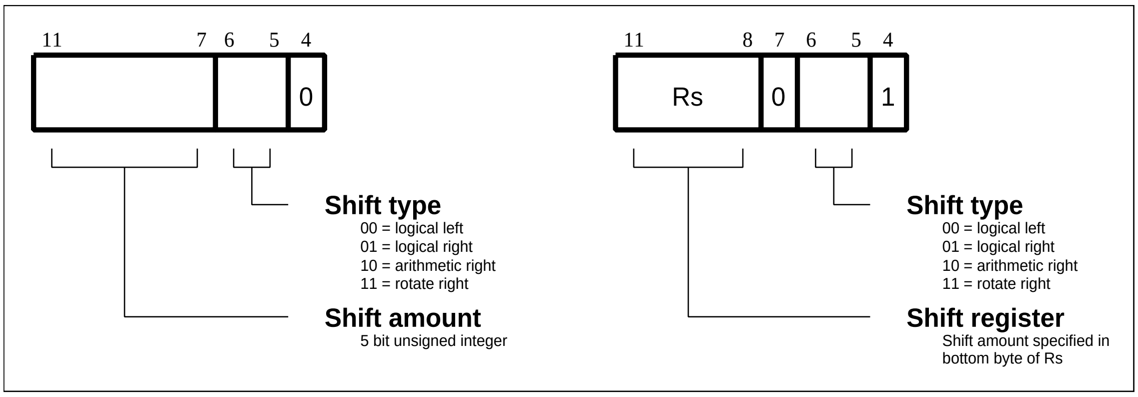


Figure 4-5: ARM shift operations

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 4-6: Logical shift left*.

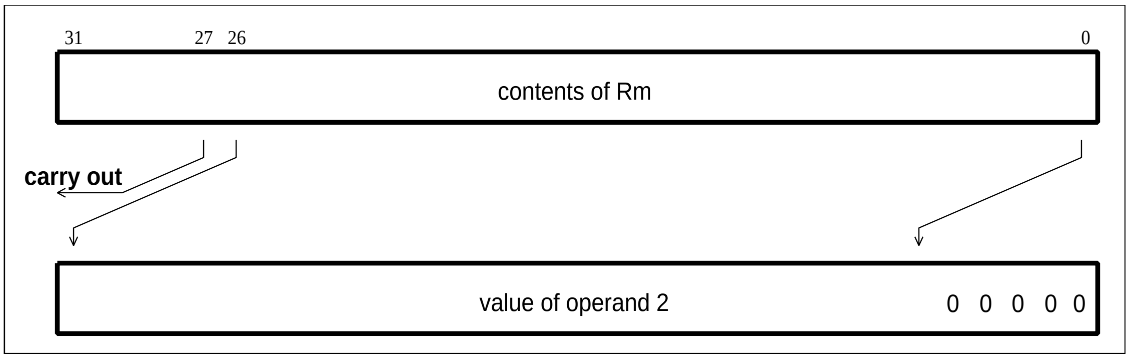


Figure 4-6: Logical shift left

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 4-7: Logical shift right*.

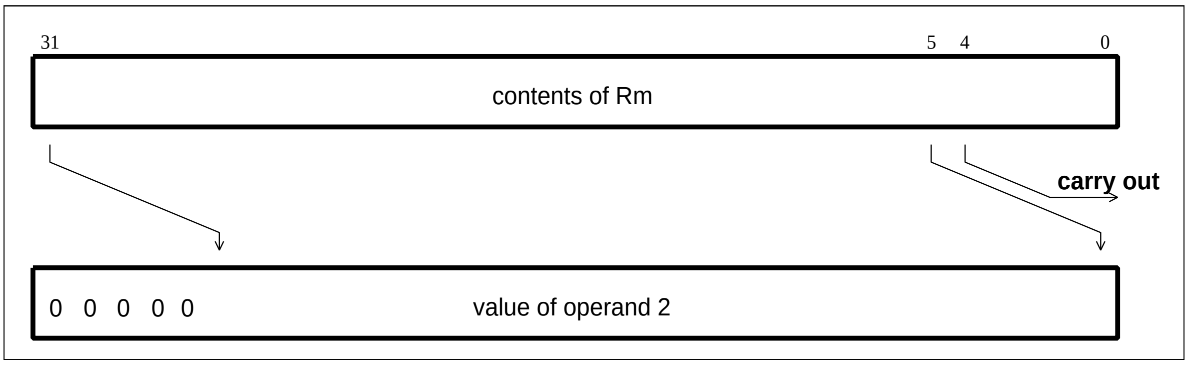


Figure 4-7: Logical shift right

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 4-8: Arithmetic shift right*.

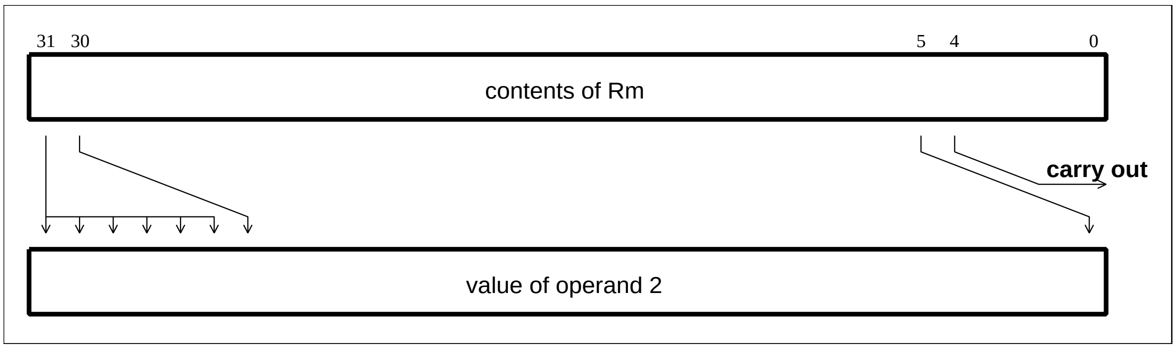


Figure 4-8: Arithmetic shift right

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 “overshoot” 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 4-9: Rotate right* on page 4-14.

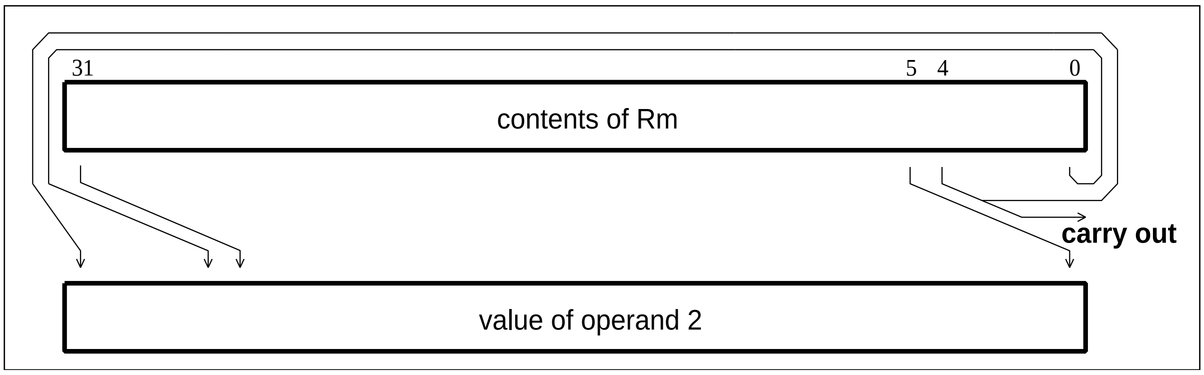


Figure 4-9: Rotate right

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 4-10: Rotate right extended*.

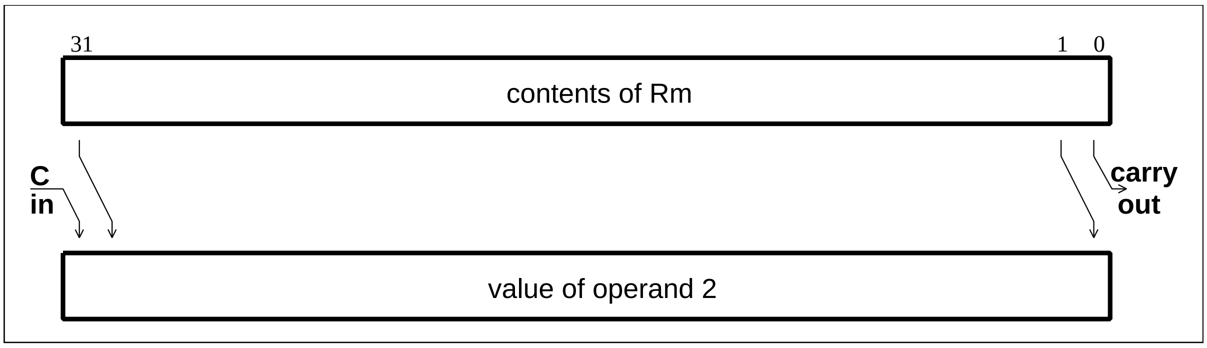


Figure 4-10: Rotate right extended

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.

4.5.3 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.

4.5.4 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.

4.5.5 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.

4.5.6 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.

4.5.7 Instruction cycle times

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

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

Table 4-4: Incremental cycle times

S, N and I are as defined in *6.2 Cycle Types* on page 6-2.

4.5.8 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 4-2: Condition code summary* on page 4-5.
{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.)

4.5.9 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.
```

4.6 PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5.

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 4-11: PSR transfer* on page 4-19.

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.

4.6.1 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.

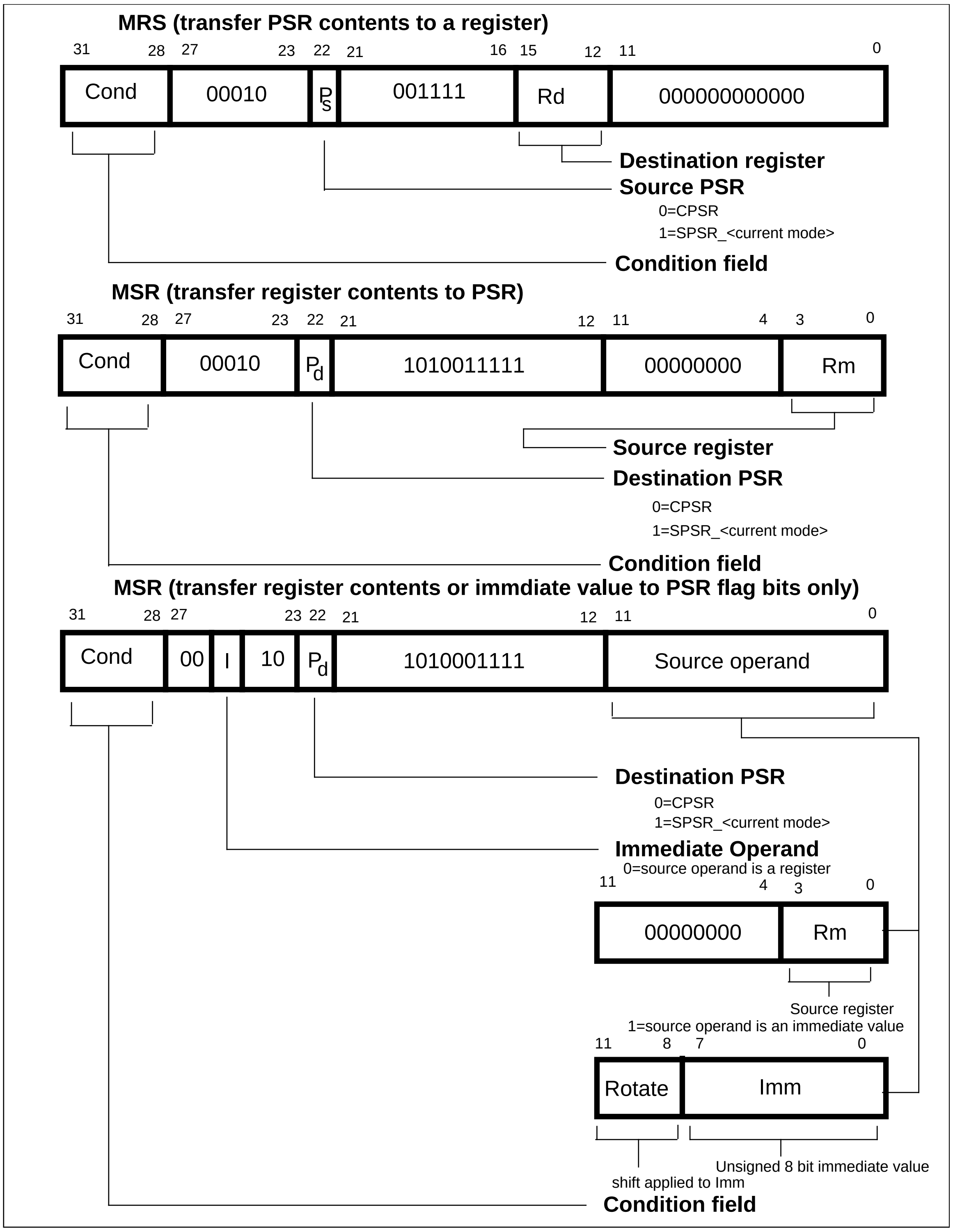


Figure 4-11: PSR transfer

4.6.2 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 3-6: Program status register format* on page 3-8 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.

4.6.3 Instruction cycle times

PSR Transfers take 1S incremental cycles, where S is as defined in *6.2 Cycle Types* on page 6-2.

4.6.4 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 4-2: Condition code summary* on page 4-5.
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.

4.6.5 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]
```

4.7 Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-12: Multiply instructions*.

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

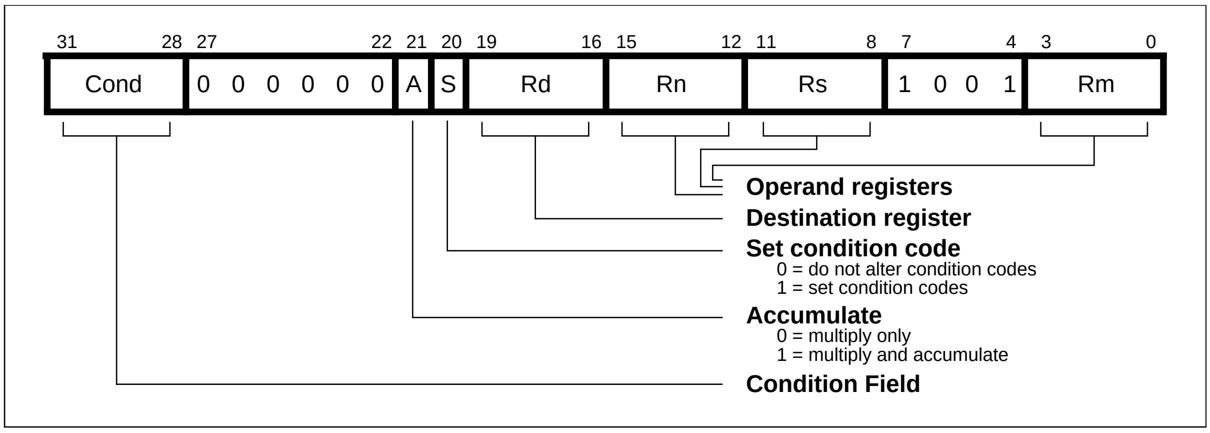


Figure 4-12: Multiply instructions

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	0x0000001	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.

Open Access

4.7.1 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.

4.7.2 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.

4.7.3 Instruction cycle times

MUL takes 1S + mI and MLA 1S + (m+1)I cycles to execute, where S and I are as defined in ➲*6.2 Cycle Types* on page 6-2.

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.

4.7.4 Assembler syntax

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

{cond}	two-character condition mnemonic. See ➲*Table 4-2: Condition code summary* on page 4-5.
{S}	set condition codes if S present
Rd, Rm, Rs and Rn	are expressions evaluating to a register number other than R15.

4.7.5 Examples

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

4.8 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 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-13: Multiply long instructions*.

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.

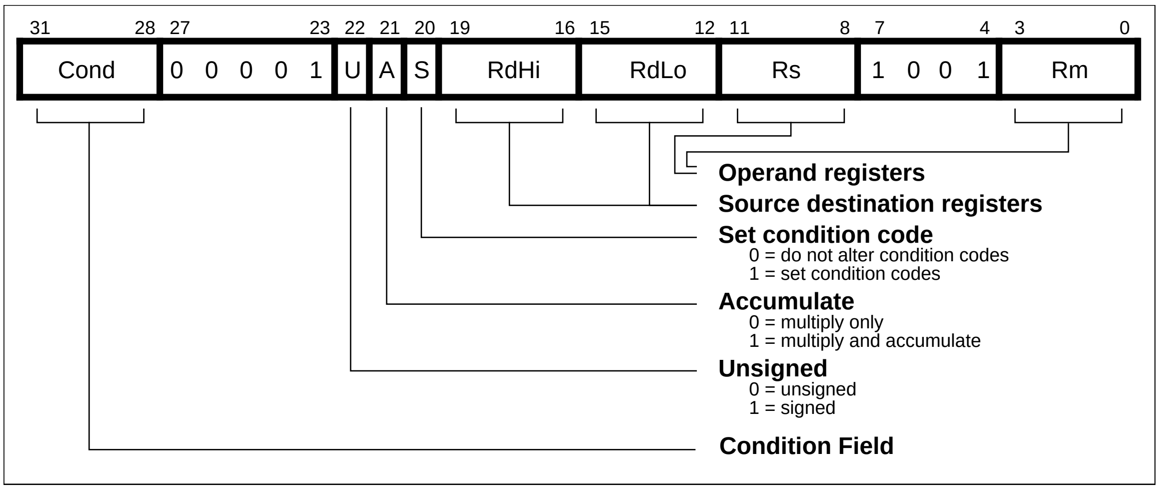


Figure 4-13: Multiply long instructions

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.

4.8.1 Operand restrictions

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

4.8.2 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.

4.8.3 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 ➲*6.2 Cycle Types* on page 6-2.

4.8.4 Assembler syntax

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

Table 4-5: Assembler syntax descriptions

where:

{cond}	two-character condition mnemonic. See *Table 4-2: Condition code summary* on page 4-5.
{S}	set condition codes if S present
RdLo, RdHi, Rm, Rs	are expressions evaluating to a register number other than R15.

4.8.5 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
```

4.9 Single Data Transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-14: Single data transfer instructions* on page 4-28.

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.

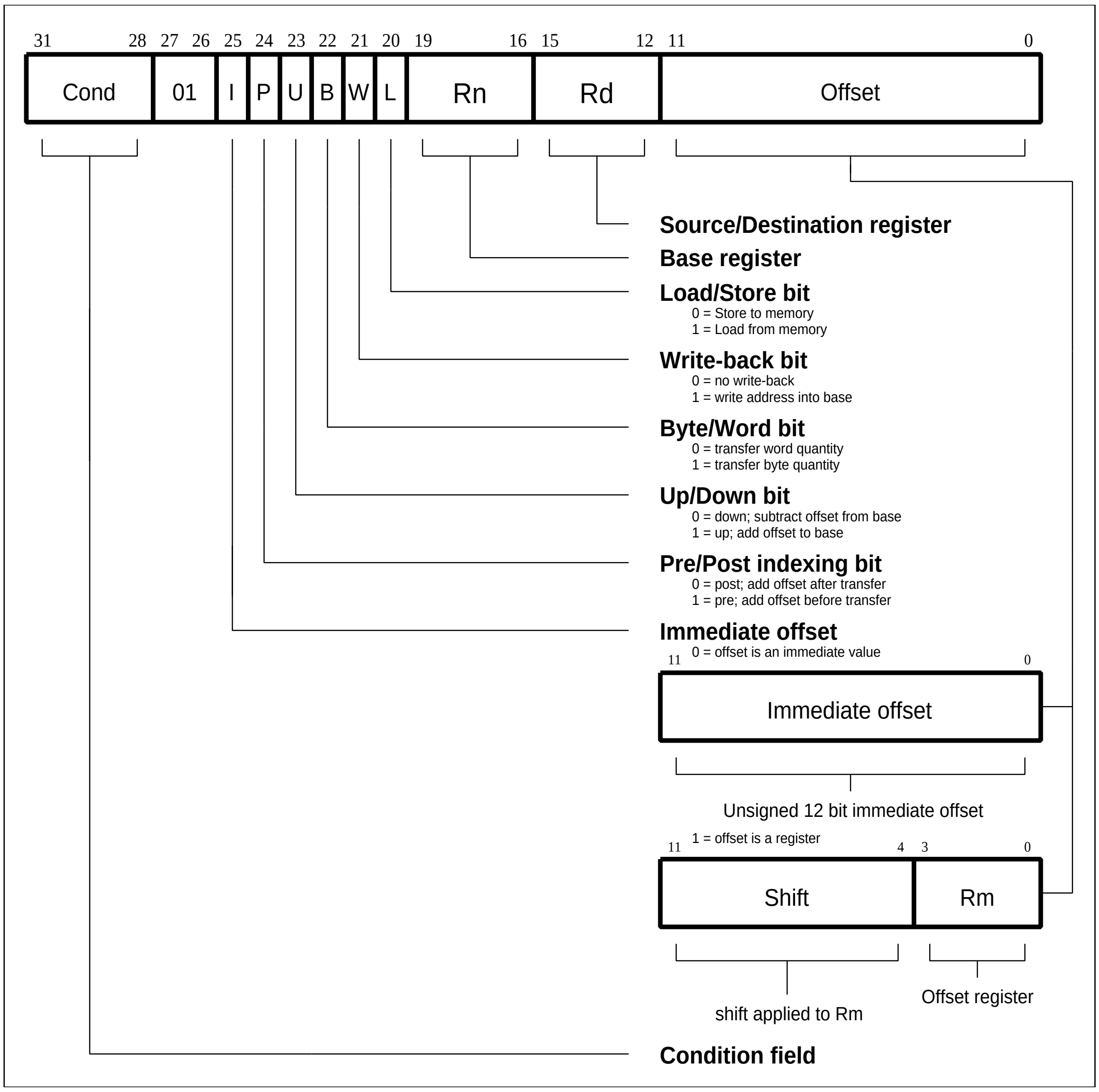


Figure 4-14: Single data transfer instructions

4.9.1 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.

4.9.2 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 *4.5.2 Shifts* on page 4-12.

4.9.3 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 3-2: Little endian addresses of bytes within words* on page 3-3.

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 4-15: Little endian offset addressing* on page 4-30.

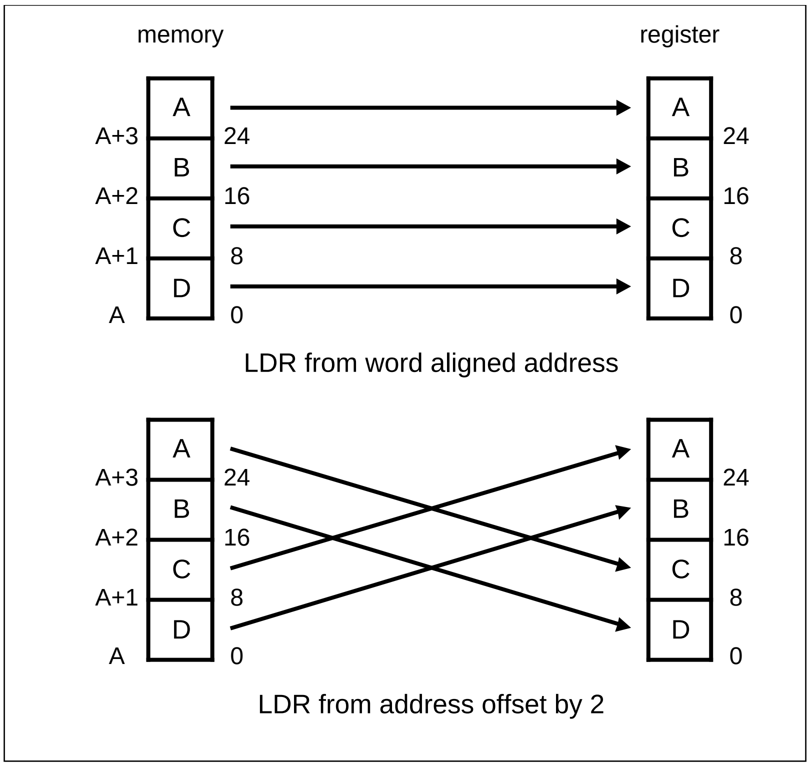


Figure 4-15: Little endian offset addressing

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 3-1: Big endian addresses of bytes within words* on page 3-3.

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.

4.9.4 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.

4.9.5 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:

```
    LDR   R0,[R1],R1
```

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

4.9.6 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.

4.9.7 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 ➲*6.2 Cycle Types* on page 6-2.

STR instructions take 2N incremental cycles to execute.

4.9.8 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 4-2: Condition code summary* on page 4-5.
{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.

4.9.9 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
```

4.10 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 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-16: Halfword and signed data transfer with register offset*, below, and *Figure 4-17: Halfword and signed data transfer with immediate offset* on page 4-35.

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.

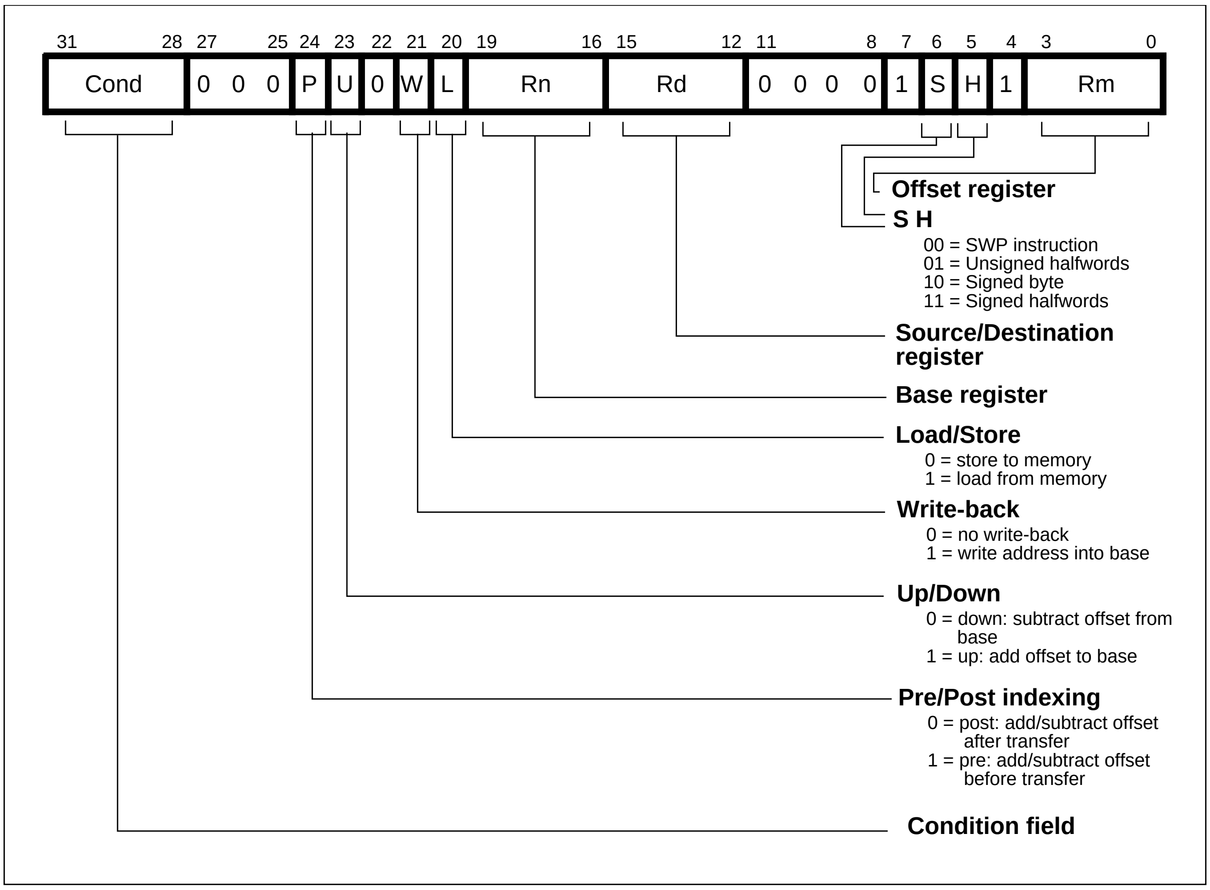


Figure 4-16: Halfword and signed data transfer with register offset

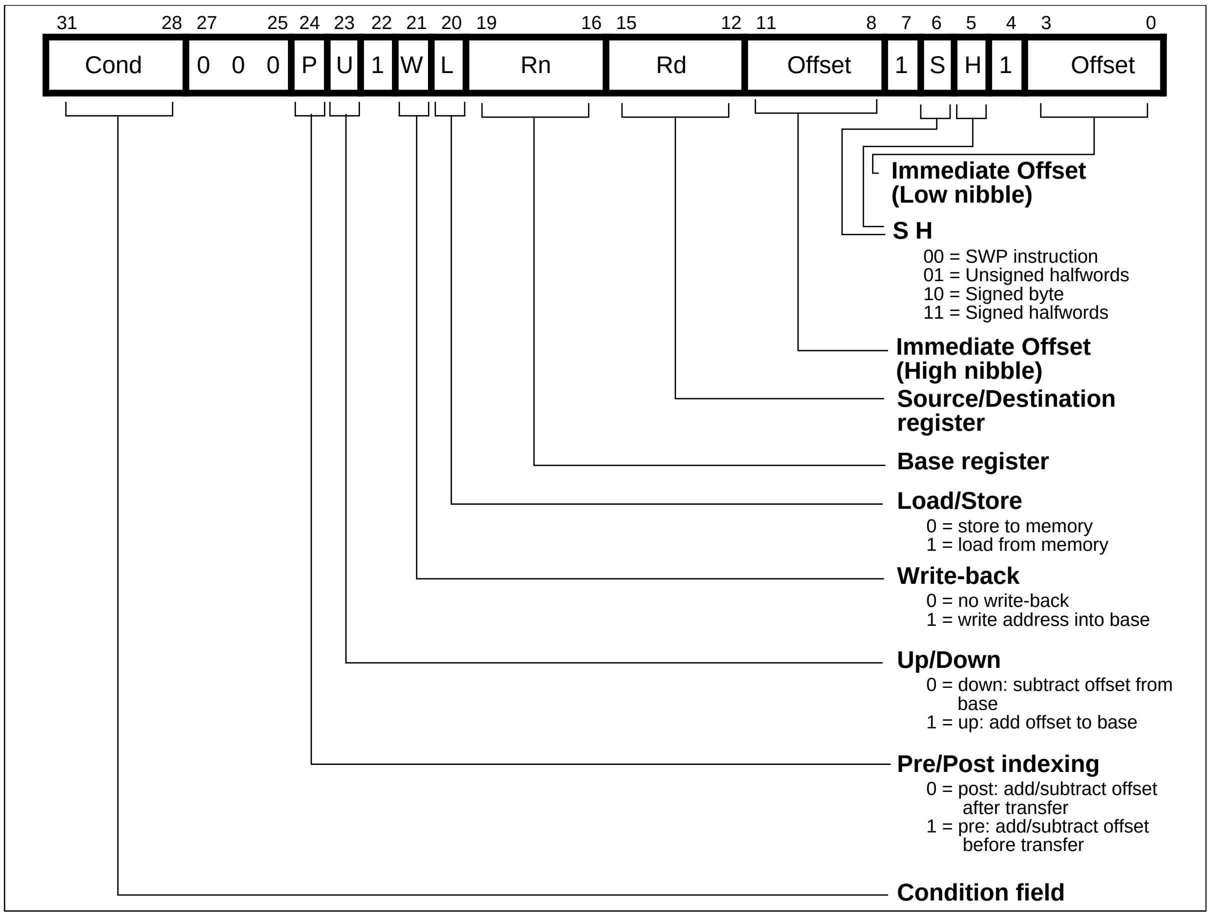


Figure 4-17: Halfword and signed data transfer with immediate offset

4.10.1 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.

4.10.2 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.

4.10.3 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.

4.10.4 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 3-2: Little endian addresses of bytes within words* on page 3-3

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 3-1: Big endian addresses of bytes within words* on page 3-3

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.

4.10.5 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.

4.10.6 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.

4.10.7 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 *6.2 Cycle Types* on page 6-2.

STRH instructions take 2N incremental cycles to execute.

4.10.8 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 4-2: Condition code summary* on page 4-5.
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.

4.10.9 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.
LDRNESH  R11,[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
```

4.11 Block Data Transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-18: Block data transfer instructions*.

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.

4.11.1 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.

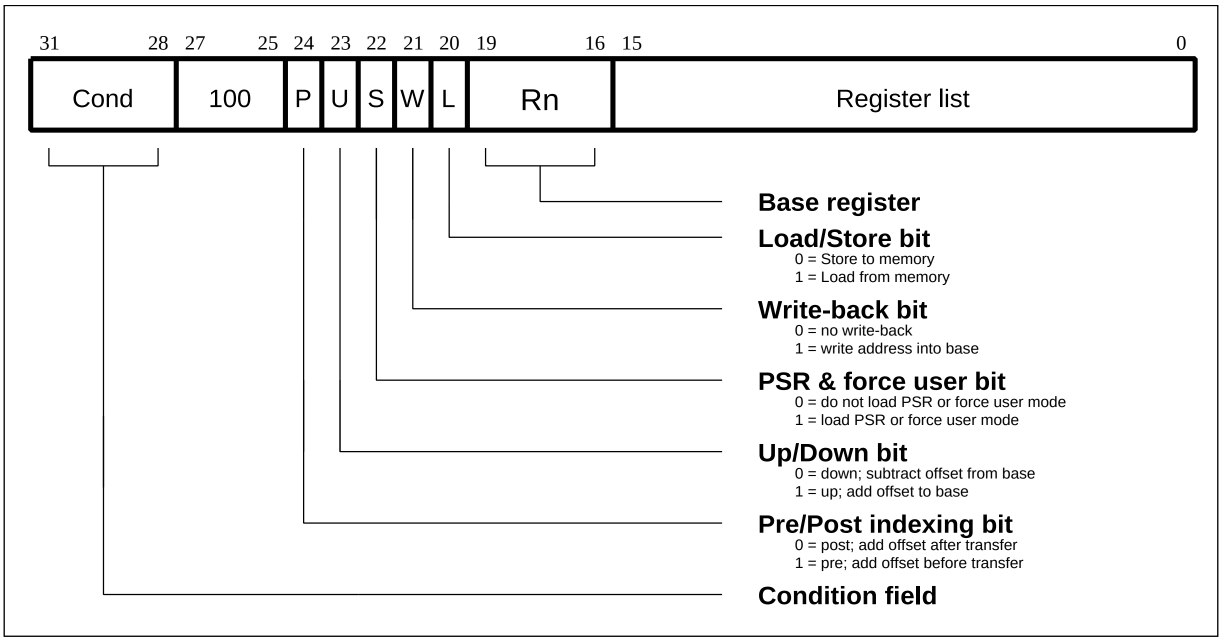

31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down 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

Figure 4-18: Block data transfer instructions

4.11.2 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 4-19: Post-increment addressing*, *Figure 4-20: Pre-increment addressing*, *Figure 4-21: Post-decrement addressing* and *Figure 4-22: Pre-decrement addressing* 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.

4.11.3 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.

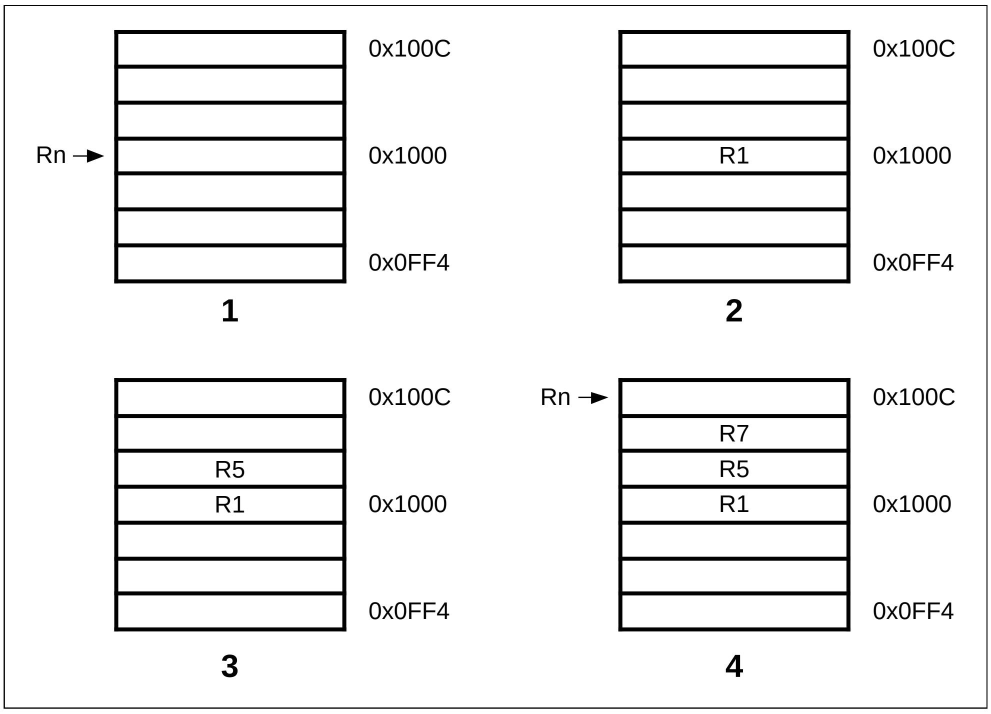


Figure 4-19: Post-increment addressing

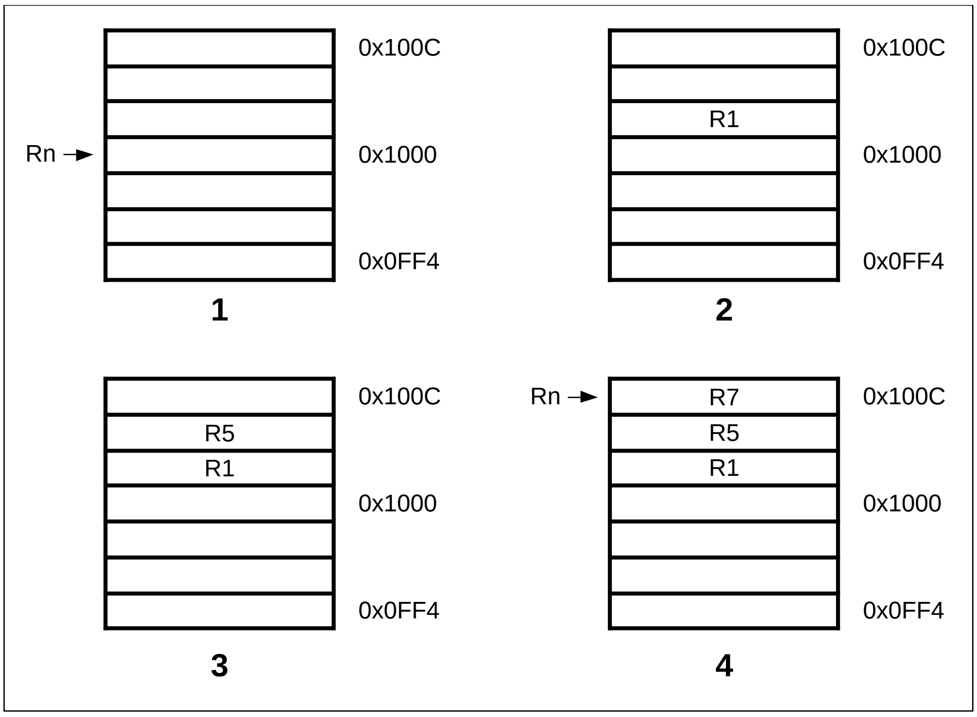


Figure 4-20: Pre-increment addressing

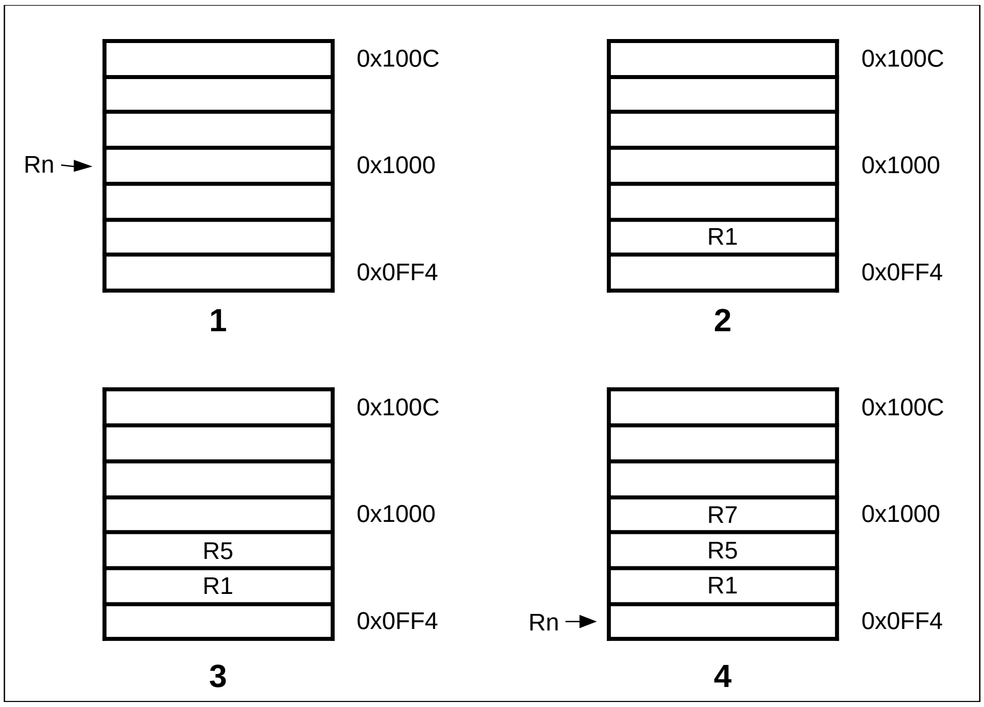


Figure 4-21: Post-decrement addressing

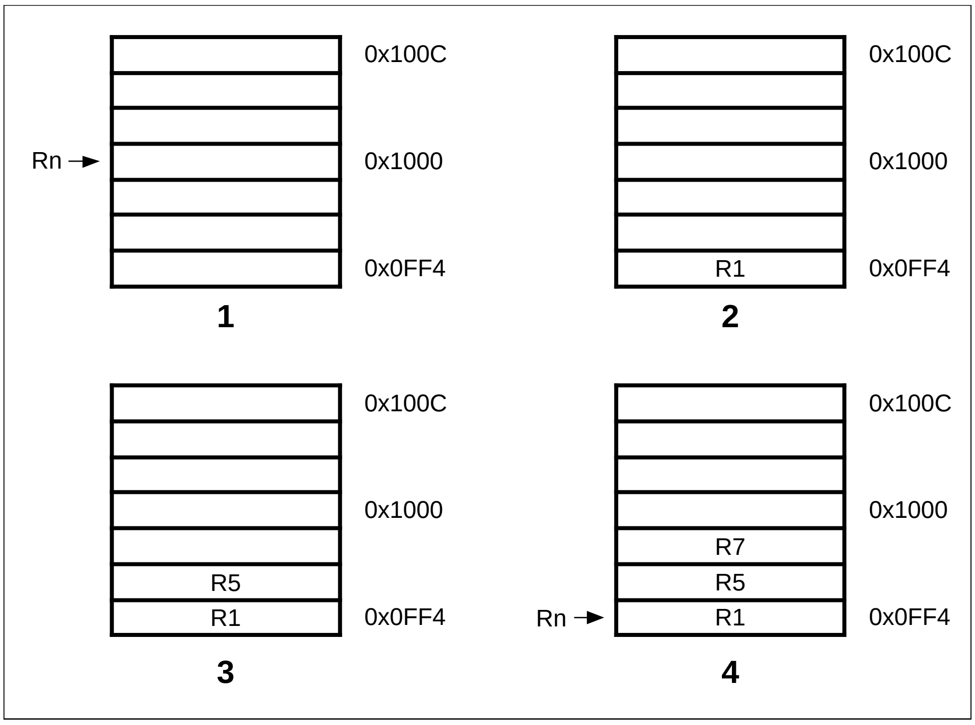


Figure 4-22: Pre-decrement addressing

4.11.4 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).

4.11.5 Use of R15 as the base

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

4.11.6 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.

4.11.7 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.

4.11.8 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 ➲*6.2 Cycle Types* on page 6-2. STM instructions take (n-1)S + 2N incremental cycles to execute, where *n* is the number of words transferred.

4.11.9 Assembler syntax

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

where:

{cond}	two character condition mnemonic. See *Table 4-2: Condition code summary* on page 4-5.
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:

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

Table 4-6: Addressing mode names

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.

4.11.10 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.
```

4.12 Single Data Swap (SWP)

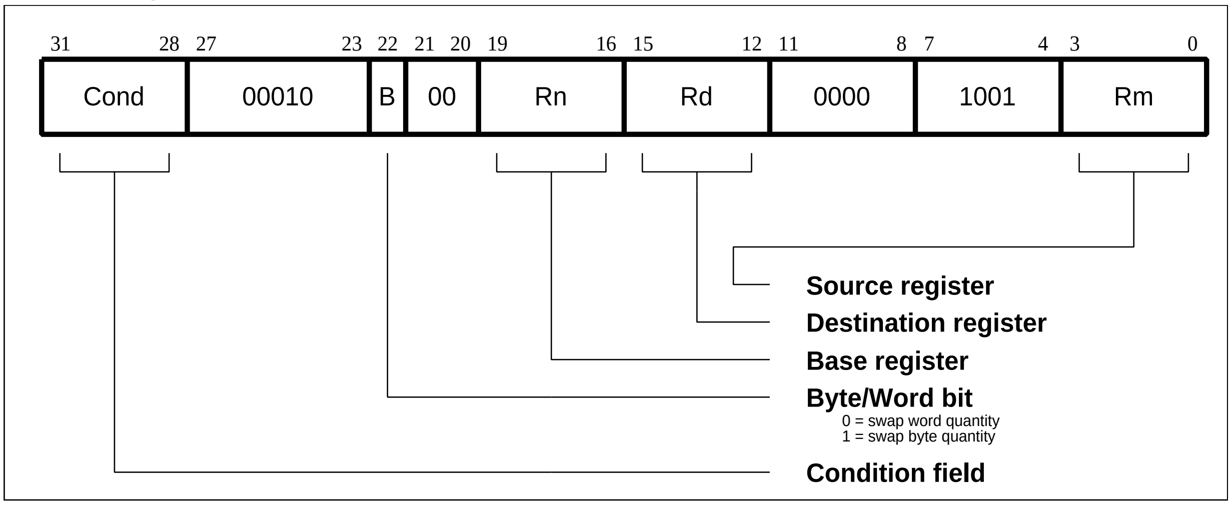


Figure 4-23: Swap instruction

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-23: Swap instruction*.

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 multi-processor 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.

4.12.1 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.

4.12.2 Use of R15

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

4.12.3 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.

4.12.4 Instruction cycle times

Swap instructions take 1S + 2N +1I incremental cycles to execute, where S,N and I are as defined in *6.2 Cycle Types* on page 6-2.

4.12.5 Assembler syntax

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

{cond}	two-character condition mnemonic. See *Table 4-2: Condition code summary* on page 4-5.
{B}	if B is present then byte transfer, otherwise word transfer
Rd,Rm,Rn	are expressions evaluating to valid register numbers

4.12.6 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.
```

4.13 Software Interrupt (SWI)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-24: Software interrupt instruction*, below.

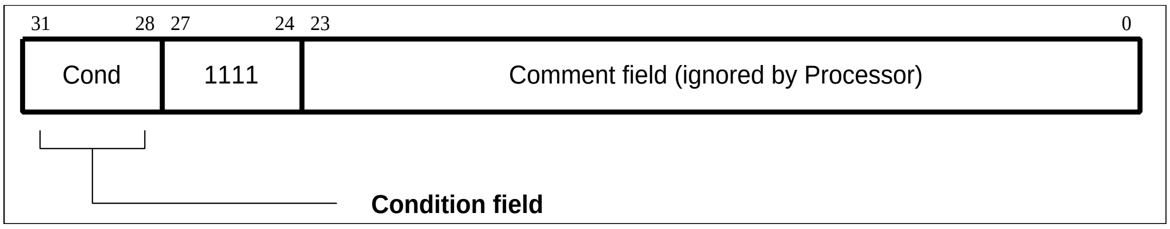


Figure 4-24: Software interrupt instruction

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.

4.13.1 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.

4.13.2 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.

4.13.3 Instruction cycle times

Software interrupt instructions take 2S + 1N incremental cycles to execute, where S and N are as defined in *6.2 Cycle Types* on page 6-2.

4.13.4 Assembler syntax

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

{cond}	two character condition mnemonic, *Table 4-2: Condition code summary* on page 4-5.
<expression>	is evaluated and placed in the comment field (which is ignored by ARM7TDMI).

4.13.5 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.
```

4.14 Coprocessor Data Operations (CDP)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-25: Coprocessor data operation instruction*.

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.

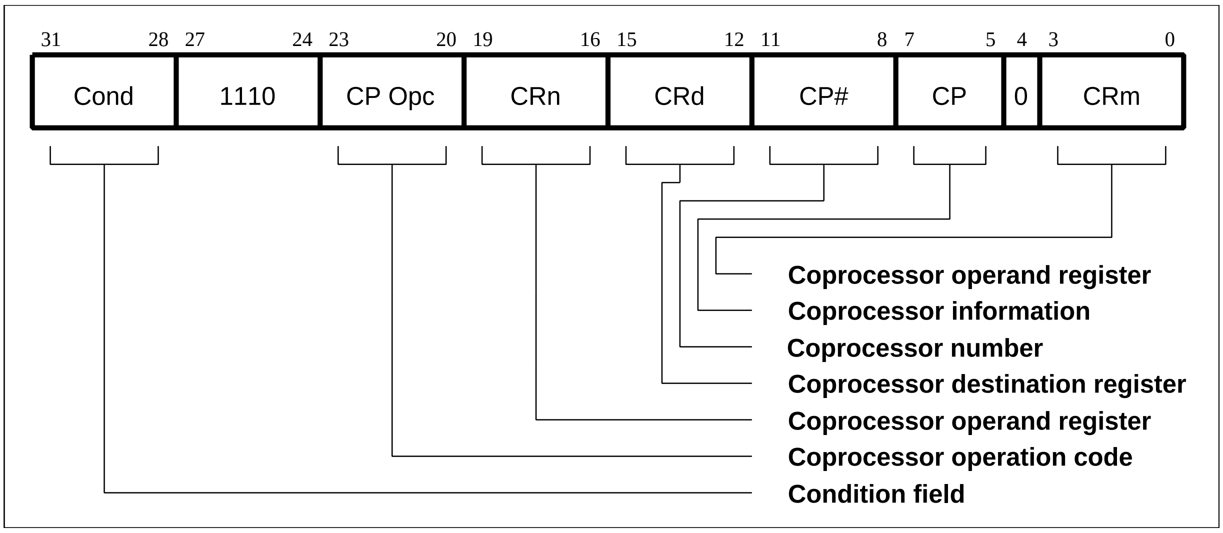


Figure 4-25: Coprocessor data operation instruction

4.14.1 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.

4.14.2 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 *6.2 Cycle Types* on page 6-2.

4.14.3 Assembler syntax

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

{cond}	two character condition mnemonic. See *Table 4-2: Condition code summary* on page 4-5.
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

4.14.4 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.
```

4.15 Coprocessor Data Transfers (LDC, STC)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-26: Coprocessor data transfer instructions*.

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.

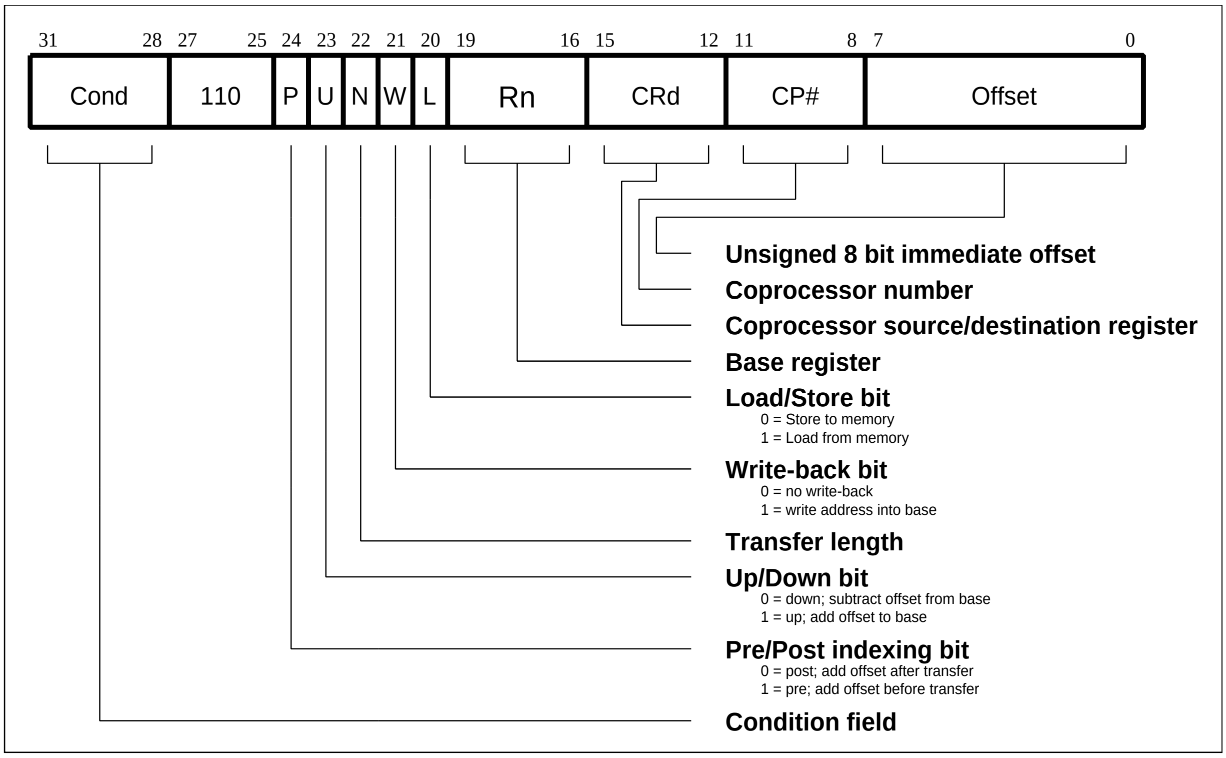


Figure 4-26: Coprocessor data transfer instructions

4.15.1 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.

4.15.2 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.

4.15.3 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.

4.15.4 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.

4.15.5 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.

4.15.6 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 ➲*6.2 Cycle Types* on page 6-2.

4.15.7 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 4-2: Condition code summary* on page 4-5.
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.*

4.15.8 Examples

```
LDC    p1,c2,table    ; Load c2 of coproc 1 from address
                      ; table, using a PC relative address.
STCEQL p2,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.*

4.16 Coprocessor Register Transfers (MRC, MCR)

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction encoding is shown in *Figure 4-27: Coprocessor register transfer instructions*.

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.

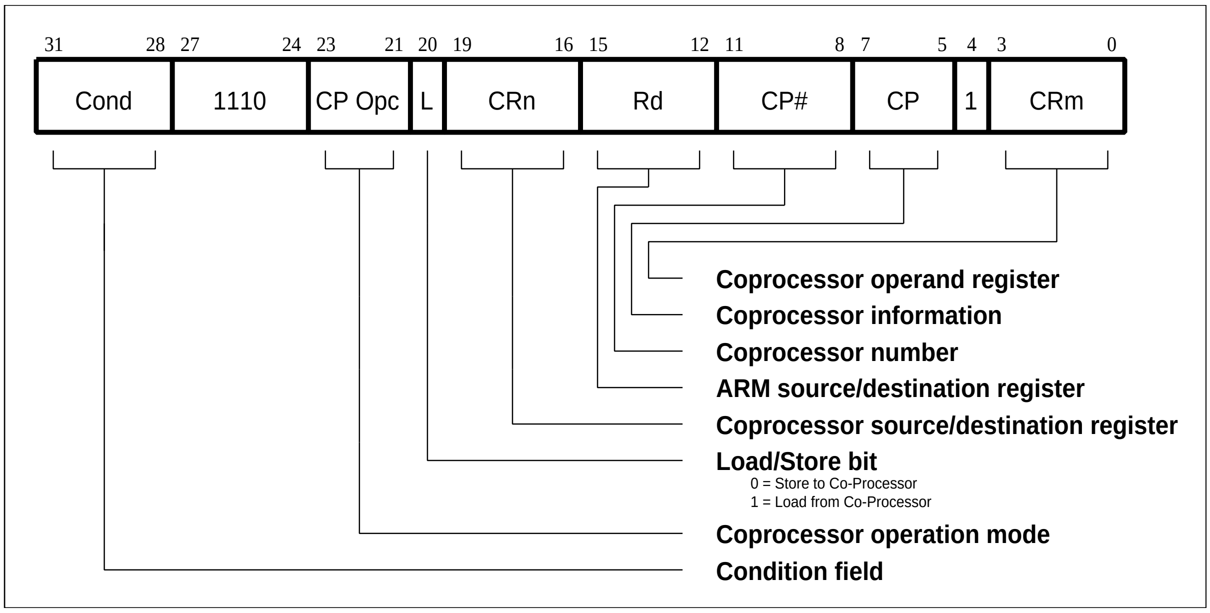

31 28 27 24 23 21 20 19 16 15 12 11 8 7 5 4 3 0
Cond
1110
CP Opc
L
CRn
Rd
CP#
CP
1
CRm
Coprocessor operand register
Coprocessor information
Coprocessor number
ARM source/destination register
Coprocessor source/destination register
Load/Store bit
0 = Store to Co-Processor
1 = Load from Co-Processor
Coprocessor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

Figure 4-27: Coprocessor register transfer instructions

4.16.1 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.

4.16.2 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.

4.16.3 Transfers from R15

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

4.16.4 Instruction cycle times

MRC instructions take 1S + (b+1)I +1C incremental cycles to execute, where S, I and C are as defined in ➲*6.2 Cycle Types* on page 6-2.

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

4.16.5 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 4-2: Condition code summary* on page 4-5.
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

4.16.6 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.
```

4.17 Undefined Instruction

The instruction is only executed if the condition is true. The various conditions are defined in *Table 4-2: Condition code summary* on page 4-5. The instruction format is shown in *Figure 4-28: Undefined instruction*.

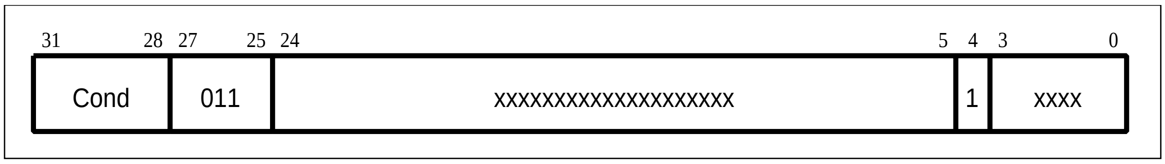


Figure 4-28: Undefined instruction

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.

4.17.1 Instruction cycle times

This instruction takes 2S + 1I + 1N cycles, where S, N and I are as defined in *6.2 Cycle Types* on page 6-2.

4.17.2 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.

4.18 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.

4.18.1 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
```

Open Access

```
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.

4.18.2 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
```

4.18.3 Multiplication by constant using the barrel shifter

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

```
MOV   Ra, 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..)

```
RSB   Ra,Ra,Ra,LSL #n
```

Multiplication by 6

```
ADD   Ra,Ra,Ra,LSL #1; multiply by 3
MOV   Ra,Ra,LSL#1; and then by 2
```

Multiply by 10 and add in extra number

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

4.18.4 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
```

5 THUMB Instruction Set

This chapter describes the THUMB instruction set.

Format Summary

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

	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	

Figure 5-1: THUMB instruction set formats

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.

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set	See Section:
ADC	Add with Carry	✔		✔	5.4
ADD	Add	✔	✔	✔①	5.1.3, 5.5, 5.12, 5.13
AND	AND	✔		✔	5.4
ASR	Arithmetic Shift Right	✔		✔	5.1, 5.4
B	Unconditional branch	✔			5.16
B*xx*	Conditional branch	✔			5.17
BIC	Bit Clear	✔		✔	5.4
BL	Branch and Link				5.19
BX	Branch and Exchange	✔	✔		5.5
CMN	Compare Negative	✔		✔	5.4
CMP	Compare	✔	✔	✔	5.3, 5.4, 5.5
EOR	EOR	✔		✔	5.4
LDMIA	Load multiple	✔			5.15
LDR	Load word	✔			5.7, 5.6, 5.9, 5.11
LDRB	Load byte	✔			5.7, 5.9
LDRH	Load halfword	✔			5.8, 5.10
LSL	Logical Shift Left	✔		✔	5.1, 5.4
LDSB	Load sign-extended byte	✔			5.8
LDSH	Load sign-extended halfword	✔			5.8
LSR	Logical Shift Right	✔		✔	5.1, 5.4
MOV	Move register	✔	✔	✔②	5.3, 5.5
MUL	Multiply	✔		✔	5.4
MVN	Move Negative register	✔		✔	5.4

Table 5-1: THUMB instruction set opcodes

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set	See Section:
NEG	Negate	✔		✔	5.4
ORR	OR	✔		✔	5.4
POP	Pop registers	✔			5.14
PUSH	Push registers	✔			5.14
ROR	Rotate Right	✔		✔	5.4
SBC	Subtract with Carry	✔		✔	5.4
STMIA	Store Multiple	✔			5.15
STR	Store word	✔			5.7, 5.9, 5.11
STRB	Store byte	✔			5.7
STRH	Store halfword	✔			5.8, 5.10
SWI	Software Interrupt				5.17
SUB	Subtract	✔		✔	5.1.3, 5.3
TST	Test bits	✔		✔	5.4

Table 5-1: THUMB instruction set opcodes (Continued)

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.

5.1 Format 1: move shifted register

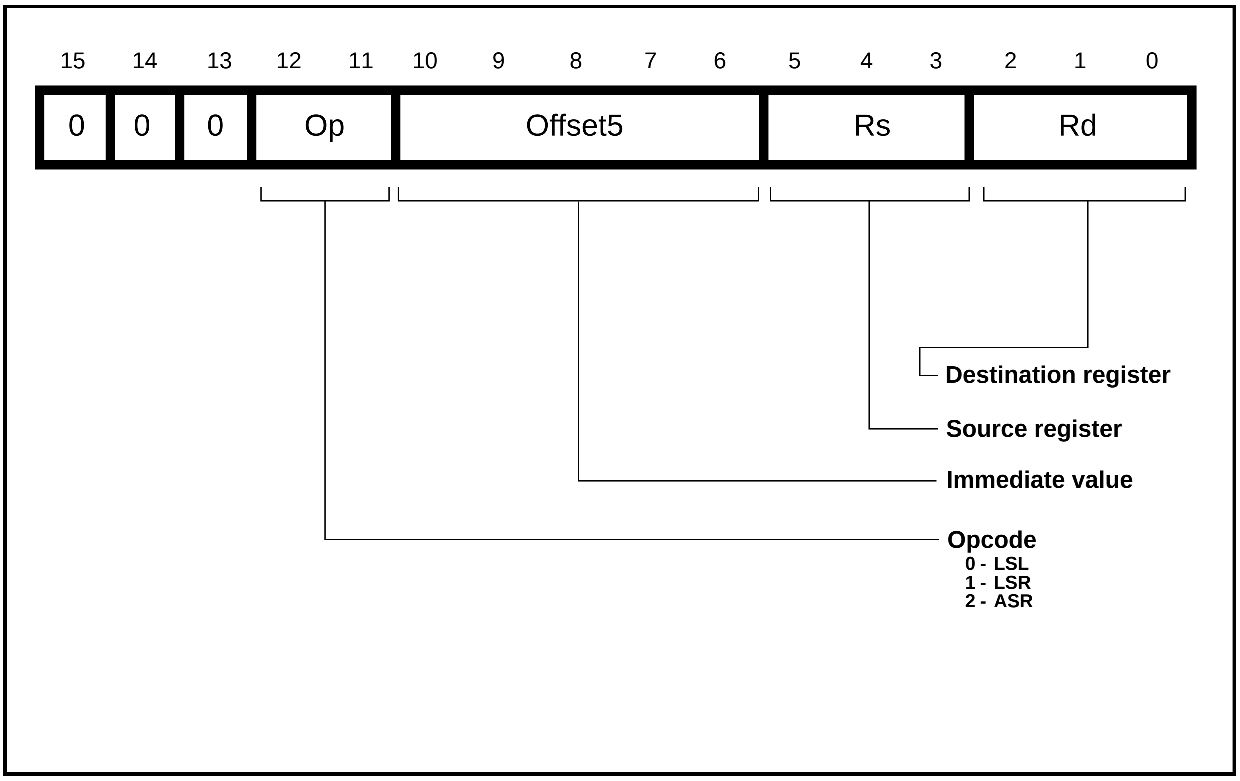


Figure 5-2: Format 1

5.1.1 Operation

These instructions move a shifted value between Lo registers. The THUMB assembler syntax is shown in *Table 5-2: Summary of format 1 instructions*.

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

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.

Table 5-2: Summary of format 1 instructions

5.1.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-2: Summary of format 1 instructions* on page 5-5. 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 *Chapter 10, Instruction Cycle Operations*.

5.1.3 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.
```

5.2 Format 2: add/subtract

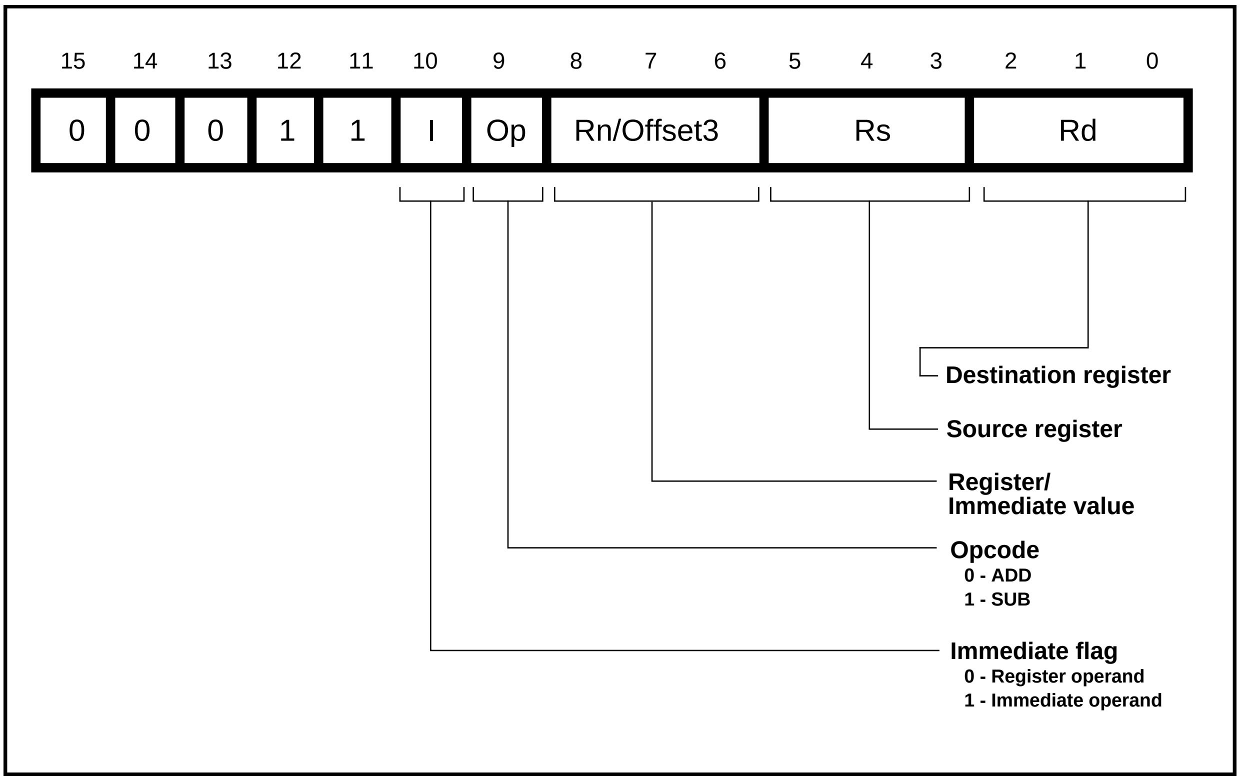


Figure 5-3: Format 2

5.2.1 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 5-3: Summary of format 2 instructions*.

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

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.

Table 5-3: Summary of format 2 instructions

5.2.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-3: Summary of format 2 instructions* on page 5-7. 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 *Chapter 10, Instruction Cycle Operations*.

5.2.3 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.
```

5.3 Format 3: move/compare/add/subtract immediate

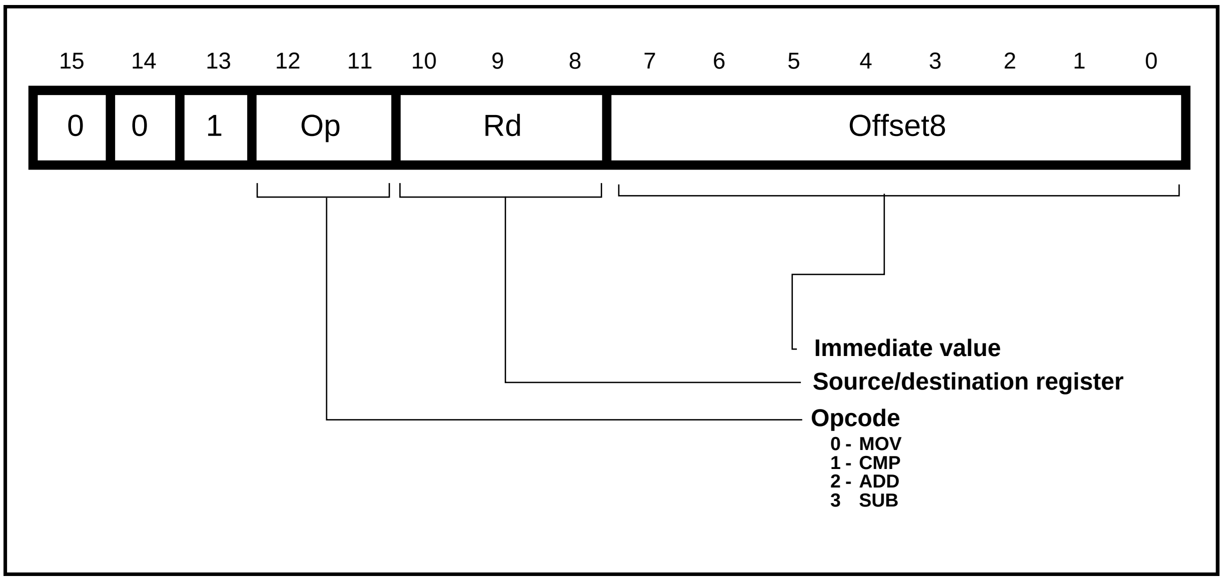


Figure 5-4: Format 3

5.3.1 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 5-4: Summary of format 3 instructions*.

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

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.

Table 5-4: Summary of format 3 instructions

5.3.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-4: Summary of format 3 instructions* on page 5-9. 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 *Chapter 10, Instruction Cycle Operations*.

5.3.3 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
```

5.4 Format 4: ALU operations

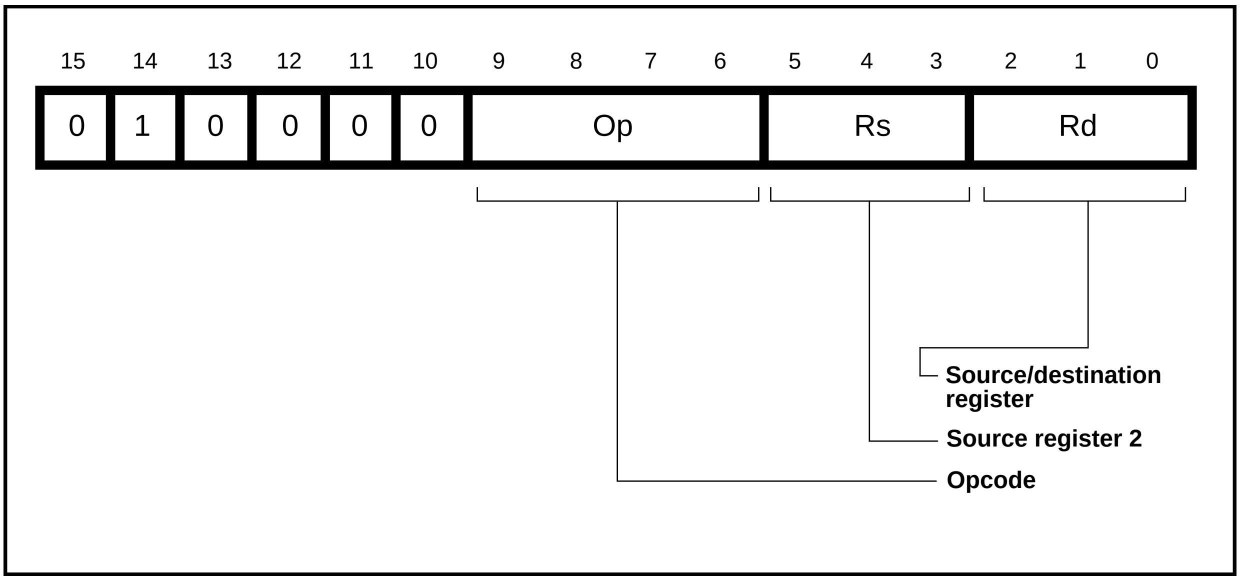


Figure 5-5: Format 4

5.4.1 Operation

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

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

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

Table 5-5: Summary of Format 4 instructions

OP	THUMB assembler	ARM equivalent	Action
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

Table 5-5: Summary of Format 4 instructions (Continued)

5.4.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-5: Summary of Format 4 instructions* on page 5-11. 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 *Chapter 10, Instruction Cycle Operations*.

5.4.3 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
```

5.5 Format 5: Hi register operations/branch exchange

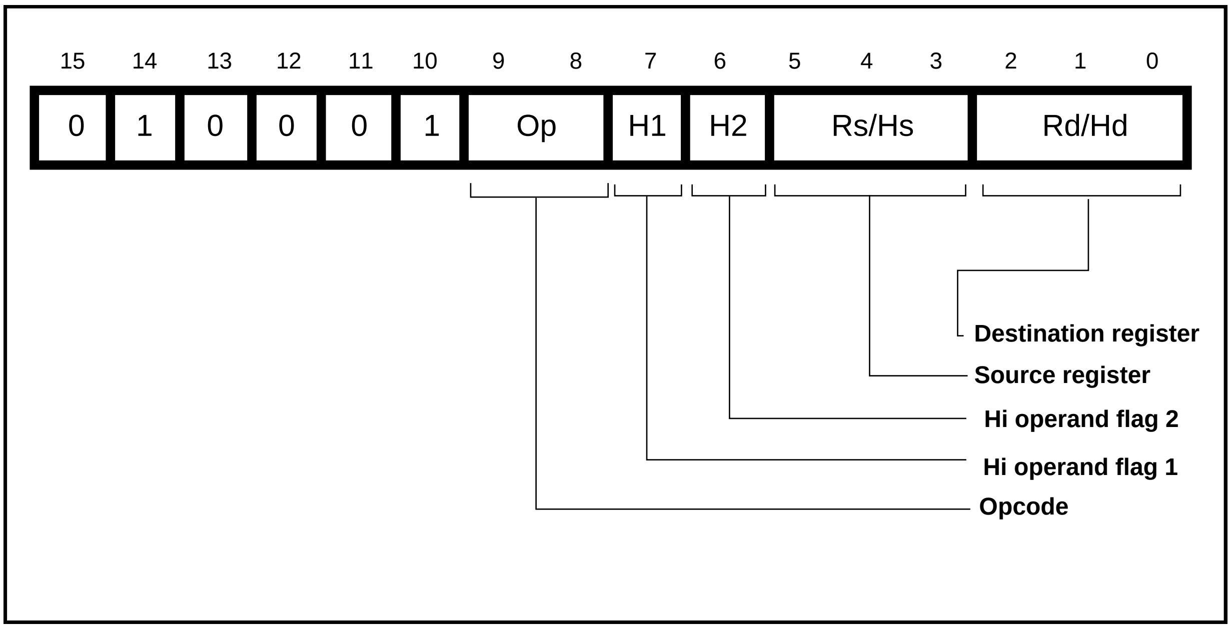


Figure 5-6: Format 5

5.5.1 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 5-6: Summary of format 5 instructions*

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.

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

Table 5-6: Summary of format 5 instructions

Op	H1	H2	THUMB assembler	ARM equivalent	Action
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.

Table 5-6: Summary of format 5 instructions (Continued)

5.5.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-6: Summary of format 5 instructions* on page 5-13. 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 *Chapter 10, Instruction Cycle Operations*.

5.5.3 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.

5.5.4 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...
```

5.5.5 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.

5.6 Format 6: PC-relative load

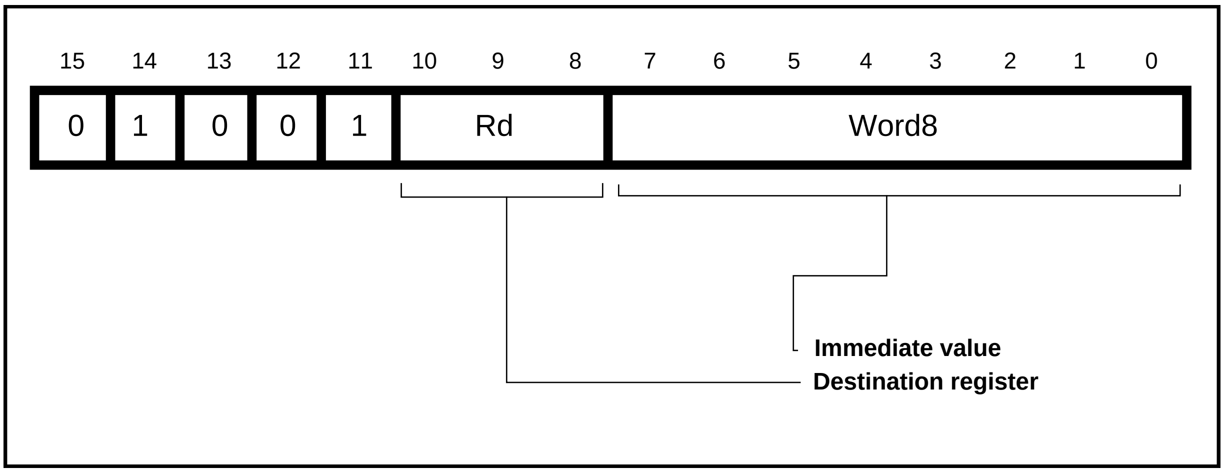


Figure 5-7: Format 6

5.6.1 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.

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.

Table 5-7: Summary of PC-relative load instruction

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.

5.6.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in ➲*Table 5-7: Summary of PC-relative load instruction* on page 5-16. 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 ➲*Chapter 10, Instruction Cycle Operations*.

5.6.3 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.
```

5.7 Format 7: load/store with register offset

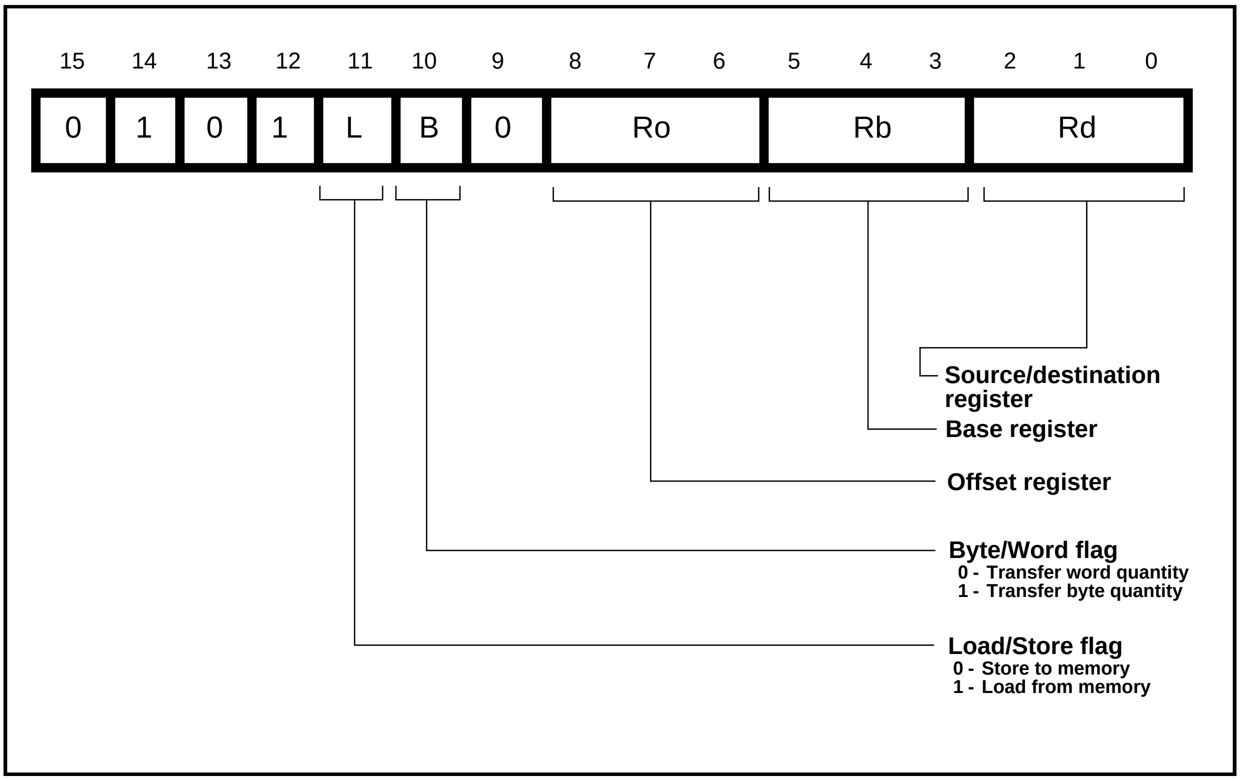


Figure 5-8: Format 7

5.7.1 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 5-8: 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 5-8: Summary of format 7 instructions

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.

Table 5-8: Summary of format 7 instructions (Continued)

5.7.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-8: Summary of format 7 instructions* on page 5-18. 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 *Chapter 10, Instruction Cycle Operations*.

5.7.3 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.
```

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

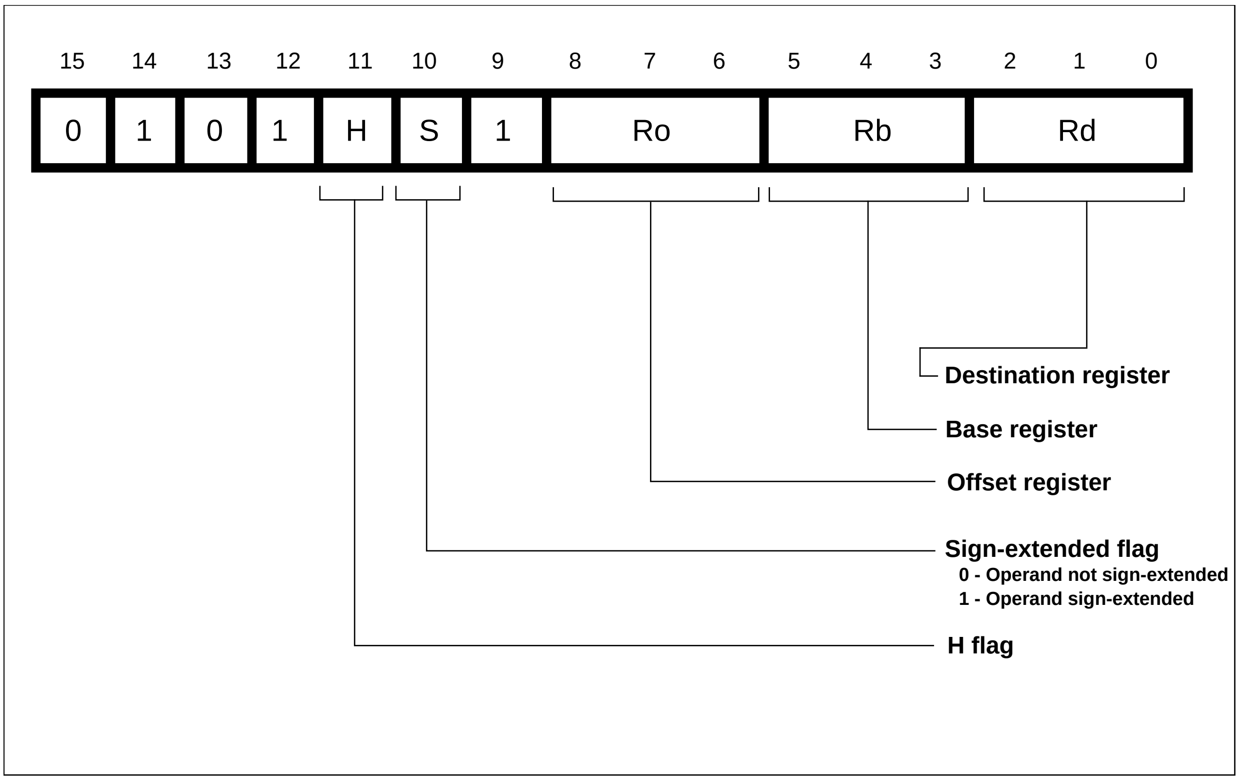


Figure 5-9: Format 8

5.8.1 Operation

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

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.

Table 5-9: Summary of format 8 instructions

S	H	THUMB assembler	ARM equivalent	Action
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.

Table 5-9: Summary of format 8 instructions (Continued)

5.8.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-9: Summary of format 8 instructions* on page 5-20. 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 *Chapter 10, Instruction Cycle Operations*.

5.8.3 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.
```

5.9 Format 9: load/store with immediate offset

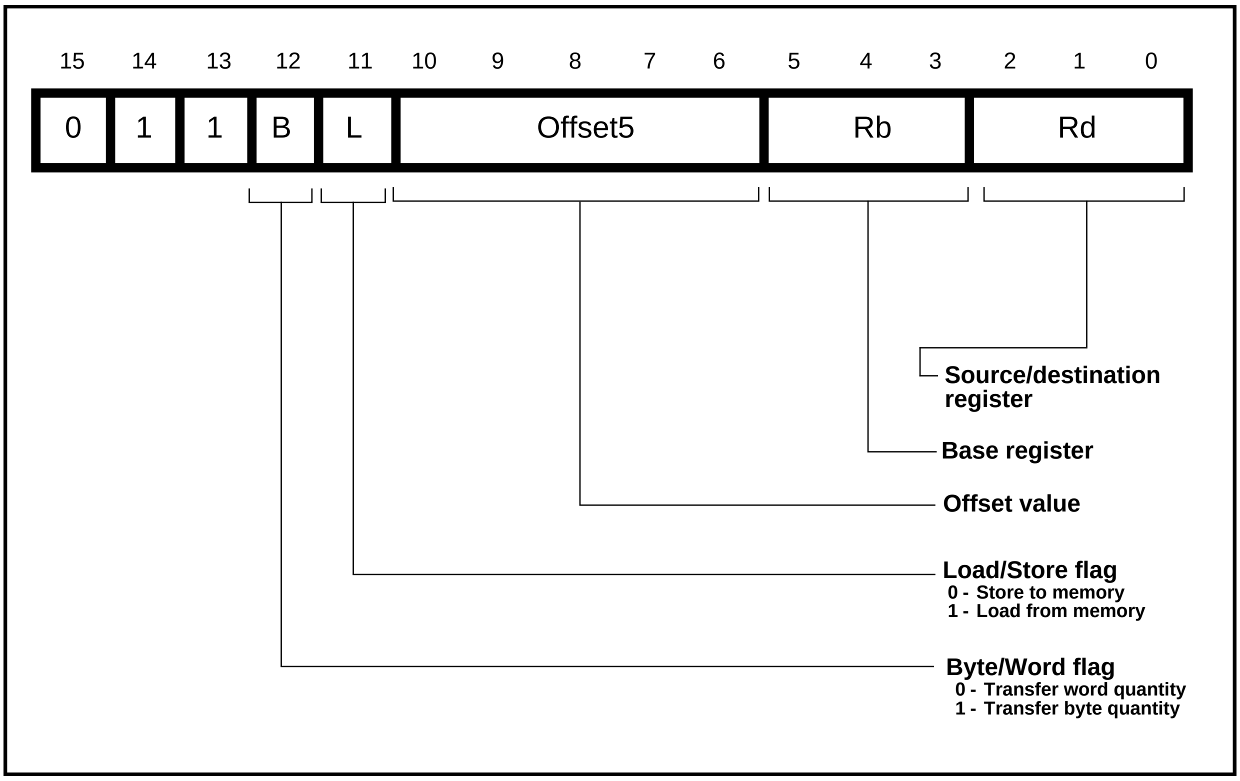


Figure 5-10: Format 9

5.9.1 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 5-10: 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.

Table 5-10: Summary of format 9 instructions

L	B	THUMB assembler	ARM equivalent	Action
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.

Table 5-10: Summary of format 9 instructions (Continued)

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.

5.9.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-10: Summary of format 9 instructions* on page 5-22. 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 *Chapter 10, Instruction Cycle Operations*.

5.9.3 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.
```

5.10 Format 10: load/store halfword

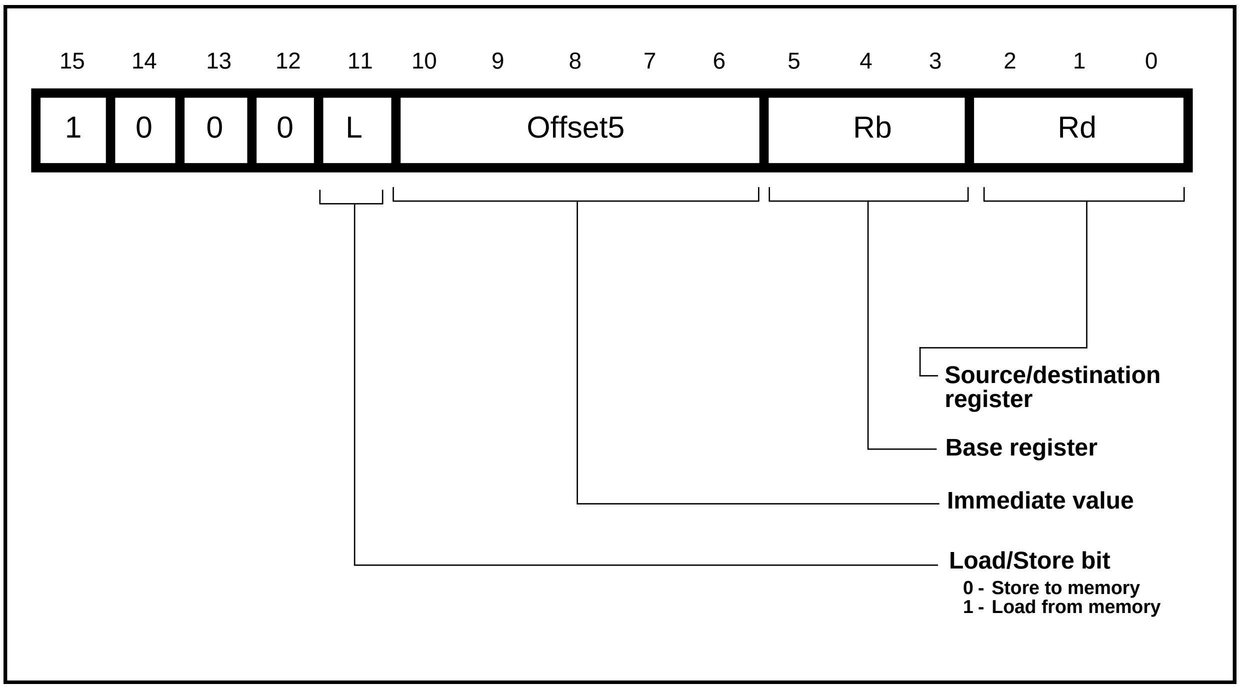


Figure 5-11: Format 10

5.10.1 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 5-11: 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.

Table 5-11: Halfword data transfer instructions

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.

5.10.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-11: Halfword data transfer instructions* on page 5-24. 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 *Chapter 10, Instruction Cycle Operations*.

5.10.3 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.
```

5.11 Format 11: SP-relative load/store

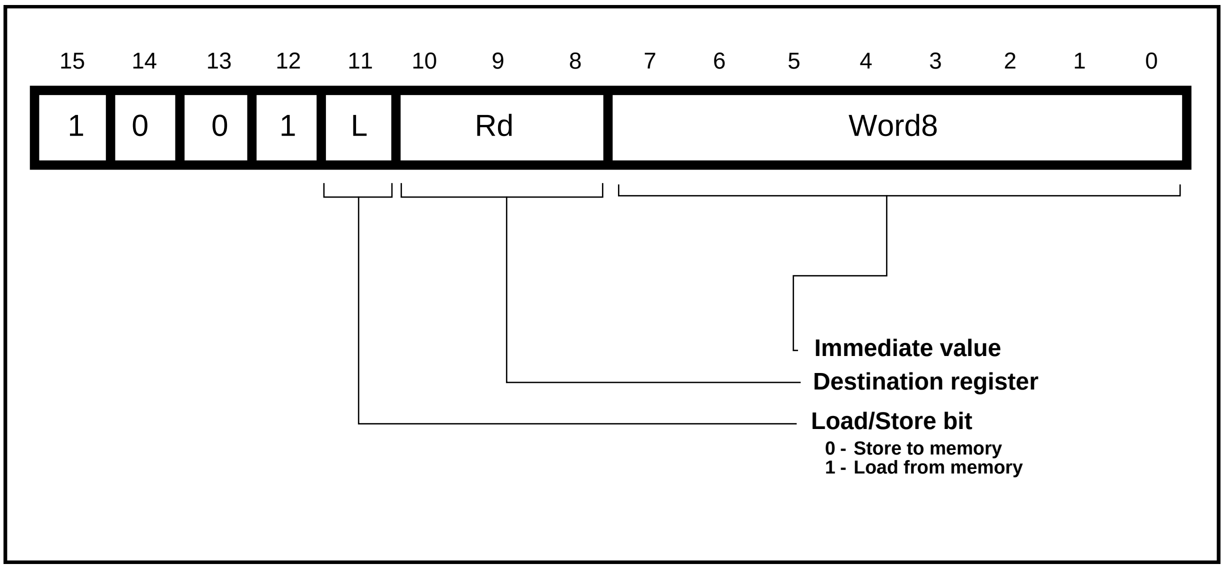


Figure 5-12: Format 11

5.11.1 Operation

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

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.

Table 5-12: SP-relative load/store instructions

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.

5.11.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-12: SP-relative load/store instructions* on page 5-26. 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 *Chapter 10, Instruction Cycle Operations*.

5.11.3 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.
```

5.12 Format 12: load address

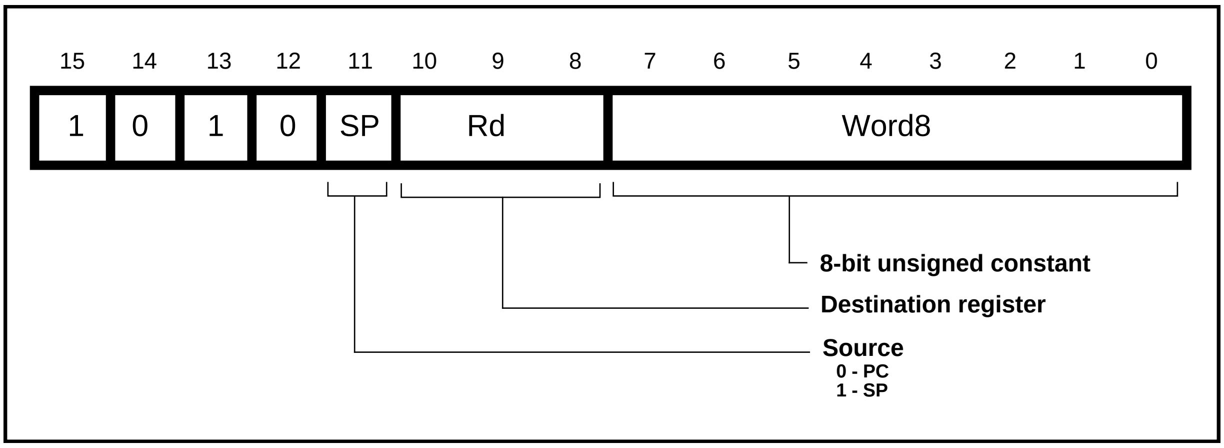


Figure 5-13: Format 12

5.12.1 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.

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.

Table 5-13: Load address

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.

5.12.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in ➲*Table 5-13: Load address* on page 5-28. 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 ➲*Chapter 10, Instruction Cycle Operations*.

5.12.3 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.
```

5.13 Format 13: add offset to Stack Pointer

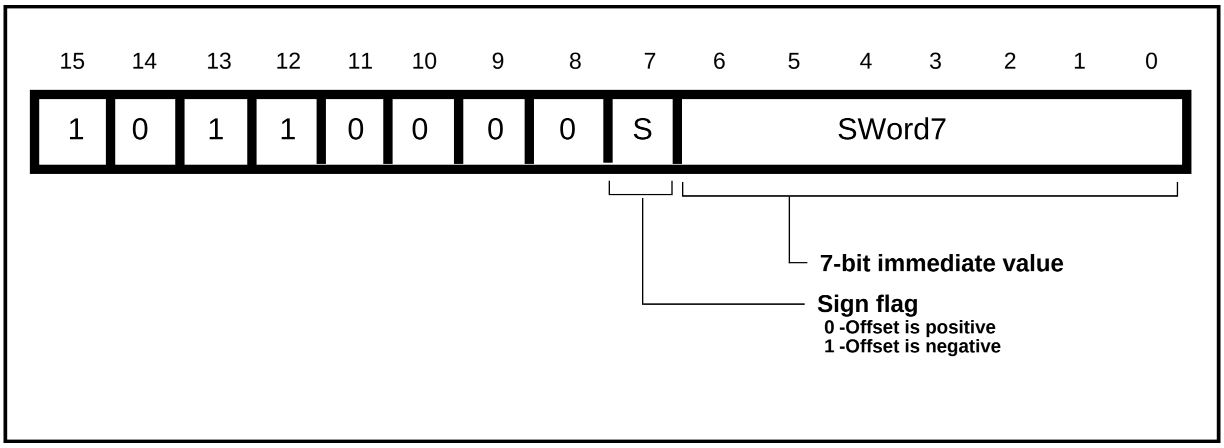


Figure 5-14: Format 13

5.13.1 Operation

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

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).

Table 5-14: The ADD SP instruction

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.

5.13.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-14: The ADD SP instruction* on page 5-30. 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 *Chapter 10, Instruction Cycle Operations*

5.13.3 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.
```

5.14 Format 14: push/pop registers

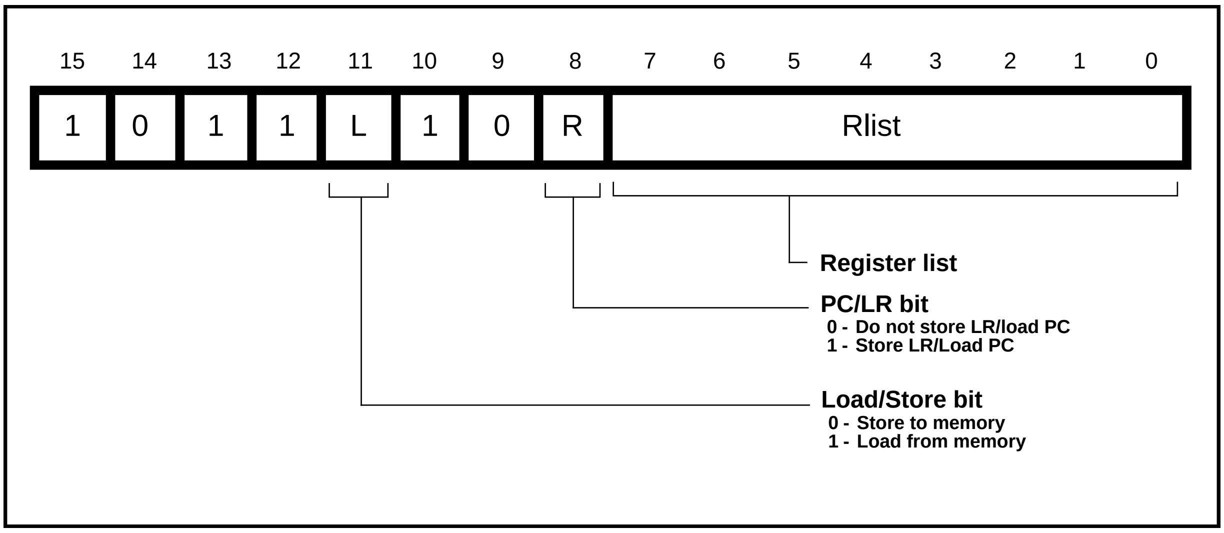


Figure 5-15: Format 14

5.14.1 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 5-15: PUSH and POP instructions*.

Note The stack is always assumed to be Full Descending.

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.

Table 5-15: PUSH and POP instructions

5.14.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-15: PUSH and POP instructions* on page 5-32. 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 *Chapter 10, Instruction Cycle Operations*.

5.14.3 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.
```

5.15 Format 15: multiple load/store

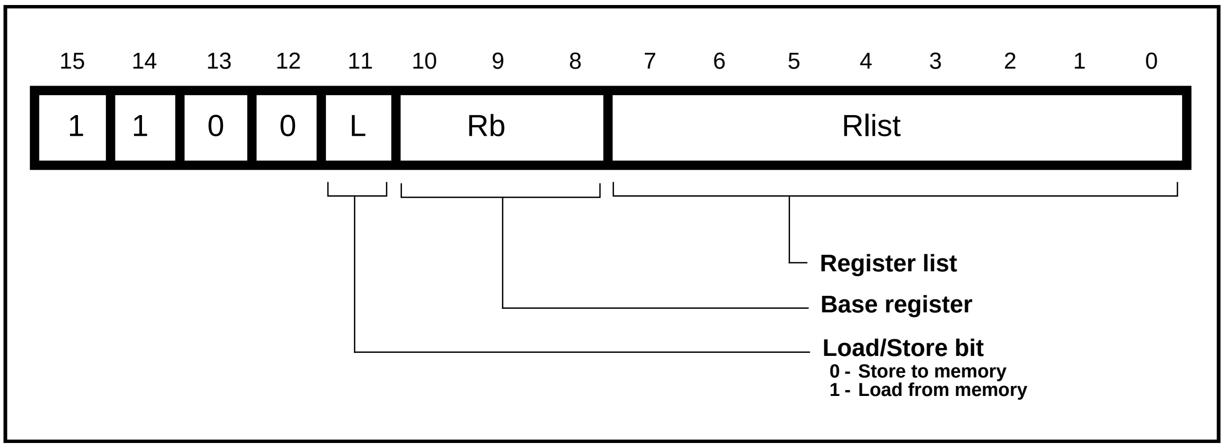


Figure 5-16: Format 15

5.15.1 Operation

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

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.

Table 5-16: The multiple load/store instructions

5.15.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-16: The multiple load/store instructions* on page 5-34. 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 *Chapter 10, Instruction Cycle Operations*

5.15.3 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.
```

5.16 Format 16: conditional branch

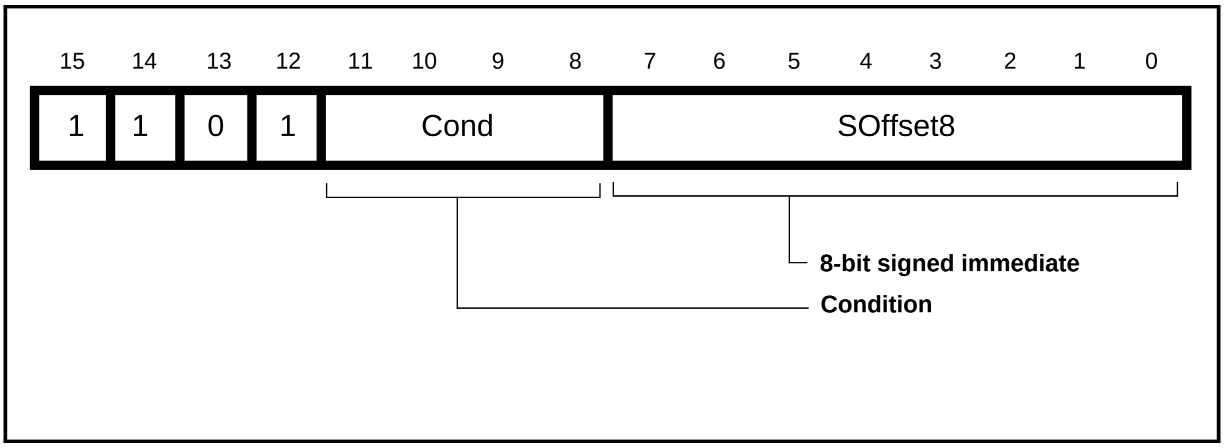


Figure 5-17: Format 16

5.16.1 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.

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)

Table 5-17: The conditional branch instructions

Cond	THUMB assembler	ARM equivalent	Action
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)

Table 5-17: The conditional branch instructions (Continued)

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 *5.17 Format 17: software interrupt* on page 5-38.

5.16.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-17: The conditional branch instructions* on page 5-36. 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 *Chapter 10, Instruction Cycle Operations*

5.16.3 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.
      ...
```

5.17 Format 17: software interrupt

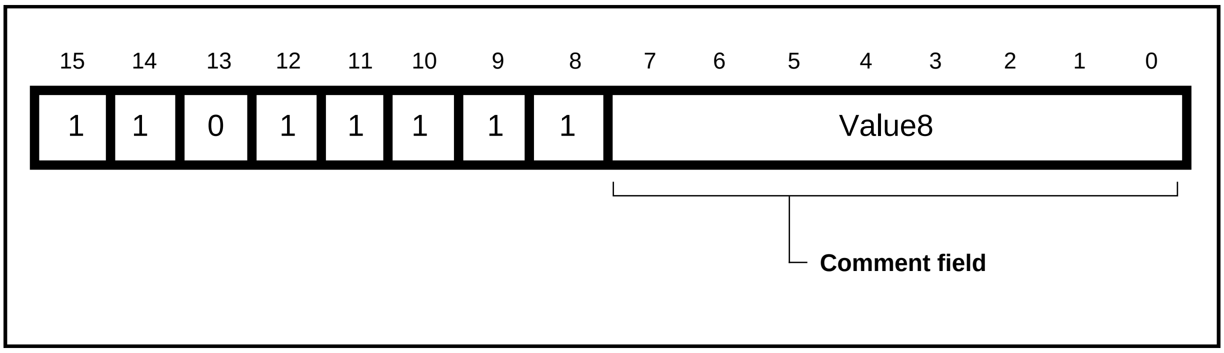


Figure 5-18: Format 17

5.17.1 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.

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.

Table 5-18: The SWI instruction

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

5.17.2 Instruction cycle times

All instructions in this format have an equivalent ARM instruction as shown in *Table 5-18: The SWI instruction* on page 5-38. 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 *Chapter 10, Instruction Cycle Operations*

5.17.3 Examples

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

5.18 Format 18: unconditional branch

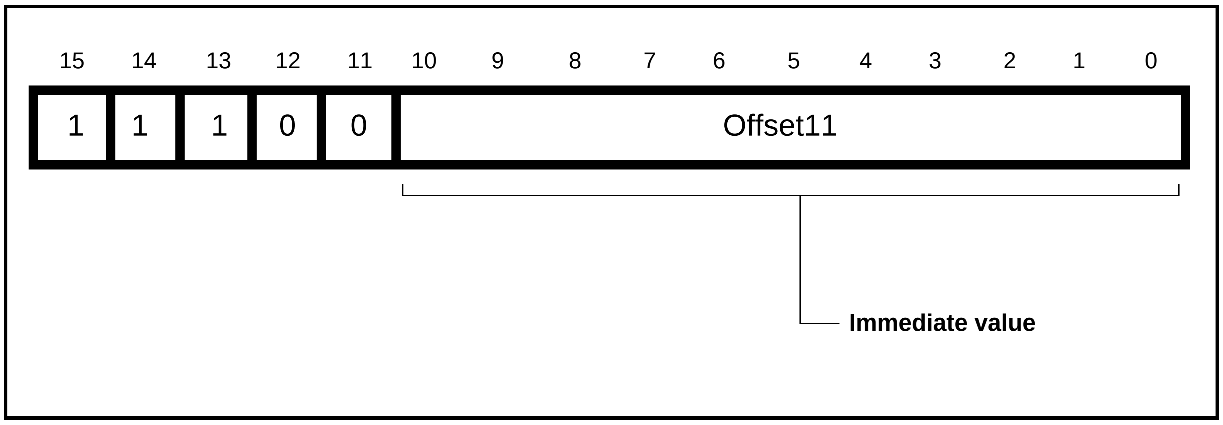


Figure 5-19: Format 18

5.18.1 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.

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

Table 5-19: Summary of Branch instruction

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.

5.18.2 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.
```

5.19 Format 19: long branch with link

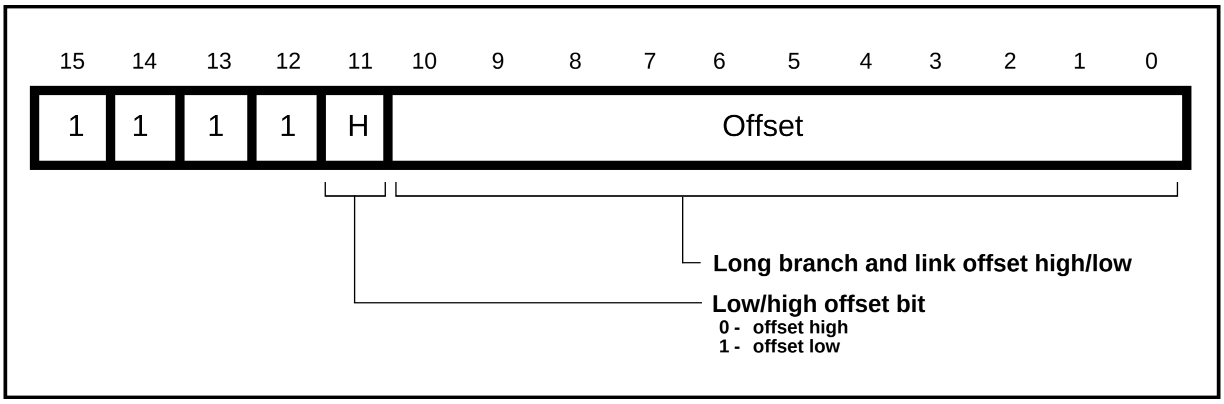


Figure 5-20: Format 19

5.19.1 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

5.19.2 Instruction cycle times

This instruction format does not have an equivalent ARM instruction. For details of the instruction cycle times, please refer to *Chapter 10, Instruction Cycle Operations*.

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

Table 5-20: The BL instruction

5.19.3 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.
```

5.20 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.

5.20.1 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
```

5.20.2 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
```

5.20.3 Division by a constant

Division by a constant can often be performed by a short fixed sequence of shifts, adds and subtracts. For an explanation of the algorithm see *The ARM Cookbook* (ARM DUYI-0005B), section entitiled *Division by a constant*.

Here is an example of a divide by 10 routine based on the algorithm in the ARM Cookbook in both Thumb and ARM code.

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
```

ARM code

```
udiv10
; takes argument in a1
; returns quotient in a1, remainder in a2
      SUB     a2, a1, #10
      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
      ADDPL   a1, a1, #1
      ADDMI   a2, a2, #10
      MOV     pc, lr
```