4.0 Instruction Set

4.1 Instruction Set Summary

A summary of the ARM60 instruction set is shown in *Figure 5: 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.

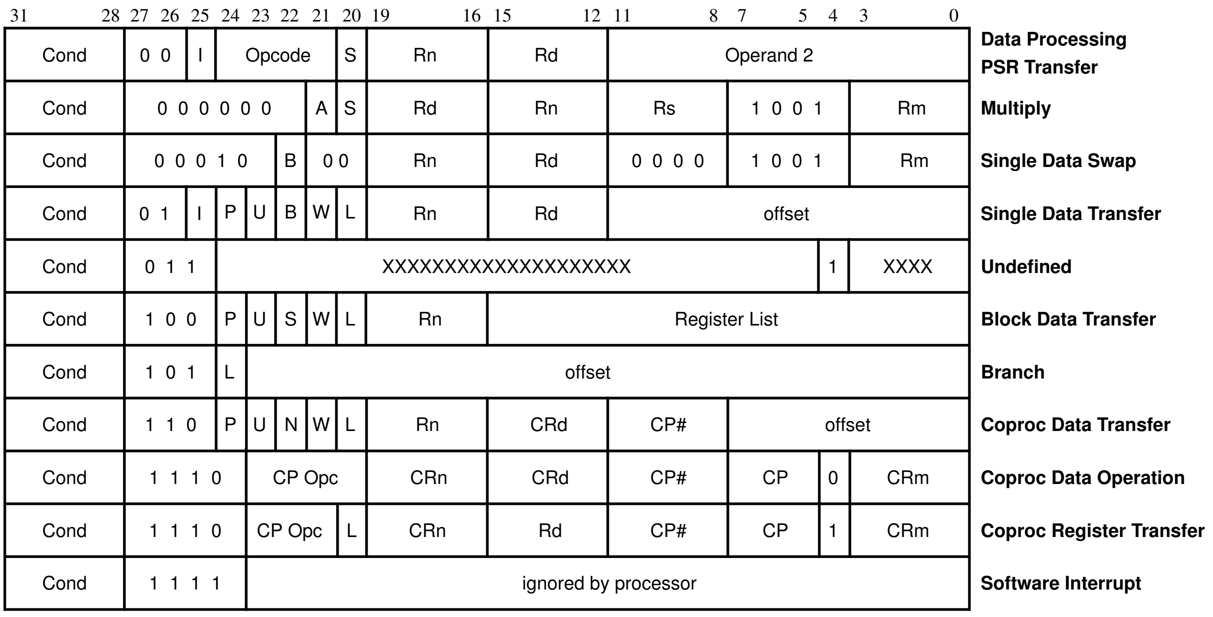
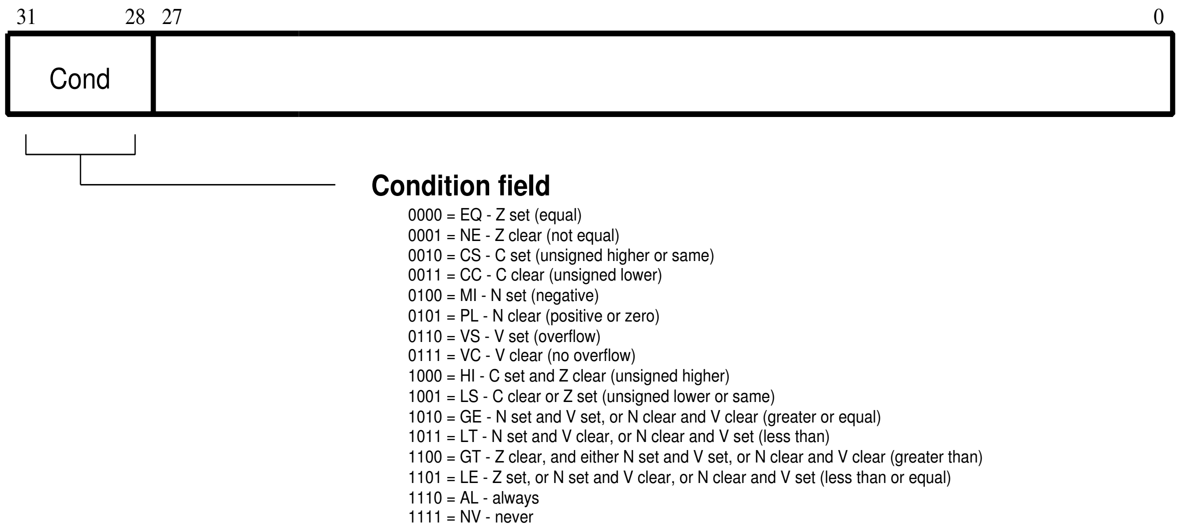

31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 5 4 3 0
Cond | 0 0 | I | Opcode | S | Rn | Rd | Operand 2 | Data Processing PSR Transfer
Cond | 0 0 0 0 0 0 | A | S | Rd | Rn | Rs | 1 0 0 1 | Rm | Multiply
Cond | 0 0 0 1 0 | B | 0 0 | Rn | Rd | 0 0 0 0 | 1 0 0 1 | Rm | Single Data Swap
Cond | 0 1 | I | 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

Figure 5: Instruction Set Summary

4.2 The Condition Field



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
1111 = NV - never

Figure 6: Condition Codes

All ARM60 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 encoding is shown in *Figure 6: Condition Codes*.

If the *always* (AL) condition is specified, the instruction will be executed irrespective of the flags. The *never* (NV) class of condition codes shall 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.

The other condition codes have meanings as detailed in *Figure 6: Condition Codes*, for instance code 0000 (EQual) causes the instruction to be executed 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 will not be executed.

4.3 Branch and Branch with link (B, BL)

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 7: 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.

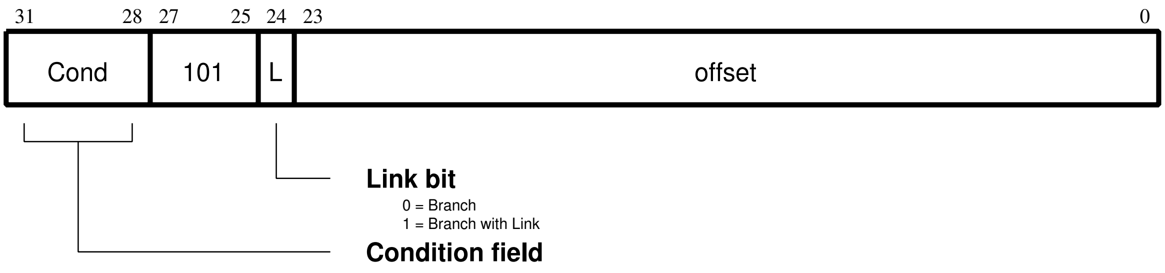


Figure 7: Branch Instructions

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

4.3.1 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 LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

4.3.2 Instruction Cycle Times

Branch and Branch with Link instructions take 2S + 1N incremental cycles, where S and N are as defined in section 5.1 Cycle types on page 65.

4.3.3 Assembler syntax

B{L}{cond} <expression>

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

{cond} is a two-char mnemonic as shown in *Figure 6: Condition Codes* (EQ, NE, VS etc). If absent then AL (ALways) will be used.

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

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

4.3.4 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
```

4.4 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 8: Data Processing Instructions*.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set. The instructions and their effects are listed in *Table 4: ARM Data Processing Instructions*.

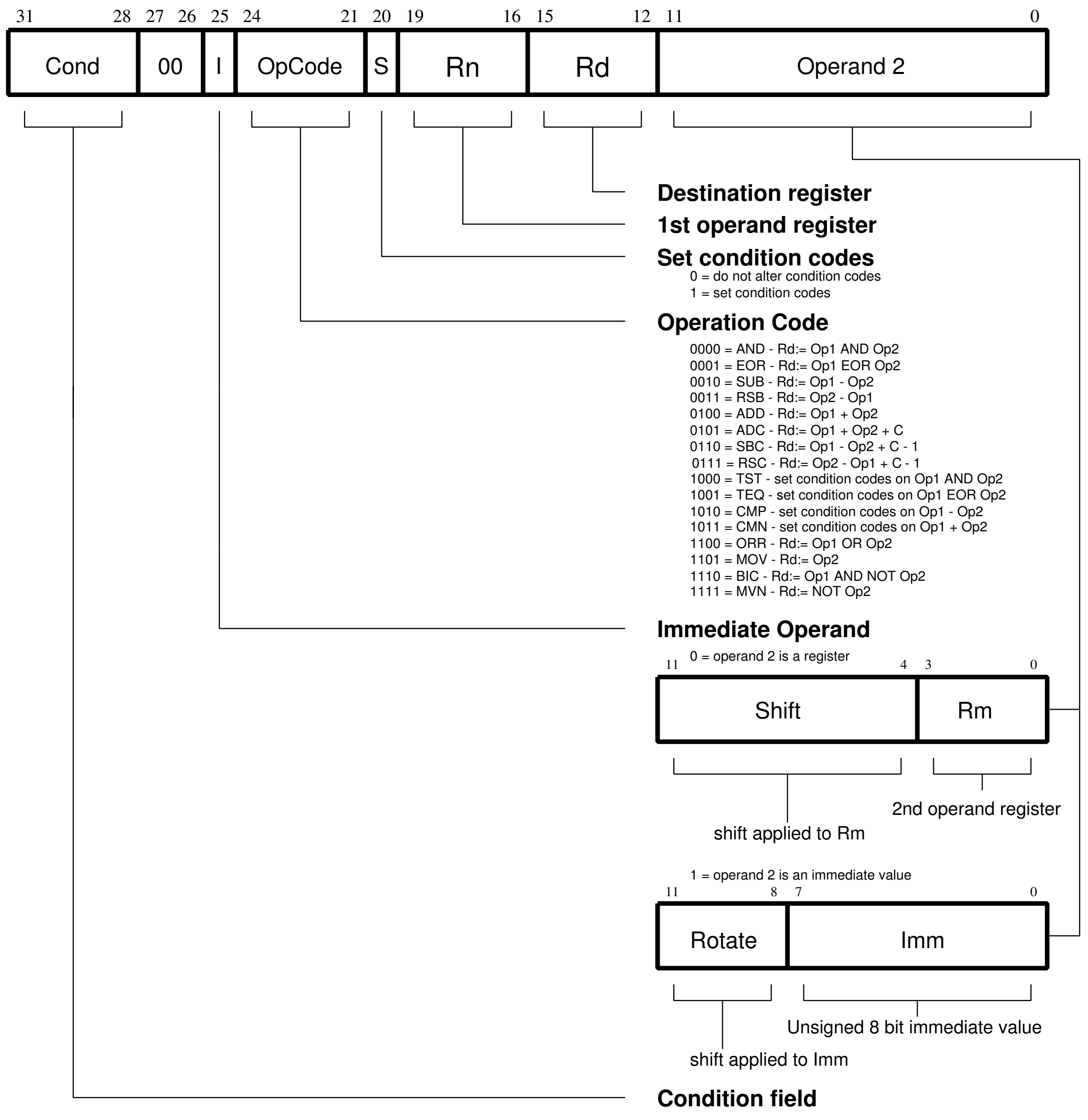

31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011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

Figure 8: Data Processing Instructions

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

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

Table 4: ARM Data Processing Instructions

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

4.4.2 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 9: ARM Shift Operations*.

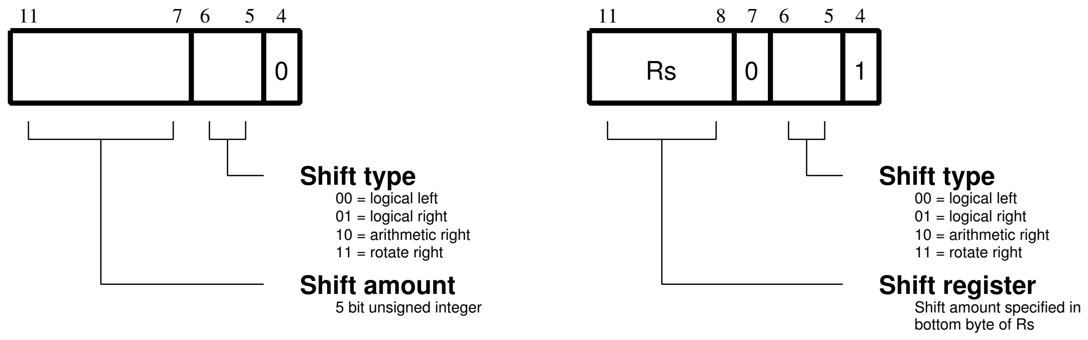


Figure 9: ARM Shift Operations

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 10: Logical Shift Left*.

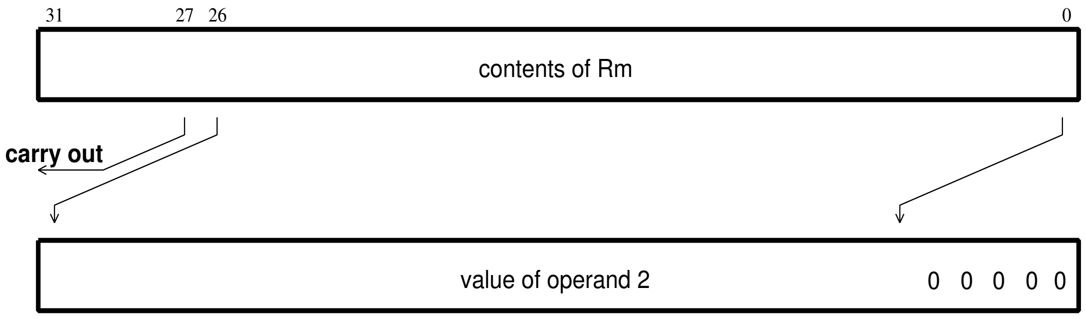


Figure 10: Logical Shift Left

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.

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 11: Logical Shift Right*.

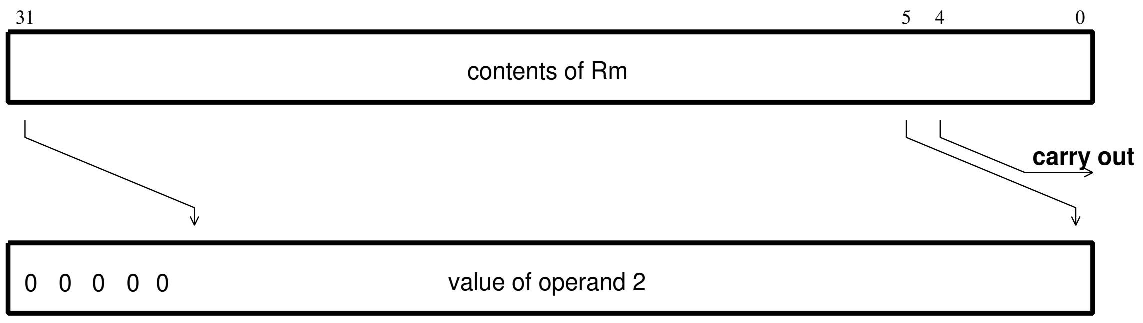


Figure 11: Logical Shift Right

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

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in *Figure 12: Arithmetic Shift Right*.

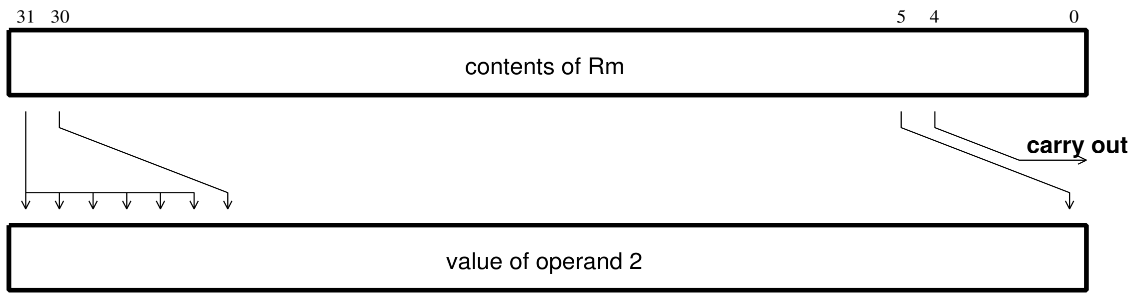


Figure 12: Arithmetic Shift Right

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

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in *Figure 13: Rotate Right*.

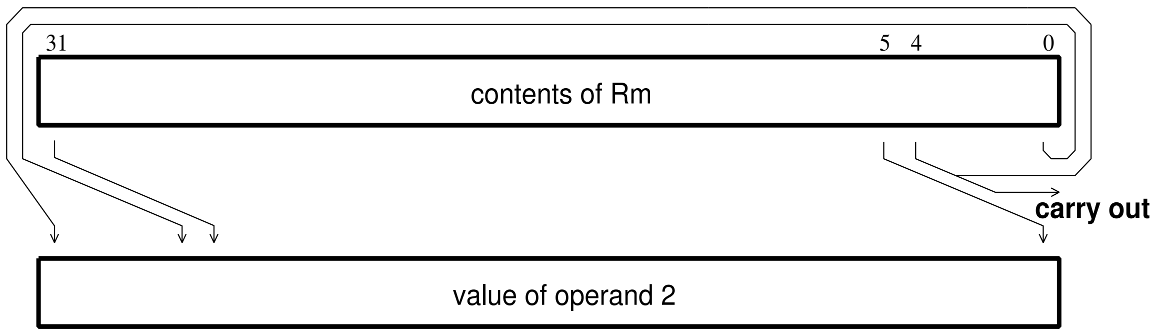


Figure 13: Rotate Right

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 14: Rotate Right Extended*.

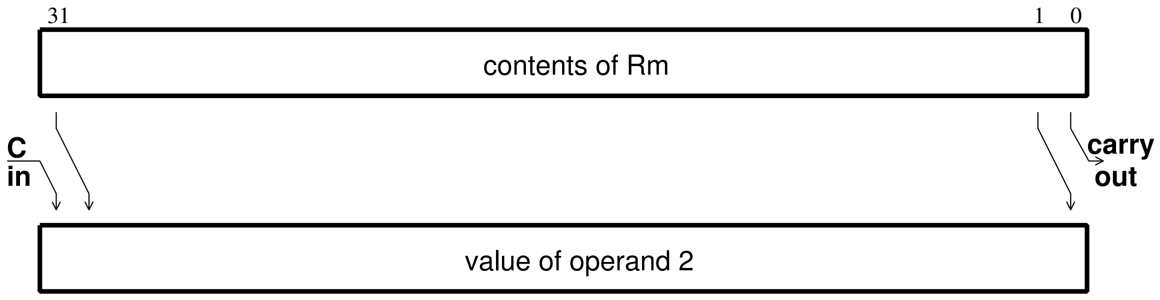


Figure 14: Rotate Right Extended

Register specified shift amount

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

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

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

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

(1) LSL by 32 has result zero, carry out equal to bit 0 of Rm.

(2) LSL by more than 32 has result zero, carry out zero.

(3) LSR by 32 has result zero, carry out equal to bit 31 of Rm.

(4) LSR by more than 32 has result zero, carry out zero.

(5) ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.

(6) ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.

(7) ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

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

4.4.3 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.

4.4.4 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. This form of instruction shall not be used in User mode.

4.4.5 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.

4.4.6 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.

4.4.7 Instruction Cycle Times

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

Normal Data Processing	1S
Data Processing with register specified shift	1S + 1I

Data Processing with PC written	2S + 1N
Data Processing with register secified shift and PC written	2S +1N + 1I

S and N are as defined in section 5.1 Cycle types on page 65.

4.4.8 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 6: Condition Codes*

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

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

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

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

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

4.4.9 Examples

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

TEQS     R4,#3                ; test R4 for equality with 3
                              ; (the S is in fact redundant as the
                              ; assembler inserts it automatically)

SUB      R4,R5,R7,LSR R2      ; logical right shift R7 by the number in
                              ; the bottom byte of R2, subtract result
                              ; from R5, and put the answer into R4

MOV      PC,R14               ; return from subroutine

MOVS     PC,R14               ; return from exception and restore CPSR
                                from SPSR_mode
```

4.5 PSR Transfer (MRS, MSR)

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

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 15: PSR Transfer*.

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

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

4.5.1 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.

R15 shall not be specified as the source or destination register.

A further restriction is that no attempt shall be made to access an SPSR in User mode, since no such register exists.

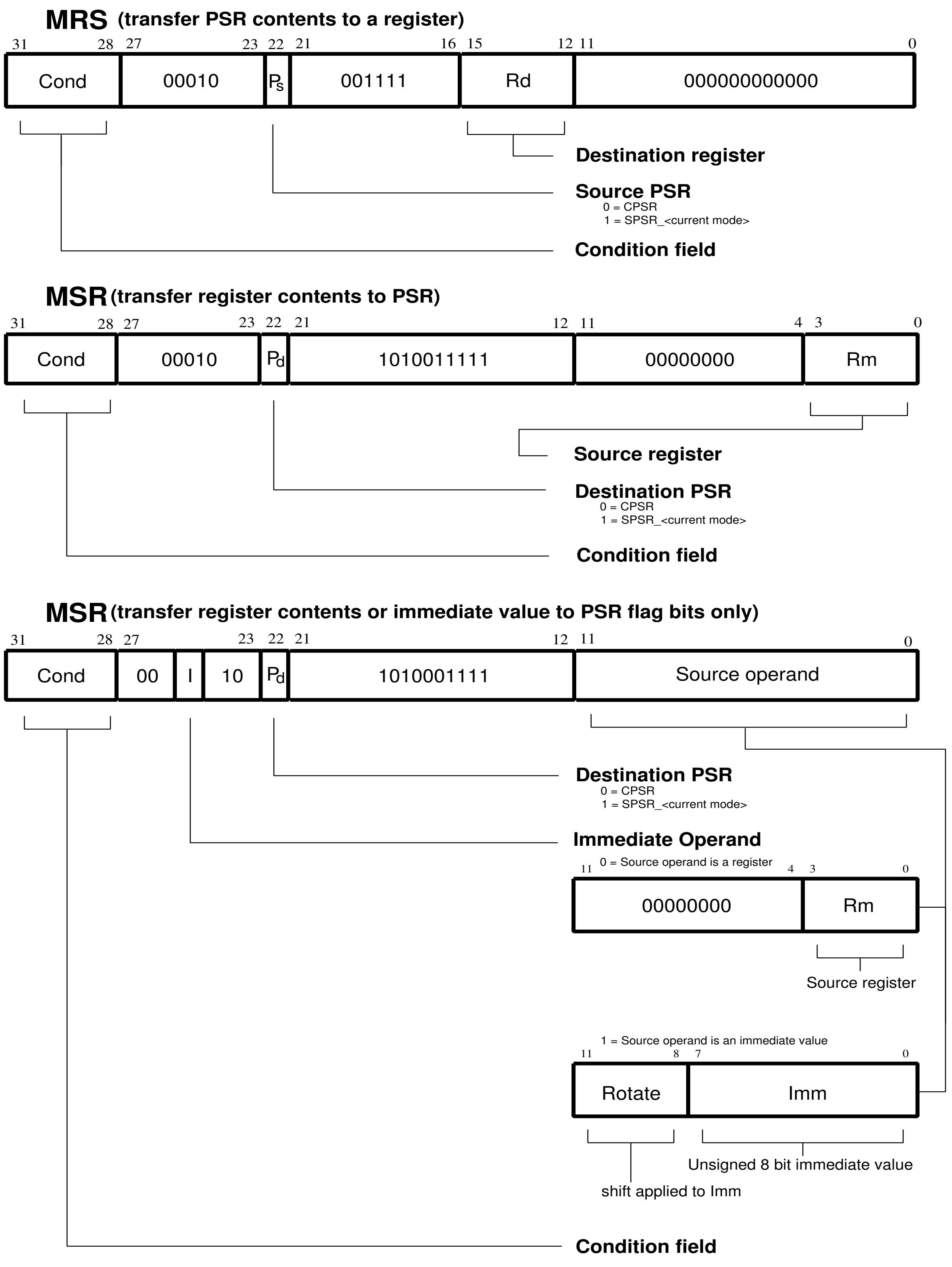


Figure 15: PSR Transfer

4.5.2 Reserved bits

Only eleven bits of the PSR are defined in ARM60 (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. To ensure the maximum compatibility between ARM60 programs and future processors, the following rules should be observed:

(1) The reserved bits shall be preserved when changing the value in a PSR.

(2) Programs shall 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.

e.g. 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, an immediate 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)
```

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

4.5.3 Instruction Cycle Times

PSR Transfers take 1S incremental cycles, where S is as defined in section 5.1 Cycle types on page 65.

4.5.4 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 symbolise a 32 bit value of which the most significant four bits are written to the N,Z,C & V flags respectively.

{cond} - two-character condition mnemonic, see *Figure 6: Condition Codes*

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

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

<psrf> is CPSR_flg or SPSR_flg

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

4.5.5 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]
```

4.6 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 16: 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.

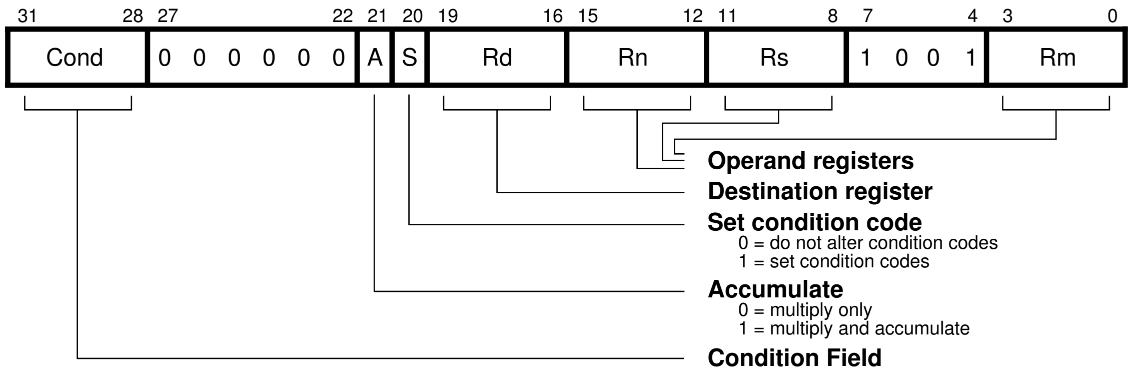


Figure 16: Multiply Instructions

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

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

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

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

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

4.6.2 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.

4.6.3 Instruction Cycle Times

The Multiply instructions take 1S + mI incremental cycles to execute, where S and I are as defined in section 5.1 Cycle types on page 65.

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 *m* 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.

4.6.4 Assembler syntax

MUL{cond}{S} Rd,Rm,Rs

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

{cond} - two-character condition mnemonic, see *Figure 6: Condition Codes*

{S} - set condition codes if S present

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

4.6.5 Examples

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

4.7 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 17: 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.

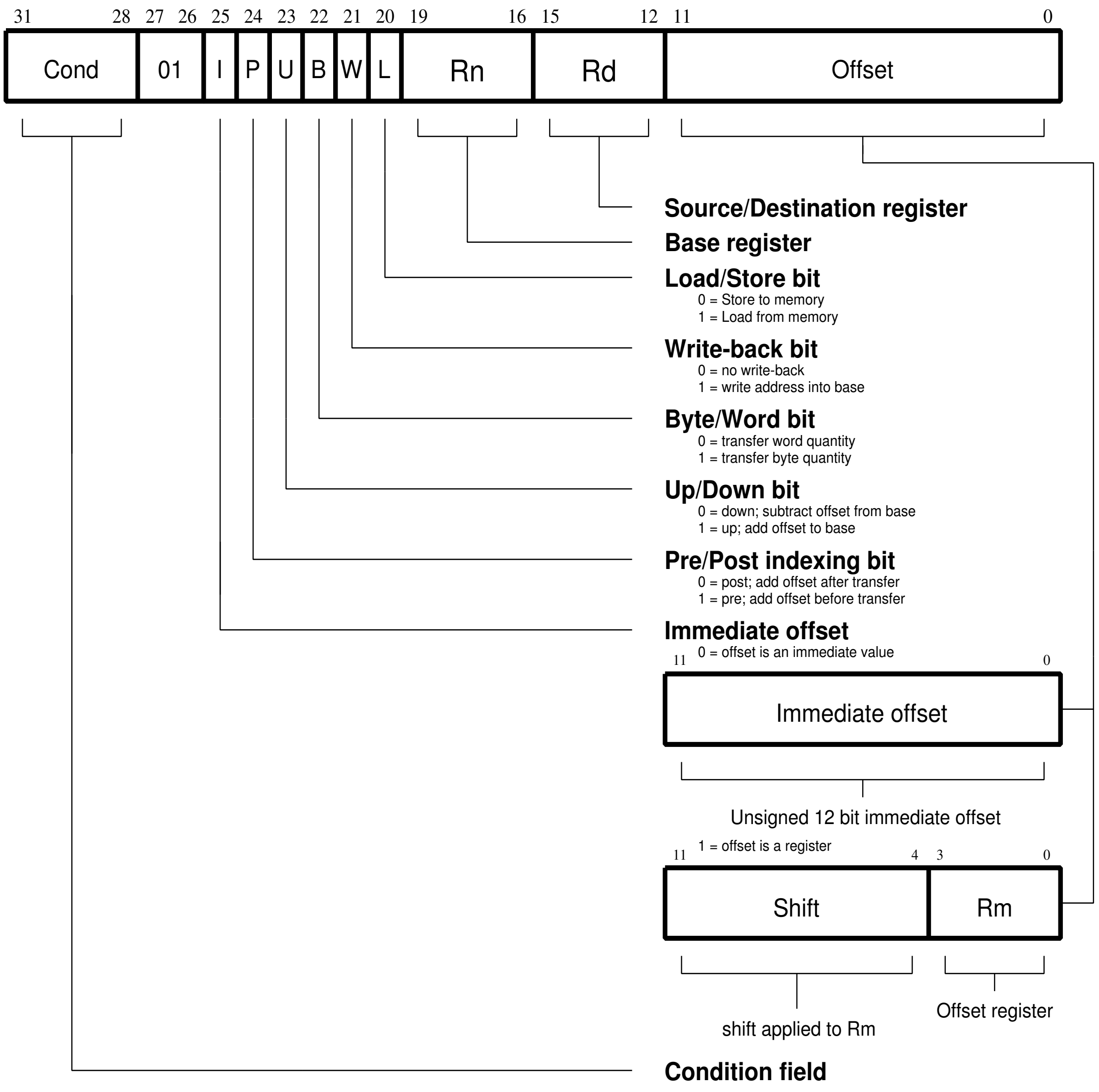


Figure 17: Single Data Transfer Instructions

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

4.7.2 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 *4.4.2 Shifts*.

4.7.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM60 register and memory.

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

Little Endian Configuration

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

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

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

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

Big Endian Configuration

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

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

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

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

4.7.4 Use of R15

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

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

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

4.7.5 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.

4.7.6 Data Aborts

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

ARM60 supports two types of Data Abort processing depending on the **LATEABT** control signal. When set for Early Aborts, any base register write-back which would have occurred is prevented in the event of an abort. When configured for Late Aborts, this write-back is allowed to take place and the Abort handler must correct this before allowing the instruction to be re-executed.

4.7.7 Instruction Cycle Times

Normal LDR instructions take 1S + 1N + 1I and LDR PC take 2S + 2N +1I incremental cycles, where S,N and I are as defined in section 5.1 Cycle types on page 65.

STR instructions take 2N incremental cycles to execute.

4.7.8 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 6: Condition Codes*

{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:

(i) 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.

(ii) 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>

(iii) 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 ARM60 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.

4.7.9 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
```

4.8 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 18: Block Data Transfer Instructions*.

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

4.8.1 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.

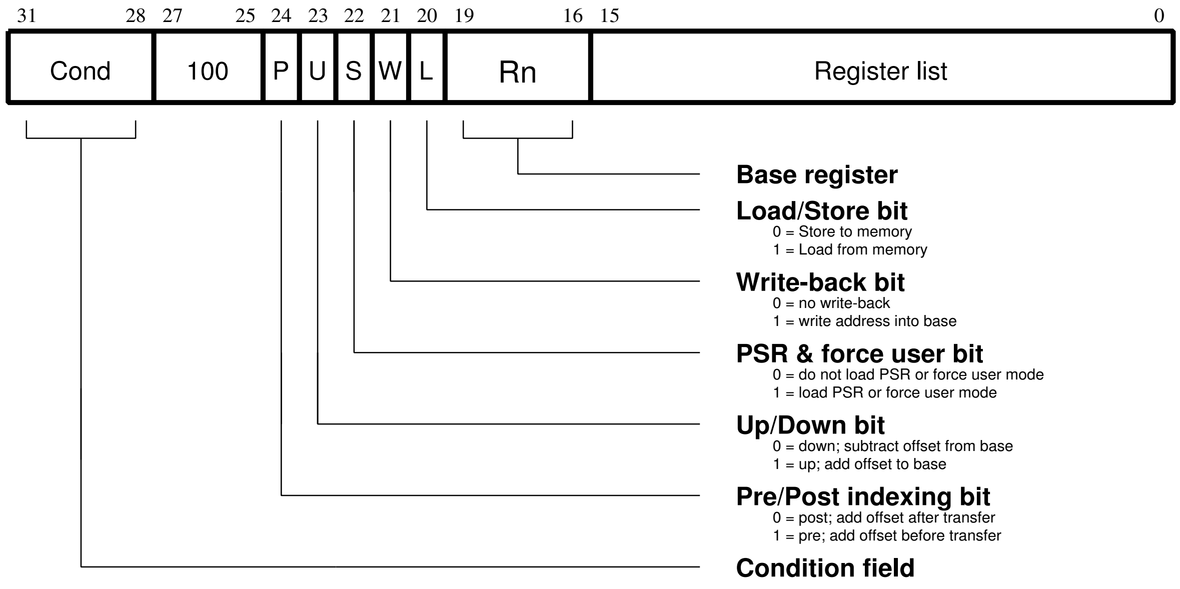

31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down 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 18: Block Data Transfer Instructions

4.8.2 Addressing modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of the modified base is required (W=1). *Figure 19: Post-increment addressing*, *Figure 20: Pre-increment addressing*, *Figure 21: Post-decrement addressing* and *Figure 22: Pre-decrement addressing* show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

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

4.8.3 Address Alignment

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

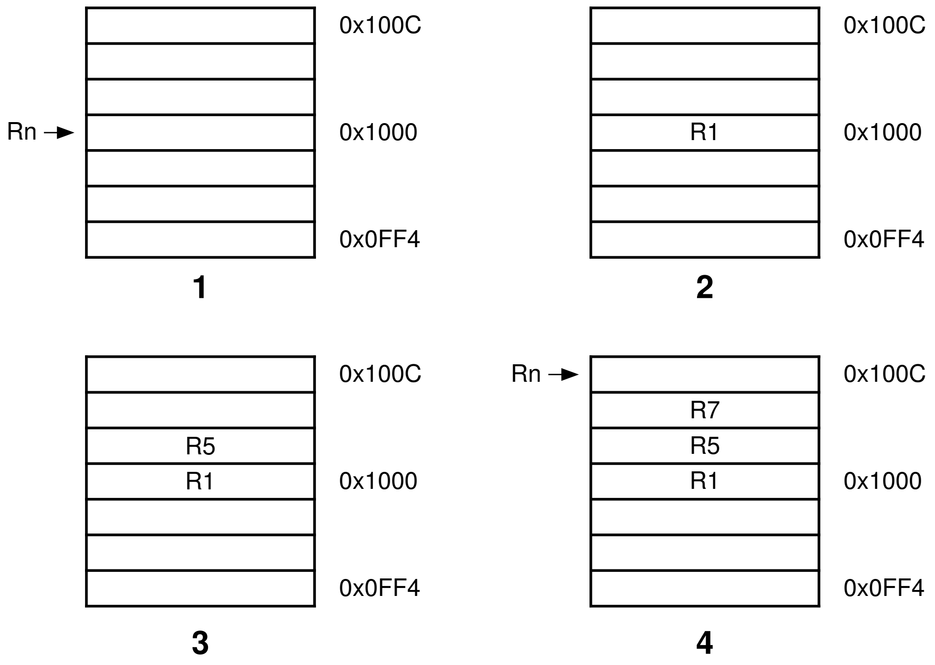


Figure 19: Post-increment addressing

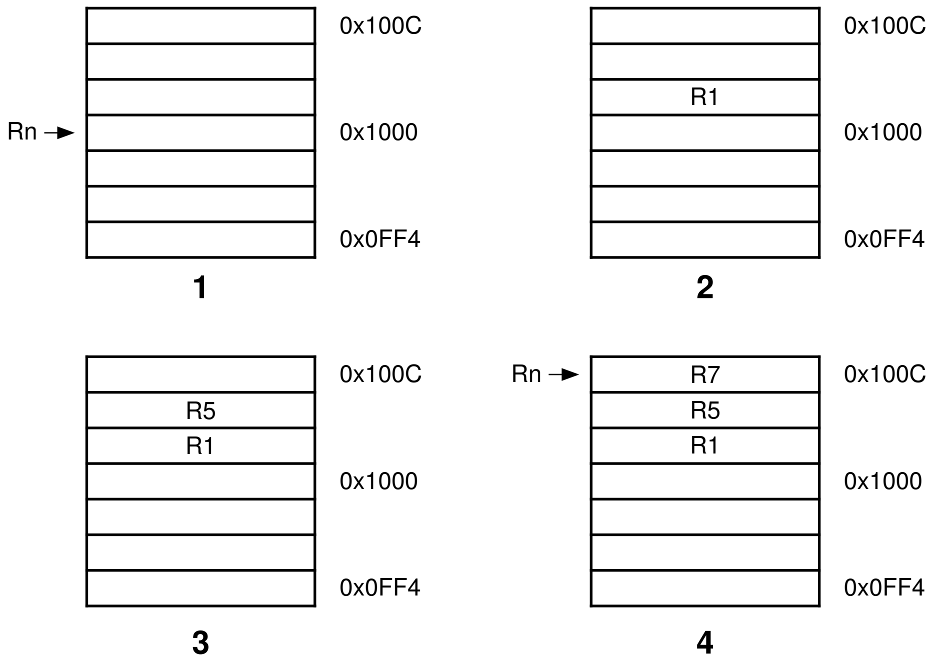


Figure 20: Pre-increment addressing

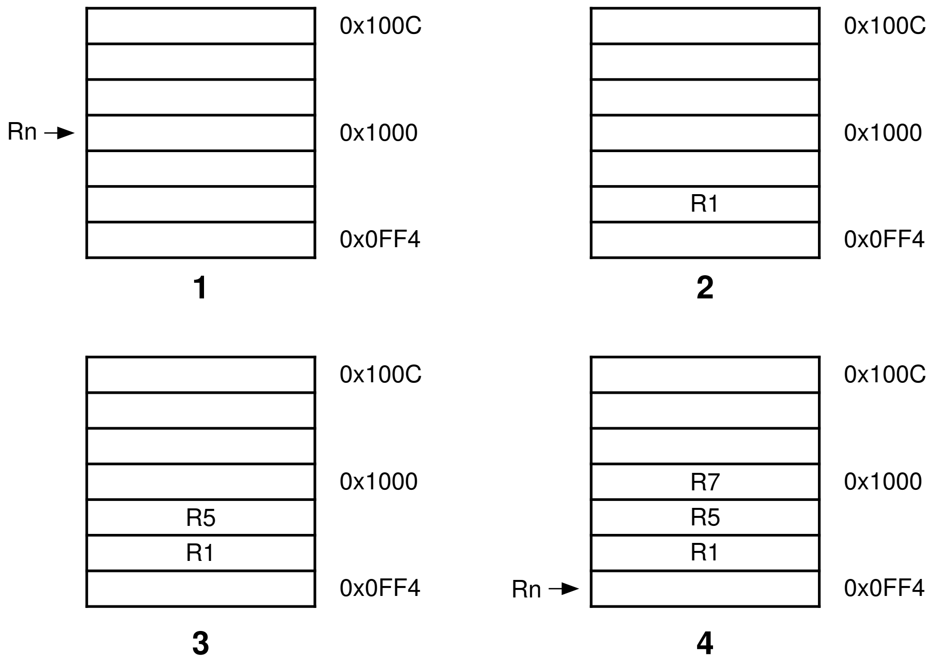


Figure 21: Post-decrement addressing

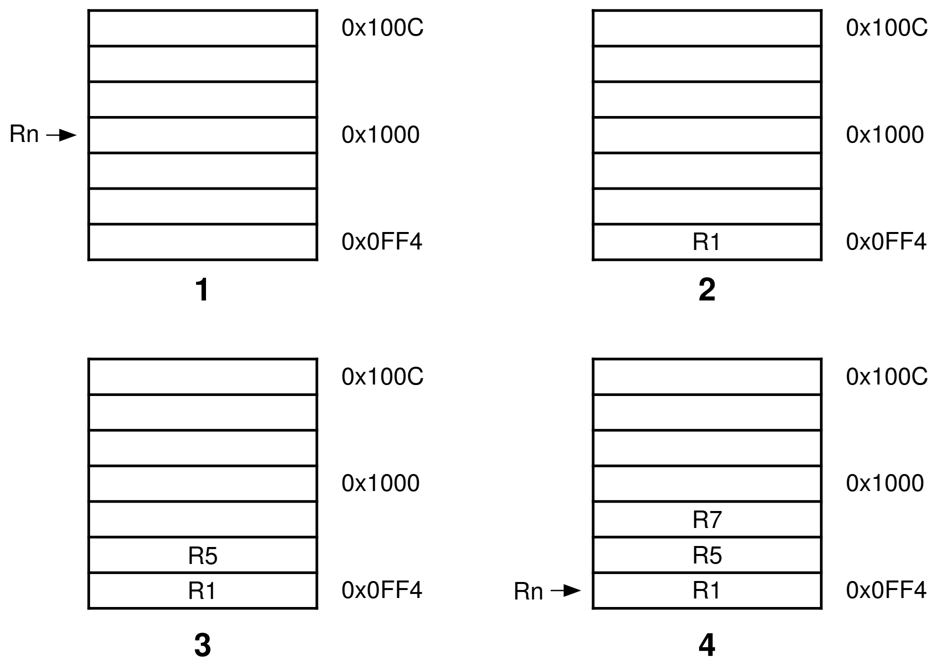


Figure 22: Pre-decrement addressing

4.8.4 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).

4.8.5 Use of R15 as the base

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

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

4.8.7 Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **ABORT** signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM60 is to be used in a virtual memory system.

The state of the **LATEABT** control signal does not affect the behaviour of LDM and STM instructions in the event of a memory abort exception.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM60 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 ARM60 detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

(i) 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.

(ii) The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

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

4.8.8 Instruction Cycle Times

Normal LDM instructions take nS + 1N + 1I and LDM PC takes (n+1)S + 2N + 1I incremental cycles, where S,N and I are as defined in section 5.1 Cycle types on page 65.

STM instructions take (n-1)S + 2N incremental cycles to execute.

n is the number of words transferred.

4.8.9 Assembler syntax

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

{cond} - two character condition mnemonic, see *Figure 6: Condition Codes*

Rn is an expression evaluating to a valid 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 set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

Addressing mode names

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

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

Table 5: Addressing Mode Names

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

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

4.8.10 Examples

```
LDMFD    SP!,{R0,R1,R2}       ; unstack 3 registers

STMIA    R0,{R0-R15}          ; save all registers

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

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

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

BL       somewhere            ; this nested call will overwrite R14

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

4.9 Single data swap (SWP)

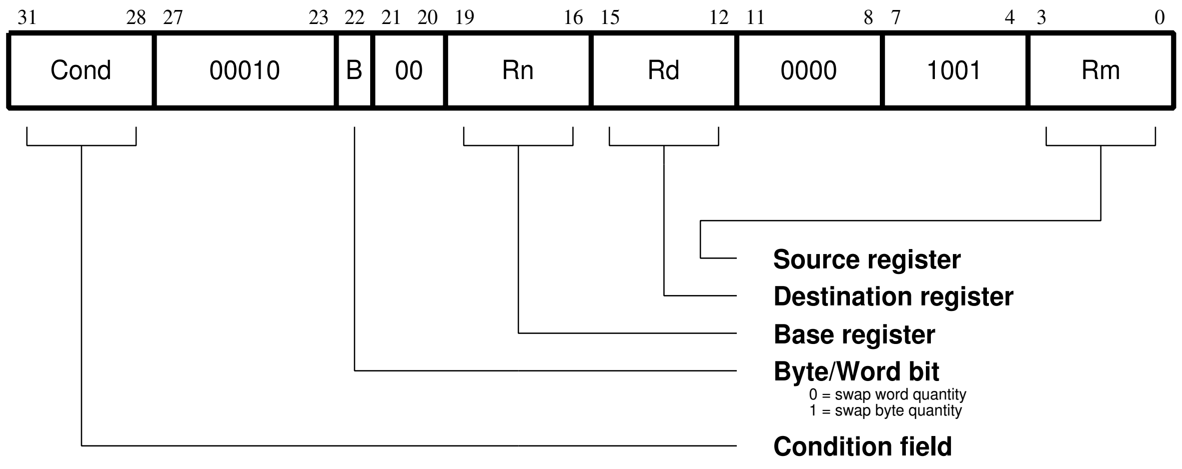


Figure 23: Swap Instruction

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 23: Swap Instruction*.

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

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

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

4.9.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM60 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

4.9.2 Use of R15

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

4.9.3 Data Aborts

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

Because no base register write-back is allowed, the behaviour of an aborted SWP instruction is the same regardless of the state of the **LATEABT** control signal.

4.9.4 Instruction Cycle Times

Swap instructions take 1S + 2N +1I incremental cycles to execute, where S,N and I are as defined in section 5.1 Cycle types on page 65.

4.9.5 Assembler syntax

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

{cond} - two-character condition mnemonic, see *Figure 6: Condition Codes*

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

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

4.9.6 Examples

```
SWP      R0,R1,[R2]            ; load R0 with the contents of R2, and
                               ; store R1 at R2

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

SWPEQ    R0,R0,[R1]            ; conditionally swap the contents of R2
                               ; with R0
```

4.10 Software interrupt (SWI)

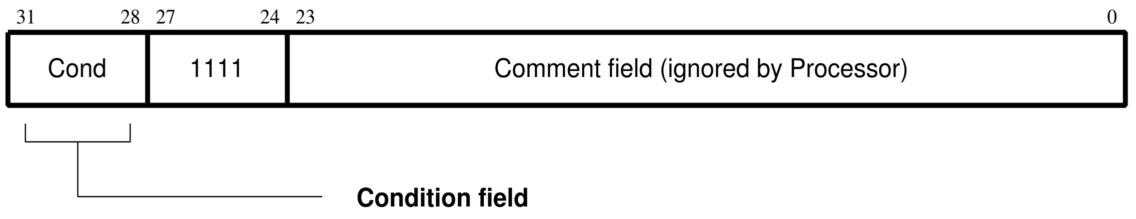


Figure 24: Software Interrupt Instruction

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 24: Software Interrupt Instruction*.

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

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

4.10.2 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.

4.10.3 Instruction Cycle Times

Software interrupt instructions take 2S + 1N incremental cycles to execute, where S and N are as defined in section 5.1 Cycle types on page 65.

4.10.4 Assembler syntax

SWI{cond} <expression>

{cond} - two character condition mnemonic, see *Figure 6: Condition Codes*

<expression> is evaluated and placed in the comment field (which is ignored by ARM60).

4.10.5 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
```

4.11 Coprocessor data operations (CDP)

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 25: Coprocessor Data Operation Instruction*.
This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to ARM60, 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 ARM60 activity allowing the coprocessor and ARM60 to perform independent tasks in parallel.

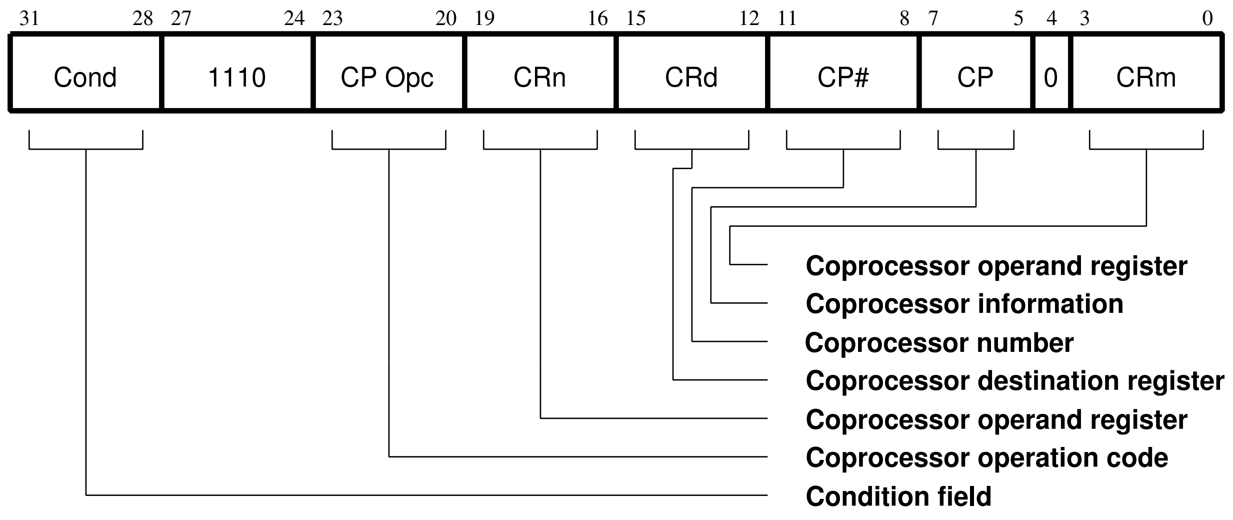


Figure 25: Coprocessor Data Operation Instruction

4.11.1 The Coprocessor fields

Only bit 4 and bits 24 to 31 are significant to ARM60; the remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

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

4.11.2 Instruction Cycle Times

Coprocessor data operations take 1S + bI incremental cycles to execute, where S and I are as defined in section 5.1 Cycle types on page 65.

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

4.11.3 Assembler syntax

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

{cond} - two character condition mnemonic, see *Figure 6: Condition Codes*

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

4.11.4 Examples

```
CDP      p1,10,c1,c2,c3        ; request coproc 1 to do operation 10
                               ; on CR2 and CR3, and put the result in CR1

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

4.12 Coprocessor data transfers (LDC, STC)

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 26: Coprocessor Data Transfer Instructions*.
This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessors's registers directly to memory. ARM60 is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

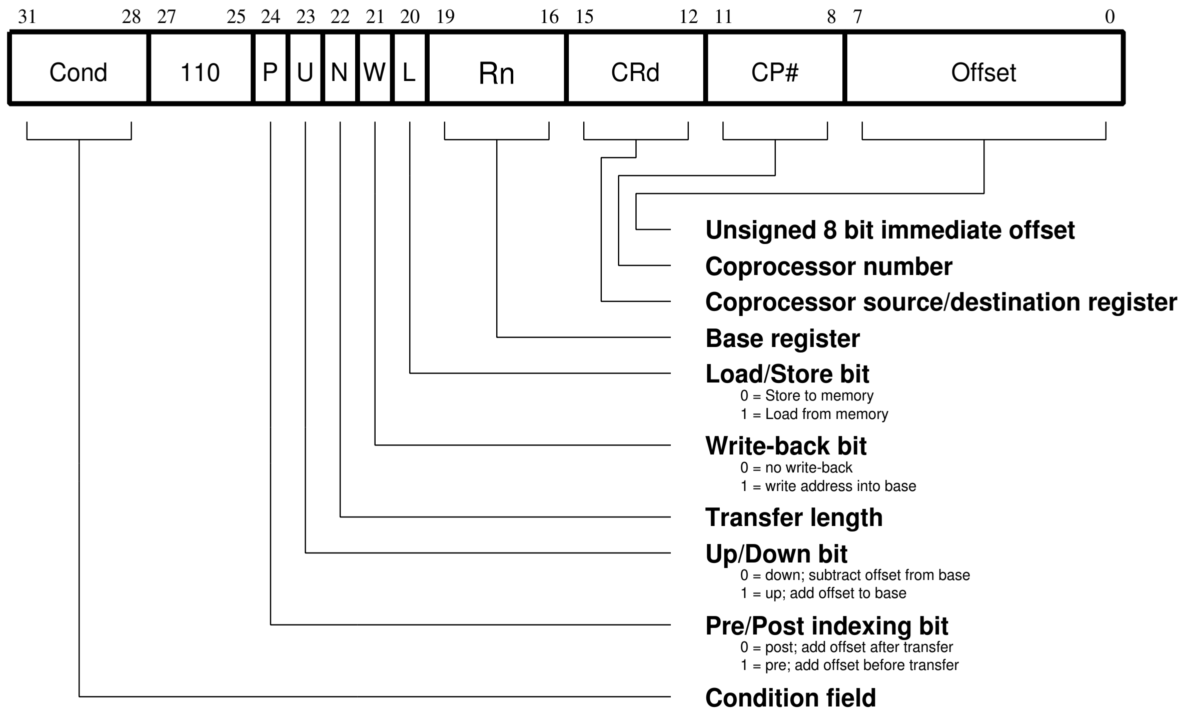


Figure 26: Coprocessor Data Transfer Instructions

4.12.1 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.

4.12.2 Addressing modes

ARM60 is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note, however, that the immediate offsets are 8 bits wide and specify word offsets for coprocessor data transfers, whereas they are 12 bits wide and specify byte offsets for single data transfers.

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

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

4.12.3 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.

4.12.4 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.

4.12.5 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.

The state of the **LATEABT** control signal does not affect the behaviour of LDC and STC instructions in the event of an Abort exception.

4.12.6 Instruction Cycle Times

Coprocessor data transfer instructions take (n-1)S + 2N + bI incremental cycles to execute, where S, N and I are as defined in section 5.1 Cycle types on page 65.

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

4.12.7 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 6: Condition Codes*

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:

(i) 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.

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

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

(iii) A post-indexed addressing specification:

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

Rn is an expression evaluating to a valid ARM60 register number. Note, if Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM60 pipelining.

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

4.12.8 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 that though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

4.13 Coprocessor register transfers (MRC, MCR)

The 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 27: Coprocessor Register Transfer Instructions*.

This class of instruction is used to communicate information directly between ARM60 and a coprocessor. An example of a coprocessor to ARM60 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 an ARM60 register. A FLOAT of a 32 bit value in an ARM60 register into a floating point value within the coprocessor illustrates the use of an ARM60 register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the ARM60 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 ARM60 has an internal coprocessor (#15) for control of on-chip functions. Accesses to this coprocessor are performed during coprocessor register transfers.

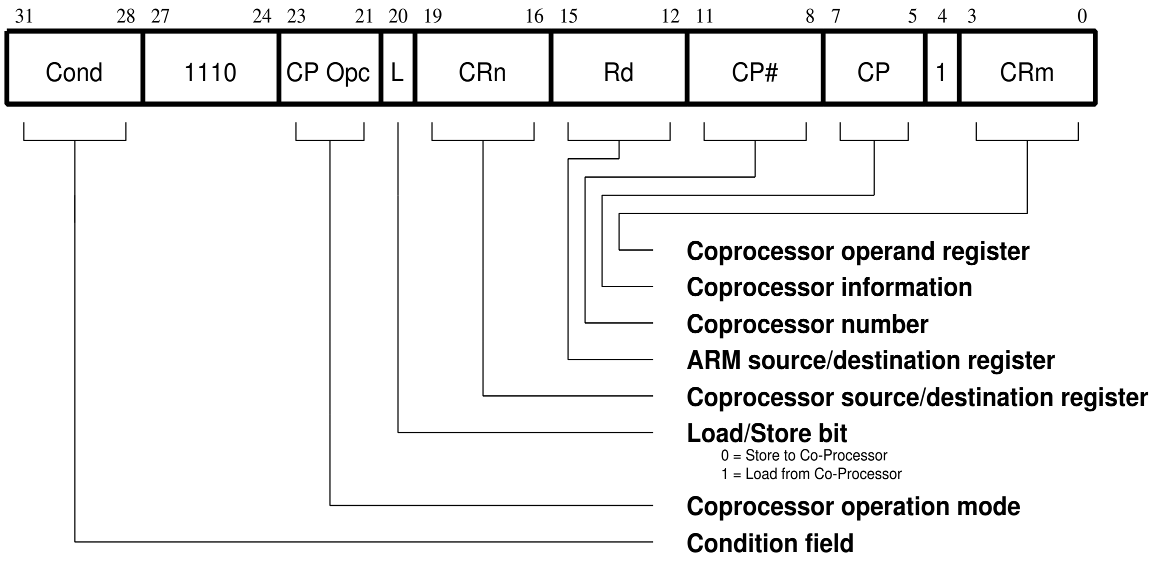

31 28 27 24 23 21 20 19 16 15 12 11 8 7 5 4 3 0
Cond
1110
CP Opc
L
CRn
Rd
CP#
CP
1
CRm
Coprocessor operand register
Coprocessor information
Coprocessor number
ARM source/destination register
Coprocessor source/destination register
Load/Store bit
0 = Store to Co-Processor
1 = Load from Co-Processor
Coprocessor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

Figure 27: Coprocessor Register Transfer Instructions

4.13.1 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.

4.13.2 Transfers to R15

When a coprocessor register transfer to ARM60 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.

4.13.3 Transfers from R15

A coprocessor register transfer from ARM60 with R15 as the source register will store the PC+12.

4.13.4 Instruction Cycle Times

MRC instructions take 1S + bI +1C incremental cycles to execute, where S, I and C are as defined in section 5.1 Cycle types on page 65.

MCR instructions take 1S + (b+1)I +1C incremental cycles to execute.

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

4.13.5 Assembler syntax

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

MRC - move from coprocessor to ARM60 register (L=1)

MCR - move from ARM60 register to coprocessor (L=0)

{cond} - two character condition mnemonic, see *Figure 6: Condition Codes*

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 ARM60 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

4.13.6 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
```

4.14 Undefined instruction

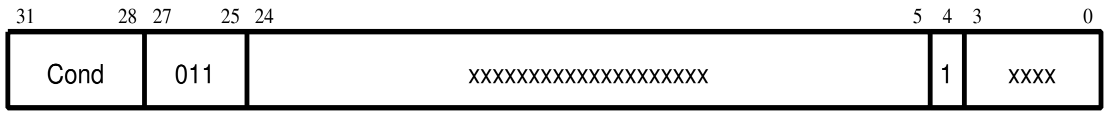


Figure 28: Undefined Instruction

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 28: Undefined Instruction*.

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

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

4.14.1 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.

4.15 Instruction Set Examples

The following examples show ways in which the basic ARM60 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.

4.15.1 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

```
                                     ; 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
```

4.15.2 Pseudo random binary sequence generator

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

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

4.15.3 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
```

4.15.4 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
```

4.15.5 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
```

4.15.6 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
```