5 ARM Processor Instruction Set

This chapter describes the ARM processor instruction set.

ARM Processor Instruction Set

5.1 Instruction Set Summary

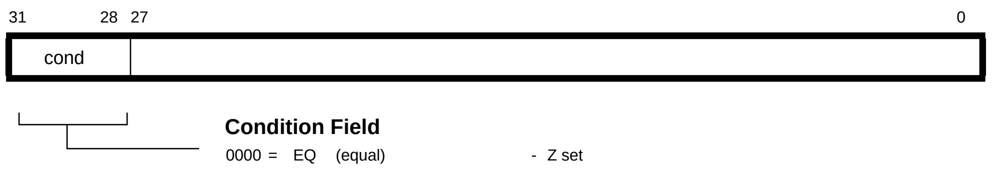
A summary of the ARM processor instruction set is shown in *Figure 5-1: Instruction set summary*.

	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
Data Processing PSR Transfer	cond	0	0	I	opcode				S	Rn	Rd	operand 2					
Multiply	cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm
Single data swap	cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm
Single data transfer	cond	0	1	I	P	U	B	W	L	Rn	Rd	offset					
Undefined instruction	cond	0	1	1	x	x	x	x	x	x x x x	x x x x	x x x x	x	x	x	1	x x x x
Block data transfer	cond	1	0	0	P	U	S	W	L	Rn	Register List						
Branch	cond	1	0	1	L	offset											
Coproc data transfer	cond	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	offset				
Coproc data operation	cond	1	1	1	0	CP opc				CRn	CRd	cp_num	CP			0	CRm
Coproc register transfer	cond	1	1	1	0	CP opc			L	CRn	Rd	cp_num	CP			1	CRm
Software interrupt	cond	1	1	1	1	ignored by processor											

Figure 5-1: Instruction set summary

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

5.2 The Condition Field



Condition Field

0000 =	EQ	(equal)	-	Z set
0001 =	NE	(not equal)	-	Z clear
0010 =	CS	(unsigned higher or same)	-	C set
0011 =	CC	(unsigned lower)	-	C clear
0100 =	MI	(negative)	-	N set
0101 =	PL	(positive or zero)	-	N clear
0110 =	VS	(overflow)	-	V set
0111 =	VC	(no overflow)	-	V clear
1000 =	HI	(unsigned higher)	-	C set and Z clear
1001 =	LS	(unsigned lower or same)	-	C clear or Z set
1010 =	GE	(greater or equal)	-	N set and V set, or N clear and V clear
1011 =	LT	(less than)	-	N set and V clear, or N clear and V set
1100 =	GT	(greater than)	-	Z clear, and either N set and Vset, or N clear and V clear
1101 =	LE	(less than or equal)	-	Z set, or N set and V clear, or N clear and V set
1101 =	AL		-	always
1111 =	NV		-	never

Figure 5-2: Condition codes

All ARM processor instructions are conditionally executed, which 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.

The condition codes have meanings as detailed in *Figure 5-2: Condition codes*, for instance code 0000 (EQual) executes the instruction only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction is not executed.

Note: *If the* always *(AL - 1110) condition is specified, the instruction will be executed irrespective of the flags. The never (NV - 1111) class of condition codes must not be used as they will be redefined in future variants of the ARM architecture. If a NOP is required it is suggested that MOV R0,R0 be used. The assembler treats the absence of a condition code as though* always *had been specified.*

5.3 Branch and Branch with Link (B, BL)

These instructions are only executed if the condition is true. The instruction encoding is shown in *Figure 5-3: Branch instructions*.

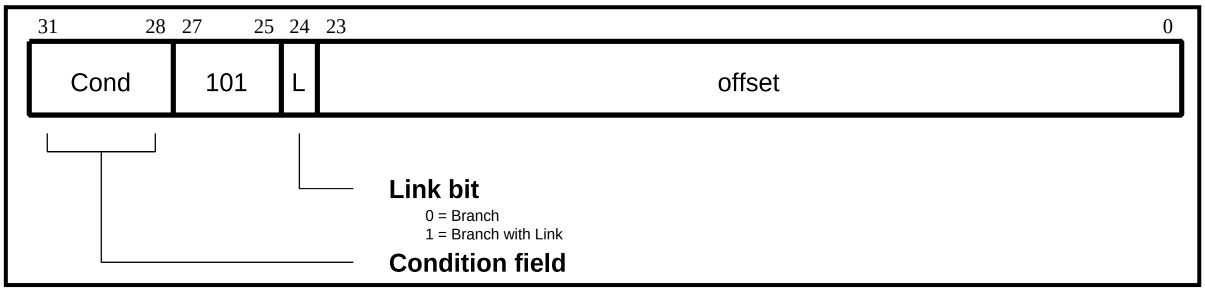


Figure 5-3: Branch instructions

Branch instructions contain a signed 2's complement 24-bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction. Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a branch with link type operation is required.

5.3.1 The link bit

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

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

5.3.2 Instruction cycle times

Branch and Branch with Link instructions take 3 instruction fetches. For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.3.3 Assembler syntax

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

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

`{L}`	requests the Branch with Link form of the instruction. If *absent, R14 will not be affected by the instruction.
`{cond}`	is a two-char mnemonic as shown in *Figure 5-2: Condition codes* on page 5-2 (EQ, NE, VS etc). If absent then AL (ALways) will be used.
`<expression>`	is the destination. The assembler calculates the offset.

5.3.4 Examples

```
here    BAL   here     ;assembles to 0xEAFFFFFE (note effect of PC
                       ;offset)
        B     there    ;ALways condition used as default
        CMP   R1,#0    ;compare R1 with zero and branch to fred if R1
        BEQ   fred     ;was zero otherwise continue to next instruction
        BL    sub+ROM  ;call subroutine at computed address
        ADDS  R1,#1    ;add 1 to register 1, setting CPSR flags on the
        BLCC  sub      ;result then call subroutine if the C flag is
                       ;clear, which will be the case unless R1 held
                       ;0xFFFFFFFF
```

5.4 Data Processing

The instruction is only executed if the condition is true, defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-4: Data processing instructions* on page 5-5.

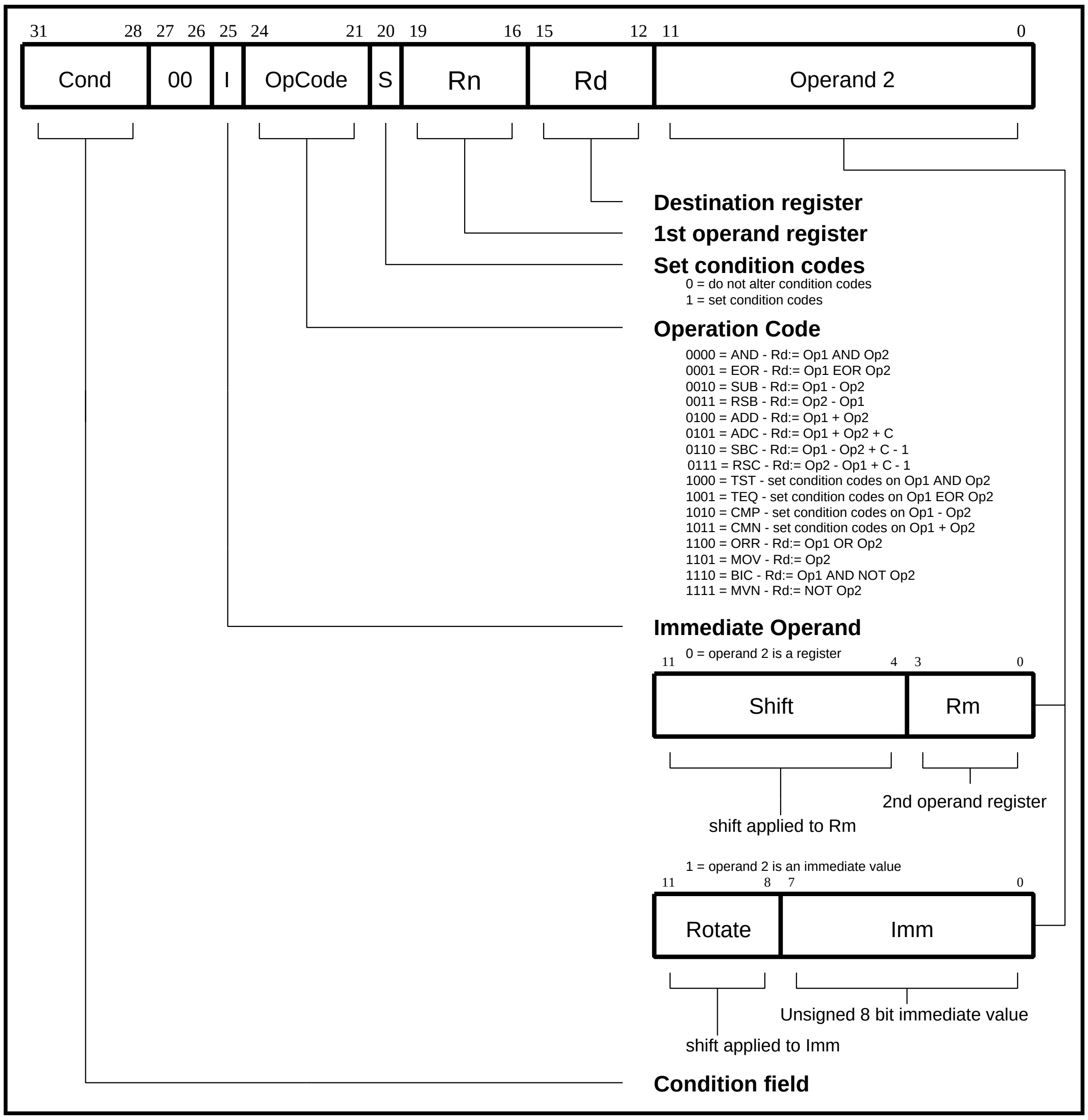
The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands.

First operand	is always a register (Rn).
Second operand	may be a shifted register (Rm) or a rotated 8-bit immediate value (Imm) according to the value of the I bit in the instruction.

The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S-bit in the instruction.

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

The instructions and their effects are listed in *Table 5-1: ARM data processing instructions* on page 5-6**.**



```
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
```

Figure 5-4: Data processing instructions

5.4.1 CPSR flags

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

If the S bit is set (and Rd is not R15):

- the V flag in the CPSR will be unaffected
- the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0)
- the Z flag will be set if and only if the result is all zeros
- the N flag will be set to the logical value of bit 31 of the result.

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

Table 5-1: ARM data processing instructions

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or 2's complement signed, the two are equivalent).

If the S bit is set (and Rd is not R15):

- the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed
- the C flag will be set to the carry out of bit 31 of the ALU
- the Z flag will be set if and only if the result was zero
- the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

5.4.2 Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in *Figure 5-5: ARM shift operations*.

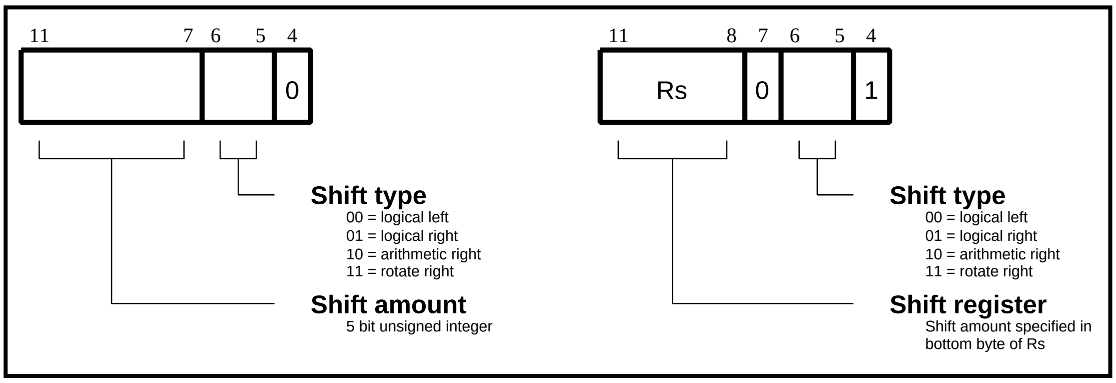


Figure 5-5: ARM shift operations

Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in *Figure 5-6: Logical shift left* on page 5-8.

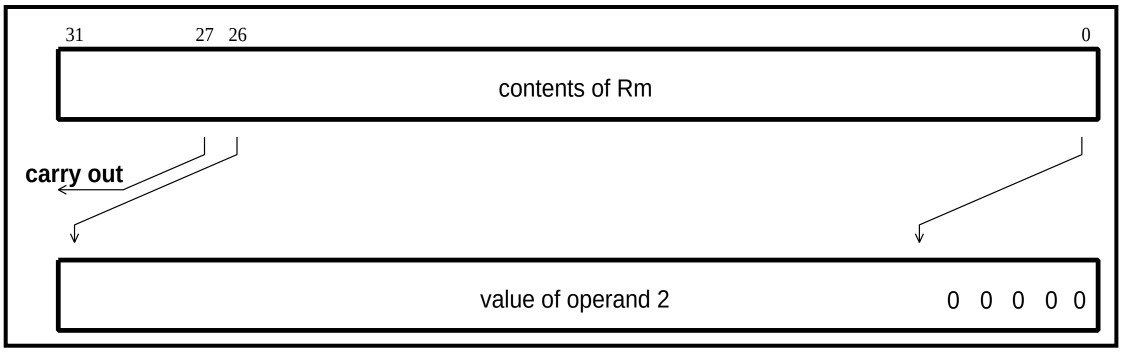


Figure 5-6: Logical shift left

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

Logical shift right

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

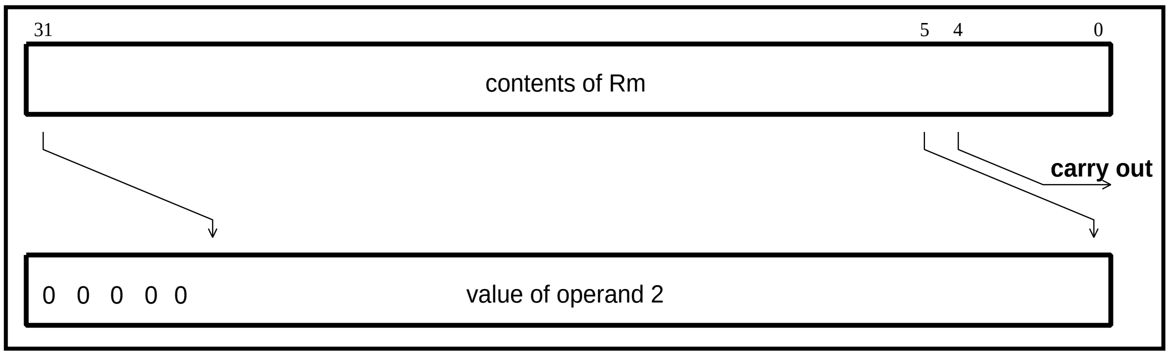


Figure 5-7: Logical shift right

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

Arithmetic shift right

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in *Figure 5-8: Arithmetic shift right* on page 5-9.

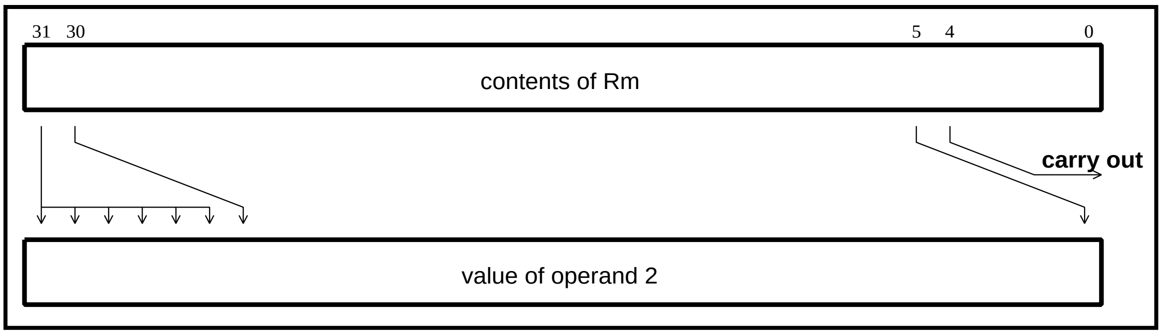


Figure 5-8: Arithmetic shift right

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

Rotate right

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in *Figure 5-9: Rotate right* on page 5-9.

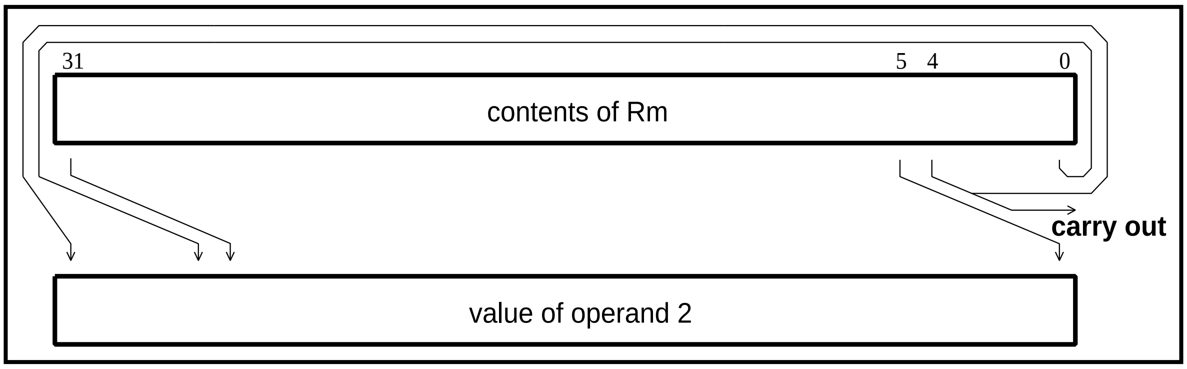


Figure 5-9: Rotate right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in *Figure 5-10: Rotate right extended* on page 5-10.

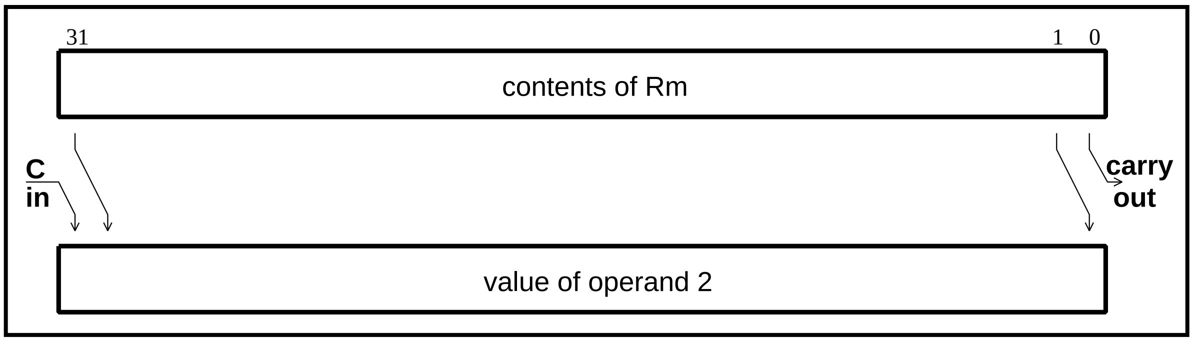


Figure 5-10: Rotate right extended

Register specified shift amount

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

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

Table 5-2: Register specified shift amount

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

5.4.3 Immediate operand rotates

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

5.4.4 Writing to R15

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

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

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR.

Note: This form of instruction must not be used in User mode.

5.4.5 Using R15 as an operand

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

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

5.4.6 TEQ, TST, CMP & CMN opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler shall always set the S flag for these instructions even if it is not specified in the mnemonic.

The TEQP form of the instruction used in earlier processors shall not be used in the 32-bit modes, the PSR transfer operations should be used instead. If used in these modes, its effect is to move SPSR_<mode> to CPSR if the processor is in a privileged mode and to do nothing if in User mode.

5.4.7 Instruction cycle times

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

Instruction	Cycles
Normal Data Processing	1instruction fetch
Data Processing with register specified shift	1 instruction fetch + 1 internal cycle
Data Processing with PC written	3 instruction fetches
Data Processing with register specified shift and PC written	3 instruction fetches and 1 internal cycle

Figure 5-11: Instruction cycle times

See *5.17 Instruction Speed Summary* on page 5-47 for more information.

5.4.8 Assembler syntax

1 MOV,MVN - single operand instructions

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

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

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

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

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

where:

`<Op2>`	is `Rm{,<shift>}` or,`<#expression>`
`{cond}`	two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
`{S}`	set condition codes if S present (implied for CMP, CMN, TEQ, TST).
`Rd, Rn` and `Rm`	are expressions evaluating to a register number.
`<#expression>`	if used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.
`<shift>`	is <shiftname> <register> or <shiftname> #expression, or RRX (rotate right one bit with extend).
`<shiftname>`	is: ASL, LSL, LSR, ASR, ROR. (ASL is a synonym for LSL; they assemble to the same code.)

5.4.9 Example

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

5.5 PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined in *5.2 The Condition Field* on page 5-2.

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in *Figure 5-12: PSR transfer* on page 5-14.

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

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

5.5.1 Operand restrictions

In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

The SPSR register which is accessed depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.

Note: *R15 must not be specified as the source or destination register.*

A further restriction is that you must not attempt to access an SPSR in User mode, since no such register exists.

5.5.2 Reserved bits

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

Compatibility

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

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

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

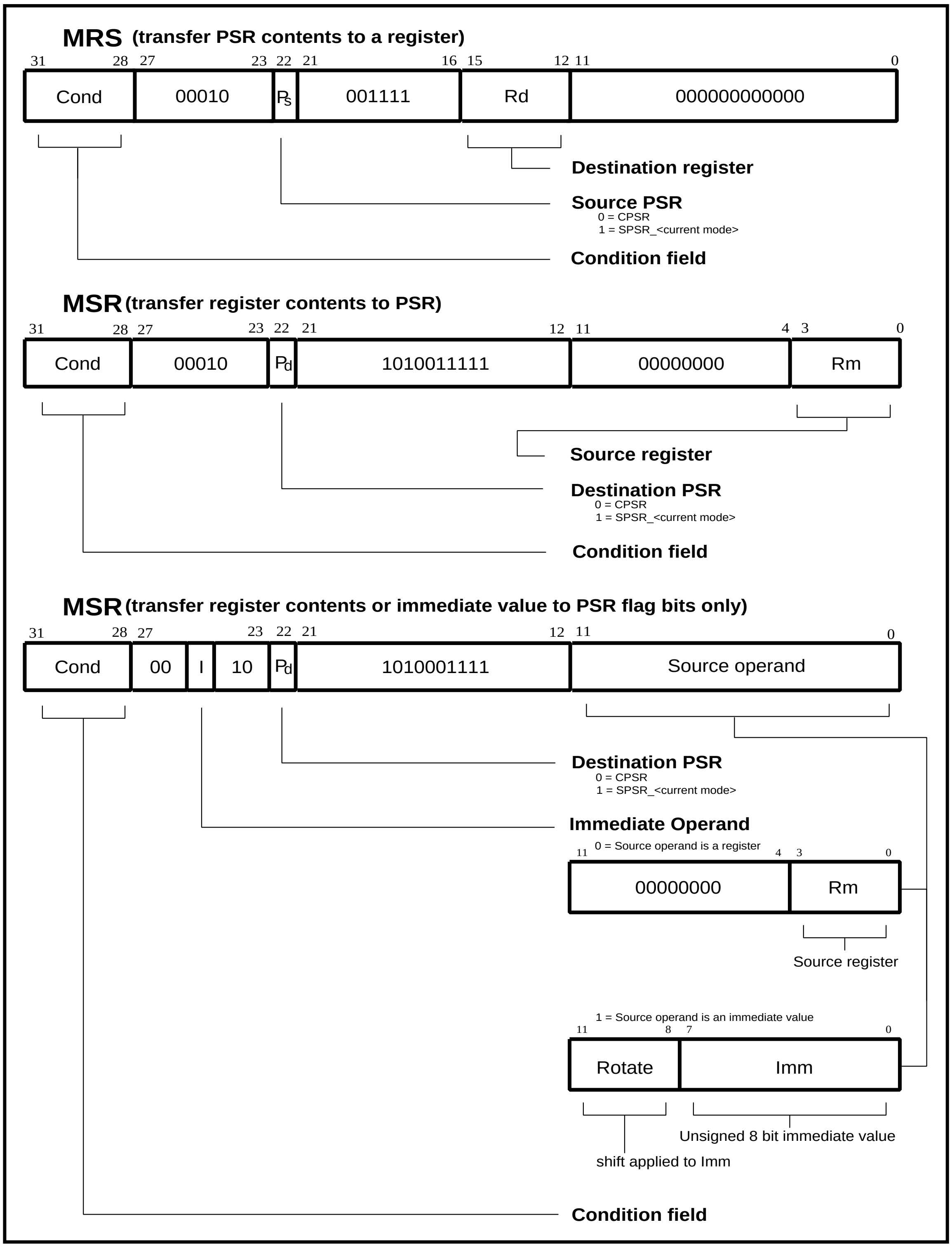


Figure 5-12: PSR transfer

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. e.g. The following instruction sets the N,Z,C & V flags:

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

Note: *Do not attempt to write an 8 bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.*

5.5.3 Instruction cycle times

PSR Transfers take 1 instruction fetch. For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.5.4 Assembler syntax

1. MRS - transfer PSR contents to a register

   MRS{cond} Rd,<psr>

2. MSR - transfer register contents to PSR

   MSR{cond} <psr>,Rm

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

   MSR{cond} <psrf>,Rm

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

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

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

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

where:

{cond}	two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
Rd and Rm	expressions evaluating to a register number other than R15
<psr>	is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)
<psrf>	is CPSR_flg or SPSR_flg
<#expression>	where used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

5.5.5 Examples

In User mode the instructions behave as follows:

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

In privileged modes the instructions behave as follows:

```
MSR   CPSR_all,Rm    ;CPSR[31:0]  <- Rm[31:0]
MSR   CPSR_flg,Rm    ;CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg,#0x50000000;
                     ;CPSR[31:28] <- 0x5
                     ;(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_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]
```

5.6 Multiply and Multiply-Accumulate (MUL, MLA)

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 5-13: Multiply instructions*.

The multiply and multiply-accumulate instructions use a 2-bit Booth's algorithm to perform integer multiplication. They give the least significant 32-bits of the product of two 32-bit operands, and may be used to synthesize higher-precision multiplications.

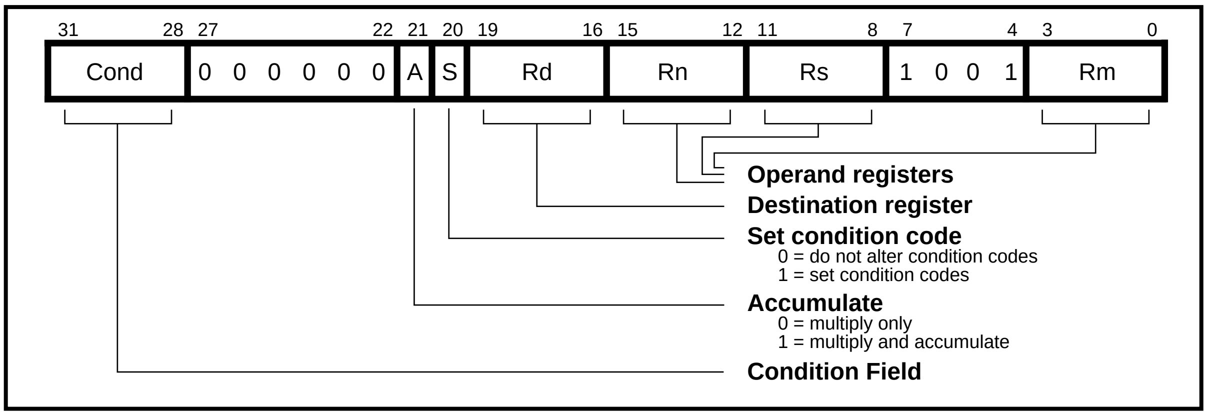


Figure 5-13: Multiply instructions

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

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

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

Example

For example consider the multiplication of the operands:

```
Operand A      Operand B      Result
0xFFFFFFF6     0x00000014     0xFFFFFF38
```

If the operands are interpreted as signed, operand A has the value -10, operand B has the value 20, and the result is -200 which is correctly represented as 0xFFFFFF38

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

5.6.1 Operand restrictions

Due to the way multiplication was implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the operand register (Rm), as Rd is used to hold intermediate values and Rm is used repeatedly during multiply. A MUL will give a zero result if Rm=Rd, and an MLA will give a meaningless result. R15 must not be used as an operand or as the destination register.

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

5.6.2 CPSR flags

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

5.6.3 Instruction cycle times

The Multiply instructions take 1 instruction fetch and m internal cycles, as shown in *Table 5-3: Instruction cycle times*. For more information see *5.17 Instruction Speed Summary* on page 5-47.

Multiplication by	**Takes**
any number between *2^(2m-3)* and *2^(2m-1)-1*	1S+mI cycles for *1<m>16*.
Multiplication by 0 or 1	1S+1I cycles

Table 5-3: Instruction cycle times

Multiplication by	Takes
any number greater than or equal to 2^(29)	1S+16I cycles.

Table 5-3: Instruction cycle times

m	is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs

The maximum time for any multiply is thus 1S+16I cycles.

5.6.4 Assembler syntax

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

where:

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

5.6.5 Examples

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

5.7 Single Data Transfer (LDR, STR)

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 5-14: Single data transfer instructions*.

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

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

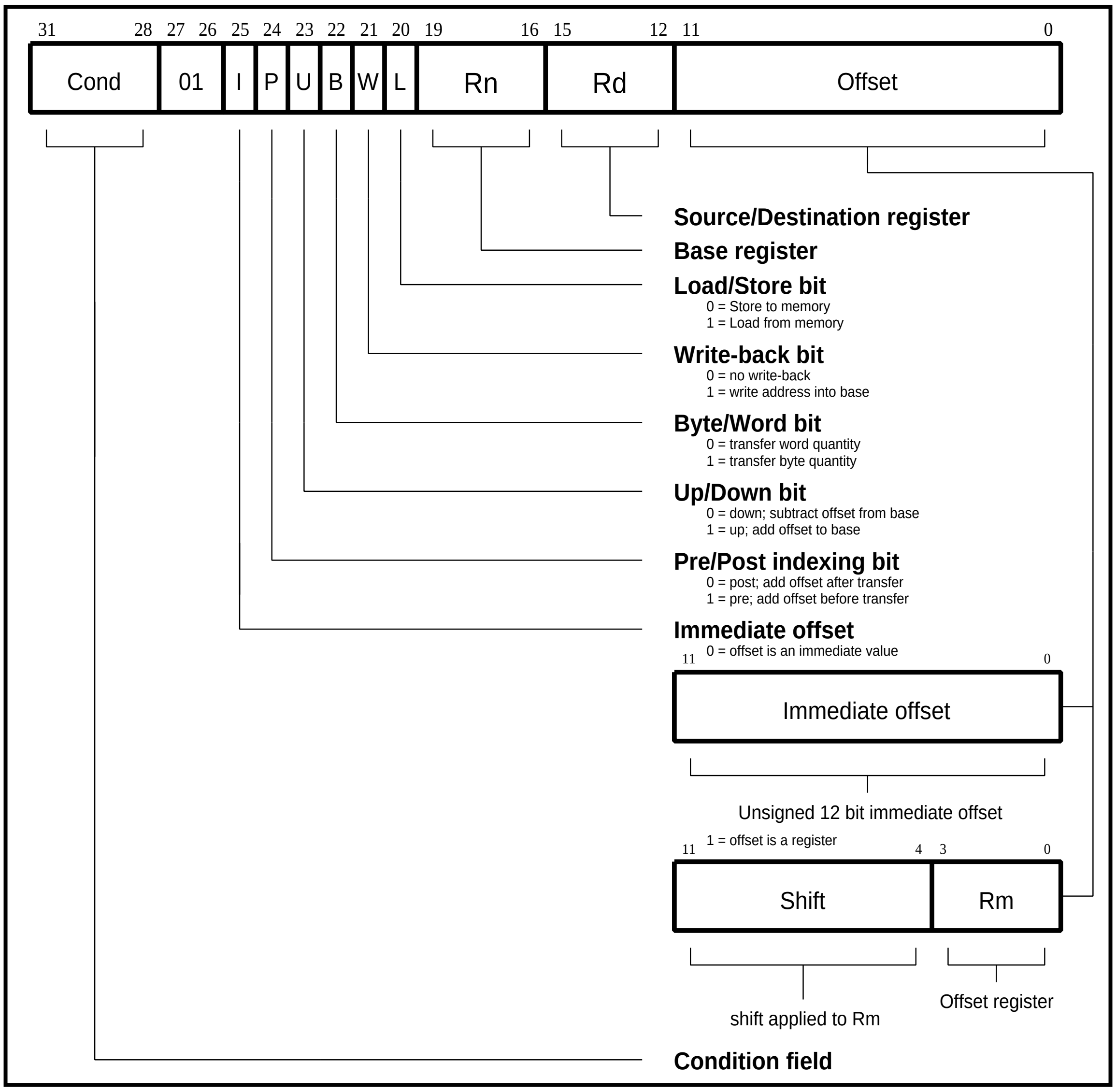


Figure 5-14: Single data transfer instructions

5.7.1 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).

Post-indexed addressing

In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces non-privileged mode for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this hardware.

5.7.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See *5.4.2 Shifts* on page 5-7.

5.7.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM processor register and memory. The following text assumes that the ARM7500FE is operating with 32-bit wide memory. If it is operating with 16-bit wide memory, the positions of bytes on the *external* data bus will be different, although, on the ARM7500FE internal data bus the positions will be as described here.

The action of LDR(B) and STR(B) instructions is influenced by the 3 instruction fetches. For more information see *5.17 Instruction Speed Summary* on page 5-47. The two possible configurations are described below.

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. See *Figure 4-1: Little-endian addresses of bytes within words* on page 4-2.
Byte store (STRB)	repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0.
Word load (LDR)	will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits. This is illustrated in *Figure 5-15: Little Endian offset addressing* on page 5-21.

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

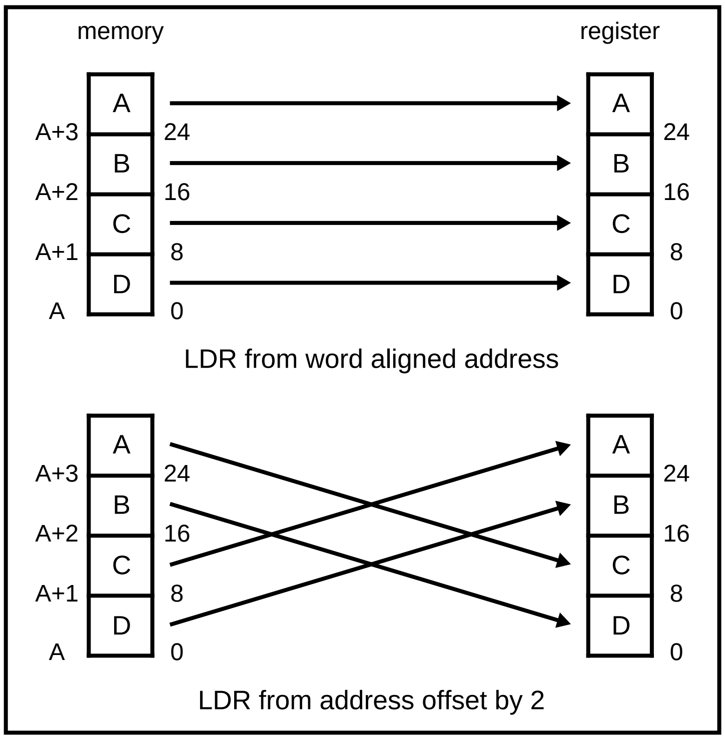


Figure 5-15: Little Endian offset addressing

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 4-2: Big-endian addresses of bytes within words* on page 4-3.
Byte store (STRB)	repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0.
Word load (LDR)	should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 15 through 8.

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

5.7.4 Use of R15

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

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

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

5.7.5 Restriction on the use of base register

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

For example:

```
LDR   R0,[R1],R1
      <LDR|STR> Rd, [Rn],{+/-}Rn{,<shift>}
```

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

5.7.6 Data aborts

A transfer to or from a legal address may cause the MMU to generate an abort. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

5.7.7 Instruction cycle times

Instruction	Cycles
Normal LDR instruction	1 instruction fetch, 1 data read and 1 internal cycle
LDR PC	3 instruction fetches, 1 data read and 1 internal cycle.
STR instruction	1 instruction fetch and 1 data write incremental cycles.

Table 5-4: Instruction cycle times

For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.7.8 Assembler syntax

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

LDR	load from memory into a register
STR	store from a register into memory
{cond}	two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
{B}	if B is present then byte transfer, otherwise word transfer
{T}	if T is present the W bit will be set in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.
Rd	is an expression evaluating to a valid register number.

`<Address>` can be:

1 An expression which generates an address:

`<expression>`

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

2 A pre-indexed addressing specification:

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

3 A post-indexed addressing specification:

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

`Rn and Rm` are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7500FE pipelining. In this case base write-back

	shall not be specified.
`<shift>`	is a general shift operation (see section on data processing instructions) but note that the shift amount may not be specified by a register.
`{!}`	writes back the base register (set the W bit) if ! is present.

5.7.9 Examples

```
STR   R1,[R2,R4]!    ;store R1 at R2+R4 (both of which are
                     ;registers) and write back address to R2
STR   R1,[R2],R4     ;store R1 at R2 and write back
                     ;R2+R4 to R2
LDR   R1,[R2,#16]    ;load R1 from contents of R2+16
                     ; 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 to 7, filling bits 8 to 31
                     ; with zeros
STR   R1,PLACE       ;generate PC relative offset to address
      •              ;PLACE
      •
PLACE
```

5.8 Block Data Transfer (LDM, STM)

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 5-16: 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.

5.8.1 The register list

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

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

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

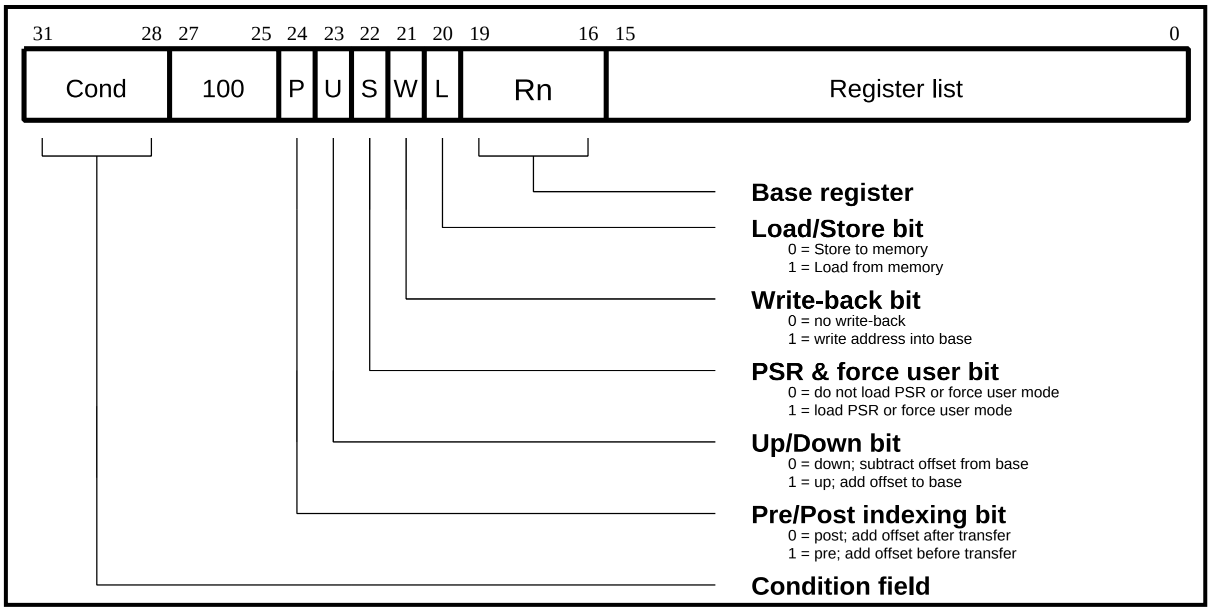

31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down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

Figure 5-16: Block data transfer instructions

5.8.2 Addressing modes

The transfer addresses are determined by:

- the contents of the base register (Rn)
- the pre/post bit (P)
- the up/down bit (U)

The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address.

By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of the modified base is required (W=1).

Figure 5-17: Post-increment addressing, *Figure 5-18: Pre-increment addressing*, *Figure 5-19: Post-decrement addressing*, and *Figure 5-20: Pre-decrement addressing* on page 5-28, show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

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

5.8.3 Address alignment

The address should always be a word aligned quantity.

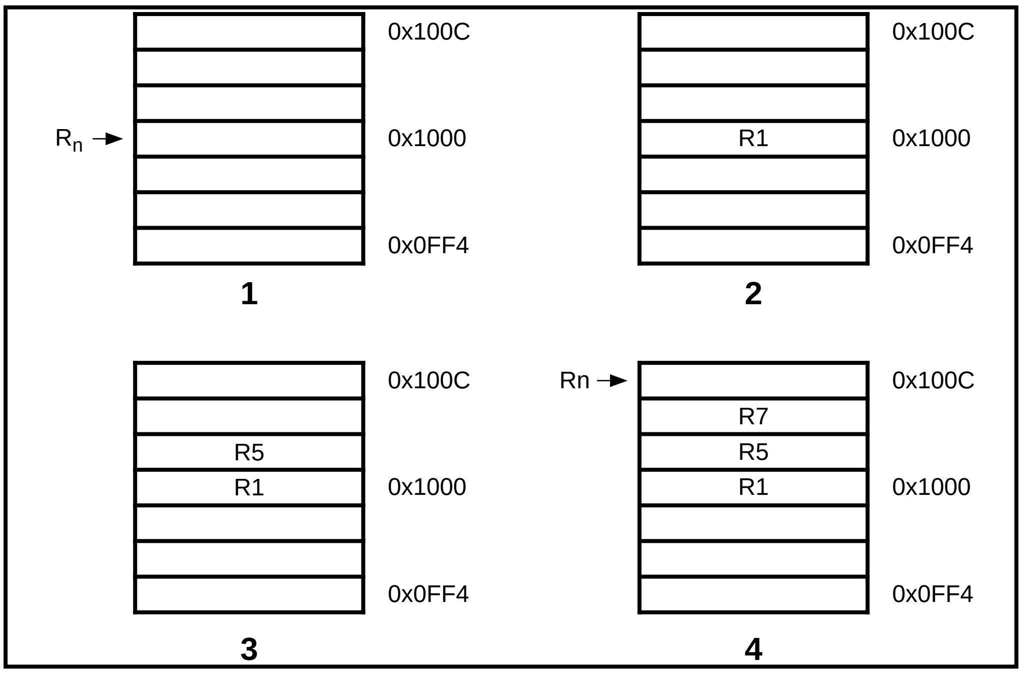


Figure 5-17: Post-increment addressing

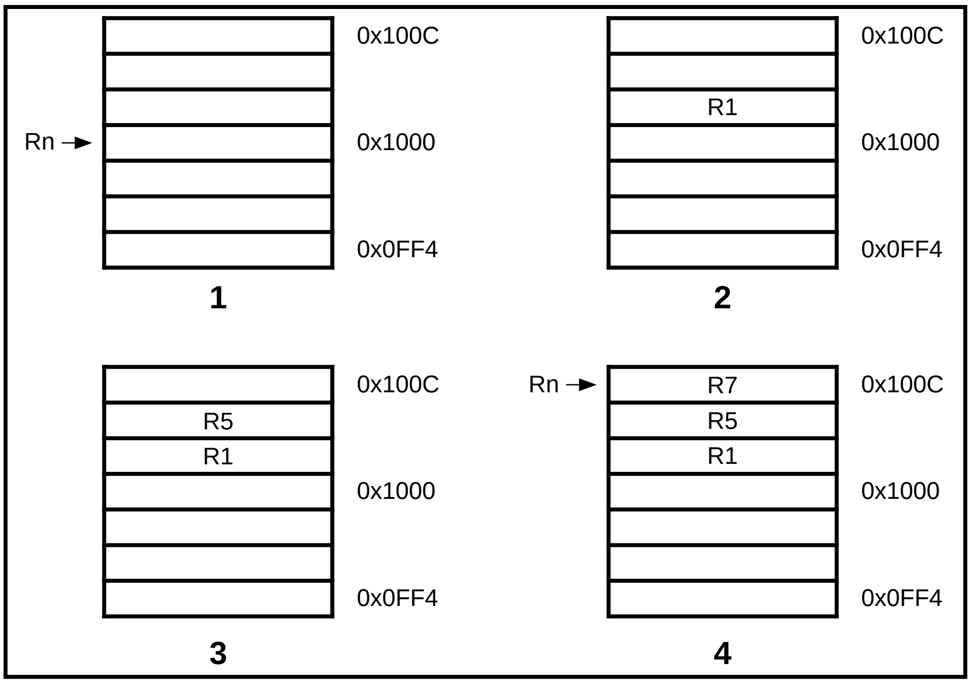


Figure 5-18: Pre-increment addressing

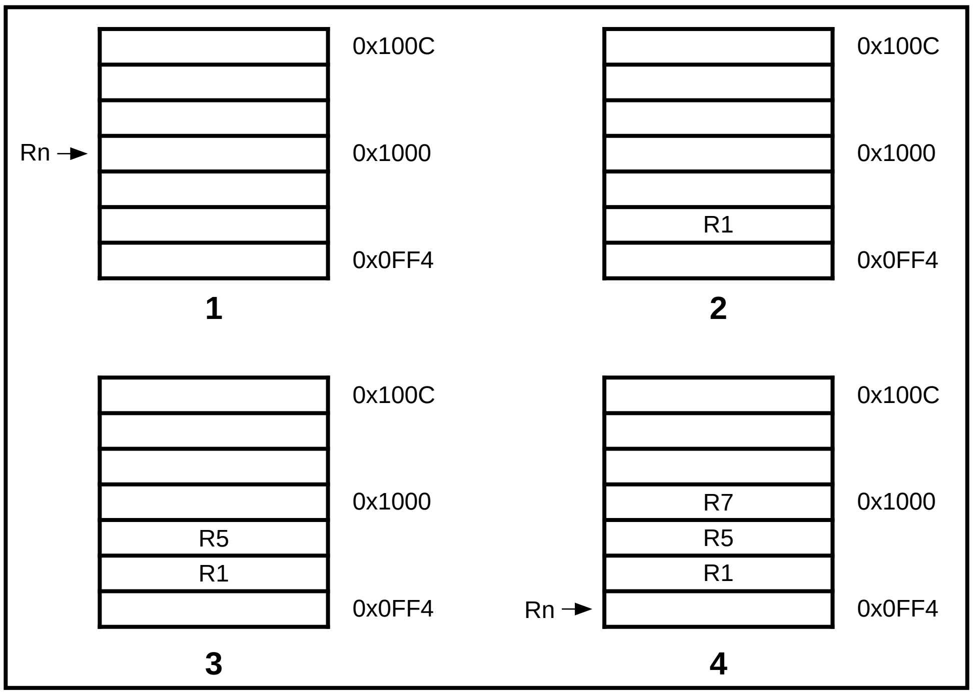


Figure 5-19: Post-decrement addressing

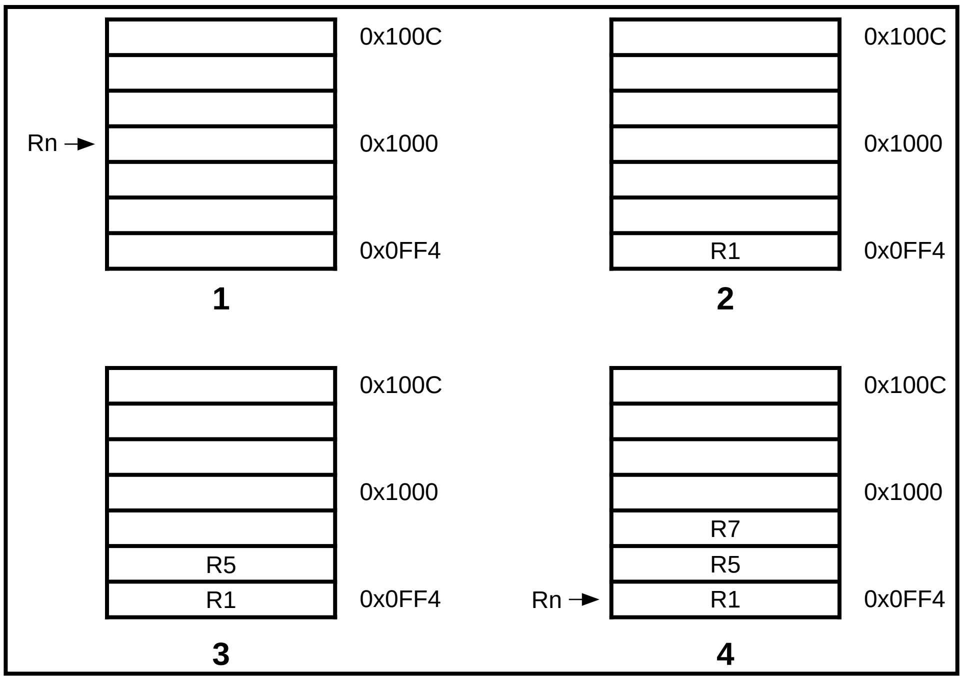


Figure 5-20: Pre-decrement addressing

5.8.4 Use of the S bit

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

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

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

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

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

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

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

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

5.8.5 Use of R15 as the base register

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

5.8.6 Inclusion of the base in the register list

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

5.8.7 Data aborts

Some legal addresses may be unacceptable to the MMU. The MMU will then cause an abort. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM7500FE is to be used in a virtual memory system.

Aborts during STM instructions

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

Aborts during LDM instructions

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

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

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

5.8.8 Instruction cycle times

Instruction	Cycles
Normal LDM instructions	1 instruction fetch, *n* data reads and 1 internal cycle
LDM PC	3 instruction fetches, *n* data reads and 1 internal cycle.
STM instructions	instruction fetch, *n* data reads and 1 internal cycle, where *n* is the number of words transferred.

Table 5-5: Instruction cycle times

For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.8.9 Assembler syntax

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

where:

{cond}	is a two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
Rn	is an expression evaluating to a valid register number
<Rlist>	is a list of registers and register ranges enclosed in {} (e.g. {R0,R2-R7,R10}).
{!}	(if present) requests write-back (W=1), otherwise W=0
{^}	(if present) set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

5.8.10 Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalencies between the names and the values of the bits in the instruction are shown in *Table 5-6: Addressing mode names*:

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	The stack is ascending (an STM will go up and LDM down)
D	The stack is descending (an STM will go down and LDM up)

The following symbols 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
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 5-6: Addressing mode names

5.8.11 Examples

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

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

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

5.9 Single Data Swap (SWP)

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 5-21: Swap instruction*.

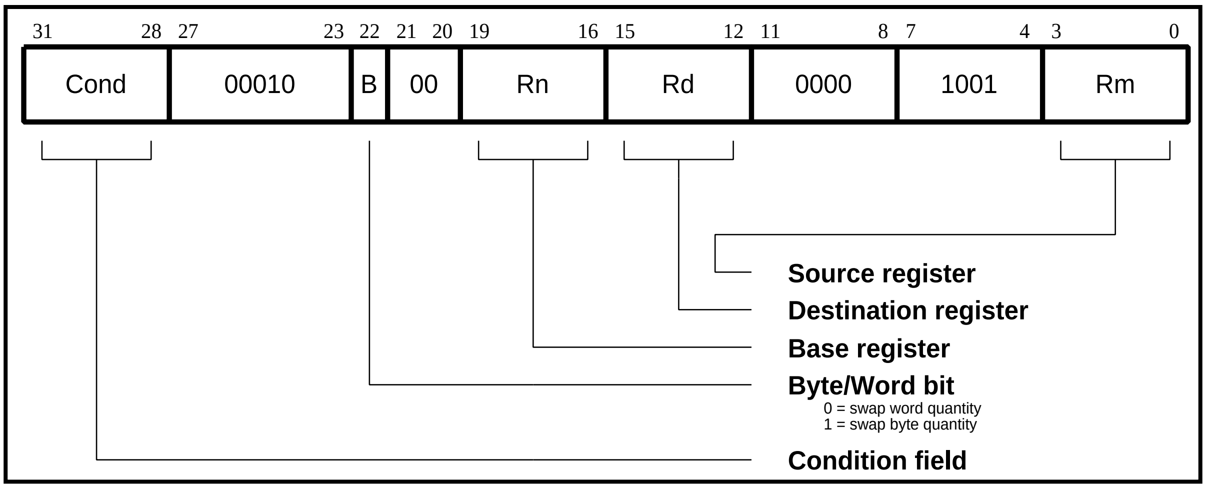


Figure 5-21: Swap instruction

Data swap instruction

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

Swap address

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

ARM710 lock feature

The ARM7500FE does not use the lock feature available in the ARM710 macrocell. You must take care to ensure that control of the memory is not removed from the ARM processor while it is performing this instruction.

5.9.1 Bytes and words

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

5.9.2 Use of R15

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

5.9.3 Data aborts

If the address used for the swap is unacceptable to the MMU, it will cause an abort. This can happen on either the read or write cycle (or both), and, in either case, the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem. The instruction can then be restarted and the original program continued.

5.9.4 Instruction cycle times

Swap instructions take 1 instruction fetch, 1 data read, 1 data write and 1 internal cycle. For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.9.5 Assembler syntax

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

{cond}	two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
{B}	if B is present then byte transfer, otherwise word transfer
Rd,Rm,Rn	are expressions evaluating to valid register numbers

5.9.6 Examples

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

5.10 Software Interrupt (SWI)

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 5-22: Software interrupt instruction*. The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

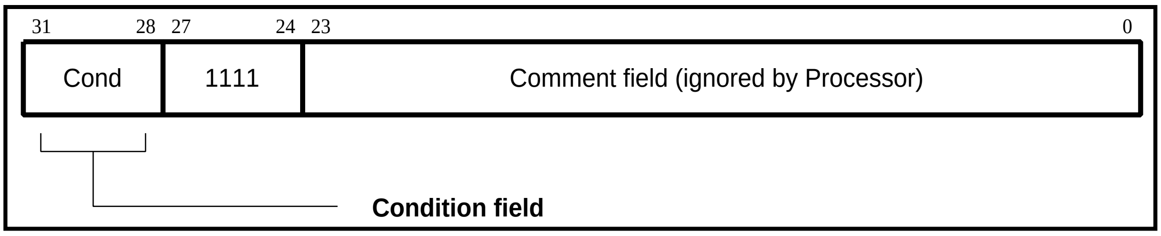


Figure 5-22: Software interrupt instruction

5.10.1 Return from the supervisor

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

Note: 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.

5.10.2 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.

5.10.3 Instruction cycle times

Software interrupt instructions take 3 instruction fetches. For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.10.4 Assembler syntax

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

{cond}	two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
<expression>	is evaluated and placed in the comment field (ignored by the ARM processor).

5.10.5 Examples

```
SWI   ReadC          ;get next character from read stream
SWI   WriteI+”k”     ;output a “k” to the write stream
```

```
SWINE 0             ;conditionally call supervisor
                    ;with 0 in comment field
```

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

```
0x08 B Supervisor    ;SWI entry point

EntryTable           ;addresses of supervisor routines
               DCD ZeroRtn
               DCD ReadCRtn
               DCD WriteIRtn
               ...
      Zero     EQU      0
      ReadC    EQU      256
      WriteI   EQU      512

Supervisor

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

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

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

5.11 Coprocessor Instructions on the ARM Processor

The core ARM processor in the ARM7500FE, unlike some other ARM processors, does not have an external coprocessor interface. It supports 2 on-chip coprocessors:

- the FPA
- on-chip control coprocessor, #15, which is used to program the on-chip control registers

For coprocessor instructions supported by the FPA, see *Chapter 10: Floating-Point Instruction Set* .

Coprocessor #15 supports only the Coprocessor Register instructions MRC and MCR.

Note: *Sections 5.12 through 5.14 describe non-FPA coprocessor instructions only.*

All other coprocessor instructions will cause the undefined instruction trap to be taken on the ARM processor. These coprocessor instructions can be emulated in software by the undefined trap handler. Even though external coprocessors cannot be connected to the ARM processor, the coprocessor instructions are still described here in full for completeness. It must be kept in mind that any external coprocessor referred to will be a software emulation.

5.12 Coprocessor Data Operations (CDP)

Use of the CDP instruction on the ARM processor (except for the defined FPA instructions) will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

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 5-23: 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 processor, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other activity allowing the coprocessor and the processor to perform independent tasks in parallel.

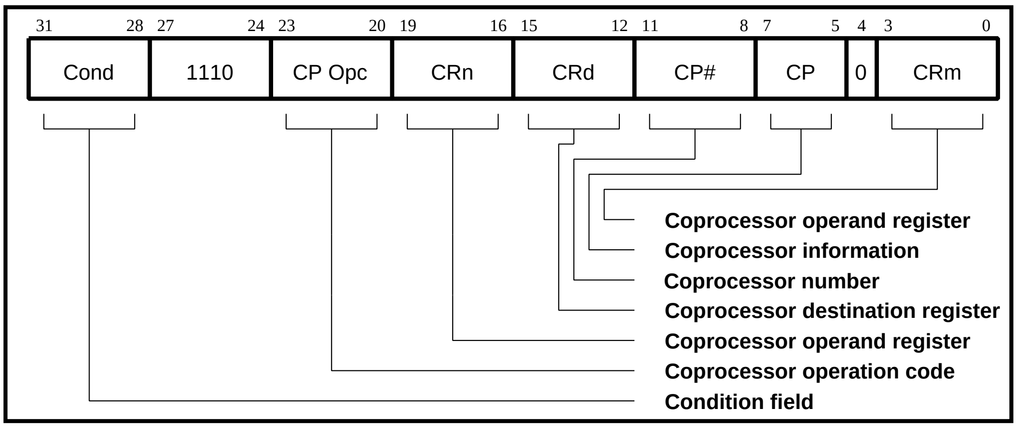


Figure 5-23: Coprocessor data operation instruction

5.12.1 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 will ignore any instruction which does not contain its number in the CP# field.

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

5.12.2 Instruction cycle times

All non-FPA CDP instructions are emulated in software: the number of cycles taken will depend on the coprocessor support software.

5.12.3 Assembler syntax

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

{cond}	two character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
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

5.12.4 Examples

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

5.13 Coprocessor Data Transfers (LDC, STC)

Use of the LDC or STC instruction on the ARM processor (except for the defined FPA instructions) will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

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 5-24: Coprocessor data transfer instructions*.

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

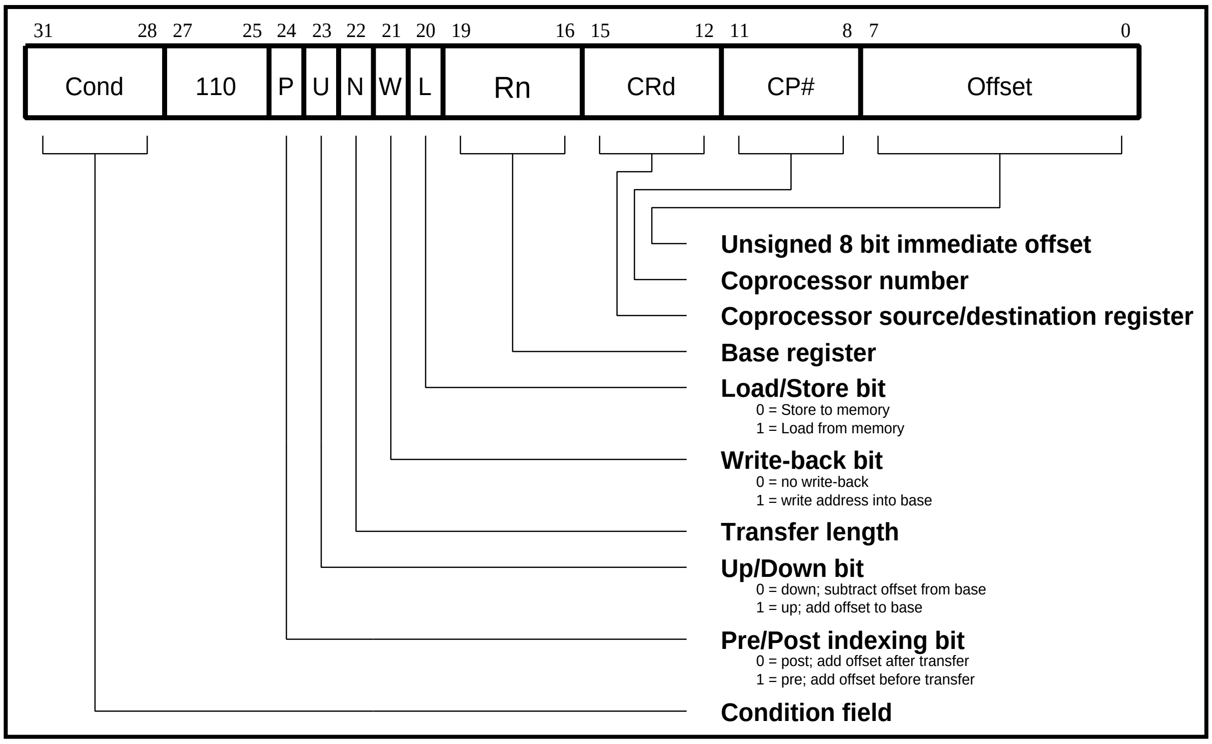


Figure 5-24: Coprocessor data transfer instructions

5.13.1 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 example:

N=0	could select the transfer of a single register
N=1	could select the transfer of all the registers for context switching.

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

The 8 bit unsigned immediate offset is shifted left 2 bits and either added to (U=1) or subtracted from (U=0) the base register (Rn); this calculation may be performed either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0).

Note: 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.

5.13.3 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.

5.13.4 Use of R15

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

5.13.5 Data aborts

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

5.13.6 Instruction cycle times

All non-FPA LDC instructions are emulated in software: the number of cycles taken will depend on the coprocessor support software.

5.13.7 Assembler syntax

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

LDC	load from memory to coprocessor
STC	store from coprocessor to memory
{L}	when present perform long transfer (N=1), otherwise perform short transfer (N=0)
{cond}	two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
p#	the unique number of the required coprocessor
cd	is an expression evaluating to a valid coprocessor register number that is placed in the CRd field

<Address> can be:

1 An expression which generates an address:

```
<expression>
```

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

2 A pre-indexed addressing specification:

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

3 A post-indexed addressing specification:

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

Rn	is an expression evaluating to a valid processor register number. Note, if Rn is R15 then the assembler will subtract 8 from the offset value to allow for processor pipelining.
{!}	write back the base register (set the W bit) if ! is present

5.13.8 Examples

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

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

5.14 Coprocessor Register Transfers (MRC, MCR)

Use of the MRC or MCR instruction on the ARM processor to a coprocessor other than to the FPA or to coprocessor #15 will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

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 5-25: Coprocessor register transfer instructions*.

This class of instruction is used to communicate information directly between the ARM processor 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.

Note: *The ARM processor has an internal coprocessor (#15) for control of on-chip functions. Accesses to this coprocessor are performed during coprocessor register transfers.*

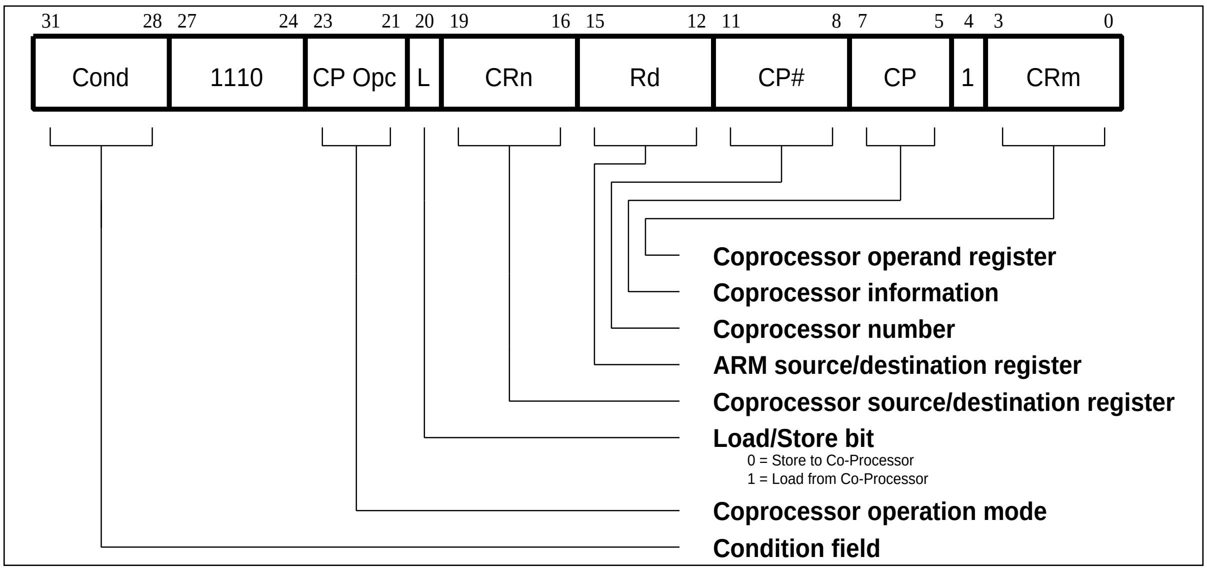

31 28 27 24 23 21 20 19 16 15 12 11 8 7 5 4 3 0
Cond
1110
CP Opc
L
CRn
Rd
CP#
CP
1
CRm
Coprocessor operand register
Coprocessor information
Coprocessor number
ARM source/destination register
Coprocessor source/destination register
Load/Store bit
0 = Store to Co-Processor
1 = Load from Co-Processor
Coprocessor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

Figure 5-25: Coprocessor register transfer instructions

5.14.1 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.

5.14.2 Transfers to R15

When a coprocessor register transfer to the ARM processor 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.

5.14.3 Transfers from R15

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

5.14.4 Instruction cycle times

Access to the internal configuration register takes 3 internal cycles. All non-FPA MRC instructions default to software emulation, and the number of cycles taken will depend on the coprocessor support software.

5.14.5 Assembler syntax

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

where:

`MRC`	move from coprocessor to ARM7500FE register (L=1)
`MCR`	move from ARM7500FE register to coprocessor (L=0)
`{cond}`	two character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
`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 ARM processor 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

5.14.6 Examples

```
MRC   2,5,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   6,0,R4,c6      ;request coproc 6 to perform operation 0
                     ;on R4 and place the result in c6
MRCEQ 3,9,R3,c5,c6,2 ;conditionally request coproc 2 to
                     ;perform
                     ;operation 9 (type 2) on c5 and c6, and
                     ;transfer the result back to R3
```

5.15 Undefined Instruction

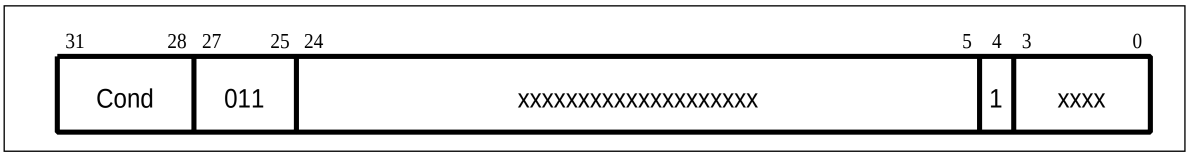


Figure 5-26: Undefined instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction format is shown in *Figure 5-26: Undefined instruction* on page 5-43.

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

5.15.1 Assembler syntax

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

5.16 Instruction Set Examples

The following examples show ways in which the basic ARM processor 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.

5.16.1 Using the conditional instructions

1 using conditionals for logical OR

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

can be replaced by

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

2 absolute value

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

3 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
```

4 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:=”.”
```

5 division and remainder

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

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

5.16.2 Pseudo random binary sequence generator

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

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

5.16.3 Multiplication by constant using the barrel shifter

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
```

5.16.4 Loading a word from an unknown alignment

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

5.16.5 Loading a halfword (Little-endian)

```
LDR   Ra, [Rb,#2]    ;get halfword to bits 15:0
MOV   Ra,Ra,LSL #16  ;move to top
MOV   Ra,Ra,LSR #16  ;and back to bottom
                     ;use ASR to get sign extended version
```

5.16.6 Loading a halfword (Big-endian)

```
LDR   Ra, [Rb,#2]    ;get halfword to bits 31:16
MOV   Ra,Ra,LSR #16  ;and back to bottom
                     ;use ASR to get sign extended version
```

5.17 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in *Table 5-7: ARM instruction speed summary* on page 5-48.

These figures assume that the instruction is actually executed.

Unexecuted instructions take one instruction fetch cycle.

Instruction	Cycle count
Data Processing - normal with register specified shift with PC written with register specified shift & PC written	1 instruction fetch 1 instruction fetch and 1 internal cycle 3 instruction fetches 3 instruction fetches and 1 internal cycle
MSR, MRS	1 instruction fetch
LDR - normal if the destination is the PC	1 instruction fetch, 1 data read and 1 internal cycle 3 instruction fetches, 1 data read and 1 internal cycle
STR	1 instruction fetch and 1 data write
LDM - normal if the destination is the PC	1 instruction fetch, n data reads and 1 internal cycle 3 instruction fetches, n data reads and 1 internal cycle
STM	1 instruction fetch and n data writes
SWP	1 instruction fetch, 1 data read, 1 data write and 1 internal cycle
B,BL	3 instruction fetches
SWI, trap	3 instruction fetches
MUL,MLA	1 instruction fetch and m internal cycles
CDP	1 instruction fetch and *b* internal cycles
LDC	1 instruction fetch, *n* data reads, and *b* internal cycles
STC	1 instruction fetch, *n* data writes, and *b* internal cycles
MCR	1 instruction fetch and *b*+1 internal cycles
MRC	1 instruction fetch and *b*+1 internal cycles

Table 5-7: ARM instruction speed summary

Where:

n	is the number of words transferred.
m	is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between *2^(2m-3)* and *2^(2m-1)-1* takes 1S+mI cycles for *1<m>16*. Multiplication by 0 or 1 takes 1S+1I cycles, and multiplication by any number greater than or equal to 2^(29) takes 1S+16I cycles. The maximum time for any multiply is thus 1S+16I cycles.
b	is the number of cycles spent in the coprocessor busy-wait loop.

The time taken for:

- an internal cycle will always be one FCLK cycle
- an instruction fetch and data read will be FCLK if a cache hit occurs, otherwise a full memory access is performed.
- a data write will be FCLK if the write buffer (if enabled) has available space, otherwise the write will be delayed until the write buffer has free space. If the write buffer is not enabled a full memory access is always performed.
- memory accesses are dealt with elsewhere in the ARM7500FE datasheet.
- coprocessor instructions depends on whether the instruction is executed by:

the FPA	See *Chapter 10: Floating-Point Instruction Set* for details of floating-point instruction cycle counts.
coprocessor #15	MCR, MRC to registers 0 to 7 only. In this case *b* = 0.
software emulation	For all other coprocessor instructions, the undefined instruction trap is taken.