4 Instruction Set

This chapter details the ARM810 instruction set.

Instruction Set

4.1 Summary

The ARM810 instruction set is summarized 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	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: ARM8 instruction set

Note The instruction cycle times given in this section assume that there is no register interlocking.

4.2 Reserved Instructions and Usage Restrictions

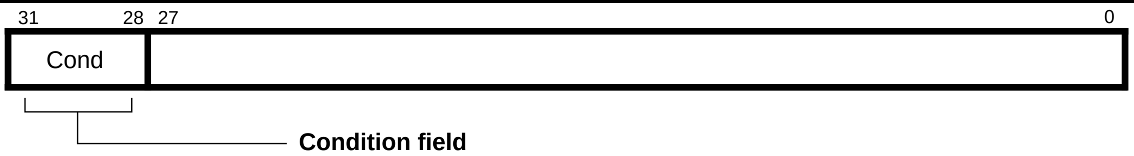
ARM810 enters an Undefined Instruction trap if it encounters an instruction bit pattern that it does not recognize. However, there are some bit patterns which are not defined, but which do not cause the Undefined Instruction trap to be taken. These *reserved* instructions must not be used, as their action may change in future ARM implementations, and may differ from previous ARM implementations.

In addition, this datasheet states that some plausible instruction usages must not be used - particular register combinations for example. In all cases where this is so, should the rules be broken, the processor will not halt or become damaged in any way, though its internal state may well be changed.

Please refer to ***4.18 Undefined Instructions*** on page 4-67 for details of which instruction bit patterns fall into the Undefined Instruction trap.

4.3 The Condition Field

All ARM810 instructions are conditionally executed. This means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. ***Figure 4-2: Condition codes*** shows the condition encoding.



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 set and V set, or N clear and V clear (greater or equal)
1011	=	LT	-	N set and V clear, or N clear and V set (less than)
1100	=	GT	-	Z clear, and either N set and V set, or N clear and V clear (greater than)
1101	=	LE	-	Z set, or N set and V clear, or N clear and V set (less than or equal)
1110	=	AL	-	Always

Figure 4-2: Condition codes

31 28	27 26	25	24	23	22	21	20	19 16	15 12	11 8	7 5	4	3 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 1 0				B	0 0		Rn	Rd	0 0 0 0	1 0 0 1		Rm	**Single Data Swap**
Cond	0 1	I	P	U	B	W	L	Rn	Rd	offset				**Single Data Transfer**
Cond	0 1 1		XXXXXXXXXXXXXXXXXXXX									1	XXXX	**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			**Coproc Data Transfer**
Cond	1 1 1 0			CP Opc				CRn	CRd	CP#	CP	0	CRm	**Coproc Data Operation**
Cond	1 1 1 0			CP Opc			L	CRn	Rd	CP#	CP	1	CRm	**Coproc Register Transfer**
Cond	1 1 1 1			ignored by processor										**Software Interrupt**

If the *always* (AL) condition is specified in an instruction, the instruction will be executed regardless of the CPSR flags.

Note: A condition field of 1111 is reserved and should not be used. Instructions with such a condition field may be redefined in future variants of the ARM architecture.

The assembler treats the absence of a condition code qualifier as though AL had been specified. If you require a NOP, use `MOV R0,R0`.

The other condition codes have meanings as detailed in ***Figure 4-2: Condition codes***. For example, code 0000 (EQual) causes an instruction to be executed only if the Z flag is set. This corresponds to the case in which a compare (CMP) instruction has found its two operands to be equal. If the two operands are different, the compare will have cleared the Z flag, and the instruction will not be executed.

4.4 Branch and Branch with Link (B, BL)

A Branch instruction is only executed if the specified condition is true: the various conditions are defined at the beginning of this chapter. ***Figure 4-3: Branch instructions*** shows the instruction encoding.

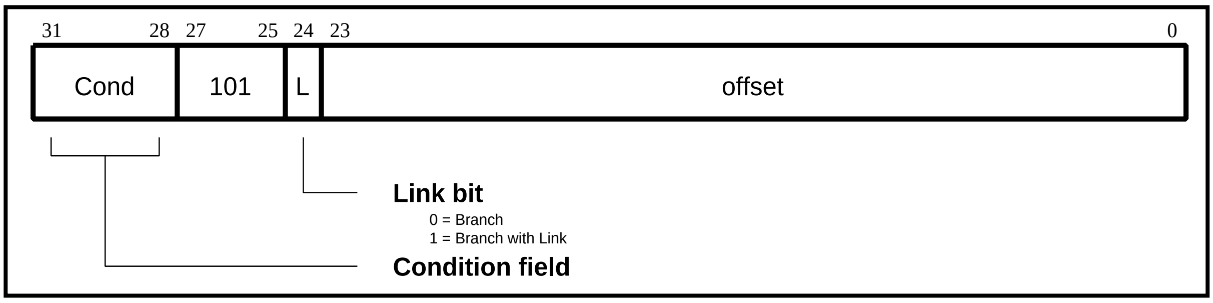


Figure 4-3: Branch instructions

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

Branches beyond +/- 32MB must use an offset or an absolute destination that has been previously loaded into a register. For Branch with Link operations that exceed 32MB, the PC must be saved manually into R14 and the offset added to the PC, or the absolute destination moved to the PC.

4.4.1 The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. In the process, 4 is subtracted from the PC value, so that R14 will contain the address of the instruction immediately following the BL instruction. The CPSR is not saved with the PC.

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 Branch prediction and removal

The ARM8 Prefetch Unit will attempt to remove a Branch instruction before it reaches the Core. If a Branch is predictable and predicted taken, the Prefetch Unit will start prefetching from the target address, so removing the Branch altogether if predicted correctly. For more information, refer to ***Chapter 6, The Prefetch Unit***.

4.4.3 Instruction cycle times

*Note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.Please refer to **Chapter 12, Bus Interface** for timing details of off-chip accesses.*

A Branch (B) or Branch with Link (BL) instruction takes 3 cycles. If optimised by the Prefetch Unit, a Branch will take fewer cycles—possibly 0—and a Branch with Link will take a minimum of 1 cycle if taken, and 0 cycles if not taken.

4.4.4 Assembler syntax

Branch instructions have the following syntax:

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

where

`{L}`	requests a Branch with Link.
`{cond}`	is one of the two-character mnemonics, shown in ***Figure 4-2: Condition codes*** on page 4-3. The assembler assumes AL (ALways) if no condition is specified.
`<expression>`	is the destination address. The assembler calculates the offset, taking into account that the PC is 8 ahead of the current instruction.

4.4.5 Examples

```
hereBAL   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
    BEQ    fred     ; if R1 was zero, otherwise continue to next
                    ; instruction

    BL     sub+ROM  ; call subroutine at address computed by
                    ; Assembler

    ADDS   R1,R1,#1 ; add 1 to register 1, setting CPSR flags
    BLCC   sub      ; on the result, then call subroutine if the
                    ; C flag is clear, which will be
                    ; the case unless R1 held 0xFFFFFFFF
```

4.5 Data Processing Instructions

A data processing instruction is only executed if the specified condition is true: the various conditions are defined at the beginning of this chapter. ***Figure 4-4: Data processing instructions*** shows the instruction encoding.

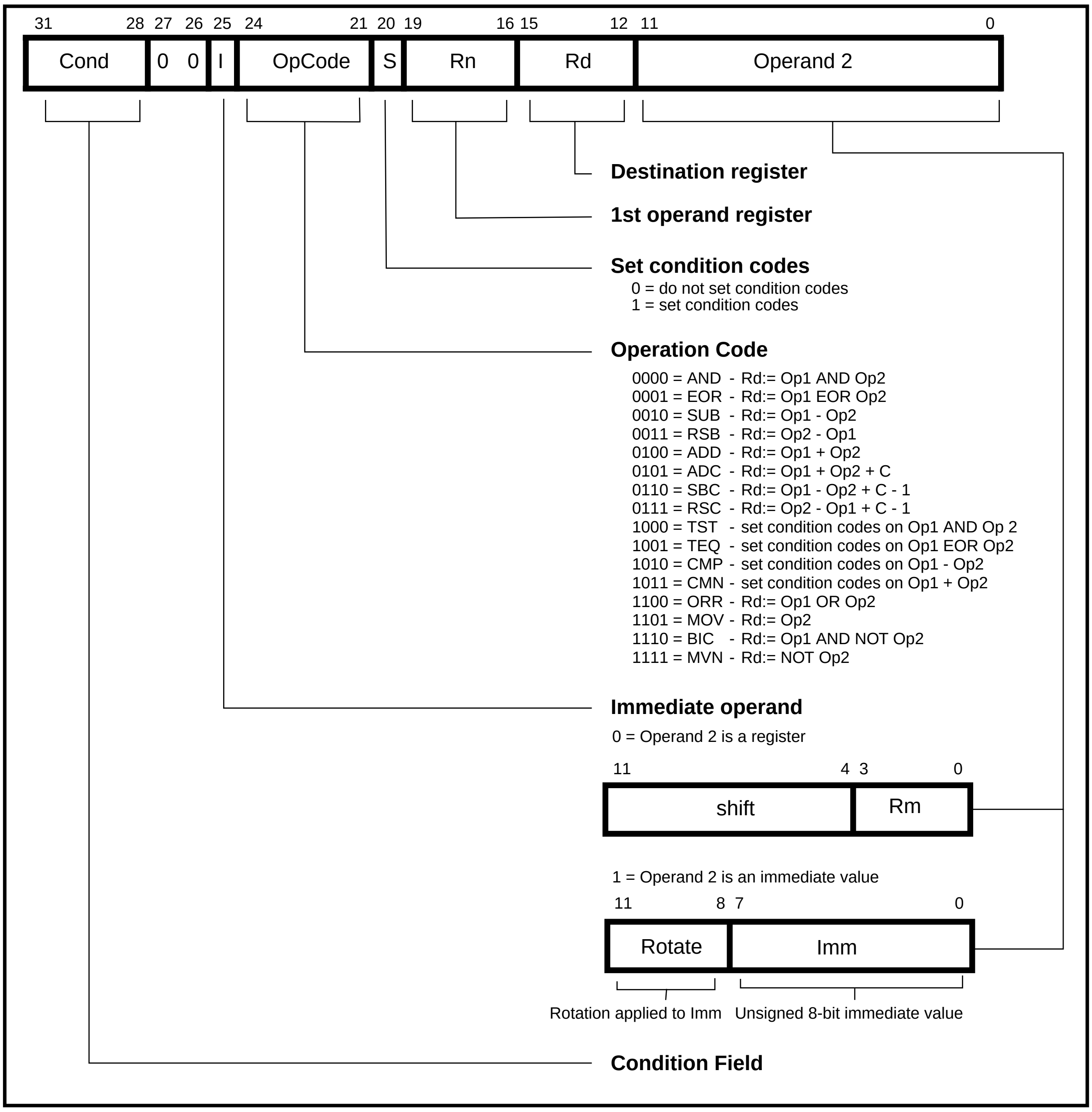

31
28
27
26
25
24
21
20
19
16
15
12
11
0
Cond
0
0
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not set condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011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

Figure 4-4: Data processing instructions

The instructions in this class produce a result by performing a specified operation on one or two operands, where:

- 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) depending on the value of the instruction's I bit.

The CPSR flags may be preserved or updated as a result of this instruction, depending on the value of the instruction's S bit.

Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the CPSR flags on the result, and therefore always have the S bit set.

The data processing instructions and their effects are listed in ***Table 4-1: ARM data processing instructions***.

Assembler mnemonic	OpCode	Action	Note
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	Rd is ignored and should be 0x0000
TEQ	1001	as EOR, but result is not written	Rd is ignored and should be 0x0000
CMP	1010	as SUB, but result is not written	Rd is ignored and should be 0x0000
CMN	1011	as ADD, but result is not written	Rd is ignored and should be 0x0000
ORR	1100	operand1 OR operand2	
MOV	1101	operand2	Rn is ignored and should be 0x0000
BIC	1110	operand1 AND NOT operand2	Bit clear
MVN	1111	NOT operand2	Rn is ignored and should be 0x0000

Table 4-1: ARM data processing instructions

4.5.1 Effects on CPSR flags

Data processing operations are classified as *logical* or *arithmetic*.

Logical operations

The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all the corresponding bits of the operand or operands to produce the result.

If the S bit is set (and Rd is not R15 - see below), they affect the CPSR flags as follows:

N	is set to the logical value of bit 31 of the result.
Z	is set if and only if the result is all zeros.
C	is set to the carry out from the shifter (so is unchanged when no shift operation occurs - see ***4.5.2 Shifts*** and ***4.5.3 Immediate operand rotates*** for the exact details of this).
V	is preserved.

Arithmetic operations

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or two's complement signed).

If the S bit is set (and Rd is not R15), they affect the CPSR flags as follows:

N	is set to the value of bit 31 of the result. This indicates a negative result if the operands are being treated as 2's complement signed.
Z	is set if and only if the result is zero.
C	is set to the carry out of bit 31 of the ALU.
V	is set if a signed overflow occurs into bit 31 of the result. This can be ignored if the operands are considered as unsigned, but warns of a possible error if they are being treated as 2's complement signed.

4.5.2 Shifts

When the second operand is a shifted register, the instruction's Shift field controls the operation of the shifter. This 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 either 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***.

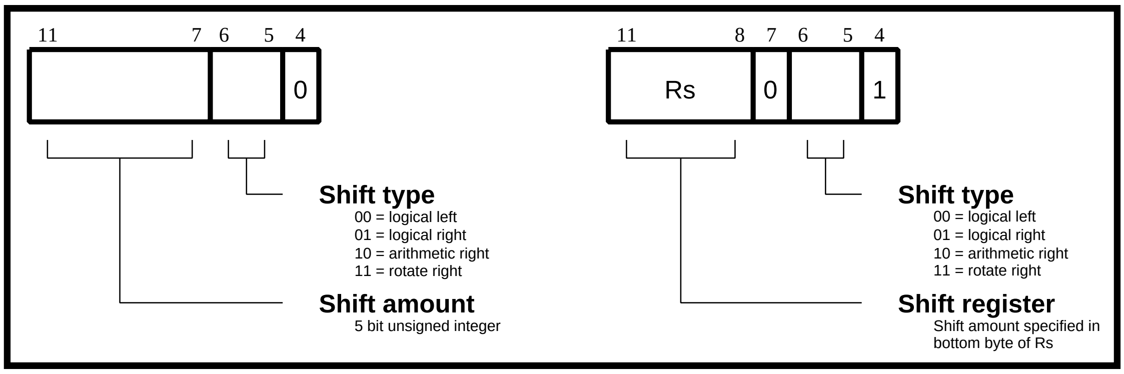


Figure 4-5: ARM shift operations

Instruction-specified shifts

When specified in the instruction, the shift amount 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 to a more significant position by the specified amount. The least significant bits of the result are filled with zeros, and the high bits of Rm that do not map into the result are discarded, with the exception of the least significant discarded bit. This 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 ***Logical operations*** on page 4-9).

As an example, ***Figure 4-6: Logical shift left*** shows the effect of LSL #5.

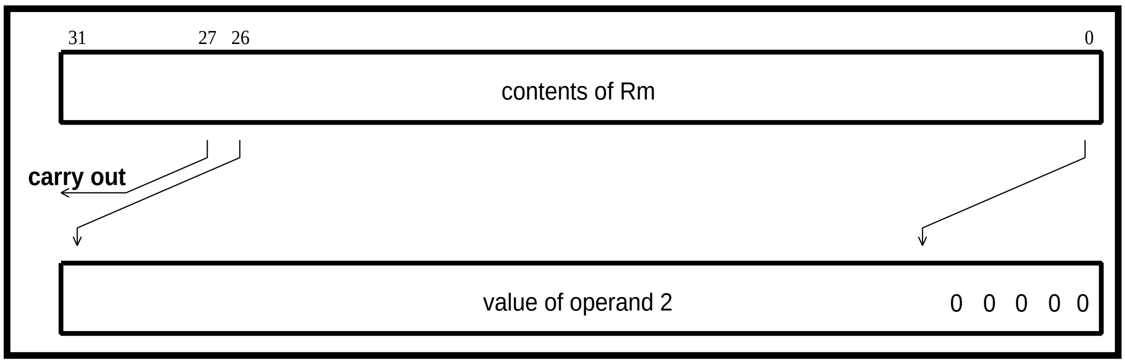


Figure 4-6: Logical shift left

Note that 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.

Logical shift right: A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. For example, LSR #5 has the effect shown in ***Figure 4-7: Logical shift right***.

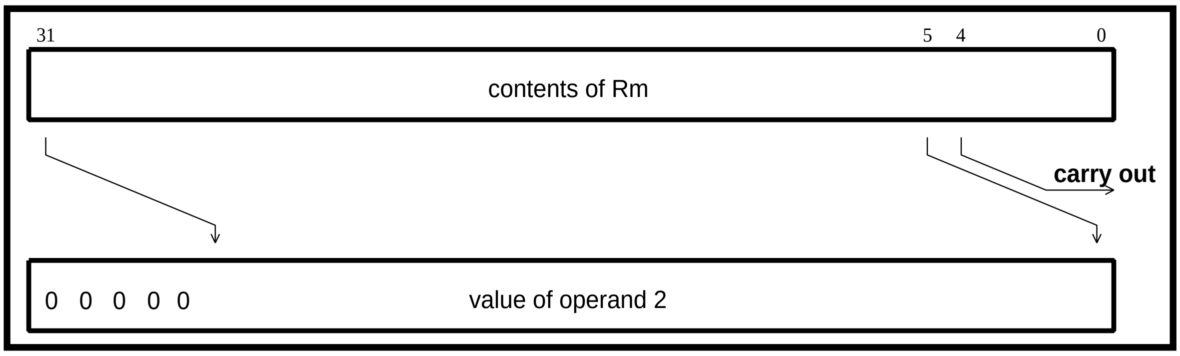


Figure 4-7: Logical shift right

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 converts LSR #0 (as well as ASR #0 and ROR #0) into LSL #0, and allows LSR #32 to be specified.

Arithmetic shift right: An arithmetic shift right (ASR) is similar to a logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in two's complement notation. ***Figure 4-8: Arithmetic shift right*** on page 4-11 shows the effect of ASR #5.

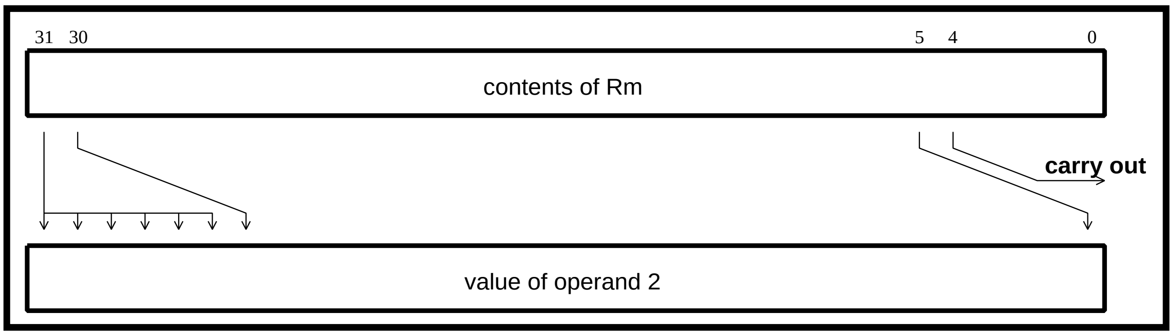


Figure 4-8: Arithmetic shift right

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, depending on the value of bit 31 of Rm.

Rotate right: Rotate right (ROR) operations re-use 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 shift right operations. To illustrate this, the effect of ROR #5 is shown in ***Figure 4-9: Rotate right***.

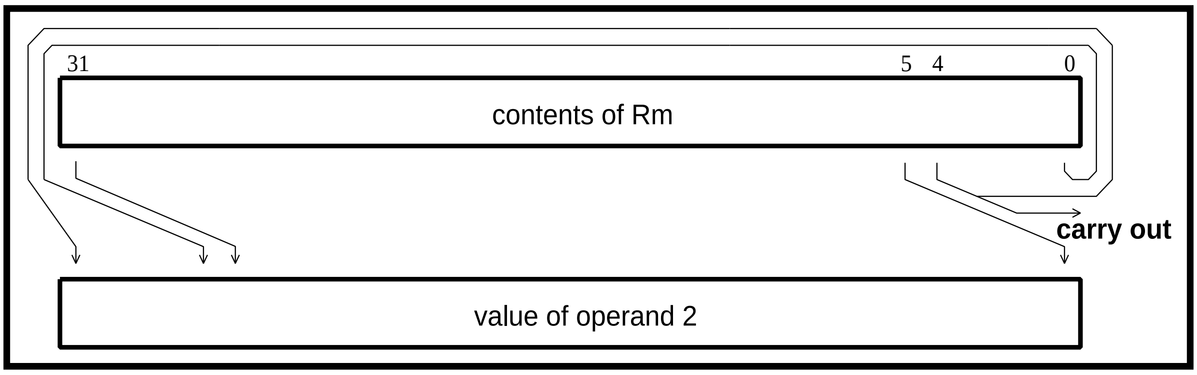


Figure 4-9: Rotate right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the 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***.

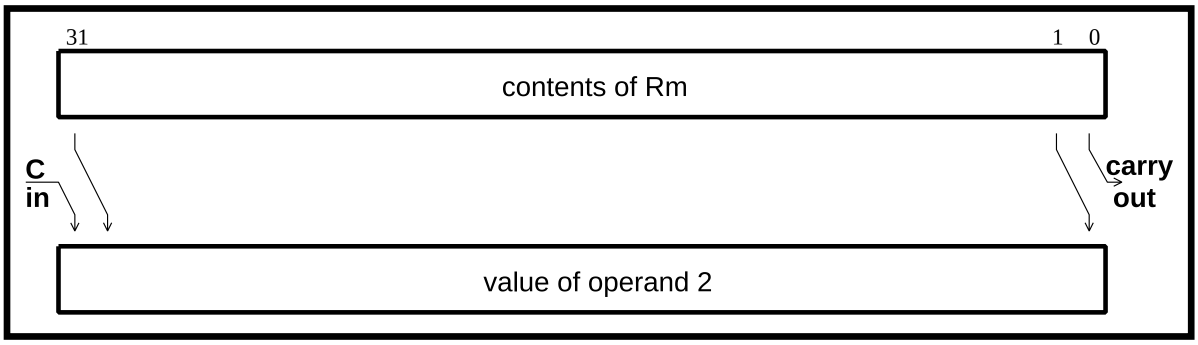


Figure 4-10: Rotate right extended

Register-specified shifts

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

Byte value	Description
0	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
1- 31	the shifted result will exactly match that of an instruction specified shift with the same value and shift operation
32	the result will be a logical extension of the shift described above: • LSL by 32 has result zero, carry out equal to bit 0 of Rm. • LSL by more than 32 has result zero, carry out zero. • LSR by 32 has result zero, carry out equal to bit 31 of Rm. • LSR by more than 32 has result zero, carry out zero. • ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm. • ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm. • 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

Note Bit 7 of an instruction with a register-controlled shift must be 0: a 1 in this bit will cause the instruction to be something other than a data processing instruction.

4.5.3 Immediate operand rotates

An immediate operand is constructed by taking the 8-bit immediate in the Imm field, zero-extending it to 32 bits, and rotating it by twice the value in the Rotate field. This enables many common constants to be generated, for example all powers of two.

If the value in the Rotate field is zero, the shifter carry out is set to the old value of the CPSR C flag. Otherwise, the shifter carry out is set to bit 31 of the shifter result, just as though an ROR had been performed (see ***Figure 4-9: Rotate right*** on page 4-12).

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 automatically restore both PC and CPSR. This form of instruction must not be used in User mode or System mode.

Note Bits [1:0] of R15 are set to zero when read from, and ignored when written to.

4.5.5 Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction and the shift amount is instruction-specified, the PC value will be the address of the instruction plus 8 bytes.

For any register-controlled shift instructions, neither Rn nor Rm may be R15.

4.5.6 MOV and MVN opcodes

With MOV and MVN opcodes, the Rn field is ignored and should be set to 0000.

4.5.7 TEQ, TST, CMP and CMN opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler will always set the S flag for these instructions, even if you do not specify this in the mnemonic. The Rd field is ignored and should be set to 0000.

In 32-bit modes, the TEQP form of the instruction used in earlier processors should not be used: the PSR transfer operations (MRS, MSR) must be used instead. Please refer to ***Appendix C, 26-bit Operations on ARM810*** for information on 26-bit mode operation.

Note The S bit (bit 20) of these instructions must be a 1; a 0 in this bit will cause the instruction to be something other than a data processing instruction.

4.5.8 Instruction cycle times

Note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information. Please refer to ***Chapter 12, Bus Interface*** *for timing details of off-chip accesses.*

Data Processing instructions vary in the number of incremental cycles taken, as shown in ***Table 4-2: Instruction cycle times*** on page 4-14.

Description	Cycles
Normal	1
If the opcode is one of ADD, ADC, CMP, CMN, RSB, RSC, SUB, SBC and there is a complex shift (anything other than LSL #0, LSL #1, LSL #2 or LSL #3)	+1
If a register-specified shift is used	+1
With PC written and the S bit is clear	+2
With PC written and the S bit is set	+3

Table 4-2: Instruction cycle times

4.5.9 Assembler syntax

The data processing instructions have the following syntax:

One operand instructions

MOV, MVN

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

Instructions that do not produce a result

CMP, CMN, TEQ, TST

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

Two operand instructions

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

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{S}`	if present, specifies that the CPSR flags will be affected (implied for CMP, CMN, TEQ, TST).
`Rd`	is an expression evaluating to a valid register number.
`Rn`	is an expression evaluating to a valid register number.
`<Op2>`	is `Rm{,<shift>}` or `#<expression>,` where `<shift>` is one of:

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

`<shiftname>` can be:

- ASL (ASL is a synonym for LSL)
- LSL
- LSR
- ASR
- ROR

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

If there is a choice of forms (for example as in #0, which can be represented using 0 rotated by 0, 2, 4,...30) the assembler will use a rotation by 0 wherever possible. This affects whether C will be changed in a logical operation with the S bit set - see ***4.5.3 Immediate operand rotates*** on page 4-13. If the rotation is 0, then C won't be modified. If the rotation is non-zero, it will be set to the last rotated bit as shown in ***Figure 4-9: Rotate right*** on page 4-12.

It is also possible to specify the 8-bit immediate and the rotation amount explicitly, by writing `<Op2>` as:

```
#<immediate>,<rotate>
```

where:

`<immediate>`	is a number in the range 0-255
`<rotate>`	is an even number in the range 0-30

4.5.10 Examples

```
ADDEQR2,R4,R5       ; if the Z flag is set make R2:=R4+R5

TEQSR4,#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

MOVSPC,R14          ; return from exception and restore CPSR
                    ; from SPSR_mode

MOVS R0,#1          ; R0 becomes 1; N and Z flags cleared;
                    ; C and V flags unchanged

MOVS R0,#4,2        ; R0 becomes 1 (4 rotated right by 2);
                    ; N, Z and C flags cleared, V flag unchanged
```

4.6 PSR Transfer (MRS, MSR)

A PSR Transfer instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter.

These instructions allow access to the CPSR and SPSR registers.

Figure 4-11: MSR (transfer register contents or immediate value to PSR) on page 4-18 and ***Figure 4-12: MRS (transfer PSR contents to a register)*** on page 4-19 show the encodings.

MRS allows the contents of the CPSR or SPSR_<mode> register to be moved to a general register. MSR allows the contents of a general register or an immediate value to be moved to the CPSR or SPSR_<mode> register, with the option of affecting any subset of bytes in the register, including:

- the flag bits only
- the control bits only
- both the flag and control bits

4.6.1 MSR operands

A register operand is any general-purpose register except R15.

An immediate operand is constructed by taking the 8-bit immediate in the Imm field, zero-extending it to 32 bits, and rotating it by twice the value in the Rotate field. This enables many common constants to be generated, for example all powers of two.

4.6.2 Operand restrictions

In User mode, the control bits of the CPSR are protected so that only the condition code flags can be changed. In other (privileged) modes, it is possible to alter the entire CPSR.

The mode at the time of execution determines which of the SPSR registers is accessible: for example, only SPSR_fiq can be accessed when the processor is in FIQ mode.

R15 cannot be specified as the source or destination register.

Note Do not attempt to access an SPSR in User mode or System mode, since no such register exists.

4.6.3 Reserved bits

Only eleven bits of the PSR are defined in ARM810 (N, Z, C, V, I, F and M[4:0]). The remaining bits (PSR[27:8,5]) are reserved for use in future versions of the processor.

To ensure the maximum compatibility between ARM810 programs and future processors, you should observe the following rules:

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

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

The reserved flag bits (bits 27:24) are an exception to this rule; they may have any values written to them. Any future use of these bits will be compatible with this. In particular, there is no need to use the read-modify-write strategy on these bits.

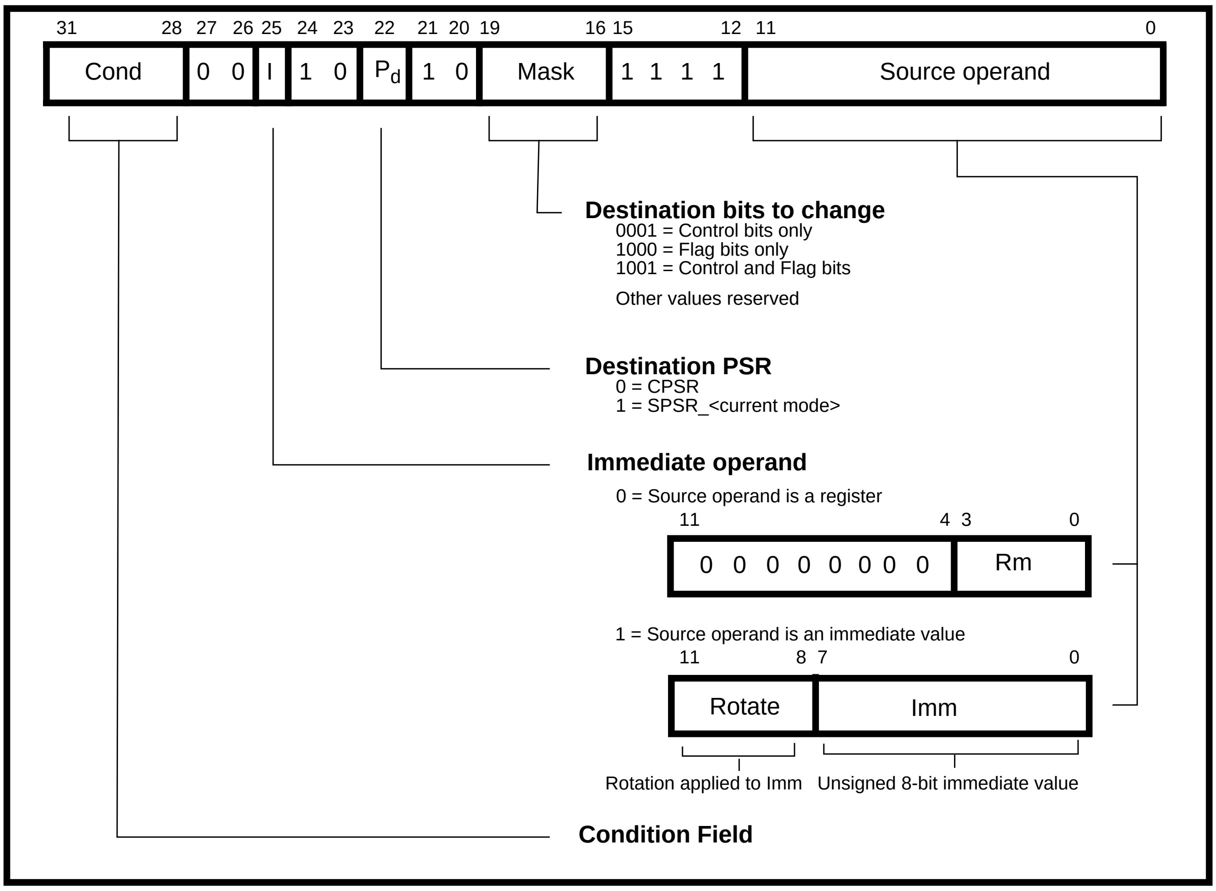


Figure 4-11: MSR (transfer register contents or immediate value to PSR)

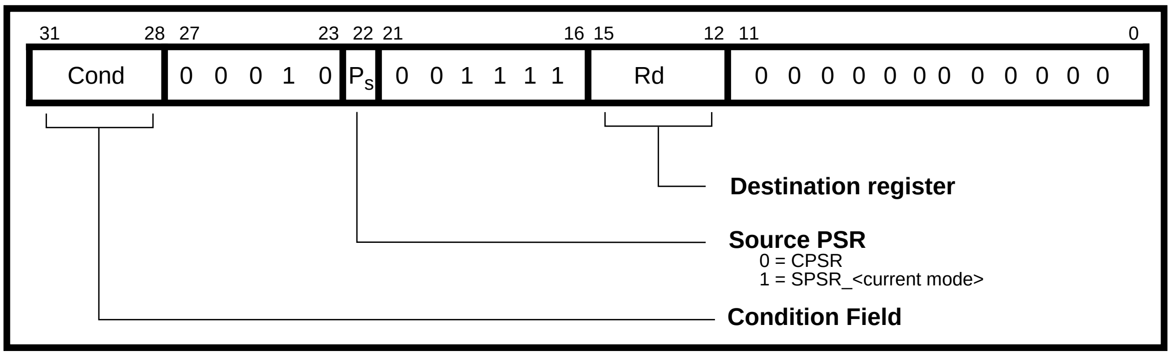


Figure 4-12: MRS (transfer PSR contents to a register)

For 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 example 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)
```

You should not attempt to write an 8-bit immediate value into the whole PSR, since such an operation cannot preserve the reserved bits.

4.6.4 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

The MRS instruction takes 1 cycle.

The MSR instruction takes 1 cycle when the flag variant is used, or the destination is SPSR_<mode>. In all other cases, MSR takes 3 cycles.

4.6.5 Assembler syntax

The PSR transfer instructions have the following syntax:

Transfer PSR contents to a register

```
MRS{cond} Rd,<psr>
```

Transfer register contents to PSR

```
MSR{cond} <psr>_<fields>,Rm
```

Transfer immediate value to PSR

```
MSR{cond} <psr>_f,#<expression>
```

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`Rd` and `Rm`	are expressions evaluating to a register number other than R15.
`<psr>`	is CPSR or SPSR.
`<fields>`	is one of:
	_c to set the control field mask bit (bit 0)
	_x to set the extension field mask bit (bit 1)
	_s to set the status field mask bit (bit 2)
	_f to set the flags field mask bit (bit 3)
`#<expression>`	is used by the assembler to generate a shifted immediate 8-bit field. If this impossible, the assembler gives an error.

4.6.6 Previous, deprecated MSR assembler syntax

This section describes the old assembler syntax for MSR instructions. These will still work on ARM8, but should be replaced by the new syntax as described in section ***4.6.5 Assembler syntax*** on page 4-20.

Transfer register contents to PSR

```
MSR{cond} <psrf>,Rm
```

Transfer immediate value to PSR

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`Rd` and `Rm`	are expressions evaluating to a register number other than R15.
`<psrf>`	is one of CPSR, CPSR_all, CPSR_flg, CSPR_ctl, SPSR, SPSR_all, SPSR_flg or SPSR_ctl.
`#<expression>`	is used by the assembler to generate a shifted immediate 8-bit field. If this is impossible, the assembler gives an error.

4.6.7 Examples

User mode

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
                              ; (i.e. set N,C; clear Z,V)

MRS   Rd,CPSR                 ; Rd[31:0] <- CPSR[31:0]
```

System mode

In system mode, 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_ctl,Rm              ; CPSR[7:0]   <- Rm[7:0]

MSR   CPSR_flg,#0x50000000     ; CPSR[31:28] <- 0x5
                               ; (i.e. set Z,V; clear N,C)

MRS   Rd,CPSR                  ; Rd[31:0] <- CPSR[31:0]
```

Other privileged modes

In other 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_ctl,Rm              ; CPSR[7:0]   <- Rm[7:0]

MSR   CPSR_flg,#0x50000000     ; CPSR[31:28] <- 0x5
                               ; (i.e. 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_ctl,Rm              ; SPSR_<mode>[7:0]   <- Rm[7:0]

MSR   SPSR_flg,#0xC0000000     ; SPSR_<mode>[31:28] <- 0xC
                               ; (i.e. set N,Z; clear C,V)

MRS   Rd,SPSR                  ; Rd[31:0] <- SPSR_<mode>[31:0]
```

4.7 Multiply and Multiply-Accumulate (MUL, MLA)

A multiply instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-13: Multiply instructions*** shows the instruction encoding.

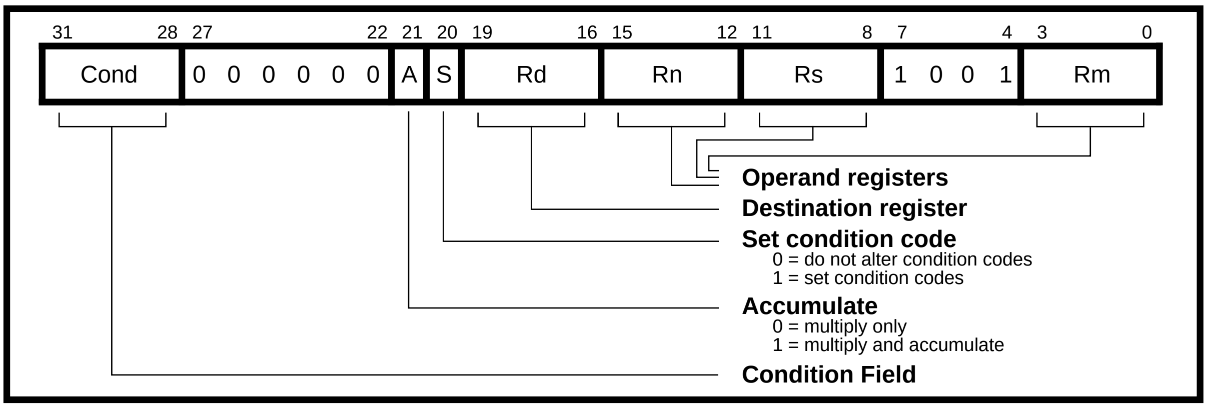


Figure 4-13: Multiply instructions

The multiply and multiply-accumulate instructions perform integer multiplication, optionally accumulating another integer to the product.

Multiply instruction

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

Multiply-accumulate

Multiply-accumulate (MLA) gives Rd:=Rm*Rs+Rn. In some circumstances this can save an explicit ADD instruction.

The result of a signed multiply of 32-bit operands differs from that of an unsigned multiply of 32-bit operands only in the upper 32 bits - the low 32 bits of signed and unsigned results are identical. Since MUL and MLA only produce the low 32 bits of a multiply, they can be used for both signed and unsigned multiplies. Consider the following:

Operand A	Operand B	Result
0xFFFFFFF6	0x00000014	0xFFFFFF38

Signed operands: When the operands are interpreted as signed, A has the value -10 and B has the value 20. The result is -200, which is correctly represented as 0xFFFFFF38.

Unsigned operands: When the operands are interpreted as unsigned, A has the value 4294967286, B has the value 20 and the result is 85899345720, which is represented as 0x13FFFFFF38, the least significant 32 bits of which are 0xFFFFFF38. Again, the representation of the result is correct.

4.7.1 Operand restrictions

- The destination register (Rd) must not be the same as Rm.
- R15 must not be used as Rd, Rm, Rn or Rs.

4.7.2 CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit. If this is set:

N	is made equal to bit 31 of the result.
Z	is set if and only if the result is zero.
C	is set to a meaningless value.
V	is unaffected.

4.7.3 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

MUL and MLA take from 3 to 6 cycles to execute, depending upon the early termination, as follows:

Basic cycle count	6 (including any accumulate)
Early termination	-(0 to 3)

4.7.4 Assembler syntax

The multiply instructions have the following syntax:

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{S}`	if present, specifies that the CPSR flags will be affected.
`Rd,Rm,Rs,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)

A multiply long instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-14: Multiply Long instructions***.

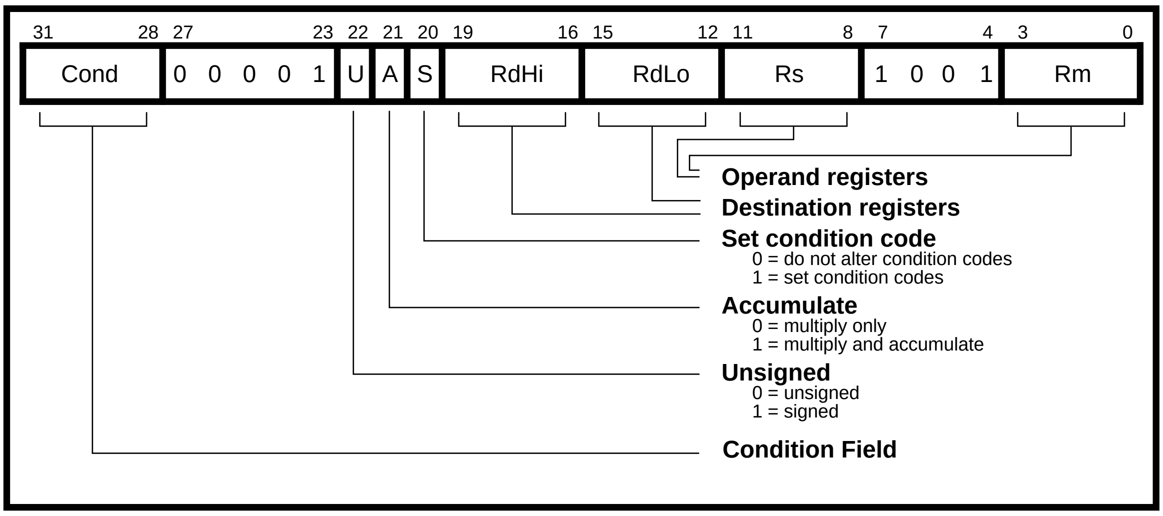


Figure 4-14: Multiply Long instructions

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.

Multiply (UMULL and SMULL)

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.

Multiply-accumulate (UMLAL and SMLAL)

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 are read from RdLo. The upper 32 bits of the 64-bit number to add are read from RdHi. The lower 32 bits of the 64-bit result are written to RdLo, and the upper 32 bits of the 64-bit result are written to RdHi.

UMULL and UMLAL 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. If this is set:

N	is made equal to bit 63 of the result
Z	is set if and only if all 64 bits of the result are zero
C	is set to a meaningless value
V	is set to a meaningless value

4.8.3 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

MULL and MLAL take from 4 to 7 cycles to execute, depending upon the early termination, as follows:

Basic cycle count	7 (including any accumulate)
Early termination	-(0 to 3)

4.8.4 Assembler syntax

The multiply long instructions have the following syntax:

Unsigned Multiply Long (32 x 32 = 64)

```
UMULL{cond}{S}    RdLo,RdHi,Rm,Rs
```

Unsigned Multiply and Accumulate Long (32 x 32 + 64 = 64)

```
UMLAL{cond}{S}    RdLo,RdHi,Rm,Rs
```

Signed Multiply Long (32x 32 = 64)

```
SMULL{cond}{S}    RdLo,RdHi,Rm,Rs
```

Signed Multiply and Accumulate Long (32 x 32 + 64 = 64)

```
SMLAL{cond}{S}    RdLo,RdHi,Rm,Rs
```

where

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{S}`	if present, specifies that the CPSR flags will be affected.
`RdLo,RdHi,Rm,Rs`	are expressions evaluating to a register number other than R15.

```
Examples

UMULL             R1,R4,R2,R3;; R4,R1:=R2*R3

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

4.9 Single Data Transfer (LDR, STR)

A single data transfer instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-15: Single data transfer instructions*** shows the instruction encoding.

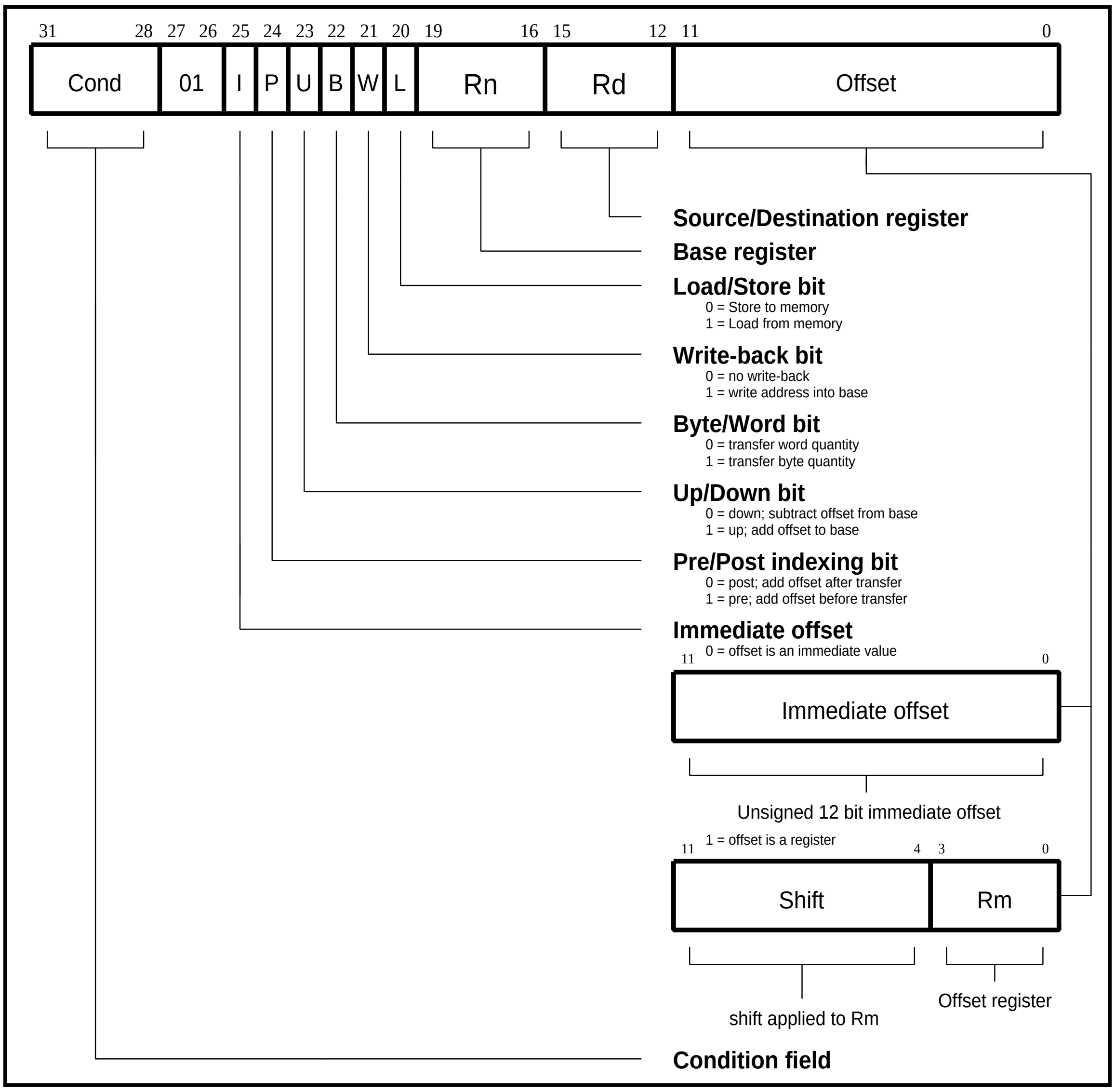


Figure 4-15: Single data transfer instructions

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 or subtracting an offset from a base register. If auto-indexing is required, the result may be written back into the base register.

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, 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 facility.

4.9.2 Shifted register offset

The 8 shift control bits are described in ***4.5.2 Shifts*** on page 4-10. However, register-specified shift amounts are not available in this instruction class.

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 ARM810 register and memory.

The action of LDR(B) and STRB instructions is influenced by the BIGEND control signal. The two possible configurations are:

- Little-endian
- Big-endian

Little-endian configuration

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 3-1: Little-endian addresses of bytes within words*** on page 3-3.
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.
Word load (LDR)	Any non-word-aligned address will cause the data read to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that halfwords 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 4-16: Little-endian offset addressing*** on page 4-29.

Note	The LDRH and LDRSH insrtuctions provide a more efficient way to load half-words on ARM810. This method of loading half-words should therefore only be used if compatibility with previous ARM processors is required. See ***4.10 Halfword and Signed Data Transfer*** on page 4-34 for further details.
Word store (STR)	will normally generate a word-aligned address. The word presented to the data bus is not affected if the address is non-word-aligned, so bit 31 of the register being stored always appears on data bus output 31.

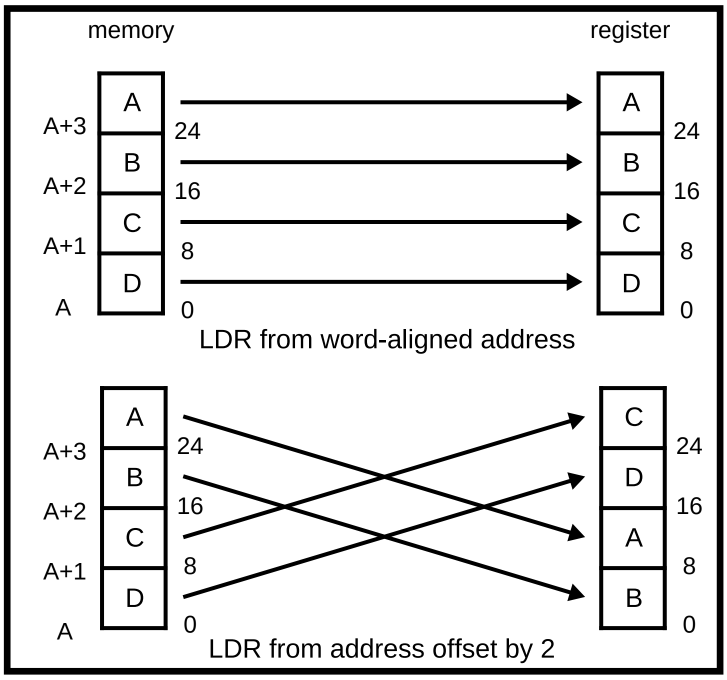


Figure 4-16: Little-endian offset addressing

Big-endian configuration

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 3-2: Big-endian addresses of bytes within words*** on page 3-3.
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.
Word load (LDR)	will normally 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 halfwords 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.

Note The LDRH and LDRSH instructions provide a more efficient way to load half-words on ARM810. This method of loading half-words should therefore only be used if compatibility with previous ARM processors is required. See ***4.10 Halfword and Signed Data Transfer*** on page 4-34 for details.

Word store (STR) will normally generate a word-aligned address. The word presented to the data bus is not affected if the address is not word-aligned, so that bit 31 of the register being stored always appears on data bus output 31.

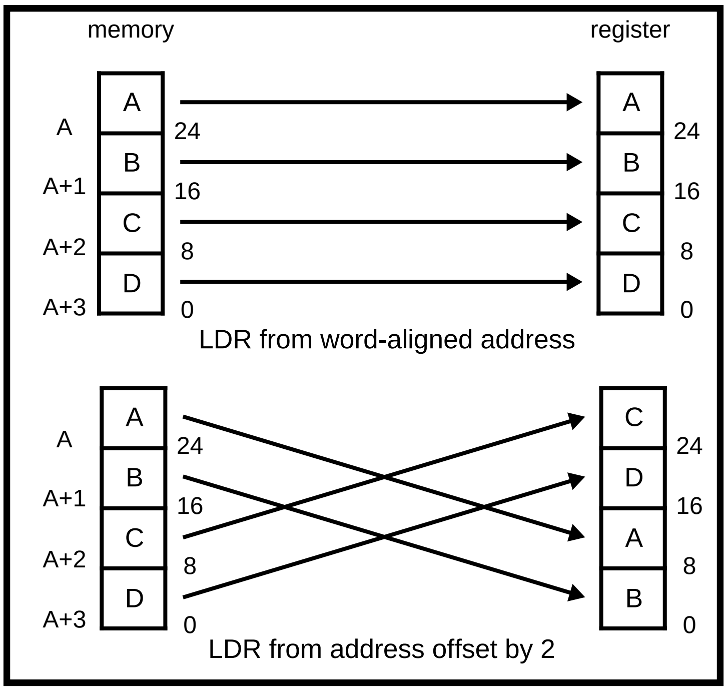


Figure 4-17: Big-endian offset addressing

4.9.4 Use of R15

Do not specify write-back if R15 is the base register (Rn). When using R15 as the base register, it must be remembered that it contains an address 8 bytes on from the address of the current instruction.

Do not specify post-indexing (forcing writeback) to Rn when Rn is R15.

Do not specify R15 as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be the address of the instruction plus 8. Note that this is different from previous ARM processors, which stored the address of the register plus 12.

When R15 is the source register (Rd) of a register store (STR) instruction, or the destination register (Rd) of a register load (LDR) instruction, the byte form of the instruction (LDRB or STRB) must not be used, *and* the address must be word-aligned.

Note Bits [1:0] of R15 are set to zero when read from, and are ignored when written to.

4.9.5 Restrictions on the use of the base register

In the following example, it may sometimes be impossible to calculate the initial value of R0 after an abort in order to restart the instruction:

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

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

When an LDR instruction specifies (or implies) base writeback, register positions Rd and Rn should not be the same register.

4.9.6 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of aborts in general.

In some situations a transfer to or from an address may cause a memory management system to generate an abort.

For example, in a system which uses virtual memory, the required data may be absent from main memory. The memory manager can signal a problem by signalling a Data Abort to the processor, whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, after which the instruction can be restarted and the original program continued.

In all cases, the base register is restored to its original value before the Abort trap is taken. In the case of an LDR or LDRB, the destination register (Rd) will not have been altered.

4.9.7 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

LDR instructions take 1 cycle:

- +1 cycle if there is a register offset with a shift other than LSL #0, LSL #1, LSL #2 or LSL #3
- +4 cycles for loading the PC

STR instructions take 1 cycle:

- +1 cycle if there is a register offset (regardless of shift type)

4.9.8 Assembler syntax

The single data transfer instructions have the following syntax:

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

where:

`LDR`	loads from memory into a register.
`STR`	stores from a register into memory.
`{cond}`	is a two-character condition mnemonic. If omitted, the assembler assumes ALways.
`{B}`	if present, specifies byte transfer. If omitted, word transfer is used.
`{T}`	if present, sets the W bit 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.
`<Addr>`	is one of:

An `<expression>` specifying an address:

The assembler will attempt to address this location by generating an instruction that uses the PC as a base, along with a corrected immediate offset. This will be a PC relative, pre-indexed address. If the address is out of range, an error is generated.

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>`

A post-indexed addressing specification:

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

Rn and Rm	are expressions evaluating to a register number. If Rn is R15, neither post-indexed addressing nor `{!}` should be specified.
`<shift>`	is one of:

`<shiftname> #expression`	
RRX	(rotate right one bit with extend)
`<shiftname>`	is ASL, LSL, LSR, ASR or ROR (ASL is a synonym for LSL)

`{!}`	if present, sets the W bit so that the base register is written back.

4.9.9 Examples

```
STR    R1,[R2,R4]!     ; store R1 at R2+R4 (both are registers)
                       ; and write back address to R2

STR    R1,[R2],R4      ; store R1 at R2. Write back R2+R4 to R2

LDR    R1,[R2,#16]     ; load R1 from contents of R2+16.
                       ; Don't write back

LDR    R1,[R2,R3,LSL#2]; load R1 from contents of R2+R3*4

LDREQB R1,[R6,#5]      ; conditionally load byte at R6+5 into R1
                       ; bits 0 - 7, filling bits 8 - 31 with 0s

STR    R1,PLACE        ; assembler generates 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 at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-18: Halfword and signed data transfer with register offset*** and ***Figure 4-19: Halfword and signed data transfer with immediate offset***.

These instructions are used to load or store halfwords of data and also load sign-extended bytes or halfwords 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.

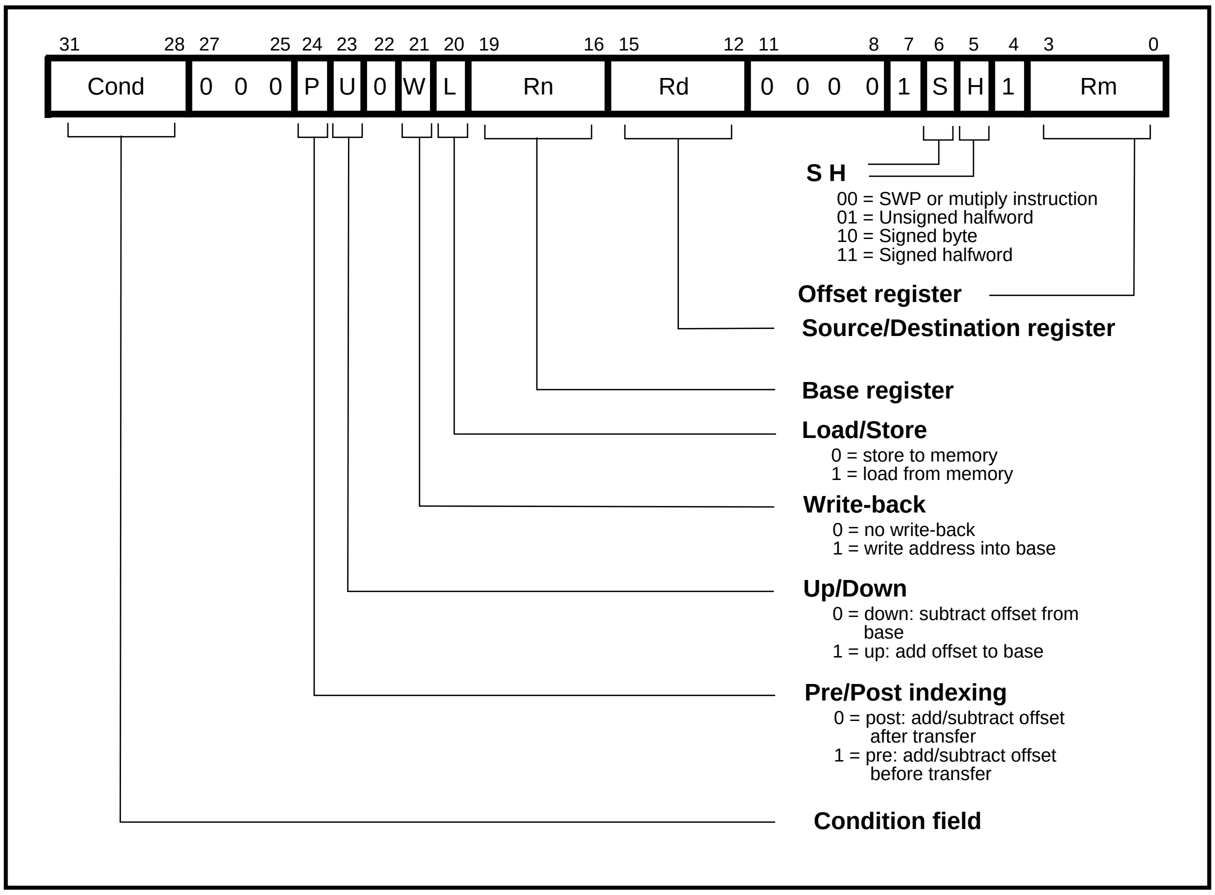


Figure 4-18: Halfword and signed data transfer with register offset

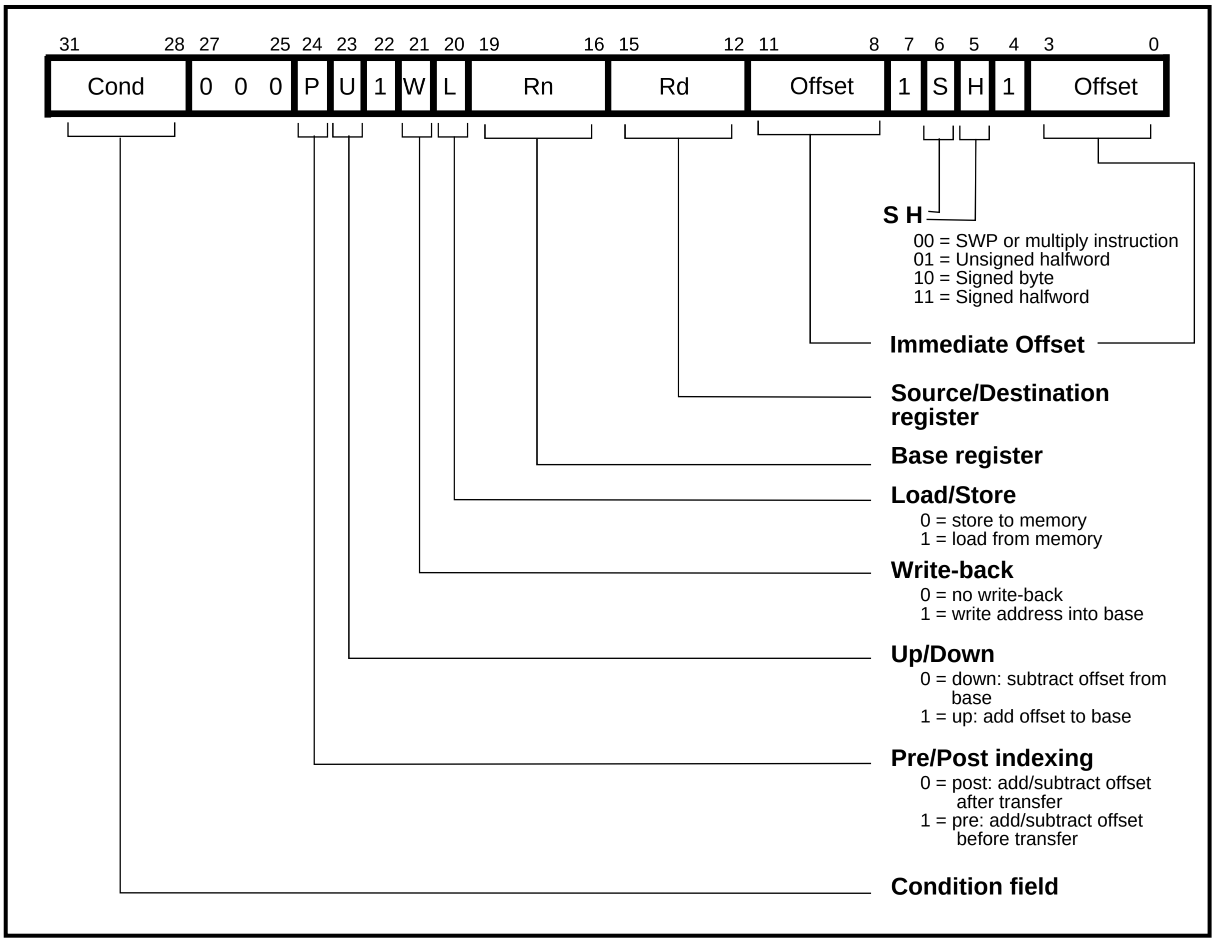


Figure 4-19: Halfword and signed data transfer with immediate offset

4.10.1 Offsets and auto-indexing

The offset from the base may be either an 8-bit unsigned binary immediate value in the instruction, or a second register. In the case of an immediate value, bits 11:8 (xxxx) and bits 3:0 (yyyy) combine to form the offset (xxxxyyyy). 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 must 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 halfwords between a 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 halfwords (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 halfword 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

Signed byte load (LDRSB): This load 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 bits of the destination register, and the remaining bits of the register are filled with the sign bit, the most significant bit of the byte. Please see ***Figure 3-1: Little-endian addresses of bytes within words*** on page 3-3.

Halfword load (LDRSH or LDRH): This load 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 on an odd halfword boundary, (A[1]=1).The supplied address should always be on a halfword boundary. If bit 0 of the supplied address is HIGH, an unpredictable value will be loaded. The selected halfword is placed in the bottom 16 bits of the destination register. For unsigned halfwords (LDRH), the top 16 bits of the register are filled with zeros and for signed halfwords (LDRSH) the top 16 bits are filled with the sign bit, the most significant bit of the halfword.

Halfword store (STRH): This store 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 The address must be halfword aligned; if bit 0 of the address is HIGH this causes unpredictable behaviour.

Big-endian configuration

Signed byte load (LDRSB): This 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 bits of the destination register, and the remaining bits of the register are filled with the sign bit, the most significant bit of the byte. Please see ***Figure 3-2: Big-endian addresses of bytes within words*** on page 3-3.

Halfword load (LDRSH or LDRH): This load 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 on an odd halfword boundary, (A[1]=1). The supplied address should always be on a halfword boundary. If bit 0 of the supplied address is HIGH, an unpredictable value is loaded. The selected halfword is placed in the bottom 16 bits of the destination register. For unsigned halfwords (LDRH), the top 16 bits of the register are filled with zeros and for signed halfwords (LDRSH) the top 16 bits are filled with the sign bit, the most significant bit of the halfword.

Halfword store (STRH): This store 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

Do not specify R15 as:

- the register offset (Rm)
- the destination register (Rd) of a load instruction (LDRH, LDRSH, LDRSB)
- the source register (Rd) of a store instruction (STRH, STRSH, STRSB)

Base register

Do not specify either write-back or post-indexing (which forces write-back) if R15 is specified as the base register (Rn). When using R15 as the base register you must remember that it contains an address 8 bytes on from the address of the current instruction.

4.10.6 Restrictions on the use of the base register

Do not specify post-indexed loads and stores where Rm and Rn are the same register, as they can be impossible to unwind after an abort.

Do not set register positions Rd and Rn to be the same register when a load instruction specifies (or implies) base write-back.

4.10.7 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of aborts in general.

In some situations a transfer to or from an address may cause a memory management system to generate an abort.

For example, in a system which uses virtual memory, the required data may be absent from main memory. The memory manager can signal a problem by signalling a Data Abort to the processor, whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, after which the instruction can be restarted and the original program continued.

In all cases, the base register is restored to its original value before the Abort trap is taken. In the case of an LDRH, LDRSB or LDRSH, the destination register (Rd) will not have been altered.

4.10.8 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

The cycle times are the same as LDR/STR for *all* cases of (H, SH, SB).

Load instructions take 1 cycle.

Store instructions take 1 cycle.

4.10.9 Assembler syntax

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

`LDR`	load from memory into a register
`STR`	Store from a register into memory
`{cond}`	two-character condition mnemonic. See ***4.3 The Condition Field*** on page 4-3
`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.
`<Addr>`	is one of:

1 An expression which generates an address:

```
<expression>
```

The assembler will attempt to generate an instruction using the PC as a base and an 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, this generates an error.

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.

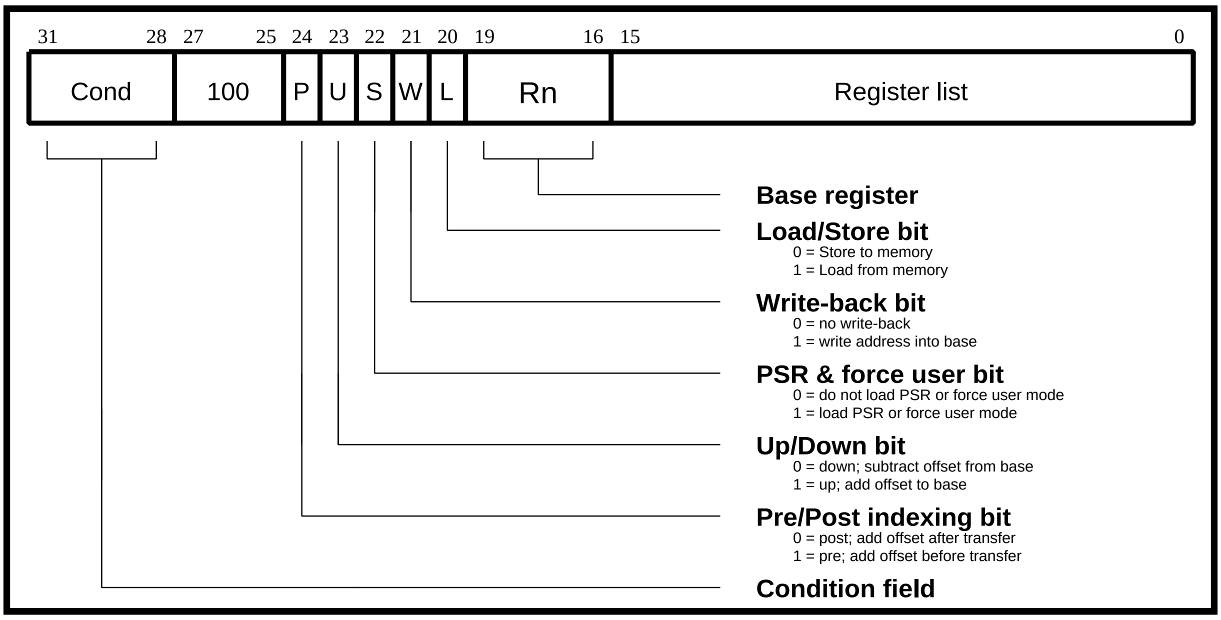
`Rn` and `Rm`	are expressions evaluating to a register number. If `Rn` is R15, neither post-indexed addressing nor `{!}` should be specified.
`{!}`	writes back the base register (sets the W bit) if ! is present.

4.10.10 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
                        ; 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.
HERESTRH  R5,[(PC, # (FRED-HERE-8)]
    .                   ; Generate PC relative offset to
    .                   ; address FRED. Store the halfword
    .                   ; in R5 at address FRED
    .
    .
FRED
```

4.11 Block Data Transfer (LDM, STM)

A block data transfer instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-20: Block data transfer instructions*** shows the instruction encoding.



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 4-20: 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 must not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 8. Note that this is different from previous ARMs which stored the address of the instruction plus 12 (or 8 if R15 is the only register in the list.)

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 stored such that the lowest register is always at the lowermost address in memory, the highest numbered register is always at the uppermost address, and the others are stored in numerical order between them.

The register transfers will occur in ascending order. 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). The figures beginning on page 4-42 show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, if write-back of the modified base was not 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. Non-word-aligned addresses do not affect the instruction: no data rotation occurs (as would happen in LDR.) However, the bottom 2 bits of the address will appear on A[1:0] and might be interpreted by the memory system.

4.11.4 Use of the S bit

When the S bit is set in a LDM/STM instruction, its meaning depends on whether R15 is in the transfer list and also on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode other than System mode.

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

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

STM with S bit set (User bank transfer)

The registers to be 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 must not be used when this mechanism is employed.

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

The user bank registers are loaded, rather than those in the bank corresponding to the current mode. This is useful for restoring the user state on process switches. Do not use base write-back when this mechanism is employed. Also, take care not to read from a banked register during the following cycle. (Inserting a NOP after the LDM will ensure safety.)

4.11.5 Use of R15

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

Note Bits [1:0] of R15 are set to zero when read from, and are ignored when written to.

4.11.6 Inclusion of the base in the register list

When write-back is specified during an STM, if the base register is the lowest numbered register in the list, then the original base value is stored. Otherwise the value stored is not specified and should not be used.

4.11.7 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of Aborts in general.

When a Data Abort occurs during LDM or STM instructions, further register transfers are stopped. The base register is always restored to its original value (before the instruction had executed) regardless of whether writeback was specified or not. As such, the instruction can always be restarted without any need to adjust the value of the base register in the Data Abort service routine code.

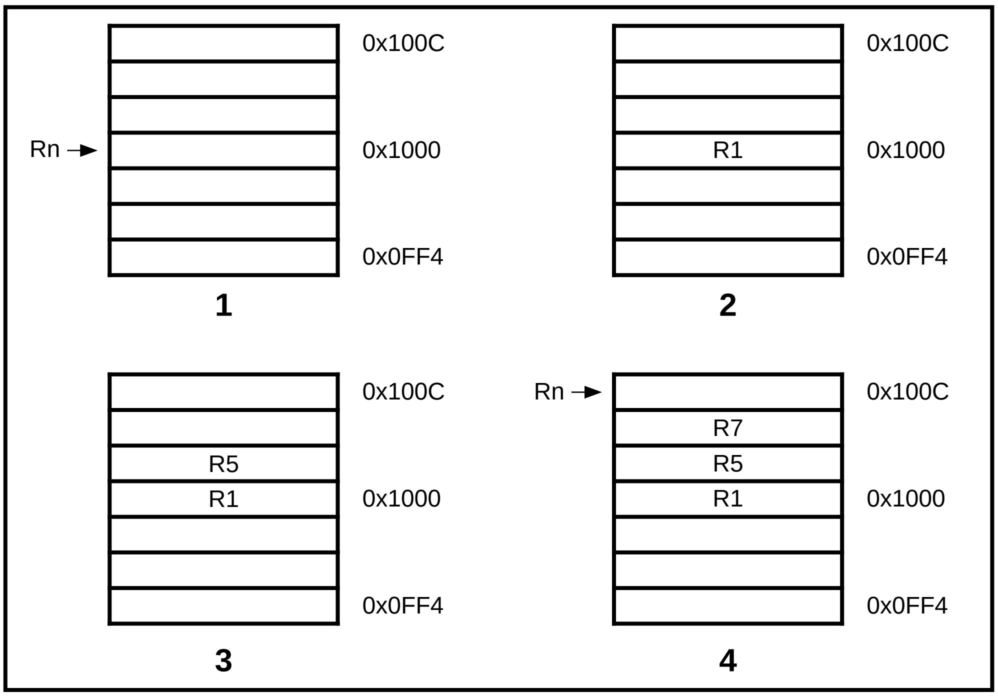


Figure 4-21: Post-increment addressing

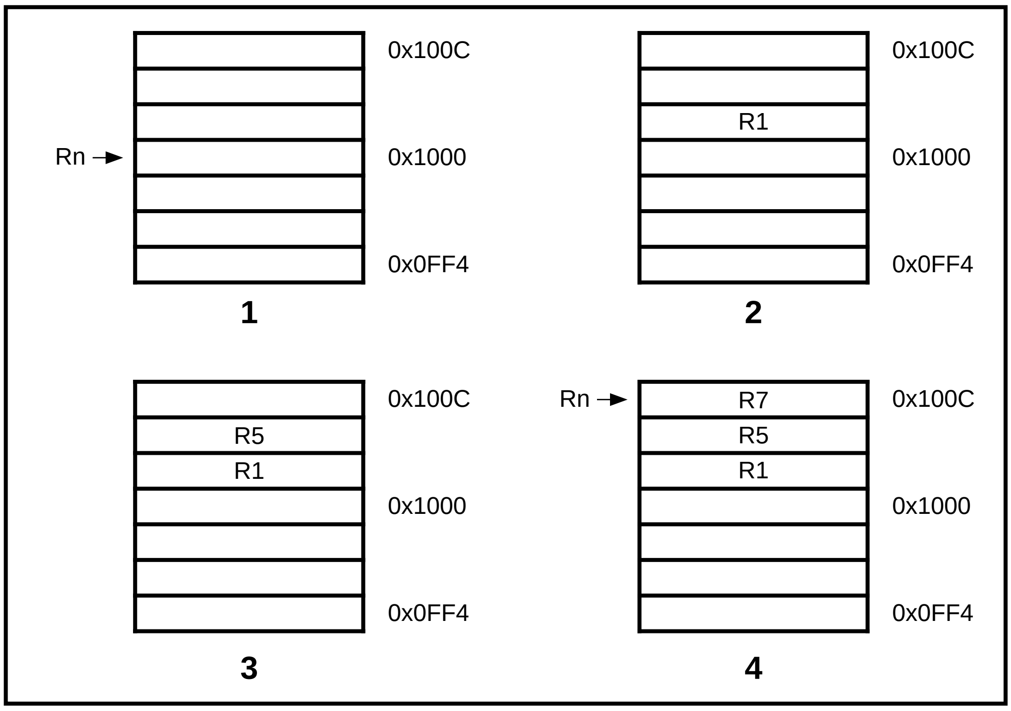


Figure 4-22: Pre-increment addressing

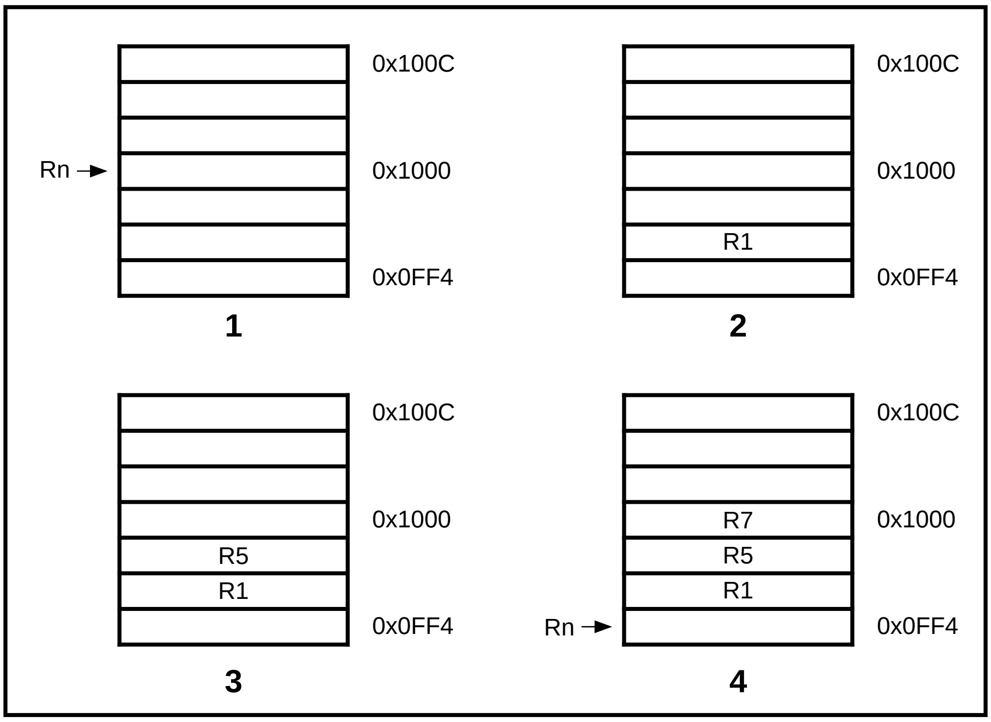


Figure 4-23: Post-decrement addressing

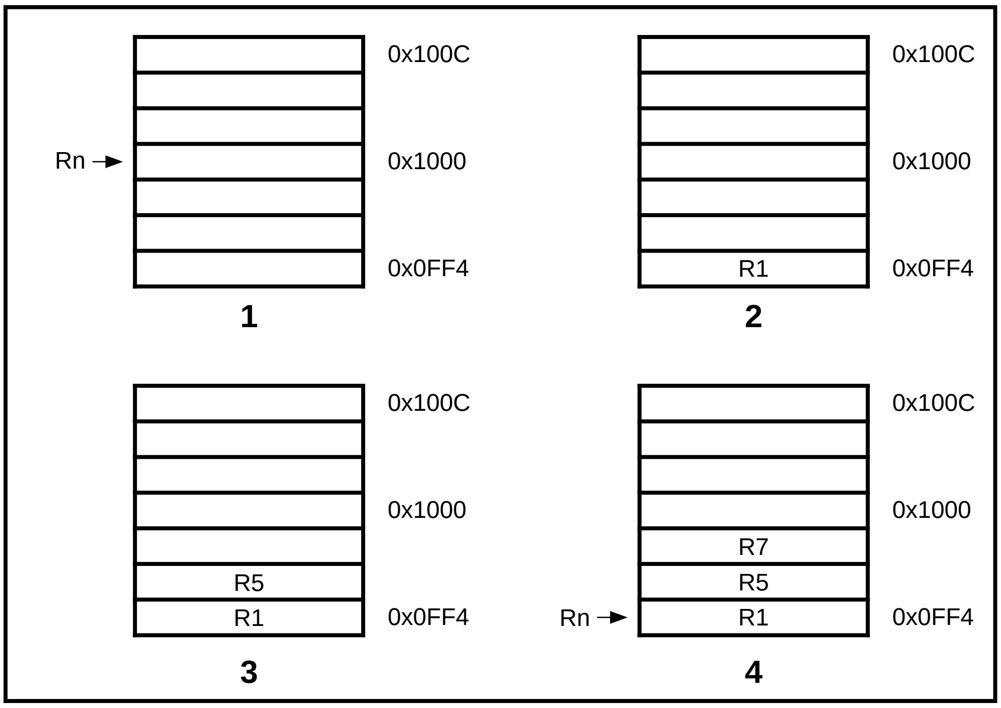


Figure 4-24: Pre-decrement addressing

4.11.8 Instruction cycle times

Note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

The cycle count for LDM instructions depends on the number of ordinary registers being loaded (excluding R15), and whether R15 is being loaded.

The following table shows the basic cycle count for LDM.

Number of Ordinary Registers transferred	Cycles when PC (R15) is not in register list	Cycles when PC (R15) is in register list
0	-	5
1	2	6
2	2	6
3	3	7
4	3	7
5	4	8
6	4	8
7	5	9
8	5	9
9	6	10
10	6	10
11	7	11
12	7	11
13	8	12
14	8	12
15	9	13

Table 4-3: Basic cycle count for LDM

The above assumes that the memory system supports double-bandwidth transfer. If this is not so, then count N cycles for the number of registers being transferred, plus 5 cycles if R15 is loaded, with a minimum of two cycles overall.

A common example of where this might happen in a cached memory system would be when uncacheable memory is being accessed.

Additional cycles may be incurred if the memory system indicates that it is only able to transfer one item of data where two were requested. For example, when accessing the last word in a cache line in a cached memory system.

The following table shows the cycle counts for STM instructions.

Number of Ordinary Registers transferred	Cycles when PC (R15) is not in register list	Cycles when PC (R15) is in register list
0	-	2
1	2	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	10
10	10	11
11	11	12
12	12	13
13	13	14
14	14	15
15	15	16

Table 4-4: Basic cycle count for STM

Note PC is stored as the address of the current instruction plus 8.

4.11.9 Assembler syntax

The block data transfer instructions have the following syntax:

```
<LDM|STM>{cond}<addressmode> Rn{!},<Rlist>{^}
```

where:

`LDM`	loads from memory to registers.
`STM`	stores from registers to memory.
`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`<addressmode>`	is one of `<FD|ED|FA|EA|IA|IB|DA|DB>`. Note that `<addressmode>` is *not* optional. (See ***Table 4-5: Addressing Mode names*** on page 4-47)
`Rn`	is an expression evaluating to a register number.
`<Rlist>`	is a list of registers and register ranges enclosed in {} (eg {R0,R2-R7,R10}).
`{!}`	if present, requests write-back (W=1), otherwise W=0.
`{^}`	if present, sets the S bit. See ***4.11.4 Use of the S bit*** on page 4-41.

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. These are shown in ***Table 4-5: Addressing Mode names*** on page 4-47.

Key to table:

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required.

F	Full stack (a pre-index has to be done before storing to the stack)
E	Empty stack
A	Ascending stack (a STM will go up and LDM down)
D	Descending stack (a STM will go down and LDM up)

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks:

IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

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

Table 4-5: Addressing Mode names

Name	Stack	Other	L bit	P bit	U bit
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 4-5: Addressing Mode names

4.11.10 Examples

```
LDMFDSP!,{R0,R1,R2}; unstack 3 registers
STMIAR0,{R0-R15}   ; save all registers

LDMFDSP!,{R15}     ; unstack R15,CPSR unchanged

LDMFDSP!,{R15}^    ; unstack R15, CPSR <- SPSR_mode
                   ; (allowed only in privileged modes)

STMFDR13,{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:

```
STMEDSP!,{R0-R3,R14}; save R0 to R3 to use as workspace
                   ; and R14 for returning

BL  somewhere      ; this nested call will overwrite R14

LDMEDSP!,{R0-R3,R15}; restore workspace and return
```

4.12 Single Data Swap (SWP)

A data swap instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-25: Swap instruction*** shows the instruction encoding.

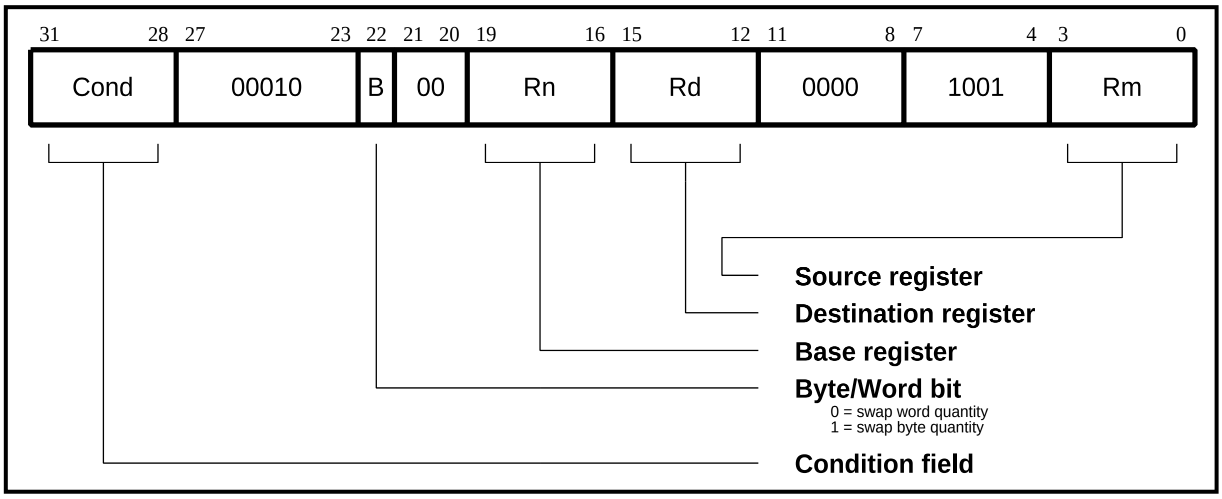


Figure 4-25: Swap instruction

The data swap instruction is used to swap a byte or word quantity between a register and external memory. It 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. It then 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.

Both the read and the write operations result in external accesses to main memory regardless of whether the cache hits or misses. In the case of a cache hit during the write operation, the cache line is updated with the new value and is not marked as dirty.

Swap Read operation:

This performs a single word or byte read that always goes to the external bus, leaving the bus locked for the subsequent write.

Swap Write operation:

This performs a single word or byte write that always goes to the external bus as an unbuffered write. If the write is a cache hit, the cache data is updated and the dirty bit is left unchanged.

The **LOCK** signal on the external interface is used to signal to the external memory manager that the read and write operations of the swap are locked together and should be allowed to complete without interruption; see ***Chapter 12, Bus Interface*** for further details. This operation is important in multi-processor systems, where the swap instruction is the only indivisible operation which may be used to implement semaphores.

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 ARM810 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in ***4.9 Single Data Transfer (LDR, STR)*** on page 4-27. In particular, the description of big and little-endian configuration applies to the SWP instruction. Note that there is no halfword SWP.

4.12.2 Use of R15

R15 must not be used as an operand (Rd, Rn or Rm) in a SWP instruction.

4.12.3 Data aborts

Please refer to ***3.6.3 Aborts*** on page 3-9 for details of Aborts in general.

In some situations, a transfer to or from an address may cause the memory management system to generate an Abort.

If the read operation is aborted, the abort will be returned to ARM8, the write will not take place and the locked indication will be removed from the external bus.

If the read operation succeeds and the write operation is aborted, the abort will be returned to ARM8 and the cache entry will be left with the updated (written) data value. The line will not be invalidated in the cache—this could be done by the abort handler if necessary.

4.12.4 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses.

SWP instructions take 2 cycles.

4.12.5 Assembler syntax

The SWP instruction has the following syntax:

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`{B}`	specifies byte transfer. If omitted, word transfer is used.
`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

SWPBR2,R3,[R4]       ; load R2 with the byte addressed by R4,
                     ; and store bits 0 to 7 of R3 at R4

SWPEQR0,R0,[R1]      ; conditionally swap the contents of the
                     ; word addressed by R1 with R0
```

4.13 Software Interrupt (SWI)

A SWI instruction is only executed if the specified condition is true. The various conditions are defined at the beginning of this chapter. ***Figure 4-26: Software interrupt instruction*** shows the instruction encoding.

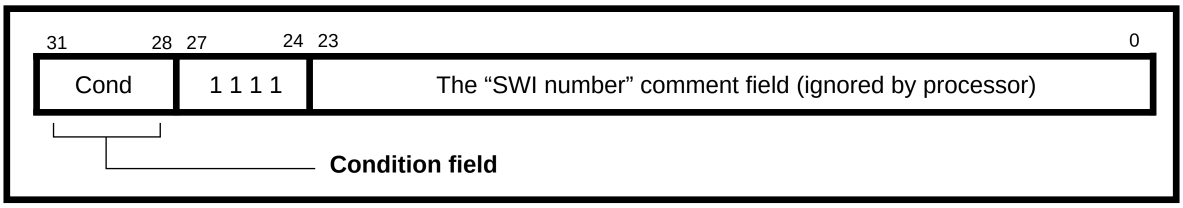


Figure 4-26: Software interrupt instruction

The software interrupt is used to enter Supervisor mode in a controlled manner. It causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to the SWI vector and the CPSR is saved in SPSR_svc. See ***3.6.4 Software interrupt*** on page 3-10 for more details.

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 and the CPSR in SPSR_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.

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. This is commonly referred to as the "SWI number".

4.13.3 Architecturally defined SWIs

The ARM Architecture V4 reserves SWI numbers 0xF00000 to 0xFFFFFF inclusive for current and future Architecturally Defined SWI functions. These SWI numbers should not be used for functions other than those defined by ARM. Please see ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 for examples of two such definitions.

Architecturally defined SWI functions are used to provide a well-defined interface between code which is:

- independent of the ARM processor implementation on which it is running, and
- specific to the ARM processor implementation on which it is running.

The implementation-independent code is provided with a function that is available on all processor implementations via the SWI interface, and which may be accessed by privileged and, where appropriate, non-priviledged (User mode) code.

The Architecturally defined SWI instructions must be implemented in the SWI handler using processor specific code sequences supplied by ARM. Please refer to ***Appendix E, Implementing the Instruction Memory Barrier Instruction*** for details.

4.13.4 Instruction cycle times

Please note that the cycle times given here are given for the ARM8 processor core, and do not give any information about the additional cycles that may be taken as a result of Cache Misses, MMU Page table walks etc. Future versions of the ARM810 Datasheet will provide such information.

Please refer to ***Chapter 12, Bus Interface*** for timing details of off-chip accesses

SWI instructions take 4 cycles to execute.

4.13.5 Assembler syntax

The SWI instruction has the following syntax:

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

where:

`{cond}`	is a two-character condition mnemonic. The assembler assumes AL (ALways) if no condition is specified.
`<expression>`	is evaluated and placed in the comment field (which is ignored by ARM810).

4.13.6 Examples

```
SWI ReadC          ; get next character from read stream
SWI WriteI+”k”     ; output a ”k” to the write stream
SWINE0             ; conditionally call supervisor
                   ; with 0 in comment field
```

The above examples assume that suitable supervisor code exists at the SWI vector address, for instance:

```
    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 entry table start address
    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
```

Note ADR is a directive that instructs the assembler to use an ADD or SUB instruction to create the address of a label, so in the above instance

```
    ADR    R2,EntryTable
```

is equivalent to

```
    SUB    R2,R15,#{PC}+8-EntryTable
```

4.14 Coprocessor Data Operations (CDP)

ARM810 will bounce all CDP instructions , forcing them to take the Undefined Instruction trap. The coprocessor instruction may then be emulated.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-27: 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 the ARM810, and it may 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 the ARM810 to perform independent tasks in parallel.

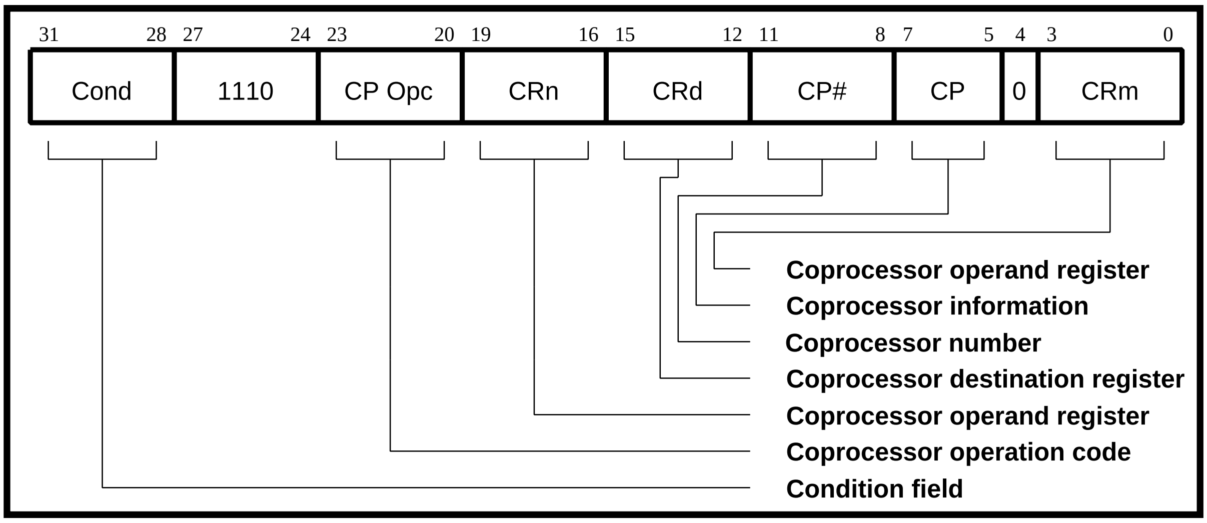


Figure 4-27: Coprocessor data operation instruction

4.14.1 The coprocessor fields

Only bit 4 and bits 24 to 31 are significant to the processor. 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 must 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

All CDP instructions must be emulated in software: the number of cycles taken will depend on the coprocessor support software.

4.14.3 Assembler syntax

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

where:

`{cond}`	two character condition mnemonic, see ***Figure 4-2: Condition codes*** on page 4-3
`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
CDPEQp2,5,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)

ARM810 will bounce all LDC and STC instructions, forcing them to take the Undefined Instruction trap. The coprocessor instruction may then be emulated.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-29: Coprocessor register transfer instructions***.

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

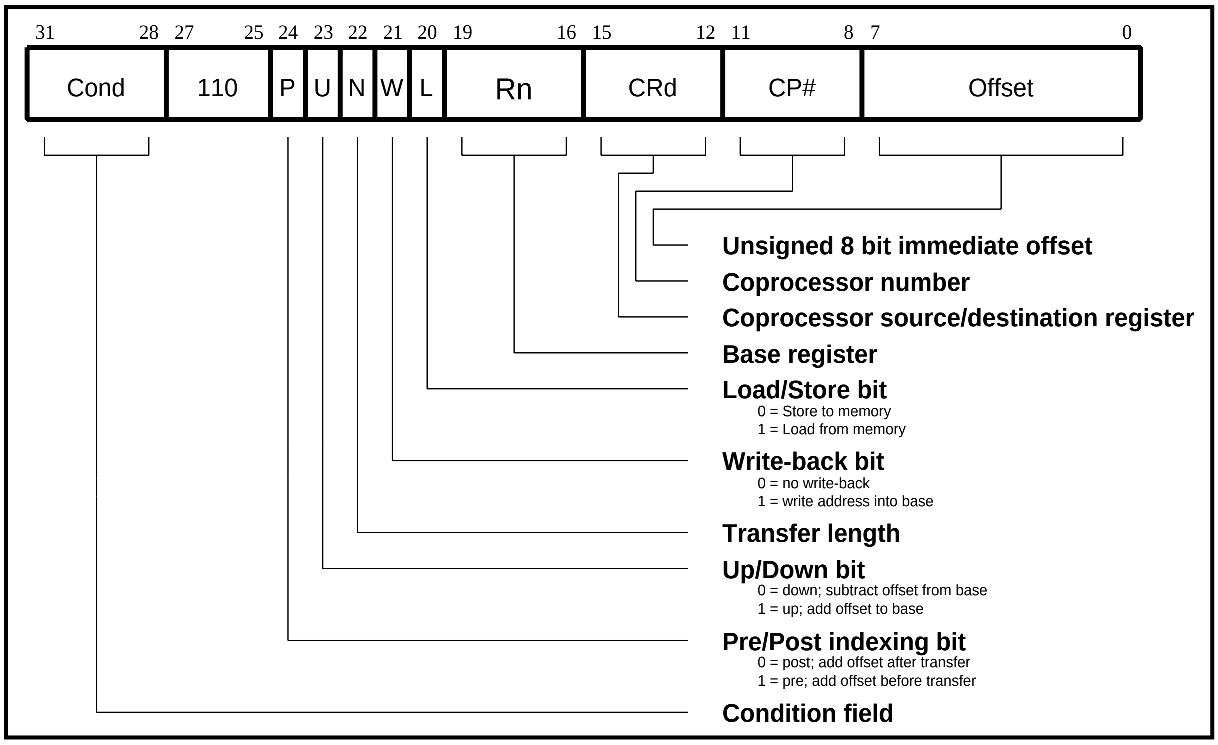


Figure 4-28: Coprocessor data transfer instructions

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

The processor 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 for coprocessor data transfers the immediate offsets are 8 bits wide and specify *word* offsets, whereas for single data transfers they are 12 bits wide and specify *byte* offsets.

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. Instructions where P=0 and W=0 are reserved, and must not be used.

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 shall not be specified.

4.15.5 Data aborts

If the address is legal but the memory manager generates an abort, the data abort trap is taken. The base register is restored to its original value, and all other processor state are preserved. Any coprocessor emulation is partly responsible for ensuring that the data transfer can restart after the cause of the abort is resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

4.15.6 Instruction cycle times

All LDC and STC instructions must be emulated in software: the number of cycles taken will depend on the coprocessor support software.

4.15.7 Assembler syntax

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

where:

`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 ***Figure 4-2: Condition codes*** on page 4-3.
`p#`	the unique number of the required coprocessor
`cd`	expression evaluating to a valid coprocessor register number that is placed in the CRd field
`<Addr>`	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	expression evaluating to a valid ARM810 register number

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 Though 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)

ARM810 has only one internal coprocessor; CP15, the system control coprocessor. The MRC and MCR instructions are used to transfer register contents between the core and the coprocessor. Please refer to ***Chapter 5, Configuration*** for details of the register arrangement and operations.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in ***Figure 4-29: Coprocessor register transfer instructions***.

This class of instruction is used to communicate information directly between ARM810 and a coprocessor. An example of a coprocessor to processor 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 a processor register. A FLOAT of a 32-bit value in a processor register into a floating point value within the coprocessor illustrates the use of a processor register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the processor 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.

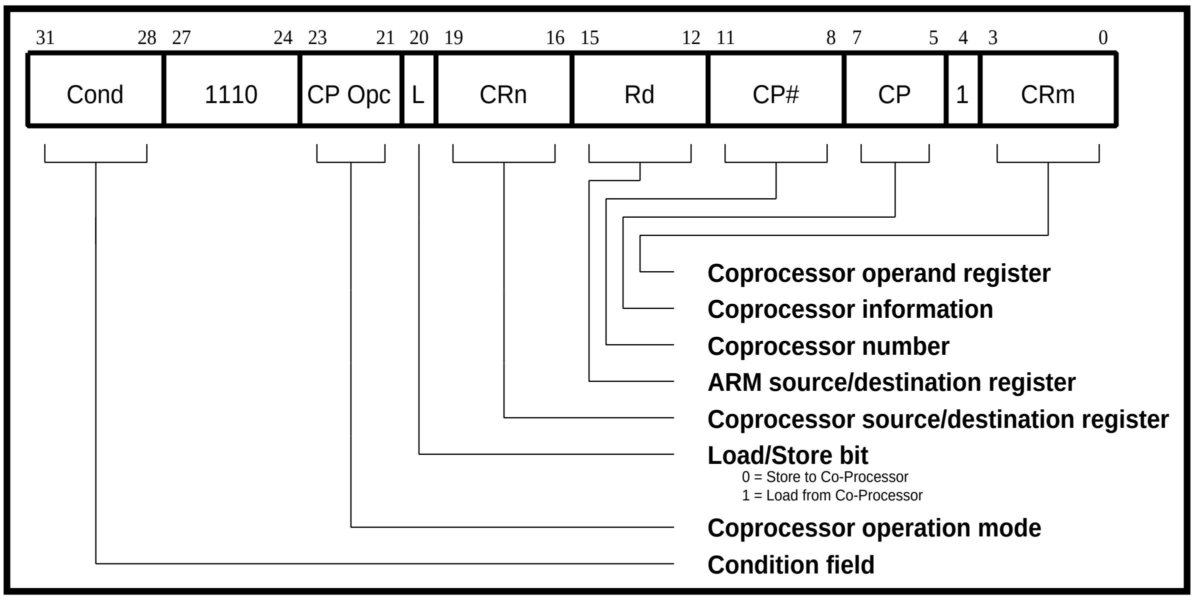

31
28
27
24
23
21
20
19
16
15
12
11
8
7
5
4
3
0
Cond
1110
CP Opc
L
CRn
Rd
CP#
CP
1
CRm
Coprocessor operand register
Coprocessor information
Coprocessor number
ARM source/destination register
Coprocessor source/destination register
Load/Store bit
0 = Store to Co-Processor
1 = Load from Co-Processor
Coprocessor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

Figure 4-29: 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 from R15

Do not specify a coprocessor register transfer from ARM810 with R15 as the source register.

4.16.3 Transfers to R15

When a coprocessor register transfer to ARM810 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.

4.16.4 Instruction cycle times

Both the MRC and MCR instructions take 1 cycle to execute, provided that the coprocessor does not “busy-wait” them.

4.16.5 Assembler syntax

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

where:

`MRC`	move from coprocessor to ARM810 register (L=1)
`MCR`	move from ARM810 register to coprocessor (L=0)
`{cond}`	two-character condition mnemonic, see ***Figure 4-2: Condition codes*** on page 4-3
`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 ARM810 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,c6,c7   ; request coproc 6 to perform operation 0
```

```
                                ; on R4 and place the result in c6, in a
                                ; way that may be influenced by c7

        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 The Instruction Memory Barrier (IMB) Instruction

An Instruction Memory Barrier (IMB) Instruction is used to ensure that correct instruction flow occurs after instruction memory locations are altered in any way - by self-modifying code for example. The recommended implementation of the IMB instructions is via an architecturally defined SWI function (see ***4.13 Software Interrupt (SWI)*** on page 4-52). The instruction encoding for the recommended IMB instruction implementations is shown below:

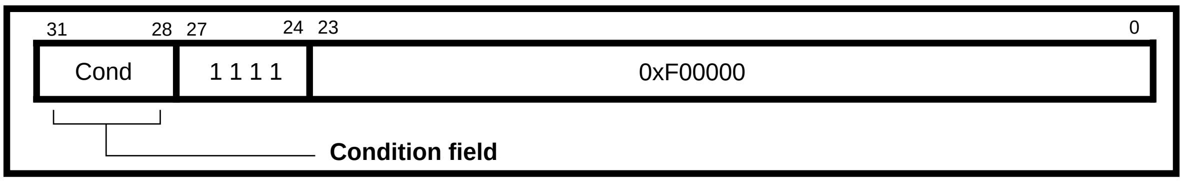


Figure 4-30: IMB instruction

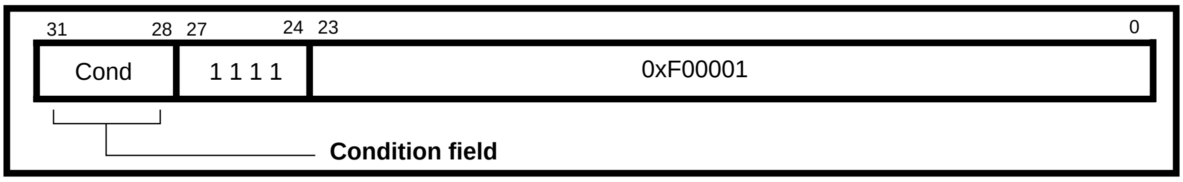


Figure 4-31: IMBRange instruction

IMBRange: Registers R0 and R1 contain the Range of addresses on entry to the SWI. R0 is the lower (inclusive) address and R1 is the upper address (not included in the range).

4.17.1 Use

During the normal operation of ARM8, the Prefetch Unit (PU) reads instructions ahead of the core in order to attempt to remove branches. It does this by predicting whether or not the branches are taken and then prefetching from the predicted address.

If a program changes the contents of memory with the intention of executing the new contents as new instructions, then any prefetched instructions and/or other stored information about instructions in the PU may be out of date because the instructions concerned have been overwritten. Thus the PU holds the wrong instructions; if passed to the execution unit they would cause unintentional behaviour.

In order to prevent such problems, an IMB instruction must be used between changing the contents of memory and executing the new contents to ensure that any stored instructions are flushed from the PU. The choice of IMB Instruction (IMB or IMBRange) depends upon the amount of code changed.

The IMB Instruction flushes all stored information about the instruction stream.

The IMBRange Instruction flushed all stored information about instructions at addresses in the range specified.

Please refer to ***Appendix E, Implementing the Instruction Memory Barrier Instruction*** for further details of the IMB implementation and use.

4.17.2 Assember syntax

```
SWI{cond} IMB           ; Where IMB = 0xF00000

; code that loads R0 and R1 with Range addresses
SWI{cond} IMBRange      ; Where IMBRange = 0xF00001
```

4.17.3 Examples

Loading code from disc

Code that loads a program from a disc, and then branches to the entry point of that program, should execute an IMB instruction between loading the program and trying to execute it.

```
IMB EQU    0xF00000
    .
    .
    ; code that loads program from disc
    .
    .
    SWI    IMB
    .
    .
    MOV    PC, entry_point_of_loaded_program
    .
    .
```

Running BitBlt code

“Compiled BitBlt” routines optimise large copy operations by constructing and executing a copying loop which has been optimised for the exact operation wanted.

When writing such a routine an IMB is needed between the code that constructs the loop and the actual execution of the constructed loop.

```
IMBRange EQU     0xF00001
         .
         .
         ; code that constructs loop code
         ; load R0 with start address of the constructed loop
         ; load R1 with the end address of the constructed loop
         SWI     IMBRange
         ; start of constructed loop code
         .
         .
```

Self-decompressing code

When writing a self-decompressing program, an IMB should be issued after the routine which decompresses the bulk of the code and before the decompressed code starts to be executed.

```
IMB EQU    0xF00000
    .
    .
    ; copy and decompress bulk of code
    SWI    IMB
    ; start of decompressed code
```

4.18 Undefined Instructions

This section shows the instruction bit patterns that will cause the Undefined Instruction trap to be taken if ARM810 attempts to execute them. This vector location is defined in ***3.6.6 Exception vector summary*** on page 3-11. There are a number of such bit pattern classes, and these can be used to cause unimplemented instructions (for example LDC) to be emulated through the Undefined Instruction trap service routine code:

Class A	Undefined instructions in previous ARM processor implementations
Class B	Unallocated MSR/MRS-like instructions
Class C	Unallocated Multiply-like instructions
Class D	Unallocated SWP-like instructions
Class E	Unallocated STRH/LDRH/LDRSH/LDRSB-like instructions

Note *Some or all of Classes B through E may not fall into the Undefined Instruction trap if further implementation restrictions dictate this. ARM reserves the right to make these decisions as necessary.*

Class	Instruction Bit Pattern								Notes
A	Cond	011x	xxxx	xxxx	xxxx	xxxx	xxx1	xxxx	
B	Cond Cond Cond	0001 0001 0011	0xx0 0xx0 0x00	xxxx xxxx xxxx	xxxx xxxx xxxx	xxxx xxxx xxxx	yyy0 0xx1 xxxx	xxxx xxxx xxxx	yyy != 000
C	Cond	0000	01xx	xxxx	xxxx	xxxx	1001	xxxx	
D	Cond	0001	yyyy	xxxx	xxxx	xxxx	1001	xxxx	yyyy !=0000 or 0100
E	Cond Cond	0000 000x	xx1x xxx0	xxxx xxxx	xxxx xxxx	xxxx xxxx	1yy1 11x1	xxxx xxxx	yy !=00

Table 4-6: Bit patterns for the undefined instruction trap

The Undefined Instruction trap is taken:

- if the condition specified by Cond is met and the instruction bit pattern is in ***Table 4-6: Bit patterns for the undefined instruction trap***

or

- by all coprocessor instructions whose condition is met and which are bounced by any coprocessor. For ARM810, the coprocessor interface must bounce all CDP, LDC and STC instructions

4.18.1 Assembler syntax

At present the assembler has no mnemonics for generating Undefined Instruction classes A through to E.

4.19 Instruction Set Examples

The following examples show ways in which the basic ARM810 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.

4.19.1 Using the conditional instructions

Using conditionals for logical OR

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

can be replaced by :

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

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.

```
; Unsigned divide of r1 by r0
; Returns quotient in r0, remainder in r1
; Destroys r2, r3
       MOV     r3, #0
       MOVS    r2, r0
       BEQ     |__rt_div0|    ; jump to divide-by-zero
                              ; error handler
; justification stage shifts r2 left 1 bit at a time
; until r2 > (r1/2)
u_loop
       CMP     r2, r1, LSR #1
       MOVLS   r2, r2, LSL #1
       BCC     u_loop
; now division proper can start
u_loop2
       CMP     r1, r2              ; perform divide step
       ADC     r3, r3, r3
       SUBCS   r1, r1, r2
       TEQ     r2, r0              ; all done yet?
       MOVNE   r2, r2, LSR #1
       BNE     u_loop2
       MOV     r0, r3
```

4.19.2 Multiply overflow detection in the ARM810

Overflow in unsigned multiply with a 32 bit result

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

Overflow in signed multiply with a 32 bit result

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

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
```

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
```

Overflow in unsigned multiply accumulate with a 64 bit result

```
         SMULL    R1,Rh,Rm,Rn    ;4 to 7 cycles
         ADDS     Rl,Rl,Ra1      ;lower accumulate
         ADCS     Rh,Rh,Ra2      ;upper accumulate
         BCS      overflow       ;2 cycles and 2 registers
```

Overflow in signed multiply accumulate with a 64 bit result

```
         UMULL    R1,Rh,Rm,Rn    ;4 to 7 cycles
         ADDS     Rl,Rl,Ra1      ;lower accumulate
         ADCS     Rh,Rh,Ra2      ;upper accumulate
         BVS      overflow       ;2 cycles and 2 registers
```

Note Overflow cannot occur in signed and unsigned multiply with a 64-bit result, so overflow checking is not applicable.

4.19.3 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 of maximal length (ie. 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 (ie. 32 bits).

```
                    ; enter with seed in Ra (32 bits),
                    ; Rb (1 bit in Rb lsb), uses Rc

TST Rb,Rb,LSR#1     ; top bit into carry
MOVSRc,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
```

4.19.4 Multiplication by constant using shifts

1 Multiplication by 2^n (1,2,4,8,16,32..)
```
    MOV      Ra, Rb, LSL #n
```
2 Multiplication by 2^n+1 (3,5,9,17..)
```
    ADD      Ra,Ra,Ra,LSL #n
```
3 Multiplication by 2^n-1 (3,7,15..)
```
    RSB      Ra,Ra,Ra,LSL #n
```
4 Multiplication by 6
```
    ADD      Ra,Ra,Ra,LSL #1; multiply by 3
    MOV      Ra,Ra,LSL#1    ; and then by 2
```
5 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
```

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

a) 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
```

b) 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
```

c) 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
```

4.19.5 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
LDMIARb,{Rd,Rc}     ; get 64 bits containing answer
AND Rb,Ra,#3        ; correction factor in bytes
MOVSRb,Rb,LSL#3     ; ...now in bits and test if aligned
MOVNERd,Rd,LSR Rb   ; produce bottom of result word
                    ; (if not aligned) for little-endian
                    ; operations (see note below)
RSBNERb,Rb,#32      ; get other shift amount
ORRNERd,Rd,Rc,LSL Rb; combine two halves to get result
                    ; for little-endian operation (see note
                    ; below)
```

Note: for Big-endian operation replace the first “LSR” with “LSL” and the final “LSL” by “LSR”.

5 Configuration

This chapter describes the configuration.

Configuration

The operation and configuration of ARM810 is controlled both directly via coprocessor instructions and indirectly via the Memory Management Page tables. The coprocessor instructions manipulate a number of on-chip registers which control the configuration of the Cache, write buffer, MMU and a number of other configuration options.

5.1 ARM810 System Control Coprocessor (CP15) Register Map

5.1.1 CP15 registers

CP15 defines 16 registers. ***Table 5-1: CP15 register summary*** on page 5-4 shows which registers are defined for reading and which for writing. All CP15 register bits which are defined and contain state are set to zero by Reset.

CP15 registers can only be accessed with MRC and MCR instructions in a Privileged mode. The instruction bit pattern of the MCR and MRC instructions is shown below:

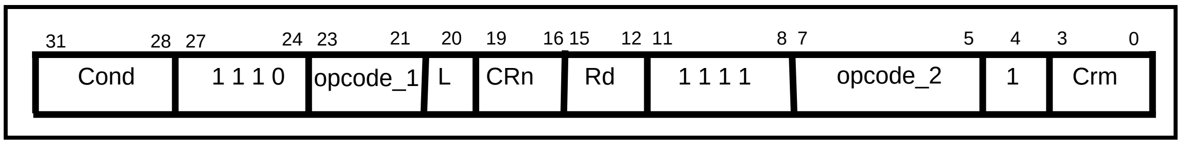


Figure 5-1: MRC, MCR bit pattern

CDP, LDC and STC instructions, along with unprivileged MRC and MCR instructions to CP15 will cause the undefined instruction trap to be taken. The CRn field of MRC and MCR instructions specify the coprocessor register to access. The CRm field and opcode_2 field are used to specify a particular action when addressing some registers.

Attempting to read from a register which is not defined for reading, or writing to a register which is not defined for writing will cause the instruction to take the undefined instruction trap. See ***5.1.2 Architectural Compliance of ARM810 CP15*** on page 5-12. In all instructions which access CP15:

- the opcode_1 field SHOULD BE ZERO
- the opcode_2 and CRm fields SHOULD BE ZERO except when accessing registers 7 and 8, when the values specified below should be used to select the desired Cache and TLB operations. Using a value other than those specified below for opcode_2 and CRm when accessing registers 7 and 8, or other than zero when accessing other registers, will cause ARM810 to take the undefined instruction trap. See ***5.1.2 Architectural Compliance of ARM810 CP15*** on page 5-12.

Throughout this section the following terms and abbreviations are used:

UNPREDICTABLE	UNP	If specified for reads: the data returned when reading from this location is unpredictable - it could have any value. If specified for writes: writing to this location will cause unpredictable behaviour or an unpredictable change in device configuration.
UNDEFINED	UND	An instruction that accesses CP15 in the manner indicated will take the undefined instruction trap.
SHOULD BE ZERO	SBZ	When writing to this location, all bits of this field should be 0.

In all cases, reading from, or writing any data values to any CP15 registers, including those fields specified as UNPREDICTABLE or SHOULD BE ZERO will not cause any permanent damage to the ARM810.

Register	Reads	Writes
0	ID Register	UNDEFINED
1	Control	Control
2	Translation Table Base	Translation Table Base
3	Domain Access Control	Domain Access Control
4	UNDEFINED	UNDEFINED
5	Fault Status	Fault Status
6	Fault Address	Fault Address
7	UNDEFINED	Cache operations
8	UNDEFINED	TLB operations
9	Cache Lock-Down	Cache Lock-Down
10	TLB Lock-Down	TLB Lock-Down
11 to 14	UNDEFINED	UNDEFINED
15	Clock and Test Configuration	Clock and Test Configuration

Table 5-1: CP15 register summary

Register 0: ID register

Reading from CP15 register 0 returns the value 0x4101810x. The CRm and opcode_2 fields SHOULD BE ZERO when reading CP15 register 0.

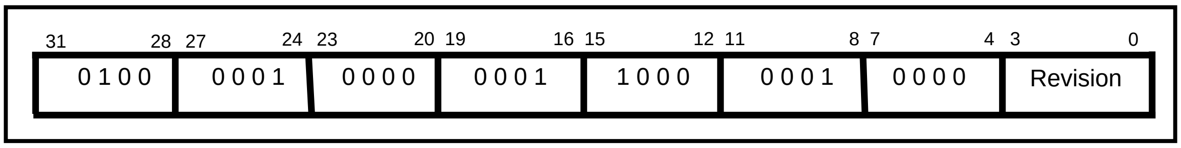


Figure 5-2: ID register read

Writing to CP15 register 0 is UNPREDICTABLE.

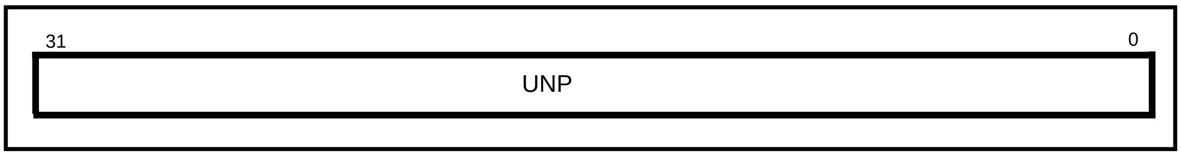


Figure 5-3: ID register write

Register 1: Control register

Reading from CP15 register 1 reads the control bits. The CRm and opcode_2 fields SHOULD BE ZERO when reading CP15 register 1.

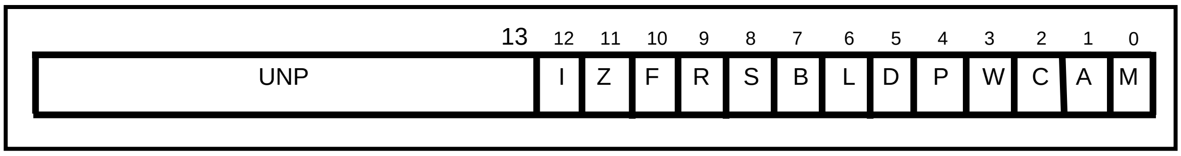


Figure 5-4: Register 1 read

Writing to CP15 register 1 sets the control bits. The CRm and opcode_2 fields SHOULD BE ZERO when writing CP15 register 1.

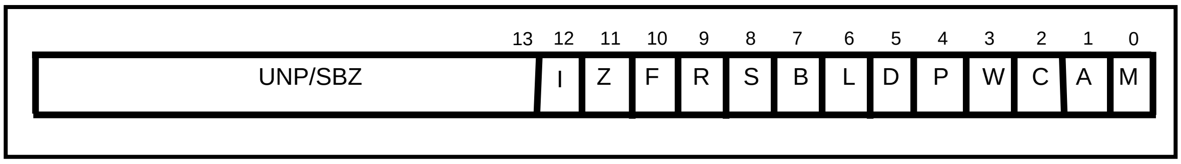


Figure 5-5: Register 1 write

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

M Bit 0	MMU Enable/Disable 0 = Memory Management Unit (MMU) disabled 1 = Memory Management Unit (MMU) enabled
A Bit 1	Alignment Fault Enable/Disable 0 = Address Alignment Fault Checking disabled 1 = Address Alignment Fault Checking enabled
C Bit 2	Cache Enable/Disable 0 = Instruction and/or Data Cache (IDC) disabled 1 = Instruction and/or Data Cache (IDC) enabled
W Bit 3	Write buffer Enable/Disable 0 = Write Buffer disabled 1 = Write Buffer enabled
P Bit 4	When read returns one, and when written is ignored.
D Bit 5	When read returns one, and when written is ignored.
L Bit 6	When read returns one, and when written is ignored.
B Bit 7	Big-endian/Little-endian 0 = Little-endian operation 1 = Big-endian operation
S Bit 8	System protection This bit modifies the MMU protection system.
R Bit 9	ROM protection This bit modifies the MMU protection system.
F Bit 10	When read returns zero. When written SHOULD BE ZERO.

Z Bit 11	Branch Prediction Enable/Disable 0 = Branch Prediction Disabled 1 = Branch Prediction Enabled.
I Bit 12	When read returns zero. When written SHOULD BE ZERO.
Bits 31:13	When read returns an UNPREDICTABLE value, and when written SHOULD BE ZERO, or a value read from these bits on the same processor. Note that using a read-write-modify sequence when modifying this register provides the greatest future compatibility.

Enabling the MMU

Care must be taken if the translated address differs from the untranslated address as the instructions following the enabling of the MMU will have been fetched using no address translation and enabling the MMU may be considered as a branch with delayed execution. A similar situation occurs when the MMU is disabled. The correct code sequence for enabling and disabling the MMU is implementation defined

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

Register 2: Translation table base register

Reading from CP15 register 2 returns the pointer to the currently active first level translation table in bits[31:14] and an UNPREDICTABLE value in bits[13:0].The CRm and opcode_2 fields SHOULD BE ZERO when reading CP15 register 2.

Writing to CP15 register 2 updates the pointer to the currently active first level translation table from the value in bits[31:14] of the written value. Bits[13:0] SHOULD BE ZERO. The CRm and opcode_2 fields SHOULD BE ZERO when writing CP15 register 2.

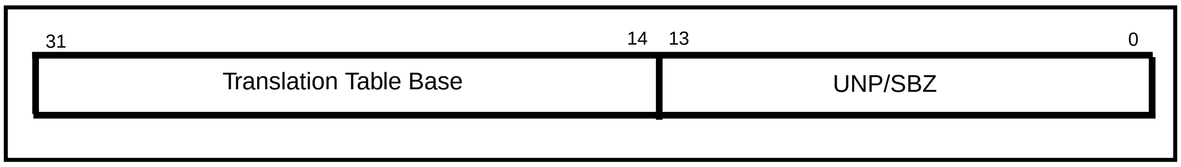


Figure 5-6: Register 2

Register 3: Domain access control register

Reading from CP15 register 3 returns the value of the Domain Access Control Register.

Writing to CP15 register 3 writes the value of Domain Access Control Register.

The Domain Access Control Register consists of sixteen 2-bit fields, each of which defines the access permissions for one of the sixteen Domains (D15-D0).

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 3.

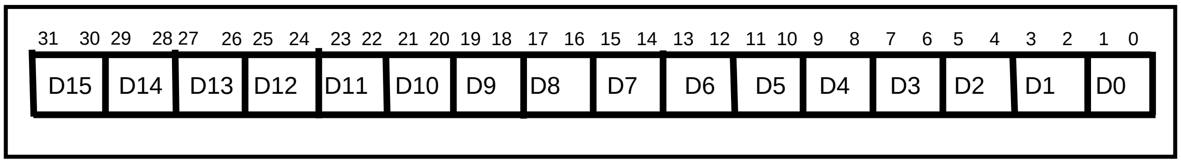


Figure 5-7: Register 3

Register 4: Reserved

Register 4 is reserved. Reading CP15 register 4 is UNDEFINED. Writing CP15 register 4 is UNDEFINED.

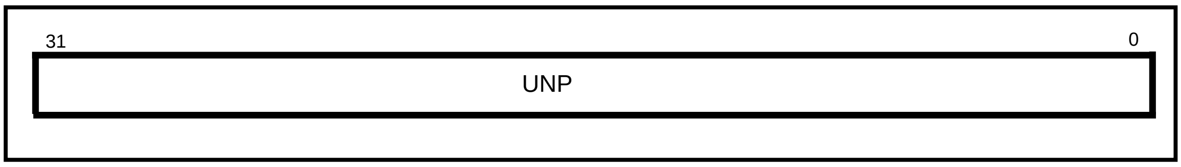


Figure 5-8: Register 4

Register 5: Fault Status Register

Reading CP15 register 5 returns the value of the Fault Status Register (FSR). The FSR contains the source of the last data fault. Note that only the bottom 9 bits are returned. The upper 23 bits are UNPREDICTABLE. The FSR indicates the domain and type of access being attempted when an abort occurred. Bit 8 is always read as zero. Bits 7:4 specify which of the sixteen domains (D15-D0) was being accessed when a fault occurred. Bits 3:1 indicate the type of access being attempted. The encoding of these bits is shown in ***8.13 Fault Address and Fault Status Registers (FAR and FSR)*** on page 8-17. The FSR is only updated for data faults, not for prefetch faults.

Writing CP15 register 5 sets the Fault Status Register to the value of the data written. This is useful for a debugger to restore the value of the FSR. The upper 24 bits written SHOULD BE ZERO. Bit 8 is ignored on writes.

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 5.

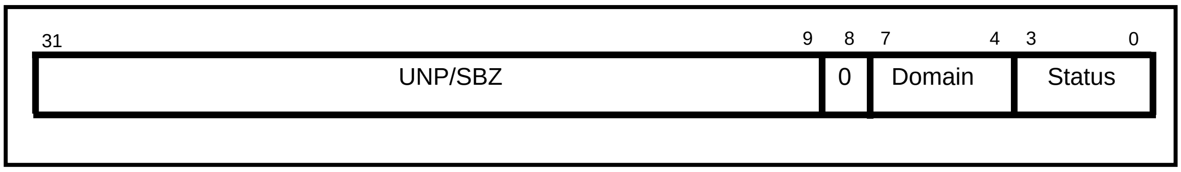


Figure 5-9: Register 5

Register 6: Fault Address Register

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

Writing CP15 register 6 sets the Fault Address Register to the value of the data written. This is useful for a debugger to restore the value of the FAR.

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 6.

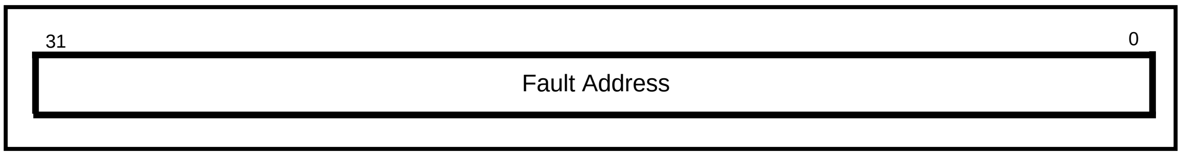


Figure 5-10: Register 6

Register 7: Cache Operations

Writing to CP15 register 7 is used to manage the ARM810 unified instruction and data cache. Four cache operations are defined, and the function to be performed selected by the opcode_2 and CRm fields in the MCR instruction used to write CP15 register 7.

Function	opcode_2 value	CRm value	Data	Instruction
Invalidate ID cache	0b000	0b0111	SBZ	MCR p15, 0, Rd, c7, c7, 0
Invalidate ID single entry	0b001	0b0111	Index, Seg Format	MCR p15, 0, Rd, c7, c7, 1
Clean ID single entry	0b001	0b1011	Index, Seg Format	MCR p15, 0, Rd, c7, c11, 1
Clean and Invalidate ID entry	0b001	0b1111	Index, Seg Format	MCR p15, 0, Rd, c7, c15, 1

Table 5-2: Cache operations

Reading from CP15 register 7 is UNDEFINED.

The “Invalidate ID cache” function invalidates all cache data, including any dirty data (data which has been modified in the cache but not yet written to main memory). Use with caution.

The “Invalidate ID single entry” function invalidates a single cache line, discarding any dirty data (data which has been modified in the cache but not yet written to main memory). Use with caution.

The “Clean ID single entry” function writes the specified cache line to main memory if the line is marked Valid and Dirty, and marks the line as not-Dirty . The Valid bit is unchanged.

The “Clean and Invalidate ID entry” function writes the specified cache line to main memory if the line is marked Valid and Dirty. It always invalidates the line.

The operations which operate upon a single cache line accept the entry's Index and Segment number as the data passed in the MCR instruction in the following format:

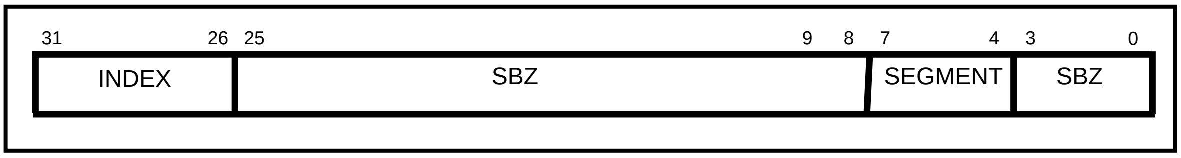


Figure 5-11: Register 7

See ***Chapter 7, Instruction and Data Cache (IDC)*** for discussion of the use of these operations.

Register 8: TLB Operations

Writing to CP15 register 8 is used to control Translation Lookaside Buffers (TLBs). The ARM810 implements a unified instruction and data TLB.

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

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

Table 5-3: TLB operations

Reading from CP15 register 8 is UNDEFINED.

The “Invalidate TLB” function invalidates all of the unlocked entries in the TLB

The “Invalidate TLB single entry” function invalidates any TLB entry corresponding to the Virtual Address given in Rd, regardless of it’s lock-down state.

Register 9: Cache Lock-Down

Writing CP15 register 9 updates the Cache Lock-Down control register. Bits 30:6 SHOULD BE ZERO when written.

Reading CP15 register 9 returns the value of the Cache Lock-Down control register. Note that only bit 31 and bits 5:0 are returned. Bits 30:6 are UNPREDICTABLE when read.

The Cache Lock-Down control register allows software to load entries into the Cache and lock them in. See ***7.7 Lock-down Features*** on page 7-3.

The CRm and opcode_2 fields SHOULD BE ZERO when reading or writing CP15 register 9.

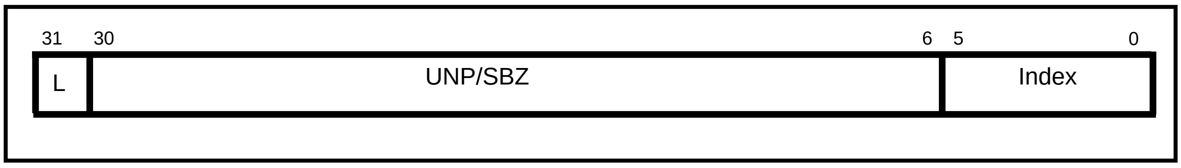


Figure 5-12: Register 9

L Bit 31	Cache Load Entry Mode 0 = Normal operation - Index Field specifies number of lock-down Indexes 1 = Load Entry Mode - Index Field specifies Index number to load into.

Register 10: TLB Lock-Down

Writing CP15 register 10 updates the TLB Lock-Down control register. Bits 30:6 SHOULD BE ZERO when written.

Reading CP15 register 10 returns the value of the TLB Lock-Down control register. Note that only bit 31 and bits 5:0 are returned. Bits 30:6 are UNPREDICTABLE when read.

The TLB Lock-Down control register allows software to load entries into the TLB and lock them in. See ***Appendix F, Cache and TLB Lock-Down Features***.

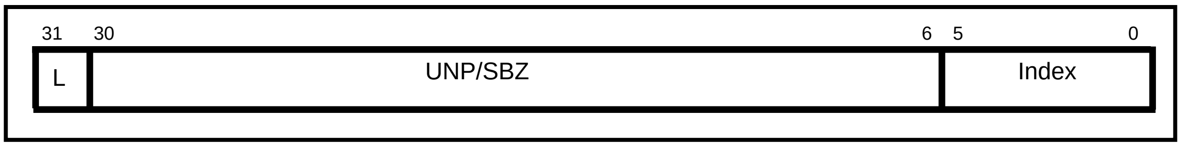


Figure 5-13: Register 10

L Bit 31TLB Load Entry Mode
0 = Normal operation - Index Field specifies number of lock-down
 Indexes.The number of lock-down Indexes must be 0 or 4.
1 = Load Entry Mode - Index Field specifies Index number to load into.

Registers 11 -14: Reserved

Accessing (reading or writing) any of these registers will cause ARM810 to take the undefined instruction trap.

Register 15: Clock and Test Configuration

Register 15 contains clocking configuration bits, test configuration bits, and the PLL Locked status bit. Writing to CP15 register 15 writes to the configuration bits. Writing a 1 to the PLL Locked status bit resets the PLL status bit for subsequent reads - see below for details.

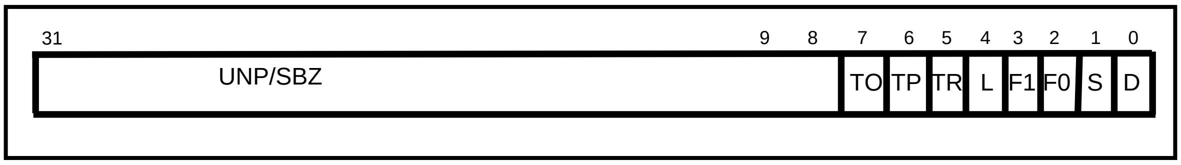


Figure 5-14: Register 15

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

D Bit 0 Enable Dynamic Clock Switching

D Bit 0	Enable Dynamic Clock Switching	
	0	Dynamic clock switching is disabled, clock synchroniser will permanently select the bus clock as the source of the processor clock.
	1	Dynamic clock switching is enabled, clock synchroniser will dynamically switch between the fast clock and the bus clock as the source of the processor clock as processor access to the Bus Interface is required.

S Bit 1	Synchronous Clock Switching	
	0	Clock synchroniser operates in asychronous mode. Use this setting if the fast clock and the bus clock *do not* obey the requirements specified in the AC parameters section for synchronous mode operation.
	1	Clock synchroniser operates in synchronous mode. Use this setting if the fast clock and the bus clock *do* obey the requirements specified in the AC parameters section for synchronous mode operation.

F1, F0 Bits 3, 2 Fast clock source configuration

F1 = 0 F0 = 0 bus clock (**MCLK** or **PCLK**) is the fast clock source.

F1 = 0 F0 = 1 **REFCLK** pin is the fast clock source.

F1 = 1 F0 = 0 Reserved. Do not use.

F1 = 1 F0 = 1 PLL output clock is fast clock source.

L Bit 4 PLL Locked indication

When Reading:

L = 1 indicates that the PLL output clock is within a small range of the target frequency.

When Writing:

Writing L = 0 is ignored.

Writing L = 1 resets the PLL Lock Detect circuitry. Following such a reset, reading the register will return L = 0 until the Lock Detect circuit again detects that the PLL output clock is within a small range of the target frequency. This is useful in systems which stop **REFCLK**, or change the frequency applied to the **REFCLK** pin, or change the PLL configuration pins under program control

Note Logic external to ARM810 would be required to implement such features.

TR, TP, and TO (Bits 5, 6, 7 and 8) are configuration bits for test features used in device production test. These bits must all be written as zero for normal device operation.

5.1.2 Architectural Compliance of ARM810 CP15

The ARM810 Coprocessor 15 complies with the definition of the ARMv4 System Control Coprocessor given in the ARM Architecture Reference (ARM DDI 0100) with the following exceptions and clarifications:

- Registers 9, 10, and 15 are not defined in the ARM Architecture Manual and should be considered implementation specific extensions to the CP15 definition.
- The ARM Architecture Reference defines read accesses to registers not defined for reading, and write accesses to registers not defined for writing, as UNPREDICTABLE. ARM810 implements these as UNDEFINED - ie, executing coprocessor instructions which attempt such accesses will cause ARM810 to take the Undefined Instruction Trap.
- The ARM Architecture Reference defines that instructions which access register 7 and 8 and which specify values of opcode_2 or CRm which do not specify an implemented operation should be IGNORED. ARM810 implements these as UNDEFINED.
- The ARM Architecture Reference defines that instructions which access register other than 7 and 8 and which specify values of opcode_2 or CRm other than zero are UNPREDICTABLE. ARM810 implements these as UNDEFINED.

6 The Prefetch Unit

This chapter describes the functions of the prefetch unit.

The Prefetch Unit

6.1 Overview

The ARM8 Prefetch Unit (PU) supplies the ARM8 Core with instructions from the memory system. The bus from the memory system to the PU is 32 bits wide but can supply two words every clock cycle. The memory system bandwidth is therefore greater than the bandwidth requirement of the Core. The Prefetch Unit makes use of this fact by buffering instructions in its FIFO and then predicting some of the branches and removing them from the instruction stream to the Core. This reduces the CPI of the Branch instruction, so increasing the processor's performance.

The Prefetch Unit is responsible for fetching and supplying instructions to the Core, and has its own PC and incrementer to provide the memory system address.

6.2 The Prefetch Buffer

Each 32-bit instruction is buffered together with its (offset) address in a FIFO. The depth of this buffer is 8 instructions. At the far end of the FIFO, the instructions are removed one at a time and presented to the Core.

6.3 Branch Prediction

ARM810 employs static branch prediction. This is based solely on the characteristics of a Branch instruction, and uses no history information. Branch prediction is performed only when the Z bit in CP15 register 1 is set to 1 (see ***Chapter 5, Configuration***).

In ARM processors that have no Prefetch Unit, the target of a Branch is not known until the end of the Execute stage; at which time it is known whether or not the Branch will be taken. The best performance is therefore obtained by predicting all Branches as *not* taken, and filling the pipeline with the instructions that follow the Branch. In this type of Core, an untaken Branch requires 1 cycle and a taken Branch requires 3 cycles.

By adding a Prefetch Buffer, it is possible to detect a Branch *before* it enters the Core. This allows the use of a different prediction scheme - for instance, one which predicts that all *forward* Branches are not taken and all *backward* Branches are taken. This scheme is the one implemented in ARM810 and because it models actual conditional branch behaviour more accurately, it reduces the average branch CPI, thus improving the processor's performance.

Using ARM8's Prefetch Unit, around 65% of all Branches are preceded by enough non-Branch cycles to be completely predicted. The Core itself deals with the Branches that the Prefetch Unit does not have time to predict.

6.3.1 Incorrect predictions and correction

Whenever a potentially incorrect prediction is made, information necessary for recovering from the error is stored. This is the fall-through address in the case of a predicted taken Branch, and the Branch's target address in the case of a predicted not taken Branch.

The Prefetch Unit uses the Core's condition codes to establish the accuracy of a prediction. If the prediction is found to be in error, the Prefetch Unit begins fetching from the saved alternate address, and cancels any instructions that have been incorrectly passed to the core.

6.3.2 Prediction details

This section describes the conditions under which prediction is made, and the result of the prediction based upon the direction of the branch.

BL is only predicted if it is an unconditional instruction. When predicted, the instruction is effectively changed into a link instruction and a branch instruction. The link part of the instruction is passed to the core as a special MOV instruction, and the branch part is predicted with the same rules as for the prediction of normal B instructions.

The following summarises the prediction scheme:

If any instruction is not predicted, then it is passed straight through to the core without change.

Instructions will not be predicted if any of the following conditions apply:

- Z bit in CP15 register 1 is 0
- Instruction[27:24]="1011" AND Instruction[31:28]!="1110" (Conditional BL)
- A prefetch abort occurs when fetching the instruction

• Instruction[31:28]==“1111”	(Invalid condition code)
• Instruction[27:25]!=“101”	(Non-branch instruction)

otherwise the instruction will be predicted as taken if:

• Instruction[31:28]==“1110”	(Always condition code)
• Instruction[24]==“0” AND Instruction[23]==“1”	(Backwards branch)

otherwise the instruction will be predicted as not-taken if:

• Instruction[24]==“0” AND Instruction[23]==“0”	(Forwards branch)

Consequences of branch prediction and the prefetch buffer

Due to the speculative prefetching of instructions that the Prefetch Unit performs, it is possible for the prefetch buffer to contain incorrect instructions. In such circumstances the prefetch buffer must be flushed, and ARM8 provides a means to do this with the IMB instruction. Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 for details of when and how to use the IMB instruction.

6.3.3 Turning off Branch Prediction

Branch prediction is disabled when the Z bit in the control register is 0 (CP15 register 1, bit 11). Clearing the Z bit does not stop speculative prefetching for a branch that has already been predicted. Branch prediction must be disabled and speculative prefetching must have completed before you disable the cache. The following code sequence disables branch prediction, and makes sure that speculative prefetching has completed:

```
Branch_Predict_Off
            MRC    p15,0,R0,c0,c0           ;Clear Control Reg Z bit.
            BIC    R0,R0,#&00000800
            MCR    p15,0,R0,c0,c0
            MSR    CPSR_f, #0xF0000000      ;set carry flag
            BCC    Branch_Predict_Off       ;branch never taken
```

Code to disable the cache should follow this code.

7 Instruction and Data Cache (IDC)

This chapter describes Instruction and Data Cache.

Instruction and Data Cache (IDC)

7.1 Introduction

ARM810 contains an 8 Kb mixed instruction and data cache which supports both write-through and write-back (also known as copy-back) operation. The IDC has 512 lines of 16 bytes (4 words), arranged as a 64-way associative, virtually addressed cache. The IDC is always reloaded a line at a time (four words). It may be enabled or disabled via the ARM810 Control Register and is disabled on **nRESET**. The operation of the cache is further controlled by the *Cacheable* (C) and *Bufferable* (B) bits stored in the Memory Management Page Table (see ***Chapter 8, Memory Management Unit***). For this reason, in order to use the IDC, the MMU must be enabled. The two functions may however be enabled simultaneously, with a single write to the Control Register.

7.2 Cacheable Bit and Bufferable Bit

The **Cacheable** bit determines whether data being read may be placed in the IDC and used for subsequent read operations. Typically main memory will be marked as Cacheable to improve system performance, and I/O space as Non-cacheable to stop the data being stored in ARM810's cache. For example if the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of initial data held in the cache. The *Cacheable* bit can be configured for both pages and sections.

When the cacheable bit associated with a memory region is 1, all write acesses to that region are bufferable and the B bit determines whether the region is cached with write-through (B=0) or write-back (B=1) cache operation.

See ***8.11 Cacheable and Bufferable Status of Memory Regions*** on page 8-14.

7.3 IDC Operation

In the ARM810 the cache will be searched regardless of the state of the C bit, only reads that miss the cache will be affected. The only effect of setting the cacheable bit to 0 is to inhibit cache replacement from occuring. If the cache is disabled by clearing bit 2 of the CP15 Control Register, no searching of the cache occurs and all regions are treated as non-cacheable.

7.3.1 Cacheable reads C = 1

A linefetch of 4 words will be performed when a cache miss occurs in a cacheable area of memory and it will be randomly placed in a cache bank.

7.3.2 Uncacheable reads C = 0

An external memory access will be performed and the cache will not be written.

7.4 IDC Validity

The IDC operates with virtual addresses, so care must be taken to ensure that its contents remain consistent with the virtual to physical mappings performed by the Memory Management Unit. If the Memory Mappings are changed, the IDC validity must be ensured.

7.4.1 Doubly mapped space

Since the cache works with virtual addresses, it is assumed that every virtual address maps to a different physical address. If the same physical location is accessed by more than one virtual address, the cache cannot maintain consistency, since each virtual address will have a separate entry in the cache, and only one entry will be updated on a processor write operation. To avoid any cache inconsistencies, both doubly-mapped virtual addresses should be marked as uncacheable.

7.5 Read-Lock-Write

The IDC treats the Read-Locked-Write instruction as a special case. The read phase always forces a read of external memory, regardless of whether the data is contained in the cache. The write phase is treated as a normal write operation (and if the data is already in the cache, the cache will be updated). Externally the two phases are flagged as indivisible by asserting the **LOCK** signal.

7.6 IDC Enable/Disable and Reset

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

7.6.1 To enable the IDC

To enable the IDC, make sure that the MMU is enabled first by setting bit 0 in Control Register, then enable the IDC by setting bit 2 in Control Register. The MMU and IDC may be enabled simultaneously with a single control register write.

7.6.2 To disable the IDC

To disable the IDC clear bit 2 in the Control Register and perform a flush by writing to the flush register.

7.7 Lock-down Features

See ***Appendix F, Cache and TLB Lock-Down Features***.